C
LAMBDA CALCULUS
AND
COMPILER VERIFICATION

A study in Haskell of purely-functional techniques
for a formal specification of imperative programming languages
and an epistemically-sound verification of their compilers

PATRYK ZADARNOWSKI

A thesis submitted in fulfilment
of the requirements for the degree of
Doctor of Philosophy

University of New South Wales
School of Computer Science and Engineering

Sydney, January 2011

ORIGINALITY STATEMENT

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Patryk Zadarnowski
17 January 2011

COPYRIGHT STATEMENT

Patryk Zadarnowski
17 January 2011

AUTHENTICITY STATEMENT

I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.

Patryk Zadarnowski
17 January 2011

C
Lambda Calculus & Compiler Verification

PATRYK ZADARNOWSKI

University of New South Wales
School of Computer Science and Engineering
Sydney, January 2011

Abstract

Formal verification of a compiler is a long-standing problem in computer science and, although recent years have seen substantial achievements in the area, most of the proposed solutions do not scale very well with the complexity of modern software development environments. In this thesis, I present a formal semantic model of the popular C programming language described in the ANSI/ISO/IEC Standard 9899:1990, in the form of a mapping of C programs to computable functions expressed in a suitable variant of lambda calculus. The specification is formulated in a highly readable functional style and is accompanied by a complete Haskell implementation of the compiler, covering all aspects of the translation from a parse tree of a C program down to the actual sequence of executable machine instructions, resolving issues of separate compilation, allowing for optimising program transformations and providing rigorous guarantees of the implementation's conformance to a formal definition of the language.

In order to achieve these goals, I revisit the challenge of compiler verification from its very philosophical foundations. Beginning with the basic epistemic notions of knowledge, correctness and proof, I show that a fully rigorous solution of the problem requires a constructive formulation of the correctness criteria in terms of the translation process itself, in contrast with the more popular extensional approaches to compiler verification, in which correctness is generally defined as commutativity of the system with respect to a formal semantics of the source and target languages, effectively formalising various aspects of the compiler independently of each other and separately from its eventual implementation. I argue that a satisfactory judgement of correctness should always constitute a direct formal description of the job performed by the software being judged, instead of an axiomatic definition of some abstract property such as commutativity of the translation system, since the later approach fails to establish a crucial causal connection between a judgement of correctness and a knowledge of it.

The primary contribution of this thesis is the new notion of *linear correctness*, which strives to provide a constructive formulation of a compiler's validity criteria by deriving its judgement directly from a formal description of the language translation process itself. The approach relies crucially on a denotational semantic model of the source and target languages, in which the domains of program meanings are unified with the actual intermediate program representation of the underlying compiler implementation. By defining the concepts of a *programming language*, *compiler* and *compiler correctness* in category theoretic terms, I show that every linearly correct compiler is also valid in the more traditional sense of the word. Further, by presenting a complete verified translation of the standard C programming language within this framework, I demonstrate that linear correctness is a highly effective approach to the problem of compiler verification and that it scales particularly well with the complexity of modern software development environments.

Acknowledgments

My own interest in programming languages started in 1994, when I chanced to come by a book titled *Compilers—Principles, Techniques and Tools* in a local library. I join countless others in thanking Alfred Aho, Ravi Sethi and Jeffrey Ullman for their "Dragon Book".

My curiosity matured and developed under the guidance and support of Manuel M.T. Chakravarty, who introduced me to Haskell, lambda calculus and, in his role as my PhD supervisor, eventually instilled in me a passion for theoretical computer science. I am also indebted to Gabriele Keller for her invaluable support during my research and an initiation into the craft of formal reasoning, Philip Wadler, who in one short conversation managed to finally confer on me the elusive understanding of Curry-Howard isomorphism, my wife Sifa, who contributed countless hours to the thankless task of proof-reading the final manuscript, and the three anonymous reviewers who were charged with the mammoth undertaking of scrutinising the finished product and who have provided me with invaluable feedback on both the substance of my project and its execution.

Further, I would like to thank the School of Computer Science and Engineering at the University of New South Wales for providing a wonderful research environment and the much-needed funding during the first year of my PhD candidature, as well as Gernot Heiser for arranging a scholarship to support the subsequent two years of my research and Hewlett-Packard for providing the actual funds.

Last but not least, I would like to thank my father for five years of nagging and the much needed motivation, without which this work would have never been possible.

To my parents and my wife,
the three people without whom this work,
as this life, would not be possible.

Preface

In the beginning there was FORTRAN. Well, not really; first, there were a bunch of mathematicians, hackers and college students punching their programs bit by bit onto pieces of cardboard, patiently, a hole at a time, reminiscent of our ancestors who, gathered around a bonfire, tried to weave their first set of pants out of a pile of loose seaweed.

Then came FORTRAN. In a valiant effort to teach a machine a little English, John Backus, together with a group of his colleagues at IBM, created the *FORmulae TRANslator* and the first compiled high-level programming language (or a *rapid coding system* as it was called in the IBM literature) was born in the winter months of 1953.

FORTRAN was perhaps the biggest blunder in the history of computer science. Even though it came into being only a few years after the era of von Neumann machine was launched in the vaults of EDSAC labs at Cambridge, it already displayed all the characteristic traits of modern compilers: it was big, slow, complex, buggy, with an *ad hoc* design and a poor specification. Backus himself was later to say that they "did not know what they were doing" and commenced work on improving specification of programming languages even while he was still completing the original compiler implementation for IBM. In 1958, only a year after the first FORTRAN system was delivered, he introduced what is arguably the single most important invention in the history of programming languages: the Backus-Naur Form of context-free grammars. Soon after, Backus followed with another ground-breaking attempt at improving compiler design, recognising that the von Neumann programming style of FORTRAN offers a poor abstraction of the underlying hardware and introducing the world to declarative programming in 1978.

Yet FORTRAN and the plethora of other imperative languages that followed in its footsteps were here to stay. The world entered a rat race in which language designers continue adding insult to the injury by inventing new languages, new tastes and flavours and new language features, as if forgetting that the whole essence of abstraction is to *reduce* the number of distinct entities in the system, thus reducing the overall system complexity.

A wise man once said that it is futile to argue about taste and computer engineers definitely seem to have a sweet tooth for imperative programming. As early as 1965, Peter J. Landin noted that the then quintessential imperative language ALGOL is nothing more than a sugaring on top of Church's lambda calculus, a beautiful, minimal declarative system of programming based on the notion of a function abstraction. In this work, I continue in Landin's tradition by investigating the use of lambda calculus for compilation and verification of mainstream imperative languages, in particular the C programming language prevailing in the modern software industry.

TABLE OF CONTENTS

1

INTRODUCTION

There's no sense being exact about something
if you don't even know what you're talking about.
— John von Neumann

The idea of using English words to program a computer is due to Grace Hopper. In 1952, Hopper developed FLOW-MATIC [Hopper 57], a system for "automatic programming of business applications", which was later used as a basis for her design of COBOL, the first programming language standardised by the American National Standards Institute [ANSI 68]. However, it was not until after IBM released the first optimising FORTRAN compiler written by John Backus *et al.* [Backus 54] that the idea gained a broader public acceptance. The difficulties posed by specification of a fully-featured programming language were noted early and, in 1959, Backus introduced what is arguably the single most important invention in the history of programming languages: the Backus-Naur Form of context-free grammars [Backus 59]. Soon, a plethora of new languages and even more numerous compilers for them followed in Hopper's and Backus's footsteps.

Today, the issues involved in crafting an industrial-strength compiler are well understood and have been the subject of countless books and research papers. Nevertheless, modern compilers remain some of the most complex pieces of software in existence. The GNU C compiler suit consists of a mind-boggling 2.1 million lines of C code. The Open Watcom C/C++ compiler is even more formidable: its depository contains over 317,000 lines of assembly source on top of almost 2.5 million lines of C and nearly 2 million lines in auxiliary files. GHC, being written in a much higher level language, fares better, but still approaches almost 260,000 lines of Haskell with an additional 120,000 lines of C in its runtime library. Given these figures, it is perhaps unsurprising that programming errors are commonplace among virtually all compilers routinely deployed in the industry. Detailed statistics are hard to come by, but some data is available in the form of bug databases for the GNU C, Borland Pascal and Sun Java compilers. The GNU C bug database contains over 2800 entries as of May 2007. The commercial compilers from Borland and Sun are much more respectable at 50 and 90 entries each, although their apparent success is probably influenced more by the lower number of users reporting faults than the actual reliability of the software.

Of course, most of the programming errors found in commercially deployed compilers pertain to corner cases of their behaviour that are unlikely to be encountered in practical applications. Nevertheless, they pose a serious issue for designers of safety-critical systems such as those found in avionics and medical equipment, given that,

when correctness is paramount, every tool employed in the course of program construction must be shown to be sound before reliability of the final product can be ascertained. Since its original identification in the 1950s, the problem of compiler correctness has attracted a lot of attention in both the academic and industrial communities and, to this day, it remains one of the busiest areas of research in the history of computer science.

1.1 History of the Problem

It is perhaps appropriate that the problem of validating a compiler was raised for the first time by Grace Hopper, the very same person who reported the famous "original bug" in 1947 while working on the Mach II computer at Harvard University. In 1952, Hopper published a design of FLOW-MATIC, arguably the first high level programming language ever developed [Hopper 57]. When, seven years later, her ideas were finally adopted in the form of a specification for the COBOL programming language [ANSI 68], Hopper instigated and led development of validation procedures for the newly created compiler, which remained the primary focus of her research throughout Hopper's long and successful career.

However, the first actual attempt at a formal verification of a compiler is due to John McCarthy and James Painter [McCarthy 67]. In their 1967 paper, McCarthy and Painter analyse a small translator for a language of arithmetic expressions that targets a simple von Neumann machine with a single accumulator register. They define an abstract dynamic semantics for their language and present a complete correctness proof for its compiler. Unfortunately, their approach proved resilient to all attempts at extending it to any of the more complete software development environments of the time. In 1973, Morris published a seminal paper on compiler correctness [Morris 73] in which he proposes to capture the essence of the problem in well-defined mathematical terms by describing a compiler as a function from source programs to their counterparts in the target language, reducing the proof of its correctness to the proof of commutativity for the following diagram:

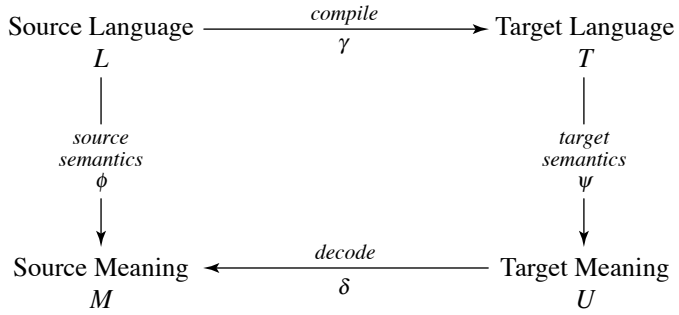


FIGURE 1-1 Schematic View of Compiler Correctness

Some authors suggest that the *decode* arrow should be reversed and labelled *encode* to reflect the fact that, ultimately, it is a semantics of the target language that govern the

actual behaviour of the final executable programs produced by the compiler [Orejas 81]. The choice of that arrow's direction is a subject of a somewhat vacuous philosophical argument and, in this work, I chose Morris's original version of the diagram merely for its visual similarity to my own design described later in Chapter 2.

This view of compiler correctness, sometimes called the *Morris Diagram*, prevails throughout most of the subsequent research in the area of compiler verification. It is, to a large extent, based on the algebraic treatment of programs and their proofs by Burstall and Landin [Burstall 69]. Morris argues that the corners of his diagram should be defined as heterogeneous universal algebras connected through homomorphisms, effectively introducing the compiler verification community to Scott's denotational semantics [Scott 71, Scott 82]. A number of follow-up articles have been published to Morris's paper, including works by Thatcher [Thatcher 80], Broy *et al.* [Broy 80], Cleaveland [Cleaveland 80] and the already-mentioned critique by Orejas [Orejas 81], all advocating similar approaches to compiler verification. In particular, the later three works argue for representation of source programs as data structures within the compiler, which is further refined by Broy *et al.* [Broy 87] to formulate denotational semantics of a programming language as a translation into a metalanguage with an established semantic model.

The approach of Morris *et al.* exemplifies a very direct extensional treatment of the problem, in which a compiler implementation and a semantics of the source and target languages are constructed independently of each other. Armed with these models, the commutativity proof is then established as a posterior fact about the system's correctness, often making little or no attempt to mould the compiler's design to the needs of formal reasoning. In this work, I call such program specifications *extensional*, in order to contrast them with my own constructive or *intentional* approach to compiler verification, under which the judgement of correctness is *derived* from the actual or expected behaviour of the system. The term is in no way intended to be derogatory, since extensional specifications tend to possess a kind of mathematical elegance that no intentional description could ever hope to achieve. Nevertheless, in Chapter 2, I demonstrate that constructive program specifications resolve certain philosophical problems left unanswered by extensional techniques and, accordingly, that intentional definitions should be preferred in the specific case of compiler construction.

Although, ultimately, no complete compiler for a high level programming language in common use has been ever verified purely through an application of extensional techniques, the approach of Morris has had some successes. Among the most notable achievements are Polak's work on verification of a compiler for a subset of Pascal [Polak 81], Despeyroux's publication of a correctness proof for a translation of an ML subset [Despeyroux 86] and Chirica and Martin's systematic treatment of correctness for language parsers [Chirica 86]. In 1990, Simpson simplified the approach of Despeyroux, developing a specification in natural semantics of a compiler from a subset of ML to the SECD machine [Simpson 90]. The method is also stud-

ied in detail by da Silva [da Silva 92] who analyses the general issues involved in proving homomorphisms of translations between source and target languages. More recently, Börger, Đurđanović and Rosenzweig published a series of papers [Börger 94, Börger 95a, Börger 95b, Börger 96] in which they describe strict mathematical models for compilation of Occam and Prolog to the Transputer architecture. Although remarkably successful, their work was hugely simplified by the natural characteristics of the chosen source and target languages and was never applied to a more pragmatic system.

It was realised early that the complexity of compiler correctness proofs would render the problem unmanageable without at least some degree of automation. The first attempt at an application of automated proof checking to the question of compiler correctness is due to Milner and Weyhauch [Milner 72b] and was carried out as part of some of the earliest work with the LCF theorem prover [Milner 72a, Milner 73]. The LCF approach is also scrutinised by Cohn [Cohn 78]. In 1989, Moore presented a detailed machine-verified system for compiling a high level assembly language Piton on the FM8502 microprocessor [Moore 89]. Moore's approach is grounded in a direct commutativity proof of the Morris Diagram using a theorem prover developed specifically for the purpose. A similar strategy was followed independently by Young, who presents a fully-verified compiler for a subset of a Pascal-like language Gypsy in Gypsy's own framework for combined program development and verification [Young 89]. In 1991, Curzon described an even more complete treatment of a mechanically-verified translation from another structured assembly language Vista to the instruction set of the formally-defined Viper microprocessor [Curzon 91] using a semantic model of Viper developed by Cohn [Cohn 88, Cohn 89] and the HOL proof assistant designed by Gordon [Gordon 88]. In 1992, Hannan and Pfenning applied Morris's approach directly in the LF logical framework [Harper 87, Twelf 98], presenting a machine-checked correctness proof for an idealised compiler of a small functional language [Hannan 93]. Most recently, 2004 saw Blech's portrayal of a machine-verifiable code generator from the static single assignment form [Blech 04, Cytron 91] formulated in the Isabelle/HOL system [Paulson 90, Paulson 05]. Tool support for mechanical verification of compiler correctness and scalability of the associated techniques is treated exhaustively in Stringer-Calvert's 1998 PhD thesis [Stringer-Calvert 98].

The advent of automated proof checking techniques increased pressures on the question of *model fidelity*, or a degree of accuracy with which a model describes the behaviour of an actual implementation. It is generally accepted that even a most stringently-verified proof has little practical value if no equally-strict correspondence can been established between the formal model used in its formulation and that model's rendition within an actual compiler implementation. A most immediate solution to this problem can be obtained by deriving an executable representation of the language translator mechanically from its abstract specification, a technique that was first explored at the IBM Research Laboratories in Vienna during 1960s and 1970s as exemplified by the Vienna Development Method and the associated Vienna Definition Language [Weg-

ner 72]. The primary goal of the Vienna Project was a complete formal specification of the PL/1 programming language and an implementation of a corresponding verified compiler [Bekic 74]. After over 30 years of research, the project was ultimately pronounced a failure, although IBM continues work on verification of a subset of PL/1 to this day. During an unpublished talk given at the 2003 ETAPS SE-WMT workshop, Cliff Jones quoted the sheer size of PL/1's semantic state as the primary reason for the project's failure [Jones 03]; Jones's talk provided a central motivation for my own research in the area, ultimately leading to the present work.

Notwithstanding the numerous obstacles encountered by the Vienna Project, compilation by program refinement remains a busy area of research. The most ambitious effort has been undertaken as part of the EU-funded ProCoS project, which delivered a number of remarkable results [Langmaack 89, Sampaio 90, Langmaack 90] although ultimately fell short of producing a fully-verified optimising compiler for a practical high level language. More recently, Okuma and Minamide [Okuma 03] and independently Berghofer and Strecker [Berghofer 03] presented complete derivations of an executable compiler from its abstract specification using the Isabelle proof assistant [Paulson 90, Paulson 05]. The later work handles a simplified version of the popular Java language, thus making an important step towards a practical application of the technique. The work on program refinement is summarised by Denney in his 1998 PhD thesis [Denney 98].

Independently of the primary compiler verification effort, a lot of attention has been given to the issue of a formal specification of the behaviour of programs itself. While this problem has far-reaching implications throughout theoretical computer science, it is particularly pressing in the context of compiler verification, which rests fundamentally on the question of semantic equivalence between two or more syntactically-distinct renditions of the same computation. The earliest attempt at assigning meanings to programs is due to Floyd, who proposed the now ubiquitous flowchart notation [Floyd 67]. An axiomatic approach to the problem, in which the system's behaviour is specified as a set of pre- and post-conditions, was formulated by Hoare [Hoare 69] and refined by Dijkstra [Dijkstra 76]. Hoare's approach was subsequently extended into a fully-featured software development environment in the form of the already-mentioned Vienna Definition Language (VDL). Typical VDL specifications of a software system describe the dynamic behaviours of programs in terms of an appropriate finite state machine, an approach that is now known under the term of *transitional semantics* [Clemmensen 84]. An alternative algebraic treatment of the problem was proposed by Burstall and Landin [Burstall 69] and refined into the notion of *denotational semantics* by Scott [Scott 71, Scott 82]. Scott depicts the meanings of programs as a set of mappings between individual language constructs and abstract mathematical objects with well-defined algebraic properties. This technique was introduced into the context of compiler verification by the original paper by Morris [Morris 73], who explicitly rejected earlier transitional treatments of language semantics.

Denotational semantics remained the predominant focus of research on the meanings of computational structures throughout the 1970s. The trend was reversed in the 1980s, which saw a revival of more direct models of program execution under a common brand of *operational semantics*. Compared with denotational treatments, operational semantics, which strive to describe the actual evaluation of a given syntactic entity, offer an immediate and highly intuitive correspondence between the formal meaning of a program and its actual behaviour on some computational system. Unfortunately, it turns out that the sparsity of clean algebraic properties in a typical operational model of a programming language often hinders reasoning about concrete properties of individual programs, although a neat solution to this problem has been proposed by Plotkin in the form of *structural* or *small-step operational semantics* [Plotkin 81]. By masquerading the evaluation function as a set of logical judgements about the structure of the underlying computation, this style of semantics alleviates most of the issues faced by earlier operational treatments and, in his 1996 PhD thesis, Diehl describes successfully the use of structural operational models in provably-correct code generation [Diehl 96]. A compromise between the denotational and structural treatments has also been proposed by Despeyroux in the form of *big-step* or *natural semantics* [Despeyroux 86] and was accompanied by a correctness proof for a translation of a small ML subset in the same work. However, the denotational approach is still pursued actively for its mathematical merits. In 1992, Palsberg published a study into the use of a denotational semantics in the context of compilation of imperative languages on the SPARC and HP-PA instruction set architectures [Palsberg 92]. It is also central to my own work on compiler verification.

All of the above formulations of program semantics share to a varying degree the problem of a lack of modularity. Since the whole language is formalised in a system of mutually dependant rules, small changes to its specification often require a complete rewrite of the entire semantic model. This issue was first addressed successfully by Moggi, who proposed a categorical approach to denotational semantics based on the concept of a *monad* [Moggi 89]. Moggi's work has been popularised in the functional language community by Wadler [Wadler 92] and remains the predominant solution to the problem regardless of the precise style employed in the course of language formalisation.

The number of different strategies developed for the task of defining a programming language leads naturally to yet another challenge in the area, namely that of establishing equivalence of computations whose meanings have been expressed using two or more distinct semantic styles. The issue is, to a large extent, innate in the nature of compiler verification, especially when a denotational model is applied in the course of language specification and an operational semantics is deployed to give the system a degree of computational relevance. Even the most rigorous proponents of the denotational paradigm agree that a more concrete axiomatic or operational definition is generally required to ground the end-product in a computational reality. Conversely,

even the most devoted advocates of operational semantics are usually bound to apply some sort of an abstract denotational, axiomatic or algebraic model somewhere in the course of compiler verification. In both cases, it is critical that an equivalence of the two formulations is established formally. The issue is usually expressed as the problem of *full abstraction*, which was first studied by Milner and Plotkin in the context of PCF [Milner 77, Plotkin 77] and is treated exhaustively by Mulmuley in his 1985 PhD thesis [Mulmuley 85]. A completely-satisfactory solution to the problem eluded the research community for almost thirty years and was one of the primary motivations for a departure from denotational models in the 1980s. In the 1990s, rediscovery of Lorenzen's *game semantics* by Abramsky, Jagadeesan and Melliès lead eventually to a complete solution of the full abstraction problem [Lorenzen 61, Abramsky 94, Abramsky 99] and, as a result of that work, denotational treatments are now enjoying a wave of renewed popularity. Recent advancements in the area of definability and full abstraction are summarised concisely by Curien [Curien 07].

The task of specifying a formal semantics for a fully-featured programming language is itself surprisingly difficult. Most of the publications mentioned above deal only with idealised languages or small subsets of existing languages. With the notable exceptions of Standard ML [Milner 90, Milner 91] and Scheme [IEEE 91, Kelsey 98], no other practical software development system has ever been supplied with a precise semantic model as part of its official definition. However, as a result of an extensive research in the area, many other languages have been retrofitted with a formal specification, including Pascal [Hoare 73, Bjørner 82b], Ada [Pedersen 80], ALGOL 60 [Bjørner 82a], Smalltalk [Wolczko 87, Blakley 92], Modula-2 [Gurevich 88, Morris 88], Occam [Gurevich 90, Börger 94, Börger 96], Eiffel [Attali 93], C++ [Wallace 93, Wallace 95], Cobol [Vale 93], Prolog [Börger 95a], Oberon [Kutter 96, Kutter 97], C [Gurevich 93, Cook 94b, Cook 94a, Black 96, Norrish 97, Papaspyrou 98] and Java [Wallace 97]. To the best of my knowledge, none of that work has been applied successfully in actual implementation of a complete language translator.

An entirely different approach to compiler verification known as *program checking* was pioneered by Blum and Kannan [Blum 89, Blum 95, Wasserman 97]. Rather than verifying an implementation of a translation system directly, program checking treats the compiler as a code-generating "black box", striving instead to establish correctness of it final product through a machine-guided verification of the actual executable instructions produced by the compiler with respect to the supplied specification of the user's program. The approach is actively pursued as part of the EU-funded Verifix project, showing some encouraging results [Goerigk 96, Zimmermann 97, Dold 98, Gaul 99, Goos 99, Goos 00]. Most recently, Glesner applied program checking to the problem of verified code generation for embedded systems [Glesner 02, Glesner 03]. The technique exhibits some promising pragmatic properties, although it remains a subject of a somewhat delicate philosophical debate. In particular, since program checking validates a specific instance of a compiler's output rather than the actual translation

process, the resulting correctness proofs cannot be generalised to arbitrary compilation tasks and, accordingly, cannot serve as a witness to a correctness of the underlying development environment itself.

An interesting and powerful mixture of program checking with the classic strategy of Morris has been also studied by Blazy, Dargaye and Leroy [Blazy 06]. Under this approach, a compiler is structured into a number of phases whose individual verification is either performed directly using standard extensional techniques, or else postponed until an ensuing phase of compilation, in which case the phase's output is annotated automatically with an appropriate witness to its conformance to a specification that is also produced mechanically by an earlier phase of compilation using the *proof carrying code* technique invented by Necula [Necula 97]. A consistency of these annotations is then verified systematically by a later stage of translation, ensuring that the eventual executable program derived by the compiler remains correct irrespective of any unverified phases incorporated into its implementation. This approach proves particularly attractive for handling many popular optimisation algorithms, which often include various corner cases that are seldom encountered during real-life applications of the compiler, but still pose a serious hurdle to a formulation of its correctness proof. Blazy, Dargaye and Leroy successfully apply their mixed verification strategy to a lightly optimising compiler for a subset of C that generates PowerPC assembly code and is implemented entirely in the Coq proof assistant [Blazy 06, INRIA 02].

Last but not least, no work on compiler verification would be complete without a mention of the approach pioneered by Morrisett and Tarditi [Morrisett 95, Tarditi 96, Morrisett 99, Crary 03]. Forfeiting ambitions of total correctness, this pragmatic technique strives to obtain a very reliable compiler by utilising a strongly typed intermediate program representation to render many common implementation errors inexpressible within the system. Originally deployed in the context of ML [Tarditi 96], *type-preserving compilation* has since been applied in construction of the highly optimising GHC compiler for Haskell [Peyton Jones 96, Peyton Jones 98]. Although the technique stops short of establishing total correctness, it promises to play an important rôle in an ultimate solution to the compiler verification problem, particularly as the guarantees provided by the intermediate representation's type system are also likely to simplify formulation of proper correctness proofs. However, given that a successful treatment of a weakly typed language such as C would require a sophisticated dependant type system with at least partially automated type inference algorithm that is difficult to foresee at the present time, I explicitly restrict this work to an untyped program representation, hoping that it will provide sufficient motivation for future research in the area. In particular, I envisage that many semantic constraints on the concrete syntax of languages such as C could be formalised directly through their translation into a calculus with a suitably expressive dependant type system, effectively expanding the methodology described in this work from compiler verification to the compiler-based verification of programs proposed by Hoare [Hoare 03].

1.2 Contributions

The central contribution of this work is a novel approach to software verification described in Chapter 2. It exposes a number of philosophical limitations inherent in the traditional extensional treatments of program correctness and proposes to resolve them through a tighter epistemic integration of a system's formal specification with its concrete implementation as an executable computer program. In the particular case at hand, the solution proposed in Chapter 2 describes a new, category-theoretic definition of a *programming language*, *compiler* and its *correctness judgement*, together with an innovative criteria of *linear correctness* which, as I prove in that chapter, constitutes a simple yet sufficient condition that alone guarantees a total correctness of the entire translation system with respect to a semantic model of the source and target languages embedded within the compiler's implementation itself. To the best of my knowledge, neither the philosophical limitations nor the linear correctness solution to the problem of a verified compiler design have been investigated prior to the publication of this work.

Besides its philosophical advantages, the main practical benefit of my approach is the relatively small number of proof obligations required for a satisfactory formal verification of the entire system. In order to demonstrate effectiveness of linear correctness, I apply the technique directly to the task of specifying and implementing a linearly-verified compiler for the C programming language as defined informally in the ANSI/ISO/IEC Standard 9899:1990 [ANSI 89]. A detailed description of that compiler occupies Chapters 4, 5 and 6 of this work and constitutes its secondary contribution, namely a new rigorous specification of that popular programming language, available as an alternative to prior research on the topic [Gurevich 93, Cook 94b, Cook 94a, Black 96, Norrish 97, Papaspyrou 98]. Unlike all existing semantic models of C, the one proposed in this work provides a clear and direct mechanism for the model's ratification within a practical compiler implementation, as demonstrated by the fact that the set of formal definitions depicted within Chapters 4 and 5 serves simultaneously as a partial implementation of such a compiler, which, in Chapter 6 is completed to provide a comprehensive translation of C programs to the executable instructions of Knuth's MMIX architecture [Knuth 05]. Due to space limitations, certain engineering aspects of the system which have already received extensive treatment in literature (namely lexical analysis, preprocessing, parsing and optimisations) are omitted from this work, since their satisfactory treatment would provide an undue distraction from the more novel aspects of my approach. Notwithstanding these limitations, however, I believe my project to represent the most complete attempt at a verified translation of C programs to date.

A final minor contribution of this work is contained in Chapter 3, whereby I demonstrate how the Haskell programming language could be extended and deployed as a unified framework for an implementation and formal specification of programs. Although this approach borrows heavily from the existing proof development environments such as Coq, Twelf and Ott [INRIA 02, Twelf 98, Sewell 07], it differs from all of these tools

in that it focuses development of provably-correct programs around their implementation rather than the specification effort, by which I hope to make theorem proving more accessible to the general programming community.

1.3 Rules of the Game

Given the enormity of the task at hand, it is advisable to commence the project with a clear statement of its goals. In particular, in this work I propose to present a formal specification of the static and dynamic semantics attributed to the parse trees of translation units derived in accordance with the BNF grammar published by the ANSI/ISO/IEC Standard 9899:1990 [ANSI 89], together with a minimal complete translation of such entities into their relocatable binary representations executable under the MMIX instruction set architecture [Knuth 05] and a detailed informal proof of that translation's correctness with respect to the included semantic model of the language. The means by which such parse trees are to be obtained from a concrete textual representation of the program is explicitly excluded, as are any issues of their operational efficiency, formalisation of the underlying system environment, the address space layout, module linking algorithm and a machine-verifiable rendition of the presented proofs.

The first and most conspicuous of these exclusions, namely my decision to omit the laborious but generally routine tasks of tokenising, preprocessing and parsing of a textual program representation, was made purely in order to reduce the length of this already-voluminous manuscript, since their precise formalisation would require a significant expenditure of effort, yet consist predominantly of techniques that are already well understood and studied extensively in existing literature. Instead, the presented implementation of a C compiler accepts, as its input, an algebraic data structure isomorphic to the program's parse tree obtained in accordance with the precise syntactic rules of the BNF grammar prescribed by the C Standard. In Section 5.11, I analyse an extended example of the compiler's behaviour using a parse tree that was obtained by a painstaking manual process before feeding the resulting structure to the actual Haskell implementation of the translator. Similarly, my compiler also abstracts from the precise binary representation of the constructed executable programs by rendering its output as an algebraic data structure, roughly equivalent to a parse tree of a symbolic relocatable object file produced by a more conventional compiler implementation. The actual algorithm by which such object files are to be combined or linked into a monolithic executable program image is left unspecified, since it is traditionally considered to constitute a part of the underlying operating system implementation which is not the subject of my project. In the same vein, I make no attempt to formalise any of the standard C libraries defined in the ANSI specification of the language, although all of its elements, including the cumbersome "va_list", "setjmp" and "longjmp" constructs, can be implemented easily within the proposed framework.

In all other respects, the Haskell specification of C presented in this work constitutes a complete compiler for the language, although the algorithm defined in Chapter 6

represents a fairly naïve code generator that, in the interest of exposition, makes absolutely no attempt to ensure that the constructed instruction stream represents an operationally-efficient rendition of the underlying computation. Having said that, the design naturally admits incorporation of almost arbitrary optimisation algorithms, so that, in general, future improvements to the compiler should not require reformulation of the underlying formal model. In similar vein, my specification of the translation process admits, without further reification, realisation of meticulous diagnostic messages within the system, although, in its current form, only the minimal requirements of the C Standard have been incorporated into the implementation.

On the other hand, to ensure accuracy of the presented specification of C, I pay particular attention to the careful distinction between well-defined, undefined, unspecified and implementation-defined aspects of the language in its proposed semantic model. In order to achieve this stratification, however, it was necessary to compromise reusability of the code generation algorithm from Chapter 6 between implementations of the compiler on different target architectures. Specifically, despite its focus on portability, the conceptual framework described in this work assumes a significant amount of knowledge about the underlying hardware protocols, rendering it unsuitable for realisation of a universal program representation in the tradition of UNCOL [Conway 58]. Chapter 7 contains a more detailed discussion of this limitation and an outline of a possible resolution to the problem.

Following the established mathematical tradition, in this work I assume the view that the specific task of program verification is, in principle, distinct from a further validation of the underlying proofs in an automated logical framework such as LCF, PVS, Isabelle/HOL, ACL2, Twelf or Coq [Milner 73, Owre 92, Paulson 05, Kaufmann 00, Twelf 98, INRIA 02]. Accordingly, while all formal properties of the presented translation are expressed carefully as symbolic theorems within the logical framework described in Chapter 3, the actual proofs of these theorems are generally described in a detailed, but ultimately informal regime of a natural language akin to the traditional "pen and paper" ratiocination common in mathematics. Repetitive reasoning is often summarised in an example of its most important or difficult cases and omitting various routine aspects of the underlying justification process whenever such details can be reasonably assumed to be derivable by a sufficiently sophisticated automated theorem proving tool. Although I fully recognise the need for a mechanical validation of proofs and, in Chapter 3, expend some effort on showing how, in principle, it could be achieved within my logical framework, given the already-overwhelming scope of the project, I consider it to be of a secondary importance. I leave all further details of such validation for a future time, trusting that the framework's grounding on the well-understood Martin-Löf's constructive type theory provides sufficient soundness guarantees for the present purpose and a solid foundation for a future work in this area.

Finally, it should be emphasised that, in this work, I concern myself solely with verification of the translation process itself and remain oblivious to any particular set

of behaviours expected of the constructed executable programs by their authors. Nevertheless, as suggested in Chapter 7, it could be speculated that the proposed language implementation framework can be naturally extended to promote creation of *verifying compilers* that accommodate such reasoning about various dynamic properties of their output if the underlying intermediate program representation was to be supplemented with a sufficiently expressive dependant type system.

1.4 Roadmap

In this work, I assume that the reader is already familiar with the C programming language and, further, possesses a working knowledge of Haskell or a similar non-strict functional software development environment. An elementary understanding of the basic principles of theorem proving in a constructive type theory is advantageous but not necessary. Finally, the definitions and proofs in Section 2.3 rely on some basic ideas from category theory, although readers without a background in this branch of mathematics may safely skip that section without impacting on their understanding of the remaining material.

The entire thesis is structured into seven chapters, together with four appendixes that contain various auxiliary information intended to be cross-referenced frequently whenever need arises for clarification of a given topic. In particular, Appendix A outlines the list of all Haskell entities that are utilised in this work without a formal definition, most of which are taken directly from the standard libraries of the language. Appendix B contains a detailed bibliography of the prior publications relevant to the discussion, while Appendix C captures the implementation-defined behaviour of C programs on the specific instruction set architecture targeted by their translation in Chapter 6. Finally, Appendix D encompasses an exhaustive index of all Haskell types, data constructors, functions and theorems defined within the included compiler implementation.

The actual core of my thesis is organised in such a way as to ensure that every section relies only on ideas presented earlier in the discussion, so that the following six chapters should generally be perused in succession, although the reader is free to vary the degree of attention paid to the three portions of the compiler's implementation described in Chapters 4, 5 and 6.

First of all, Chapter 2 examines the epistemic roots of program verification and its principal notions of knowledge, correctness and proof, exposing the core issues with the prior approaches to compiler design that are addressed in my work. By reviewing the standard treatment of compiler validity as a commutativity of the translation system [Morris 73], I gradually develop the principle of *linear correctness* on which my project is fundamentally based. The final two sections of Chapter 2 analyse the critical formal properties of this approach with its underlying *linear correctness criteria*, prove that criteria's sufficiency with respect to a category-theoretic definition of compiler correctness and, finally, outline the practical requirements imposed by the technique on the design and implementation of a concrete language translation system.

Before applying the principle of linear correctness in an actual compiler implementation, I devote Chapter 3 to a detailed overview of the notation and formatting rules exerted on all Haskell definitions included in this work. These rules are of most interest to readers already accustomed to Haskell programming, since, in the interest of exposition, I heavily mould the actual source code of my implementation into a presentation that is much closer to a traditional notation of mathematical logic than to an ordinary syntax of the language. In the second half of Chapter 3, I describe a set of semantic extensions to standard Haskell which enable its use as a representation of rigorous program specifications and proofs. Although these extensions are fundamentally based on a constructive type theory similar to that of the Coq proof assistant [INRIA 02], they differ substantially from all past attempts at incorporating a dependant type system into the language and, accordingly, all readers should ensure that they familiarise themselves with the proposed logical framework before proceeding to any of the subsequent material in this thesis.

The next two and a half chapters of this work are devoted entirely to a meticulous specification of the C programming language and an intermediate representation of C programs within my compiler. It should be observed that, until this task is completed, very few statements can be made about formal properties of the presented translation system, so that Chapters 4 and 5 are, by far and large, preoccupied solely with the definitional aspects of the project. Readers interested in the actual compiler correctness proofs are advised to seek them later in Chapter 6, once the entire formal model and implementation of the system has been presented. This conspicuous absence of formal justifications from the bulk of my work should not be taken as a flaw, since it illustrates one of the most important features of my verification methodology, which explicitly proposes to shift emphasis from proofs to specification in order to focus the development effort on those portions of the entire system which are of a most immediate consequence to its ultimate users.

Since my approach to compiler design and verification rests crucially on finding an intermediate program representation that meets the requirements of both the language specification and the translation process itself, in Chapter 4, I commence the actual compiler implementation effort with an introduction of the precise variant of lambda calculus employed in the remainder of this work. After a brief review of the central ideas behind the pure lambda calculus of Church [Church 40], I define the complete syntax and an architecturally neutral portion of an algebraic semantics for a purely functional monadic language *Etude* that is suitable for use as the intermediate program representation within a linearly correct C compiler. Without committing to a particular set of computational facilities supported by a given combination of the instruction set and hardware designs, Etude provides for a description of programs with a sufficient degree of detail as to capture precisely the portable fragment of the C programming language. A thorough familiarity with the syntax and algebraic semantics of Etude is crucial for comprehension of almost all of the material included in the subsequent two

chapters, since the language constitutes the precise formalism in which the meanings of both C and assembly programs are described within the presented system. However, the actual operational semantics of Etude and a detailed proof of their conformance to the generic algebraic specification are only discussed later in Chapter 6, after all operational details of the language have been assigned a resolution appropriate for the chosen target architecture.

The core translation of C programs into Etude is the topic of Chapter 5. In order to preserve the distinction between portable and implementation-defined aspects of the language, all specifications pertaining to the later topic are defined separately in Appendix C. As already mentioned, Chapter 5 and Appendix C define the precise semantics of a single translation unit of a C program through its mapping to an appropriate Etude representation of the same computation and include all diagnostics of inadmissible syntactic entities mandated by the C Standard.

Finally, in Chapter 6, I present both the final translation stage of the resulting Etude programs into an actual sequence of executable machine instructions and a formal proof of the linear correctness principle for the compiler, which, as shown earlier in Chapter 2, is sufficient to establish validity of the entire translation system. The chapter begins with a brief outline of Knuth's MMIX architecture [Knuth 05], chosen as the target of my implementation for its elegance and simplicity of form that goes a long way to enhance and consolidate presentation of the following material. In Section 6.2, the meanings of all MMIX instructions are defined by their mapping onto analogous Etude constructs. Next, Section 6.3 describes the precise incarnation of Etude on that architecture, complete with its detailed semantic model and a proof of that model's conformance to the earlier algebraic specification of the same language in Chapter 4. Finally, Section 6.4 contains both the actual translation of Etude functions to MMIX instruction sequences and a proof of the linear correctness principle for the entire system, formulated as reversibility of that mapping with respect to the translation semantics of MMIX programs defined in Section 6.2. With that, the goal of developing a provably correct compiler for the standard C language is finally attained.

Last but not least, Chapter 7 contains some concluding remarks, discussing a number of benefits and pitfalls inherent in the linear approach to compiler correctness as applied in my compiler. I also outline future work envisaged in the area and its possible impact on the more conventional approaches to the problem of a verified program translation.

2

EASIER BY DESIGN

There are two ways of constructing software.
One way is to make it so simple that there are
obviously no deficiencies. The other is to make it so
complicated that there are no obvious deficiencies.
— C.A.R. Hoare

Before we can attempt to verify a compiler, it is useful to consider precisely what is meant by the term "verification." Fundamentally, verification of a program intends to demonstrate, establish or otherwise expose its *correctness*. It does not purpose to make a program correct (since, by definition, no incorrect program can be verified!) but rather to convince ourselves of its suitability for a particular job. That is, once we have verified a program, we *know* that it behaves as expected. In short, software verification is an epistemic task. It would be entirely bogus for us to ignore this philosophical aspect of the problem, since, unlike other areas of logic and applied mathematics, it proposes to bridge the gulf between the realms of physical entities and abstract thought in a way that no other application of logic has ever attempted before. The subject of verification is an actual tactile entity, the physical behaviour of some computer hardware, while the tools available for gaining knowledge about that behaviour are fundamentally abstract in nature. When building a bridge, an engineer performs some experiments to establish the mathematical principles underlying behaviours of the physical objects he or she is dealing with, such as the properties of steel and the laws of gravity. He or she uses these principles to devise an abstract *model* of the bridge being constructed in a form of a sequence of mathematical symbols and, from then on, the remainder of the verification process is performed entirely within that mathematical framework. In contrast, computer software *is* a sequence of symbols and it is the symbols themselves that manifest concrete physical behaviour through the magic of digital hardware. Somehow, in the task of program verification, we must connect that physical behaviour with the rules of logic governing the symbols from which the program has been constructed.

Since the times of Aristotle, epistemologists have defined *knowledge* as a *justified true belief*. The *truth* aspect of knowledge can be hardly debated, so it is the problems of justification and belief that must be resolved in the course of program verification. This is also where the gulf between the physical and the abstract emerges in the context of computer science. To the users of a program, the statement of belief is simple: they want to know that their system is *correct*, in that the response they anticipate from the machine matches its actual behaviour exhibited during the program's execution. Justification materialises as some *evidence* that supports this belief. However, to a logician, belief takes a rather different form, namely that of a *statement* which models the

system's behaviour, a logical *judgement* in Martin-Löf's terminology, whose justification is represented by a *proof* showing the judgement to hold [Martin-Löf 96]. There are, therefore, two kinds of knowledge involved in program verification: on one hand, knowledge of correctness (i.e., knowledge that the expected and actual behaviours of the program are in some sense equivalent) and, on the other, that the judgement of this behaviour is provable. These two notions are fundamentally different and must be reconciled somewhere in the course of verification.

Perhaps the best way to explain this difference is to paraphrase the standard philosophical puzzle originally proposed by Gettier [Gettier 63]. Suppose you are driving through a beautiful Scottish countryside. You see a sheep in the field and exclaim "Oh, now I *know* that there are sheep in Scotland!". However, imagine that, unknown to you, two further facts hold: (1) the animal you saw was actually a dog dressed up as a sheep by a cunning farmer intent to fool your senses and (2), a flock of sheep is grazing happily on a neighbouring paddock behind a hill, ignorant of your philosophical torment. Do you really *know* that there are sheep in Scotland? Your belief in the existence of Scottish sheep is certainly true. It is also justified by strong sensory evidence (you saw one!) and also proven by the unseen sheep behind the hill. But, somehow, your belief misses a crucial causal connection with that proof and, indeed, most people would agree that you do not possess knowledge of the fact in question.

This problem has revitalised epistemology of the twentieth century, but it is, by no means, restricted to sensory evidence. In 1637, Pierre de Fermat wrote his now-famous note on the margin of *Arithmetica*, claiming to have discovered a "truly marvellous proof" of what is now known as "Fermat's Last Theorem". Today, we are almost certain that Fermat's proof was incorrect, although, eventually, a correct one has been discovered by Taylor and Wiles in 1994 [Aczel 96]. Accordingly, Fermat could not have *known* his theorem, despite holding a firm belief in it, which was both true, justified by the knowledge available to him and proven 358 years later.

Suppose that your customer, one Mr. Smith, has only ever had an experience with a perverted computer in which all available memory is utilised for the buffer of a sound hardware. One day Mr. Smith goes to collage and learns about the C programming language, firmly believing that all assignment operations make an audible noise. His evidence for this (false) belief is Smith's past experience with computers. Although the evidence is unsound, it is precisely of the kind on which typical unskilled users of a computer program rely when forming their often exasperating expectations and understanding of a computer system. One day, Mr. Smith comes to you, a consultant, charging you with the task of formally verifying the following program "Ding":

```
int main () {
        volatile int *x = (int *)0xC00157AF;
        *x = 440;
}
```

Smith asks you to prove the proposition that "Ding makes noise", which he believes

to be true by the virtue of his mistaken understanding of the C assignment operation, regardless of the specific behaviour of Ding itself. Nevertheless, this statement is in fact true since, by sheer coincidence, the memory location $\mathrm{C00157AF}_{16}$ happens to represent an appropriate register of the system's sound card. Indeed, after months of sweat-raising toil, you develop a rigorous Coq model of the computer system, its sound hardware and arrive at a machine-verifiable proof that Ding does, in fact, play a sound of precisely 440 Hz. But, given that Mr. Smith is not trained to read Coq models, does he really *know* the statement to be true? Somehow, his justification misses an essential connection with the reasons for the statement's validity. Every educator of computer science is painfully aware of the grief that such misconceptions about the nature of computers can bring into their lives.

Accordingly, when verifying a computer program, we are faced with three separate tasks: stating the judgement of correctness (known as *program specification*), stating the evidence (known as *proof*) and, finally, establishing a correspondence between the proof of a judgement and the knowledge of correctness. It is only after all three of these objectives have been accomplished that we can claim real knowledge of the correctness of the program under scrutiny.

Epistemologically, we can assume one of two general approaches to justification, known as *externalism* and *internalism*, respectively. Under externalist approaches, justification is obtained through a reference to some external, usually empirical evidence which, in the context of computer science, implies measurements, testing and, in general, some form of an observation of the program's behaviour. Since we are seeking absolute certainty rather than degrees of confidence, it is doubtful whether observation alone will ever deliver a fully satisfactory justification of the belief in the correctness of a program. On the other hand, internalism seeks to formulate precisely that justification which constitutes the very reason for correctness. This is exactly the notion of proof as understood in constructive mathematics and, as such, it is that approach that I single out for investigation in the present work.

But how can we reconcile internalism with the user's informal expectations of the program's behaviour? Epistemology gives three competing answers to this question: *contextualism*, which essentially states that some doubts are not real doubts (for example, doubts about the meaning of the C assignment operator should not even be raised), *foundationalism*, which claims that beliefs such as the notion of assignment don't need justification and, finally, *coherentism*, which simply requires the complete system inclusive of both the belief and its justification to be *coherent*, in the loose understanding of the term as "collectively making sense." Both contextualism and foundationalism are somewhat unsatisfactory in the context of program verification, since contextualism requires an unlikely consensus on the scope of intuition and even the unchallenged concepts of foundationalism must be defined formally and connected to reality. It is therefore the general path of coherentism that I pursue from now on.

In a way, coherentism merges the notions of belief and judgement, since both must

form a part of a single coherent system of knowledge. Let us assume an existence of a *cognitive framework* in which all of specification, implementation and verification of the program are performed. In this context, the notion of a *framework* is somewhat broader than the standard application of the term in mathematical logic: it goes beyond the mere concept of a *notation* used to express judgements and proofs, to also include some degree of a mathematical intuition about the meaning and purpose behind these judgements. For example, under the typical approach to compiler design, different phases of a compiler verification process are described in different notations such as BNF grammars in parsing, some programming language in implementation and some expression of a constructive logic in a formulation of its correctness proofs. Nevertheless, all of these notations can happily coexist in the mind of the reader and, in that, they can be viewed as a single coherent system of knowledge.

In order to verify a computer program in a coherentist way, one must not only devise a formal specification of its properties, prove these properties to hold and communicate both the specification and the proofs to the user, but also somehow ensure that, in the user's mind, the specification remains coherent with his or her expectations regarding the program's behaviour. This is by far the most difficult task in formal verification, in light of which the challenges of proofs pale in comparison. For example, if a user believes a C assignment to make an audible sound, then even a most stringent specification of the system will leave him or her unsatisfied with the actual behaviour, since his or her interpretation of symbols differs fundamentally from that of their author. To counter this, a coherentist approach to software verification must strive not only for a constructive proof, but also a constructive judgement of the correctness criteria itself. Such judgement must specify not only the core properties required for the correctness of a program, but also establish a causal connection between these properties and the program's actual behaviour. Somehow, it must inject into the consciousness of the user the very reasons for a validity of the program.

Perhaps the only viable way of achieving this is goal is to redefine the correctness criteria itself in terms of the system's operational behaviour instead of its core algebraic properties. In other words, rather than proclaiming the program to have properties P, Q and R or to produce output X, Y and Z, the correctness criteria should state that, if one requires A, B and C, then the program will do the job. This is no easy feat, but, as I hope to show in this work, it can be achieved in the context of a compiler design and ensure that the user's understanding of the program's behaviour is derived constructively from the correctness criteria and its proof. In a sense, I propose to treat the three tasks of specification, implementation and proof as inseparable, so that the compiler implementation is not only the program, but also its specification and the very reason for the system's validity. Only by fusing the three phases of the design can a usable verified compiler be delivered. I hope to show that the resulting single phase is not only more useful than the sum of three individual efforts, but also simpler in its execution. In effect, I purpose to make compiler verification *easier by design*.

2.1 Defining Correctness

Consider again the schematic view of compiler correctness proposed by Morris and depicted again in Figure 2-1 [Morris 73]. It describes a verified compiler as a composition of four structures that represent the source and target languages together with their corresponding "meanings". The structures are connected using four transformation functions, whose collective commutativity represents the correctness criteria of the entire system:

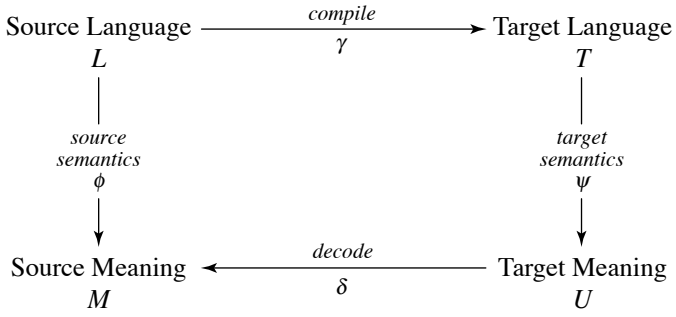


FIGURE 2-1 Morris Diagram

Observe that something important happened here: whereas an implementer of a compiler is chiefly concerned with the task of translating every input program into its target representation, Morris chose a static view of the system in which the notion of an individual program itself is not even mentioned. In fact, the program being compiled remains the same throughout the translation process and only its representation changes as the compiler's job progresses.

What sort of structures are, then, placed in the corners of a Morris diagram? Morris himself suggests them to be universal algebras, but, generally, they are simply some formalisms (logics, if you like) for manipulating symbols. In the top row we place two concrete programming languages such as C and the MMIX instruction set. These are *languages* in the formal sense of the word: each describes a set of words or sentences, defined, typically very precisely, using formalisms such as BNF grammars. Every element of such a set is simply a finite sequence of discrete symbols such as characters, lexical tokens or bit values. While, in the context of a compiler, most sentences are intended to somehow represent computable functions, this fact is, for now, incidental to the discussion. In particular, the two top-row algebras from Figure 2-1 do not assign any deeper meaning to the individual sentences (or programs) of their corresponding languages. Such meanings can be only obtained through the mappings ϕ and ψ, which provide a pairing of these programs with their respective semantic interpretations described by the algebras M and U from the bottom row of the diagram. Accordingly, under an alternative interpretation of the compiler structure proposed by Morris, it is the *columns* rather than the corners of his diagrams that form the algebras of the system, so

that each of the translation and verification components in a compiler design constitutes one single, rather than two individual homomorphisms.

The correctness criteria depicted in Figure 2-1 can be paraphrased into a definition of a valid compiler as one that, for every sentence s in the input language L, produces some sentence t in the target language T, such that there exist translations ϕ, ψ and δ for which $\phi(s)$ and $\psi(\delta(t))$ are equivalent. Observe that the equivalence relation is formulated outside of the described system and belongs to some larger meta-logical framework that is itself not scrutinised by the diagram. More so, the correctness criteria does not specify what t itself is supposed to look like. I call such definitions of correctness *extensional*, in order to contrast them with *intentional* or *constructive* definitions that are formulated in terms of the actual behaviour of the software being verified. In the case at hand, this means that a constructive judgement of a compiler's correctness should itself describe the relation between the input and output of the program, presumably by defining the actual operational process of translating an arbitrary source sentence s into a specific target sentence t. The properties of a system and its output should be *derived* from the judgement of its correctness instead of being merely scrutinised by that judgement. On the contrary, under a typical extensional approach, the relation between a program's input and output is a subject, rather than the object of correctness, which is itself formulated as an abstract property of the translation system instead of its operational behaviour or intended purpose. Necessarily, such extensional definitions resort to auxiliary concepts akin to the ϕ, ψ and δ transitions in the Morris diagram, whose only purpose is to provide means of formalising some set of desired system properties. In general, these auxiliary definitions are of little consequence to the actual user of the program, resulting in a notion of correctness that is chronically prone to epistemic paradoxes akin to the Gettier problem mentioned earlier in this chapter.

I will now describe three successive refinements to the above conventional view of compiler correctness, arriving, by the end of this chapter, at a constructive notion of *linear correctness*, which yields particularly well to the coherentist understanding of knowledge and its justification.

2.2 Accounting for Optimisations

Let us now consider the structure of a typical industrial-strength compiler. Rather than performing the translation of a source program into the target language directly as suggested by the Morris Diagram from Figure 2-1, most compilers first transform their input into an intermediate representation or, more often, a series of intermediate representations tailored to the needs of the individual optimising program transformation algorithms. By superimposing the structure of such a compiler onto the Morris Diagram, we obtain the schematic view of compiler correctness portrayed in Figure 2-2.

The two double arrows in that figure are intended to represent the multitude of intermediate representations and optimising transformations present in a typical compiler. As in the original approach of Morris, compiler correctness rests on commutativity of

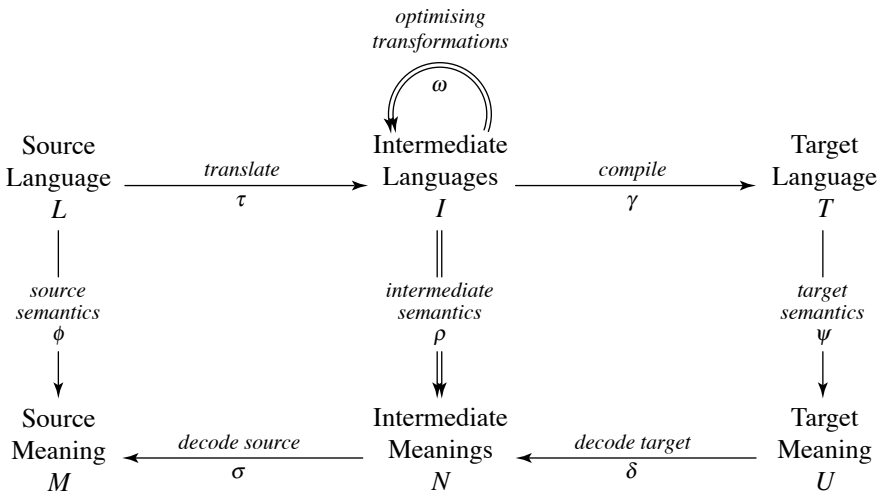


FIGURE 2-2 Schematic View of Correctness of an Optimising Compiler

the seven transitions in the diagram. Although the ρ transitions in the middle of the diagram are not, strictly speaking, necessary, they are included for modularity and the pleasant symmetry that they introduce into the view. In particular, one desirable property of this design is its clean separation of the compiler's structure into the *front-end*, concerned solely with translation of the source language L, and the *back-end*, charged with the task of generating the actual target representation of the program. Such separation is common in the industry and making it explicit in the very definition of correctness seems to be a positive step towards improving the user's understanding of the system's behaviour.

Observe that establishment of the program equivalence relations σ and δ in the bottom row of Figure 2-2 is performed independently and in parallel to the actual compilation processes τ and γ. Further, note that the semantic translations ϕ, ρ and ψ are orthogonal to σ, δ, τ and γ. Nevertheless, all of these translations are clearly interrelated and, accordingly, the approach exhibits a significant amount of redundancy, whose primary purpose is to preserve an independence of definition of the three program representations involved. However, in view of the fact that, in real life, few practical programming languages and hardware architectures are given a formal semantics as part of their definition, this separation of meanings is largely spurious and could be eliminated altogether without disturbing the basic principles of the system, by folding the semantic languages M, N and U into a single formalism. Schematically, this results in a simplified view of compiler correctness presented in Figure 2-3.

This folded compiler design significantly reduces the effort required for a formal verification of the system by eliminating the need for establishing an equivalence between the three distinct formulations of the program's meaning. Yet it is unquestionably as sound as the original view of Morris. One could still interpret the three semantic

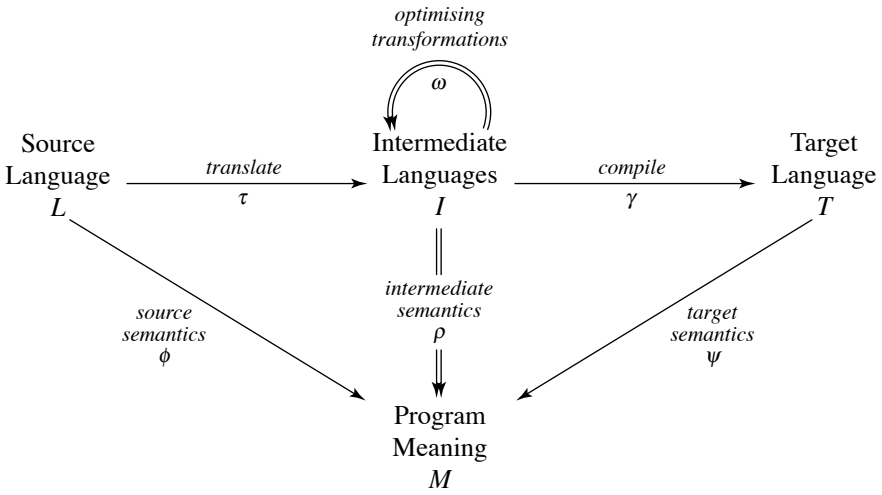


FIGURE 2-3 Correctness for a Compiler with a Unified Representation of Meaning

translations independently, with an equivalence of their meanings following directly from reflexivity of all equivalence relations. The simplification does not, however, come without a cost, which we are to pay in a reduced flexibility of a formulation of the semantic translations. The lack of "transformation buffers" between the meanings of the three languages requires that all semantic translations in the system produce structurally-identical meanings for the program irrespective of its actual representation. Effectively, the folded view of correctness requires all semantic translations to be strongly-normalising, which, in practice, can make it difficult to apply the approach to concurrent and otherwise non-deterministic systems. This issue is treated in some detail in Chapter 7, although, in general, this work assumes such limitations to be acceptable and does not attempt to alleviate them at the present time.

2.3 Correctness as a Category

Every design considered so far has been essentially extensional, in that it examines some crucial property of the translation system instead of defining the actual means by which that property is to be attained in an implementation. In particular, it describes the compiler's structure without a reference to those aspects of the system which are of the most immediate interest to its users, namely the properties of programs manipulated by the compiler. To alleviate this problem, let us reformulate the notion of compiler correctness in the language of category theory. In the following discussion, I reserve the term *function* exclusively for total (or injective) relations between objects. I use the notation "$f : X \rightarrowtail Y$" to represent a morphism from X to Y, "id" for the identity function, "id_X" for the identity morphism of X, "$\circ$" for the standard function composition combinator and "$\bullet$" to depict the more general case of morphism composition.

First, let us formalise the notion of a *programming language* as a category whose morphisms assign meanings to the sentences of some formal language:

DEFINITION: *(Programming Language)* Let M be the set of meanings and properties assigned to programs and let A be the set of all well-formed sentences in some formal language under consideration. Let μ be a function from A to M and id_X be the identity over an arbitrary set X. A *programming language* is a triple written as $\mathcal{L}\langle A, M, \mu\rangle$, which forms a category consisting of every morphism of the form $\mu_a : a \rightarrow \mu(a)$, where $a \in A$, together with the required identities $\mathrm{id}_a : a \rightarrow a$ and $\mathrm{id}_{\mu(a)} : \mu(a) \rightarrow \mu(a)$. For every $a \in A$, the set of objects of the category includes the *sentence* object a and the *meaning* object of the form $\mu(a)$. When the choice of M and μ is clear from the context, the programming language $\mathcal{L}\langle A, M, \mu\rangle$ will be written simply as $\mathcal{L}_A$.

We can now use this category-theoretical notion of a programming language to formalise the meaning of a compiler as a morphism between a pair of such categories:

DEFINITION: *(Compiler)* Let $\mathcal{L}\langle A, M, \mu\rangle$ and $\mathcal{L}\langle B, N, \nu\rangle$ be two programming languages. Let τ be a function from A to B and η be a function from M to N. Then a *compiler* is a functor $\mathcal{C}\langle\tau, \eta\rangle : \mathcal{L}_A \rightarrow \mathcal{L}_B$, such that, for every $a \in A$, $\mathcal{C}\langle\tau, \eta\rangle(a) = \tau(a)$, $\mathcal{C}\langle\tau, \eta\rangle(\mu(a)) = \eta(\mu(a))$ and $\mathcal{C}\langle\tau, \eta\rangle(\mu_a : a \rightarrow \mu(a)) = \nu_{\tau(a)} : \tau(a) \rightarrow \eta(\mu(a))$. When the choice of τ and η are clear, the compiler $\mathcal{C}\langle\tau, \eta\rangle$ from the source language A to its target B will be written simply as $\mathcal{C}_{AB}$.

DEFINITION: *(Compiler Correctness)* Let $\mathcal{C}_{AB} : \mathcal{L}_A \rightarrow \mathcal{L}_B$ be a compiler. Let $\mathcal{L}_C$ be some subcategory of $\mathcal{L}_B$. Then $\mathcal{C}_{AB}$ is *correct* if and only if it establishes an equivalence between the categories $\mathcal{L}_A$ and $\mathcal{L}_C$.

In other words, a compiler is deemed correct if and only if it maps every source program to some target representation and, further, establishes a corresponding mapping between these programs' respective meanings. In this, the above definition provides a category theoretic description of a picture similar to the original Morris diagram, which should not come as a surprise given the essentially homomorphic nature of functors. The main difference between the two notions of correctness is the fact that my proposal relies on a semantic encoding in the style of Orejas [Orejas 81] rather than the usual decoding of target meanings into their source counterparts. This results in a very broad notion of correctness, which admits many trivial cases of valid compilers that are not particularly useful to anyone. For example, it can be easily shown that a compiler $\mathcal{C}_b$ which translates all programs in A to a single program b in B is considered correct by the above criterion. However, while admitting such illogical and potentially-harmful constructions, the above definition also provides a clear means of their identification. In particular, an equivalence between categories is itself a *structure* represented by a functor (in this case, the compiler itself), so that a categorically-defined compiler correctness describes the precise *manner* in which a pair of programming languages are rendered equivalent, rather than merely asserting that equivalence without further scrutiny. Accordingly, while $\mathcal{C}_b$ may be *in some sense* correct, it is deemed correct only

in a way that should, in practice, be so obviously illogical as to readily emphasise its inadequacy to all potential users of the system. At the very least, any users actually interested in a formal analysis of programs should be able to identify such pathological compilers while applying the underlying semantic framework to the verification of their own programs translated within the system. This is precisely the kind of an intentional formulation of the correctness criteria that is necessary for a coherentist view of program correctness, and its topical confusion of internal consistency for correctness seems to be the necessary price extracted for the many benefits discussed earlier.

2.4 Linear Correctness

Finally, let us consider how the above notion of correctness can be utilised in a pragmatic task of constructing a verified compiler. First, we need to introduce the notion of a *linear compiler*, which is visualised in Figure 2-4 and captured formally as follows:

DEFINITION: *(Linear Compiler)* Let L, I and T be three formal languages (intuitively, these represent the source, intermediate and target languages of the compiler, respectively.) Let ϕ be a function from L to I, ψ be a function from I to T and $\hat{\psi}$ be a function from T to I. Let M be the set of meanings and ρ be the operational semantics of I defined as a function from I to M. Let $\mathcal{L}_I$ be the programming language $\mathcal{L}\langle I, M, \rho\rangle$. Further, let $\mathcal{L}_L$ and $\mathcal{L}_T$ be the programming languages $\mathcal{L}\langle L, M, \rho \circ \phi\rangle$ and $\mathcal{L}\langle T, M, \rho \circ \hat{\psi}\rangle$, respectively. Then, a *linear compiler* is a functor of the form $\mathcal{C}\langle \psi \circ \phi, \mathrm{id}\rangle : \mathcal{L}_L \twoheadrightarrow \mathcal{L}_T$.

In other words, a linear compiler is formed from a triple of the programming languages $\mathcal{L}_L$, $\mathcal{L}_I$ and $\mathcal{L}_T$ which share the common set of meanings M and whose semantics μ are connected through the functions $\rho \circ \phi$, ρ and $\rho \circ \hat{\psi}$, respectively. Figure 2-4 portrays the design of such a compiler visually in terms of transitions between the various program representations in the system. Observe that this diagram can be derived from Figure 2-3 by merging the semantic language M with an appropriately chosen intermediate representation I. In other words, the denotational semantics of both the source and target languages are defined in terms of a single semantic language, which is also used as an intermediate representation in the compiler. Accordingly, the semantic translations ϕ and ψ are folded with the program translations τ and γ, respectively. The combined ψ–γ arrow becomes double-headed; taken in the forward direction, it represents the compilation process, while in the reverse direction it defines a denotational semantics of the target language.

Since this approach essentially projects the multi-dimensional transitions of the original Morris Diagram onto a single axis of translation, I call the correctness criteria mandated by this design *linear correctness*. Formally:

DEFINITION: *(Linear Correctness or L.C.)* Given the three languages $\mathcal{L}\langle L, M, \rho \circ \phi\rangle$, $\mathcal{L}\langle I, M, \rho\rangle$ and $\mathcal{L}\langle T, M, \rho \circ \hat{\psi}\rangle$, a linear compiler represented by a functor of the form $\mathcal{C}\langle \psi \circ \phi, \mathrm{id}\rangle : \mathcal{L}\langle L, M, \rho \circ \phi\rangle \twoheadrightarrow \mathcal{L}\langle T, M, \rho \circ \hat{\psi}\rangle)$ is said to be *linearly correct* whenever $\rho \circ \hat{\psi} \circ \psi = \rho$.

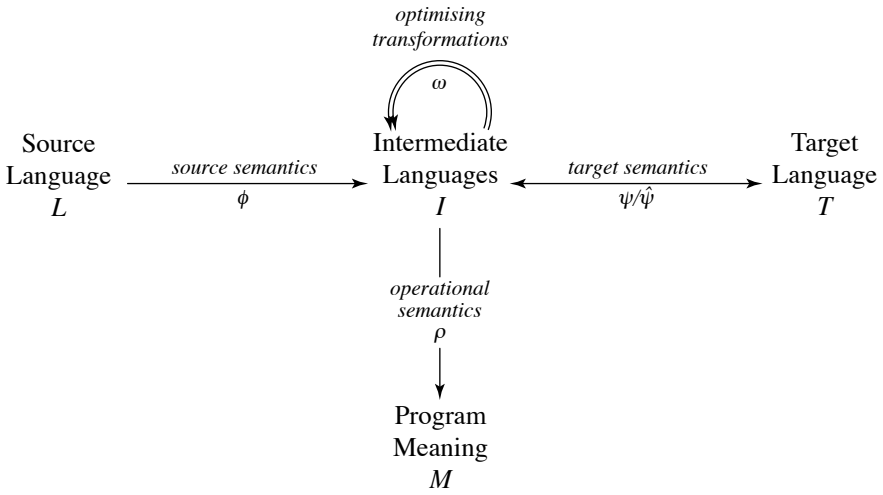


FIGURE 2-4 Schematic View of Linear Compiler Correctness

Intuitively, linear correctness requires that the result of translating every program $i \in I$ into T and back again delivers a new program with the same operational semantics as i. It turns out that linear correctness alone is sufficient to establish an absolute correctness of a linear compiler, which gives it a central rôle in the effective formulation of the correctness criteria for the compiler design described in this work.

THEOREM 2-1: *(Linearly-Correct Compiler)* If a functor $\mathcal{C}\langle\psi \circ \phi, \mathrm{id}\rangle : \mathcal{L}_L \nrightarrow \mathcal{L}_T$ is linearly correct, then it also represents a correct compiler.

Without getting unduly distracted by the nature of category equivalence, recall that, in order to establish a functor $F : A \nrightarrow B$ as an equivalence between the categories A and B, it is sufficient to show that F is *full*, *faithful* and *essentially surjective*. By definition, a functor F is said to be *full* if, for every morphism $g : b_1 \nrightarrow b_2 \in B$, there exists a morphism $f : a_1 \nrightarrow a_2 \in A$, such that $F(a_1) = b_1$, $F(a_2) = b_2$ and $F(f) = g$. Further, a functor $F : A \nrightarrow B$ is deemed *faithful* if, for every $f : a_1 \nrightarrow a_2 \in A$, the morphism $F(f) : F(a_1) \nrightarrow F(a_2) \in B$. A functor $F : A \nrightarrow B$ is *essentially surjective* if, for every $b \in B$, there exists $a \in A$, such that b and $F(a)$ are *isomorphic*. Finally, two objects $x, y \in C$ are *isomorphic* if there exist morphisms $f : x \nrightarrow y$ and $g : y \nrightarrow x$ in C, such that $f \circ g = \mathrm{id}_y$ and $g \circ f = \mathrm{id}_x$.

Before we attempt to prove theorem 2-1, let us summarise the facts we know so far about the source language $\mathcal{L}_L$, the target language $\mathcal{L}_U$ and the compiler functor $\mathcal{C}_{LU}$:

① The objects of $\mathcal{L}\langle L, M, \rho \circ \phi\rangle$ consist of sentences ℓ and meanings $(\rho \circ \phi)(\ell)$, for all $\ell \in L$. By definition of function composition "$\circ$", every such meaning can be also expressed in the form $\rho(\phi(\ell))$.

② The morphisms of $\mathcal{L}\langle L, M, \rho \circ \phi\rangle$ consists of identities and morphisms of the form $\mu_\ell : \ell \nrightarrow (\rho \circ \phi)(\ell)$, for every $\ell \in L$, which we can also write as $\mu_\ell : \ell \nrightarrow \rho(\phi(\ell))$.

③ The objects of $\mathcal{L}\langle U, M, \rho \circ \hat{\psi}\rangle$ consist of the sentences $\psi(\phi(\ell))$ and all meanings $(\rho \circ \hat{\psi})(\psi(\phi(\ell)))$, for every $\ell \in L$. By definition of function composition, every such meaning can be also expressed in the form $\rho(\hat{\psi}(\psi(\phi(\ell))))$.

④ The morphisms of $\mathcal{L}\langle U, M, \rho \circ \hat{\psi}\rangle$ consist of identities and morphisms of the form $\nu_u : u \twoheadrightarrow (\rho \circ \hat{\psi})(u)$, for every sentence $u \in U$. Given that u must have the form $\psi(\phi(\ell))$ for some $\ell \in L$, these can be rephrased as morphisms of the form $\nu_{\psi(\phi(\ell))} : \psi(\phi(\ell)) \twoheadrightarrow (\rho \circ \hat{\psi})(\psi(\phi(\ell)))$ or $\nu_{\psi(\phi(\ell))} : \psi(\phi(\ell)) \twoheadrightarrow \rho(\hat{\psi}(\psi(\phi(\ell))))$.

⑤ The functor $\mathcal{C}\langle \psi \circ \phi, \mathrm{id}\rangle$ induces the following mapping for every $\ell \in L$:

$$\begin{array}{lll} \mathcal{C}_{LU}(\ell) & = (\psi \circ \phi)(\ell) & = \psi(\phi(\ell)) \\ \mathcal{C}_{LU}((\rho \circ \phi)(\ell)) & = \mathrm{id}((\rho \circ \phi)(\ell)) & = \rho(\phi(\ell)) \\ \mathcal{C}_{LU}(\mu_\ell : \ell \twoheadrightarrow (\rho \circ \phi)(\ell)) & = \nu_{(\psi \circ \phi)(\ell)} : (\psi \circ \phi)(\ell) \twoheadrightarrow \mathrm{id}((\rho \circ \phi)(a)) & \\ & = \nu_{\psi(\phi(\ell))} : \psi(\phi(\ell)) \twoheadrightarrow \rho(\phi(\ell)) & \end{array}$$

⑥ Finally, by the linear correctness criteria, we have $\rho \circ \hat{\psi} \circ \psi = \rho$.

The above facts are obtained directly from the earlier definitions of a programming language and a compiler. We can now prove theorem 2-1 by showing that a linearly correct compiler forms a full, faithful and essentially surjective functor between the source language and some subcategory of the target language, namely, the language $\mathcal{L}\langle U, M, \rho \circ \hat{\psi}\rangle$, where U is a subset of T formed by restricting the sentences of T to the range of $\psi \circ \phi$, i.e., to the sentences of the form $\psi(\phi(\ell))$ for every $\ell \in L$. Intuitively, U contains precisely those sentence forms which the compiler can actually generate.

LEMMA 2-2: *(Fullness)* Let $\mathcal{C}_{LU}$ be a linearly correct compiler. Then, for every morphism $g : b_1 \twoheadrightarrow b_2 \in \mathcal{L}_U$, there exists a morphism $f : a_1 \twoheadrightarrow a_2 \in \mathcal{L}_L$, such that $\mathcal{C}_{LU}(a_1) = b_1$, $\mathcal{C}_{LU}(a_2) = b_2$ and $\mathcal{C}_{LU}(f) = g$.

PROOF: By definition of $\mathcal{L}_U$, every morphism $g : b_1 \twoheadrightarrow b_2$ must have the following form:

$$g = \nu_{\psi(\phi(\ell))} : \psi(\phi(\ell)) \twoheadrightarrow \rho(\hat{\psi}(\psi(\phi(\ell))))$$

so that:

$$\begin{aligned} b_1 &= \psi(\phi(\ell)) \\ b_2 &= \rho(\hat{\psi}(\psi(\phi(\ell)))) \end{aligned}$$

for some $\ell \in L$. Take $a_1 = \ell$, $a_2 = (\rho \circ \phi)(\ell)$ and $f = \mu_\ell : \ell \twoheadrightarrow (\rho \circ \phi)(\ell)$, all of which, by definition, exist in $\mathcal{L}_L$. Then:

$$\begin{aligned} \mathcal{C}_{LU}(a_1) &= \mathcal{C}_{LU}(\ell) \\ &= \psi(\phi(\ell)) \\ &= b_1 \\ \mathcal{C}_{LU}(a_2) &= \mathcal{C}_{LU}((\rho \circ \phi)(\ell)) \\ &= \rho(\phi(\ell)) \\ &= (\rho \circ \hat{\psi} \circ \psi)(\phi(\ell)) && \textit{(by L.C.)} \\ &= \rho(\hat{\psi}(\psi(\phi(\ell)))) \\ &= b_2 \\ \mathcal{C}_{LU}(f) &= \mathcal{C}_{LU}(\mu_\ell : \ell \twoheadrightarrow (\rho \circ \phi)(\ell)) \\ &= \nu_{\psi(\phi(\ell))} : \psi(\phi(\ell)) \twoheadrightarrow \rho(\phi(\ell)) \\ &= \nu_{\psi(\phi(\ell))} : \psi(\phi(\ell)) \twoheadrightarrow \rho(\hat{\psi}(\psi(\phi(\ell))))) && \textit{(by L.C.)} \\ &= g \quad \square \end{aligned}$$

LEMMA 2-3: *(Faithfulness)* Let $\mathcal{C}_{LU}$ be a linearly correct compiler. Then, for every $f : a_1 \twoheadrightarrow a_2 \in \mathcal{L}_L$, there exists a morphism $g : b_1 \twoheadrightarrow b_2 \in \mathcal{L}_U$, such that $\mathcal{C}_{LU}(a_1) = b_1$, $\mathcal{C}_{LU}(a_2) = b_2$ and $\mathcal{C}_{LU}(f) = g$.

PROOF: Intuitively, faithfulness of $\mathcal{C}_{LU}$ follows trivially from its fulness and the fact there there is at most one morphism between any given pair of objects in $\mathcal{L}_L$. In particular, by definition of $\mathcal{L}_L, f, a_1$ and a_2 must have the following forms:

$$\begin{aligned} f &= \mu_\ell : \ell \twoheadrightarrow \rho(\phi(\ell)) \\ a_1 &= \ell \\ a_2 &= \rho(\phi(\ell)) \end{aligned}$$

for some $\ell \in L$. Take $b_1 = \psi(\phi(\ell))$, $b_2 = \rho(\phi(\ell))$ and $g = \nu_{\psi(\phi(\ell))} : b_1 \twoheadrightarrow b_2$. Observe that b_1 exists in $\mathcal{L}_U$ by definition. Further, observe that $b_2 = \rho(\hat{\psi}(\psi(\phi(\ell))))$, since, by L.C., $\rho \circ \hat{\psi} \circ \psi = \rho$. Accordingly, b_2 and g also exist in $\mathcal{L}_U$. □

LEMMA 2-4: *(Essential Surjectivity of Sentences)* For every sentence $u \in U$, there exists $\ell \in L$, such that $u' = \mathcal{C}_{LU}(\ell) \in U$ and such that there exist morphisms $f : u \twoheadrightarrow u'$ and $g : u' \twoheadrightarrow u$ in $\mathcal{L}_U$, such that $f \circ g = \mathrm{id}_{u'}$ and $g \circ f = \mathrm{id}_u$.

PROOF: By definition of $\mathcal{L}_U$, every u has the form $\psi(\phi(\ell'))$, for some $\ell' \in L$. Take $\ell = \ell'$, so that:

$$\begin{aligned} u' &= \mathcal{C}_{LU}(\ell') \\ &= \psi(\phi(\ell')) \\ &= \psi(\phi(\ell)) \\ &= u \end{aligned}$$

Since u and u' represent the same object, an isomorphism between them can be established trivially by the identity morphism $f = g = \mathrm{id}_u$. □

LEMMA 2-5: *(Essential Surjectivity of Meanings)* For every meaning $n \in \mathcal{L}_U$, there exists a meaning $m \in \mathcal{L}_L$, such that $n' = \mathcal{C}_{LU}(m) \in \mathcal{L}_U$ and such that there exist morphisms $f : n \twoheadrightarrow n'$ and $g : n' \twoheadrightarrow n$ in $\mathcal{L}_U$, such that $f \circ g = \mathrm{id}_{n'}$ and $g \circ f = \mathrm{id}_n$.

PROOF: By definition of $\mathcal{L}_U$, every meaning $n \in \mathcal{L}_U$ has the form $\rho(\hat{\psi}(\psi(\phi(\ell))))$, for some $\ell \in L$. Take $m = (\rho \circ \phi)(\ell)$, which exists in $\mathcal{L}_L$ by definition. Then:

$$\begin{aligned} n' &= \mathcal{C}_{LU}((\rho \circ \phi)(\ell)) \\ &= \rho(\phi(\ell)) \\ &= \rho(\hat{\psi}(\psi(\phi(\ell)))) && \textit{(by L.C.)} \\ &= n \end{aligned}$$

Once again, the isomorphism of n and n' can then be established trivially using the identity morphisms $f = g = \mathrm{id}_u$. □

The remaining cases pertain to the identity morphisms of $\mathcal{L}_L$ and $\mathcal{L}_U$, which are trivially surjective and injective by definition of the identity function "id". Accordingly, the four Lemmas 2-2, 2-3, 2-4 and 2-5 establish the sufficient criteria for the correctness of a linearly correct compiler.

The reader will doubtless notice the embarrassing simplicity of the above proofs: all three lemmas follow directly from our earlier assumptions about the natures of the

two languages $\mathcal{L}_L$ and $\mathcal{L}_U$, together with the all-important linear correctness criteria, which is used solely to ensure a consistent mapping of meaning assignments (or morphisms) across the two program representations. In particular, it is perhaps remarkable that the compiler functor plays essentially no part in the proof of its own correctness: in effect, the entire translation system is *inferred* from the various pieces of the underlying language definitions. All of the relevant properties of the functor are instead embedded within the structure of the underlying programming language categories, so that a correctness of the entire system can be secured simply by ensuring that the particular source and target languages adhere to that structure. By emphasising scrutiny of the actual language definitions over that of their compiler's implementation details, I hope to improve relevance of the resulting correctness proof to the ultimate users of the system.

Observe that the correctness of a linear compiler does not rely on commutativity of *every* transition in the system. The only criteria required of such a design is the identity $\rho \circ \hat{\psi} \circ \psi = \rho$, which alone suffices to guarantee sensibility of the entire translation process. In this, linear correctness proves to be a very powerful tool for tackling compiler verification, since the vast majority of proofs called for by more traditional approaches are completely incidental in the above design. However, since it relies on a definition of the correctness judgement as a category equivalence, every such compiler must be accompanied by some form of a formal description that captures the *structure* of its correctness. In this work, that structure is depicted by the actual implementation of the three functions ϕ, $\hat{\psi}$ and ρ, all of which constitute a part of the compiler's specification, although the reader should observe that there is no need to expose the more elaborate implementation details of the ψ and ω translations to the user.

In a linearly correct compiler, both the source and target languages are *defined* through translations into a common universal algebra of computable functions, formed from the intermediate representation I and its semantics ρ. Each of these translations is depicted by a total function from sentences of the formal language L or T to the sentences of I and is formulated in a suitable declarative programming language such as Haskell, so that an actual compiler implementation can be extracted directly from its specification simply by executing the actual declarative representations of these two functions (or, if you prefer, a formal specification may be extracted from the implementation through a suitable typesetting of the compiler's source code as done in this work.) Critically to the design, no other formal interpretation of $\mathcal{L}_L$ and $\mathcal{L}_T$ is admitted into the system, rendering the two translations ϕ and ψ correct *by definition*. Observe that the intermediate representation I must not only satisfy the specific requirements of all possible optimisation algorithms ω, but also constitute a formal calculus of computable functions. Most importantly for a coherentist view of knowledge, this calculus must be *well known* and commonly accepted as an authoritative description of computation, so that the semantics of any language presented via a translation into that calculus can be communicated without a danger of linguistic ambiguity. In other words, the calculus I must satisfy not only the mathematical requirements of a universal algebra, but also the

philosophical requirements of a *language* in the Wittgensteinian sense of the word.

The algebraic properties of I are defined through the semantics ρ that, intuitively, ground the entire system in an external definition of the program's meaning M. In order to reduce the axiomatic base of the compiler to a bare minimum, it is advisable to define these semantics in operational terms since, as argued by Abramsky [Abramsky 93], it is precisely such an operational interpretation that renders an algebra into a programming language. In practice, it is also wise to equip I with an alternative algebraic formulation of its meaning, given that such formulations are often invaluable to the commutativity proofs of any optimising transformations ω incorporated into the system. However, to avoid the gruesome complexities of a full abstraction proof, we can easily restrict the scope of these algebraic semantics to an appropriate subset of I, as is done, for example, in Chapter 4, where the algebraic formulation is utilised to capture the portable fragment of Etude, while the full operational specification is reserved for the representation's rendition on a particular instruction set architecture.

Observe that, save for the optimising transformations ω, the three translations of a linearly correct compiler form a tree rather than a cyclic graph. This eliminates the need for administrative commutativity proofs traditionally assigned a central rôle in compiler verification. What, then, makes a linearly correct compiler correct? The simple answer is, for better or worse: *itself*. A compiler does not *implement* a language, but rather *defines* one. This notion is not, by itself, new, since it has often been stated that a compiler provides the ultimate semantics of a language. However, in a traditional compiler design such *implementation semantics* are obscured by a plethora of details stemming from the idiosyncrasies of a particular implementation environment and no compiler, up to now, could provide an *effective* definition of the translation process. The central contribution of this work is its utilisation of a suitable intermediate representation to eliminate such idiosyncrasies to the point where the implementation is arguably as readable as any other formal specification of a programming language can ever hope to be. The following three chapters contain a complete design and execution of such a linearly correct compiler for the standard C language, thus providing a solid empirical evidence for a scalability and practically of the technique.

It should also be noted that linear correctness does not guard against every conceivable programming error. In particular, a systematic specification error such as a confusion of the addition and subtraction instructions may not only pass the correctness criteria, but, more seriously, establish itself within the implementation itself. Unfortunately, errors of this nature cannot be eliminated completely under any approach to software verification, as long as no formal specification exists for the underlying computational hardware. Fortunately, the constructive nature of linear correctness ensures that every such error is readily exposed in the formal description of the system and therefore less likely to remain unnoticed. Nevertheless, in the most stringent application of the linear correctness principle, it is essential that the hardware implementation be verified within the same coherent system of knowledge as a compiler that targets its

instruction set architecture. Although the thorny issue of hardware validity is beyond the scope of the present work, it should be observed that the translations $\hat{\psi}$ and ρ provide a natural means of expressing and justifying the appropriate correctness criteria in the same cognitive framework as that utilised in the remainder of a linearly correct compiler design.

It is perhaps natural to ask if further simplifications to a compiler verification methodology can be applied beyond those already enacted in Figure 2-4. Readers proposing to do so are forewarned that all prior attempts to merge the intermediate representation I with either of the source or target languages have proven unfeasible in the past. Due to common syntactic restrictions, most source languages are not rich enough to express the output of many common transformation algorithms. Conversely, past researchers have had only limited success in applying such program optimisations directly to realistic target languages. Accordingly, a distinction between the program representations L, I and T seems to be intrinsic to a design of a practical optimising compiler. Further, as already discussed, a separation of the concrete representation I from its meaning M seems to be necessary, lest the price for the simplification extracted by Gödel's incompleteness theorem in terms of consistency proves prohibitive [Gödel 31, Gödel 40]. Finally, it should be clear from the formal outline of the design presented in this chapter, that a division of the compiler into a front and back end simplifies not only its implementation but also a formal verification of that implementation, since it eliminates most proof obligations that pertain to the input representation of programs and effectively detaches the task of verifying a translation system from a detailed semantic study of its source language.

2.5 Designing a Linear Compiler

In order to obtain an efficient implementation of a practical linearly correct compiler, its design should generally proceed in a number of discrete steps as follows:

① First, the syntax I and the operational semantics ρ are defined for an intermediate program representation utilised by the compiler. In Chapter 4, I present a purely functional monadic language *Etude* that is suitable for this purpose and, in Chapter 6, I complete Etude's specification with a presentation of its operational semantics ρ, defined as a reduction of its terms into their various *beta normal forms*, which are taken to represent the meanings of programs M in the system.

② A mapping ϕ of all well-defined sentences in the source language L to the sentences of I is established. This definition is taken as the authoritative source of information about the programming language $\mathcal{L}_L$ accepted by the compiler and users are expected to utilise it directly in any verification efforts of programs developed in $\mathcal{L}_L$, ensuring that both the correctness criteria and the user's understanding of the compiler's behaviour are derived constructively from the actual implementation of the translation system. In Chapter 5, I present such a mapping for the standard C

programming language [ANSI 89], which is translated into the calculus from Chapter 4. A totality and termination properties of this mapping is a direct consequence of its inductive formulation over the structure of C programs.

③ A correspondence between the implementation and the formal translation ϕ is established. In Chapter 5, this is achieved by formulating the translation itself in the Haskell programming language, so that the required correspondence is derived implicitly by the Haskell compiler.

④ Optionally, if another well known semantic interpretation of L exists (such as the standard operational semantics of ML), then full abstraction may be established between the denotational semantics defined through the above mapping and that auxiliary semantic interpretation. However, since no formal semantics of C have been accepted as an authoritative definition of the language, this step is not required for a verification of the C compiler presented in this work.

⑤ For every optimising program transformation ω implemented by the compiler, commutativity of that transformation with respect to the operational semantics ρ is established. Observe that the transformations themselves need not be included in the compiler's definition and, due to space and time constraints, the present work refrains from divulging details of any such optimisation algorithms.

⑥ A total mapping ψ of all well-defined sentences in the target language T to the sentences of I is formulated and made available to the user as an authoritative specification of the computational system targeted by the compiler. In Section 6.2, I present such a mapping between the machine instructions of Knuth's MMIX architecture [Knuth 05] and the corresponding Etude functions from Chapter 4.

⑦ A mapping $\hat{\psi}$ of every well-defined sentence in the intermediate representation I to an appropriate target language construct from T is implemented as part of the compiler's back end. A complete example of a typical execution of such a mapping is given in Section 6.4 for a translation of Etude programs into their MMIX counterparts.

⑧ Finally, the linear correctness criteria must be established for the ρ, ψ and $\hat{\psi}$ functions. In Section 6.4, this is achieved directly within my logical framework described in Chapter 3. Once again, a totality of ψ follows immediately from its inductive formulation over the structure of input programs.

The most striking feature of the linear correctness approach to compiler verification is the relatively low number of proof obligations required for establishment of the implementation's correctness. After perusing the following four chapters, the reader will doubtless notice that the bulk of this work consists of definitions instead of a formal reasoning about these definitions' properties. This feature of linear correctness is innate to the design and constitutes one of its main achievements. The weight of the formal work is shifted from proof obligations to a precise specification of the system, since it is that specification rather than the proofs that are examined by the product's eventual

users. Only a limited number of crucial properties must be established to guarantee correctness and every one of these properties is carefully scrutinised in the remainder of this work. Accordingly, the translation of C programs into their MMIX counterparts defined in the following chapters constitutes a fully-verified compiler for the complete C programming language, whose correctness is formulated not in terms of some critical property of the translation system such as its commutativity with respect to a set of auxiliary translations, but, rather, as an expression of the fact that the implementation is consistent with the formal mapping of the source and target languages to the Etude representation of computable functions defined in Chapter 4. As long as the users are satisfied that this mapping represents the desired processing intended for their C program, they can rest assured that the compiler will perform the required job with an absolute precision, a belief that is not only provable, but also justifiable by the users' own expectations of the program's behaviour. Neither the Gettier problem nor any other issues of model fidelity can ever arise within this cognitive framework, demonstrating conclusively that the linear correctness approach provides a compelling constructive statement of the system's correctness.

3

HASKELL AS A NOTATION

The most essential gift for a good writer
is a built-in, shock-proof, shit detector. This is
the writer's radar and all great writers have had it.
— Ernest Hemingway

Philosophy and mathematics have been always interlocked in a tight love-and-hate relationship, but never more so than in the fall of the eighteenth century, when Frege proposed to ground all of mathematics in a strict formalism of Cantor's set theory. After Russell's discovery of the famous paradox in Frege's work, the two philosophers entered into a long and never quite resolved metaphysical debate on the nature of mathematics, that culminated in Russell's publication of *Principia Mathematica* and eventually caused Frege to abandon his own involvement with the project. However, despite their stated differences, both men shared the same fundamental desire to furnish mathematical reasoning with an absolute rigour and certainty.

These ideas were perhaps best expressed by Hilbert, who believed that the only completely satisfactory way of achieving Frege's and Russell's objectives is to base mathematics on manipulation of such entities that have no meanings in themselves and therefore do not require a further scrutiny. For Hilbert, these entities were represented by *symbols*: marks on paper whose only significance is derived from their visually distinguishable features. Of course, Gödel has since demonstrated that Hilbert's objective is not achievable in its full generality. In 1930, he published the fabled proof of the *incompleteness theorem* [Gödel 31, Gödel 40], which, however, did not prevent others from electing the very incompleteness of mathematics as a subject of logical scrutiny on its own, eventually giving rise to the new discipline of *meta-mathematics*.

In the age of powerful digital computers, meta-mathematics has been given a breath of new life in the form of capable theorem specification and proof development environments such as LCF, PVS, Isabelle/HOL, ACL2, Twelf and Coq [Milner 73, Owre 92, Paulson 05, Kaufmann 00, Twelf 98, INRIA 02]. Most recently, Sewell *et al.* designed the Ott system tailored explicitly towards specification of programming languages and their semantics [Sewell 07]. Nevertheless, in this work I chose to present all formal definitions directly in Haskell [Peyton Jones 03], in order to focus the discussion on a compiler's implementation rather than a specification of its source language. Since Haskell, by itself, is incapable of expressing any but the simplest forms of logical reasoning, most formal properties described in this work are depicted in a set of language extensions described in Section 3.6 below.

3.1 Presenting Algorithms

Much of the formal specification of C defined in this work is expressed in the form of a concrete compiler for the language, executed in some seven thousand lines of literate Haskell source code and formatted into the current presentation using Knuth's TEX typesetting system [Peyton Jones 03, Knuth 86]. Nevertheless, since the resulting model is likely to be of most interest to C, rather than Haskell programmers, detailed familiarity with Haskell should not be a prerequisite for its thorough understanding. To this end, an elaborate typesetting has been applied to the exposition of all Haskell definitions appearing in this work, relying on a highly sophisticated lambdaTEX macro package written by myself and available from http://www.jantar.org/lambdaTeX/. The primary purpose of lambdaTEX is to detach the abstract semantic content of these definitions from their concrete rendition in a syntax of a chosen implementation language, so that they may be absorbed by any reader possessing an adequate understanding of the relevant mathematical ideas, with only a minimal exposition to the practice of functional programming *per se*. Nevertheless, in the following section, I also attempt to accommodate those readers who are already accustomed to the Haskell school of expression, by describing the precise relation between the actual source code and its often-opaque rendition in my work.

Since the present document has been constructed directly from a set of literate Haskell source files, processed in TeX without recourse to any external typesetting tools or code generators, it already constitutes a well-typed source code of an actual C compiler, which may be itself assembled into an executable program under any standard Haskell development environment. The implementation relies solely on the plain Haskell 98 language [Peyton Jones 03] amended only with a few common extensions and a handful of popular set manipulation libraries, whose precise usage is described exhaustively in Appendix A. The lambdaTEX package achieves the elaborate formatting of all programs presented in this work by applying a small number of carefully designed typesetting rules to the stream of Haskell tokens from their literate source files.

In particular, an appearance of all Haskell names is detached from their source representation by allowing individual Haskell lexemes to be typeset as arbitrary visual symbols, usually chosen to resemble common mathematical notation as closely as possible within the constraints of the language. For example, the Haskell operators "+", "∗", "==", "/=" and "\" are presented as "+", "×", "=", "≠" and "λ", respectively. The typesetting can be also applied to infix uses of a variable or a constructor. For example, "`compositeType`" appears as "⊔" in the presentation. Some symbols may be even typeset as a white space. For example, following the standard mathematical practice, the multiplication operator "×" is often omitted when computing the product of two visually-atomic entities, so that the Haskell expression "4 ∗ x" may appear in this work as "4*x*", avoiding any possible confusion by ensuring that the intended meaning is clear from the context of such a construct. Usually, symbols used to represent entities with discrete Haskell names are chosen to be visually distinctive; however, this requirement

is sometimes relaxed whenever the context in which the entity is applied renders any uncertainty of meaning unlikely.

Most Haskell variables are given short descriptive names that are typeset in an italic font such as in "x", "*var*" or "α", while Haskell keywords and names of functions are presented in the standard text font such as "log" or "max". An italic font is also used to describe types corresponding to the non-terminal elements of programming language grammars, in line with the conventions adopted by the C Standard [ANSI 89]. For example, the "AbstractDeclaratorT" data type defined in Chapter 5 is typeset as "*abstract-declarator*". Terminal elements of the C grammar are presented in bold monospaced font such as that used in "**`while`**" and "**`*=`**". Terminal elements of the Etude syntax defined in Chapter 4 are typeset in a sans-serif font to distinguish them visually from similarly named elements of the Haskell and C grammars; for example the "LET" term constructor of Etude is presented uniformly as "LET".

For conciseness, variables are often assigned short names indicative of their type. For example, variables representing Etude terms have names derived from the Greek letter "τ", such as "τ", "τ'" or "τ_7", while C expressions are most often represented by variables such as "e", "e'" or "e_1". The names of list and set-valued variables are either represented by a single italicised upper-case letter such as "A", or else decorated with the macron accent, so that a variable bound to a list or set of Etude terms would be generally written as "$\bar{\tau}$". Similarly, variables that represent functions of the Haskell type "$a \to a$" are decorated with a ring, so that a function from Etude terms to other term structures would be generally bound to a name such as "$\mathring{\tau}$".

Following the time-honoured mathematical tradition, certain Haskell operators such as "^" and "^^" can be also typeset by modifying an appearance of their operands. In particular, the Haskell expressions "a^n" and "a^^n" are always presented using the familiar mathematical notation "a^n", while "ceiling x" and "listToSet [1, 2, 3]" materialise as "$\lceil x \rceil$" and "$\{1, 2, 3\}$", respectively. This facility of lambdaTEX proves particularly useful for exposition of parse trees. For example, in the actual Haskell source code, a C statement of the form "**`if (`***e***`)`** s_1 **`else`** s_2" appears simply as the construct "IfElseStatement e s1 s2".

A feature of Haskell's Hindley-Milner type system that has traditionally hindered its applications as a mathematical notation is the lack of support for untagged union types, which frequently results in a significant visual clutter throughout typical Haskell renditions of a formal algorithm. For example, the fact that the set of postfix expressions in C includes all primary expressions and that the set of primary expressions also incorporates all constants is captured in Chapter 5 by a data type "PostfixExpressionT" with a constructor "PrimaryExpression PrimaryExpressionT" and a data type "PrimaryExpressionT" with a constructor "Constant ConstantT". In order to use a given constant c in the context of a postfix expression, it is necessary to wrap it in a pair of constructors "PrimaryExpression (Constant c)". Such deeply nested terms are common and can easily obscure the often-simple idea behind the program. For example,

at one point in Chapter 5, a Haskell program contains the term "ConstantExpression (LogicalOrExpression (LogicalAndExpression (InclusiveOrExpression (ExclusiveOrExpression (AndExpression (EqualityExpression (RelationalExpression (ShiftExpression (AdditiveExpression (MultiplicativeExpression (CastExpression (UnaryExpression (PostfixExpression (PrimaryExpression (Constant (IntegerConstant integer_constant1))))))))))))))))", which simply depicts a pattern used to match an arbitrary C expression against an integer constant. In the presentation, typesetting of such *coercion constructors* is always suppressed together with the following pair of parentheses, resulting in the above expression being formatted simply as "*integer-constant*$_1$". Care has been taken to ensure that such suppression does not introduce visual ambiguities into the work. In particular, the reader should observe that every Haskell variable whose name is formatted identically to that of some data type defined earlier (possibly decorated with various subscripts and diacritical marks) or is written as a single letter such as e or τ that was explicitly proclaimed to range over a particular type earlier in the discussion, always represents a value of that type. Accordingly, whenever such name is applied in any other context, it should be assumed that an appropriate hidden coercion constructor has been inserted into the actual Haskell program, so that "*integer-constant*$_1$" is always bound to a value of type "IntegerConstantT", while e_k and τ_k can be safely assumed to depict, respectively, C expressions and Etude terms regardless of the apparent context in which they appear.

3.2 Type Signatures

For the benefit of readers already fluent in Haskell, every function declaration presented in this work is always accompanied by its precise type signature, intended as an additional clarification of the construct's intuitive purpose. In order to accommodate the visual formatting of functions such as "^", their declarations are given a special treatment by lambdaTEX, whereby the placement of each argument within a fully-saturated application of a Haskell function is marked in its type signature by the symbol "·". For example, the declaration of the "^" operator is presented as:

$$[\![\cdot]\!]^{[\cdot]} :: (\text{num } T) \Rightarrow T \rightarrow \mathit{integer} \rightarrow T$$

Observe that the cardinality of the function is equal to the number of "·" symbols in its signature and that each of these symbols corresponds to one type in the list to the right of the "::" sign, with the final entry in that list representing the type of the function's fully-saturated application. In all cases, the "·" symbols are introduced implicitly into the presentation by the lambdaTEX macro package and do not require any extensions to the standard Haskell grammar.

3.3 Language Syntax

The concrete syntax of all programming languages presented in this work is defined using an appropriate collection of Haskell data types. For example, the hypothetical

BNF grammar fragment "*expr* ::= *expr* + *atom* | *expr* − *atom*" would be represented in Haskell as the data type "data ExprT = ExprT `Plus` AtomT | ExprT `Minus` AtomT" and typeset in the following manner:

expr :
    expr + *atom*
    expr − *atom*

Observe that the "data" keyword and the "|" operator are suppressed in such definitions, while the "=" symbol materialises simply as ":". Similarly, the "type" and "newtype" keywords are always suppressed to reduce the amount of incidental implementation detail in the presentation. Although the parse trees formed from such data constructors are represented by ordinary Haskell expressions, for the sake of exposition their occurrences within general Haskell terms are usually surrounded by the semantic brackets "⟦⟧". Further, in order to establish a visual separation between the syntactic and semantic aspects of a given linguistic construct, every appearance of a non-trivial Haskell expression within a parse tree is itself surrounded by the same semantic brackets, as in "⟦**if (**e**)** ⟦$f\,s_1$⟧ **else** s_2⟧", wherein the subexpression "$f\,s_1$" constitutes a single operand of the "IfElseStatement" data constructor mentioned earlier. The "⟦⟧" brackets have no operational significance beyond their grouping effect and, in the actual Haskell implementation, materialise simply as a pair of ordinary parentheses symbols "()". Whenever a given syntactic entity has a context-sensitive interpretation, a suitable formal representation of that context is also inserted into the semantic brackets, separating it from the entity by the "▷" symbol as in "$\mathcal{D}_{\mathrm{V}}[\![\Sigma \triangleright e_1 \star e_2]\!]$".

In many ways, the lambdaTEX approach shares many features with Ott [Sewell 07], which also strives to provide a holistic environment for development of language semantics. However, unlike Ott, it is more lightweight and therefore applicable to a broader class of problems beyond compiler design. Most importantly, it allows us to focus the discussion on a system's implementation instead of its specification, which, as argued in Chapter 2, is essential for the compiler verification methodology applied in this work.

I consider the above approach to the typesetting of programs a minor but very effective contribution in the area of formal program verification, since it facilitates a clear and precise definition of formal concepts without obscuring their essential mathematical meaning with idiosyncrasies of a given software development environment, while retaining an ability to verify many aspects of the presentation mechanically using standard type checking tools. In particular, the GHC Haskell compiler has proven invaluable during writing of this thesis, by eliminating many embarrassing errors that have frequently crept into the work in the course of its development. To the best of my knowledge, this level of clarity in presentation of executable programs has been never attempted in the past, not even in the major feats of literate programming such as Donald Knuth's *Computers and Typesetting* series [Knuth 84].

3.4 Curry-Howard Isomorphism

As many readers may be already aware, a strongly typed language such as Haskell represents more than just a convenient program development environment. Its type system provides a powerful means for a precise expression of many program properties that are statically verified as part of the compilation process. In effect, such languages incorporate a restricted form of theorem proving capabilities.

The correspondence between type systems and proofs has been recognised since the 1950s and is generally known as the *Curry-Haward isomorphism* [Howard 69]. Under this isomorphism, every type can be viewed variously as a theorem, proposition or logical judgement, whose proof is always represented by a term of that type. For example, in Haskell (whose type system implements a second-order propositional logic) the following type declarations introduce a pair of two new theorems into the program:

```
data FACT1 = PROOF1
data FACT2 = PROOF2
```

Formally, the data types FACT_1 and FACT_2 represent *judgements* in Martin-Löf's constructive type theory [Martin-Löf 71, Martin-Löf 83]. Intuitively, it is convenient to think of such propositions as claims of an existence of a program that exhibits the property being named. Accordingly, the data constructors PROOF_1 and PROOF_2 constitute proofs in the sense of providing examples of a program for which the corresponding properties FACT_1 and FACT_2 hold.

Although such primitive judgements are rarely of any significant interest, under the Curry-Howard isomorphism more complex reasoning can be encoded within the type system and proven by an example of a program exhibiting the property being judged. For example, a conjunction of n theorems is depicted by a product type or a polymorphic n-ary type constructor, so that the proposition "$P \wedge Q$" can be captured by the following type definition:

```
data FACT3 P Q = PROOF3 P Q
```

where PROOF_3 corresponds directly the the $\wedge$-introduction rule of classical logic. For example, the judgement "$\text{FACT}_1 \wedge \text{FACT}_2$" is justified by the following program:

```
THEOREM1 :: FACT3 FACT1 FACT2
THEOREM1 = PROOF3 PROOF1 PROOF2
```

Similarly, a disjunction "$P \vee Q$" can be described by the following Haskell type:

```
data FACT4 = PROOF4A A | PROOF4B B
```

and the judgement of "$\text{FACT}_1 \vee \text{FACT}_2$" is supported by either of the two proof expressions "$\text{PROOF}_{4A}\ \text{PROOF}_1$" or "$\text{PROOF}_{4B}\ \text{PROOF}_2$".

Perhaps the most interesting aspect of the Curry-Howard isomorphism pertains to its representation of logical implications, which correspond precisely to the notion of a function type. For example, the proposition "if FACT_1 then FACT_4" may be expressed as

the Haskell type "$\text{FACT}_1 \rightarrow \text{FACT}_4$". To prove such conditional proposition, it is merely necessary to construct a function that, given an arbitrary FACT_1, delivers a FACT_4:

$$\begin{aligned}&\text{THEOREM}_2 :: \text{FACT}_1 \rightarrow \text{FACT}_4\\&\text{THEOREM}_2\ p = \text{PROOF}_{4\text{A}}\ p\end{aligned}$$

in which the operand p may be viewed as a hypothetical proof of the antecedent FACT_1.

These three primitive judgement forms can be used to capture and justify many interesting theorems of the first-order propositional logic. For example, transitivity of implication can be expressed as the following theorem:

$$\text{THEOREM}_3 :: (P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow (P \rightarrow R)$$

which we can read as "if P implies Q and Q implies R, then P implies R". This theorem can be easily established in Haskell using the following definition:

$$\text{THEOREM}_3\ p_1\ p_2\ p_3 = p_2\ (p_1\ p_3)$$

Observe that THEOREM_3 is parameterised on two operands p_1 and p_2 which represent proofs of the respective assumptions "P implies Q" and "Q implies R". It is not difficult to convince oneself that the resulting function does, indeed, have the required type and every Haskell compiler will happily vouch for the fact. An astute reader will also notice that the resulting proof term describes the standard function composition operator "$\circ$" defined in the Haskell prelude.

3.5 Coverage and Termination

In standard Haskell, every type is always populated with at least one value "$\perp$", generally introduced into the program using a non-terminating function, a non-exhaustive pattern match or an explicit "error" construct. For example, the existing type checker makes it easy to prove an arbitrary judgement P using the following recursive construct:

$$\begin{aligned}&\text{PARADOX} :: P\\&\text{PARADOX} = \text{PARADOX}\end{aligned}$$

Since such proofs make it impossible to distinguish between proper theorems and fallacies, they are said to be *inconsistent* and are commonly considered to be meaningless as far as their logical interpretation is concerned. Accordingly, all Haskell functions used as proofs of judgements must be total and provably terminating. Unfortunately, these requirements are generally unenforceable within the constraints of the standard Haskell language and, to this end, I propose to extend its type system with a new kind of *property types*, such that every expression of a property type always represents a total and provably-terminating function. In the actual source code, this kind may be rendered as the symbol "$**$", although, for aesthetic reasons, it is typeset as "$\star$" in this presentation, in order to distinguish it from the usual kind of ordinary Haskell types "$*$". Although the scope of the current project does not permit me to develop the complete theory of the resulting logical framework, in the remainder of this chapter I describe its syntax and semantics with sufficient detail as to facilitate the usage of property types in

a convenient "pen and paper" notation for definition of all formal theorems and proofs appearing throughout the balance of this work. In particular, I do not concern myself with the precise means by which the totality and termination of proofs (or property-valued expressions) is to be established by the Haskell compiler. In practice, every theorem included in this work is justified by a simple inductive argument over the structure of the corresponding Haskell definitions, so that its termination follows directly from the underlying structural induction principle and its totality can be ascertained with the well-known coverage checking techniques of Coq and Twelf [Shürmann 03], raising a very real possibility of a future Haskell implementation that would permit a mechanical verification of these proofs.

Under the proposed language extension, new properties are introduced into the program using the syntax of generalised abstract data types akin to those described by Peyton-Jones *et al.* [Peyton Jones 06]. In general, all such definitions must be accompanied by a complete type signature in order to aid the unfortunate type checker charged with their verification. For example, before accepting the PROOF_1, PROOF_2, PROOF_3, PROOF_{4A} and PROOF_{4B} data constructors as sound proofs of their corresponding theorems, we should rewrite the earlier definitions of FACT_1, FACT_2, FACT_3 and FACT_4 as follows:

$$\begin{aligned}
&\text{data } \text{FACT}_1 :: \star \text{ where}\\
&\quad \text{PROOF}_1 :: \text{FACT}_1\\
&\\
&\text{data } \text{FACT}_2 :: \star \text{ where}\\
&\quad \text{PROOF}_2 :: \text{FACT}_2\\
&\\
&\text{data } \text{FACT}_3 :: \star \to \star \to \star \text{ where}\\
&\quad \text{PROOF}_3 :: P \to Q \to \text{FACT}_3\ P\ Q\\
&\\
&\text{data } \text{FACT}_4 :: \star \to \star \to \star \text{ where}\\
&\quad \text{PROOF}_{4A} :: P \to \text{FACT}_4\ P\ Q\\
&\quad \text{PROOF}_{4B} :: Q \to \text{FACT}_4\ P\ Q
\end{aligned}$$

By the end of this chapter, it should be obvious that property types can be modelled naturally within the well-known calculus of *co-inductive constructions* [Coquand 86] and therefore enjoy the soundness and strong confluence perquisites of that established logical framework.

3.6 Reasoning About Programs

While the above examples go a long way towards exposing the deeply-rooted logical interpretation of the Hindley-Milner type system, at the end of the day, it is the dynamic properties of algorithms rendered as Haskell functions that are of a most immediate interest in our program verification project. Although, in principle, it should be possible to encode many such properties using various established extensions to the Haskell language such as functional dependencies and associated types [Jones 00, Chakravarty 03], the syntax of these extensions is rather arcane and not particularly well suited for speci-

fication of complex theorems such as those that we are likely to encounter during formal verification of our compiler.

The main challenge faced by all polymorphic type systems (whether dependant or not) is the unification of types with syntactically-distinct structures. While good solutions to this problem have been developed for the standard Hindley-Milner design and even for the more complex dependently-typed calculi of Coq and Twelf [Milner 77, Coquand 86, Harper 87], all of these algorithms break down when faced with types parameterised on diverging expressions, since they rely crucially on structural comparison of certain normal term forms that may fail to exist in the presence of infinite reduction sequences. Traditionally, term comparisons are formulated as an implicit notion of *definitional equality*, which is typically integrated directly into the system's type checking mechanism. However, in principle, it should be possible to expose this definitional equality to the programmer as a specialised property of the "$\star$" kind, whose individual axioms (or property constructors) are provided by the plethora of function declarations included in the program. For example, the well-known definition of the boolean "||" operator (typeset in this work as "$\vee$") is given in the standard Haskell prelude as:

$$[\![\cdot]\!] \vee [\![\cdot]\!] :: \mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool}$$

$$\begin{array}{ll} [\![\mathrm{true}]\!] \vee [\![b]\!] = \mathrm{true} & \\ [\![\mathrm{false}]\!] \vee [\![b]\!] = b & \\ [\![\bot]\!] \vee [\![b]\!] = \bot & \textit{(pseudo-code for exposition only)} \end{array}$$

observing that the final "$\bot \vee b$" case in this definition is completely determined by the semantics of the earlier two cases and is made explicit above only as an aid to the following discussion. In particular, the declaration of "$\vee$" can be seen as introducing three definitional equality rules of the form "$[\![\mathrm{true} \vee b = \mathrm{true}]\!]$", "$[\![\mathrm{false} \vee b = b]\!]$" and "$[\![\bot \vee b = \bot]\!]$". If we admit all such terms as property types, together with a set of suitable substitution-like primitives for their manipulation, then the programmer should, in principle, be able to describe arbitrarily-complex unification proofs explicitly within a program's source code, relieving the type checker from asserting equality between any but the simplest structurally-aligned dependant type parameters.

Under the proposed extension to the Haskell type system, definitional equality is taken to epitomise a polymorphic property type of the kind "$\forall\, T :: * \Rightarrow T \rightarrow T \rightarrow \star$" depicted by the same equality sign "=" as that used within ordinary Haskell function declarations, so that the syntax "$[\![x = y]\!]$" represents the definitional equality property of the two appropriately-typed Haskell terms x and y. A reader should observe that this treatment requires support for dependant property types that are parameterised on arbitrary Haskell expressions, with a pair of such properties considered to be equivalent whenever these expressions have identical structures after suitable renaming of their bound variables. For convenience, some limited amount of restricted term reduction may be admitted into the system, provided that only provably-terminating evaluation sequences are ever considered by the type checker.

The primitive constructors of the definitional equality property are provided by the declaration cases present in the actual Haskell program and, within other property proofs, are represented simply by the keyword "DEFN". If the constructor's precise type is not clear from its context within the proof, the keyword can be accompanied by a suitable expression type signature as in "DEFN :: $[\![\text{length}(x{:}xs) = 1 + \text{length}(xs)]\!]$". Although I leave the detailed type checking of "DEFN" proofs open for future research, in principle we can think of these constructs as means of saying to the compiler: "go on, reduce both sides of the "=" sign and compare their normal forms for equivalence modulo alpha conversions, I can assure you that you won't run into any nonsense such as non-terminating expressions."

Since every use of the "DEFN" keyword represents a property constructor, it is possible (and generally essential) to permit application of this keyword in the context of a Haskell pattern, so that a proof of some property possessed by a Haskell function may be devised by scrutiny of the individual cases in its declaration. Such pattern matching may be applied either directly using an appropriate "case" expression or indirectly in a theorem's argument list, relying on the standard Haskell translation of such constructs into their corresponding "case" form. As an example, consider a hypothetical theorem with the following type signature:

$$\text{THEOREM}_4 :: \forall\, a, b, c :: \mathit{bool} \Rightarrow [\![a \vee b = c]\!]$$

whose proof would generally follow the following pattern:

$$\begin{aligned}
&\text{THEOREM}_4\ [\![\text{DEFN} :: [\![\text{true} \ \vee b = \text{true}]\!]]\!] = \ldots \text{ where } b :: \mathit{bool}\\
&\text{THEOREM}_4\ [\![\text{DEFN} :: [\![\text{false} \vee b = b]\!]]\!] = \ldots \text{ where } b :: \mathit{bool}\\
&\text{THEOREM}_4\ [\![\text{DEFN} :: [\![\bot \ \vee b = \bot]\!]]\!] = \ldots \text{ where } b :: \mathit{bool}
\end{aligned}$$

In the type declaration of THEOREM_4, the "$\forall$" symbol (written as "forall" in the actual Haskell source code) represents a dependant product binder from the calculus of constructions and the double arrow "$\Rightarrow$" is used to mark the preceding term as an inferred argument whose value must be derived internally by the type system for every occurrence of the corresponding function within the program. In particular, binders of both the "$\forall\, a_1, a_2 \ldots a_n :: T \Rightarrow U$" and "$\forall\, a_1, a_2 \ldots a_n :: T \rightarrow U$" forms introduce a set of n free variables into the definition, all of which have the type of T. If the the "$\rightarrow$" form of "$\forall$" is used, then values of these variables must be supplied explicitly by the user, whereas, under the "$\Rightarrow$" form, these values are always reconstructed implicitly by the type inference algorithm. Incidentally, in the above example, T and U need not represent a Haskell type; they may equally well depict a property or even a kind, with the later case allowing explicit formulation of polymorphic theorems in the system.

The reader should observe that, since the type annotations on the "DEFN" keyword include arbitrary Haskell terms, we also require some means of distinguishing between free and bound occurrences of any variables within such parameters. To this end, I always assume that every Haskell identifier appearing within a parameter of a dependant refers to an entity bound in the surrounding scope, unless the associated "where"

clause declares the identifier's type without a corresponding value, so that the signature "$b :: bool$" in the above example marks the variable b as free within THEOREM_4. Although, strictly speaking, such *binding signatures* violate the standard scoping rules of Haskell, they are essential to an effective application of my type system extensions and, in all cases, the reader should assume that any Haskell variables introduced into a property-typed declaration by bindings within its "where" clause must be either defined somewhere within that declaration's scope, or else their values must be reconstructable by the type inference algorithm from the term's remaining parameters.

Since the definitional equality property "=" depicts an equivalence relation, we must also furnish it with the expected reflexivity, symmetry and transitivity axioms. In our hypothetical extended Haskell prelude, these axioms can be depicted as follows:

$$\begin{array}{ll}
\text{REFL} & [\![\cdot]\!] \quad :: \forall\, T :: *,\, \tau :: T \Rightarrow T \rightarrow [\![\tau = \tau]\!] \\
\text{SYMM} & [\![\cdot]\!] \quad :: \forall\, T :: *,\, \tau_1,\, \tau_2 :: T \Rightarrow [\![\tau_1 = \tau_2]\!] \rightarrow [\![\tau_2 = \tau_1]\!] \\
\text{TRANS} & [\![\cdot]\!]\ [\![\cdot]\!] :: \forall\, T :: *,\, \tau_1,\, \tau_2,\, \tau_3 :: T \Rightarrow [\![\tau_1 = \tau_2]\!] \rightarrow [\![\tau_2 = \tau_3]\!] \rightarrow [\![\tau_1 = \tau_3]\!]
\end{array}$$

Given an arbitrary well-typed Haskell expression τ, "REFL τ" proves the trivial definitional equality $[\![\tau = \tau]\!]$. Further, given a proof P of $[\![\tau_1 = \tau_2]\!]$, "SYMM P" constructs the proof of $[\![\tau_2 = \tau_1]\!]$. Finally, given a proof P of $[\![\tau_1 = \tau_2]\!]$ and a proof Q of $[\![\tau_2 = \tau_3]\!]$, "TRANS P Q" constructs the proof of $[\![\tau_1 = \tau_3]\!]$.

By themselves, the above "DEFN" proofs are insufficient to capture many useful cases of reasoning about practical Haskell programs. For these, one additional proof schema of the form "SUBST $P :: [\![\tau_1 = \tau_2]\!]$ IN $x \rightarrow \tau$" allows us to assert definitional equality between two variants of the same term τ in which all free occurrences of the variable x have been replaced with the respective terms τ_1 and τ_2, provided that the definitional equality between τ_1 and τ_2 can be guaranteed by some explicitly-specified proof P. In literature, this final axiom of definitional equality goes variously by the name of *referential transparency* or *compatibility* of Haskell terms, which, intuitively, dictates that two equal entities should behave identically in every context permitted by its type, so that a pair of such terms can be always interchanged freely within a larger Haskell program or proof term.

As an example of the above rules in action, let us attempt to prove the following simple theorem, which states that, for every pair of boolean values a and b, "$a \vee b$" is greater than or equal to a:

$$\text{THEOREM}_5 :: \forall\, a,\, b :: bool \rightarrow [\![(a \vee b) \geq a = \text{true}]\!]$$

The theorem assumes the following well-known ordering of the two boolean constructors "false" and "true", which, in the Haskell prelude, is introduced by the "ord" class instance derived implicitly for the standard "*bool*" type as follows:

$$[\![\cdot]\!] \geq [\![\cdot]\!] :: bool \rightarrow bool \rightarrow bool$$

$$\begin{array}{lll}
[\![\text{false}]\!] & \geq [\![\text{false}]\!] & = \text{true} \\
[\![\text{false}]\!] & \geq [\![\text{true}]\!] & = \text{false} \\
[\![\text{true}]\!] & \geq [\![b]\!] & = \text{true} \\
[\![\bot]\!] & \geq [\![b]\!] & = \bot
\end{array}$$

First, we consider the case in which the left operand of THEOREM_5 is true. The proof constructs four lemmas, with the final L_4 having the desired type of THEOREM_5:

$$\begin{aligned}
&\text{THEOREM}_5\ [\![\text{true}]\!]\ [\![b]\!] = \text{L}_4 \\
&\quad \text{where } \text{L}_1 = \text{DEFN} && :: [\![\text{true} \vee b = \text{true}]\!] \\
&\qquad\ \ \text{L}_2 = \text{SUBST L}_1 \text{ IN } x \to x \geq \text{true} && :: [\![(\text{true} \vee b) \geq \text{true} = \text{true} \geq \text{true}]\!] \\
&\qquad\ \ \text{L}_3 = \text{DEFN} && :: [\![\text{true} \geq \text{true} = \text{true}]\!] \\
&\qquad\ \ \text{L}_4 = \text{TRANS L}_2\ \text{L}_3 && :: [\![(\text{true} \vee b) \geq \text{true}]\!]
\end{aligned}$$

In the next two cases, the first operand is false and the proof proceeds as follows:

$$\begin{aligned}
&\text{THEOREM}_5\ [\![\text{false}]\!]\ [\![\text{true}]\!] = \text{TRANS } (\text{SUBST L}_1 \text{ IN } x \to x \geq \text{false})\ \text{L}_2 \\
&\quad \text{where } \text{L}_1 = \text{DEFN} :: [\![\text{false} \vee \text{true} = \text{true}]\!] \\
&\qquad\ \ \text{L}_2 = \text{DEFN} :: [\![\text{true} \geq \text{false} = \text{true}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{THEOREM}_5\ [\![\text{false}]\!]\ [\![\text{false}]\!] = \text{TRANS } (\text{SUBST L}_1 \text{ IN } x \to x \geq \text{false})\ \text{L}_2 \\
&\quad \text{where } \text{L}_1 = \text{DEFN} :: [\![\text{false} \vee \text{false} = \text{false}]\!] \\
&\qquad\ \ \text{L}_2 = \text{DEFN} :: [\![\text{false} \geq \text{false} = \text{true}]\!]
\end{aligned}$$

Since these terms do not contain any free variables or recursive reductions, we could also rewrite them into the following concise form:

$$\begin{aligned}
&\text{THEOREM}_5\ [\![\text{false}]\!]\ [\![\text{true}]\!] = \text{DEFN} :: [\![(\text{false} \vee \text{true}) \geq \text{false} = \text{true}]\!] \\
&\text{THEOREM}_5\ [\![\text{false}]\!]\ [\![\text{false}]\!] = \text{DEFN} :: [\![(\text{false} \vee \text{false}) \geq \text{false} = \text{true}]\!]
\end{aligned}$$

At this point, readers proficient in Haskell will rightly observe that we are still missing the final $[\![(\bot \vee b) \geq \bot]\!]$ proof case in the definition of "$\geq$", which, as luck has it, evaluates to "$\bot$" rather than the desired "true", so that the theorem fails miserably when applied to diverging argument values. Fortunately, we can easily reformulate THEOREM_5 in such a way that it will only ever concern itself with terminating operands. First, we must capture the precise termination criteria for boolean expressions, which can be easily achieved, for example, using the following property type:

$$\begin{aligned}
&\text{data WF}[\![\cdot]\!] :: \mathit{bool} \to \star \text{ where} \\
&\qquad \text{WF}_{\text{TRUE}}\ :: \text{WF}[\![\text{true}]\!] \\
&\qquad \text{WF}_{\text{FALSE}} :: \text{WF}[\![\text{false}]\!]
\end{aligned}$$

A boolean Haskell expression τ for which the property $\text{WF}(\tau)$ can be established is said to be *well-formed* or *valid*. Armed with this property definition, we can easily restrict THEOREM_5 to hold only for well-formed boolean terms as follows:

$$\text{THEOREM}_5' :: \forall\, a, b :: \mathit{bool} \Rightarrow \text{WF}(a) \to \text{WF}(b) \to [\![(a \vee b) \geq a = \text{true}]\!]$$

If we repeat the earlier proof of THEOREM_4 for $\text{THEOREM}_4'$, then the compiler's coverage inspection algorithm can comfortably validate the theorem as well-formed, since no diverging expression will ever come into play during its type checking. Observe that, in the above definition, both a and b represent inferred parameters, since their values can be easily reconstructed from the structures of $\text{WF}(a)$ and $\text{WF}(b)$, respectively. In fact, the system should have no trouble inferring their types, either, so that we can also rewrite the above theorem without explicit type annotations as follows:

$$\text{THEOREM}_5' :: \forall\, a, b \Rightarrow \text{WF}(a) \to \text{WF}(b) \to [\![(a \vee b) \geq a = \text{true}]\!]$$

Properties such as "WF" are commonplace and it would be rather cumbersome to construct them manually for every Haskell data type introduced into the program. Fortunately, it is generally possible to infer them automatically for arbitrary data types in a manner similar to the mechanical derivation of the "eq", "ord" and "show" class instances in the standard Haskell 98 compiler. In particular, in this work I assume that the compiler always supplies an implicit definition of "WF" for every Haskell data type T for which no such property has been specified directly in the source code. Such implicit definitions always include an appropriate axiom or property constructor "WF_C" for every data constructor C appearing in the definition of T. For example, given a type T with the following structure:

$$\begin{aligned}\text{data } T = {} & C_1\ T_{11}\ T_{12}\ ...\ T_{1i} \\ \mid {} & C_2\ T_{21}\ T_{22}\ ...\ T_{2j} \\ & \vdots \\ \mid {} & C_n\ T_{n2}\ T_{n2}\ ...\ T_{nk}\end{aligned}$$

the derived property "$\text{WF}[\![\cdot]\!] :: T \to \star$" produced implicitly by the compiler assumes the following schematic form:

$$\begin{aligned}&\text{data WF}[\![\cdot]\!] :: \forall\ T :: * \Rightarrow T \to \star \text{ where} \\ &\quad \text{WF}_{C_1} :: \forall\ x_1, x_2\ ...\ x_i \Rightarrow \text{WF}(x_1) \to \text{WF}(x_2) \to ... \to \text{WF}(x_i) \to \text{WF}[\![C_1\ x_1\ x_2\ ...\ x_i]\!] \\ &\quad \text{WF}_{C_2} :: \forall\ x_1, x_2\ ...\ x_j \Rightarrow \text{WF}(x_1) \to \text{WF}(x_2) \to ... \to \text{WF}(x_j) \to \text{WF}[\![C_2\ x_1\ x_2\ ...\ x_j]\!] \\ &\qquad \vdots \\ &\quad \text{WF}_{C_n} :: \forall\ x_1, x_2\ ...\ x_k \Rightarrow \text{WF}(x_1) \to \text{WF}(x_2) \to ... \to \text{WF}(x_k) \to \text{WF}[\![C_n\ x_1\ x_2\ ...\ x_k]\!]\end{aligned}$$

In the following presentation, I always decorate well-formedness variables with the circumflex accent such as $\hat{x}$ in order to distinguish them from the actual term being scrutinised. With the help of this final extension, $\text{THEOREM}'_5$ can be rephrased correctly as follows:

$$\text{THEOREM}''_5 :: \forall\ a, b :: \mathit{bool} \Rightarrow \text{WF}(a) \to \text{WF}(b) \to [\![(a \vee b) \geq a = \text{true}]\!]$$

$$\text{THEOREM}''_5\ [\![\text{WF}[\![\text{true}]\!]]\!]\ [\![\hat{b}]\!] = \text{L}_4$$

$$\begin{aligned}&\text{where } b :: \mathit{bool} \\ &\qquad \hat{b} :: \text{WF}(b) \\ &\qquad \text{L}_1 = \text{DEFN} &&:: [\![\text{true} \vee b = \text{true}]\!] \\ &\qquad \text{L}_2 = \text{SUBST L}_1\ \text{IN}\ x \to x \geq \text{true} &&:: [\![(\text{true} \vee b) \geq \text{true} = \text{true} \geq \text{true}]\!] \\ &\qquad \text{L}_3 = \text{DEFN} &&:: [\![\text{true} \geq \text{true} = \text{true}]\!] \\ &\qquad \text{L}_4 = \text{TRANS L}_2\ \text{L}_3 &&:: [\![(\text{true} \vee b) \geq \text{true}]\!]\end{aligned}$$

$$\begin{aligned}&\text{THEOREM}''_5\ [\![\text{WF}_{(\text{false})}]\!]\ [\![\text{WF}_{(\text{true})}]\!] = \text{DEFN} :: [\![(\text{false} \vee \text{true}) \geq \text{false} = \text{true}]\!] \\ &\text{THEOREM}''_5\ [\![\text{WF}_{(\text{false})}]\!]\ [\![\text{WF}_{(\text{false})}]\!] = \text{DEFN} :: [\![(\text{false} \vee \text{false}) \geq \text{false} = \text{true}]\!]\end{aligned}$$

in which the binding signatures "$b :: \mathit{bool}$" and "$\hat{b} :: \text{WF}(b)$" are used to scrutinise the proof argument $\hat{b}$, binding the variable b to the actual boolean value whose well-formedness is asserted by $\hat{b}$.

Occasionally, effective specification of a program's behaviour mandates formulation of well-formedness in terms of a boolean predicate or a partially-computable

Haskell function of type "$T \rightarrow bool$" rather than as an inductive property of kind "$T \rightarrow \star$". Fortunately, such functions can be readily reused within a theorem by asserting their definitional equality to the constant "true". For example, a theorem that holds for all integers n greater than 7 can be paraphrased by the property type "$\forall\, n \Rightarrow [\![n > 7 = \text{true}]\!] \rightarrow ...$". In fact, in this work definitional equalities of the form "$[\![P(x) = \text{true}]\!]$" occur so frequently that, in most cases, I drop the "= true" portion of these types, so that the property application $[\![n > 7 = \text{true}]\!]$ will be often rendered simply as $[\![n > 7]\!]$ whenever the context of the discussion makes any confusion of meaning unlikely.

3.7 Inductive Proofs

It turns out that the approach described in Section 3.6 can be also used to reason about inductive definitions. As an example, let us prove a well-known result which states that the combined length of two finite lists is equal to the sum of their individual lengths. In other words, we seek a proof of the following theorem:

$$\begin{aligned}&\text{THEOREM}_6 :: \forall\, T :: *,\, \ell_1,\, \ell_2 :: [T] \Rightarrow \\ &\quad \text{WF}(\ell_1) \rightarrow \text{WF}(\ell_2) \rightarrow [\![\text{length}(\ell_1 +\!\!+ \ell_2) = \text{length}(\ell_1) + \text{length}(\ell_2)]\!]\end{aligned}$$

To begin with, let us deal with the case in which the first list argument is empty:

$$\text{THEOREM}_6\ [\![\text{WF}_{(\emptyset)}]\!]\ [\![\hat{\ell}_2]\!] = \text{L}_8$$

where $T :: *$
$\ell_2 :: [T]$
$\hat{\ell}_2 :: \text{WF}(\ell_2)$

$$\begin{aligned}
\text{L}_1 &= \text{DEFN} && :: [\![\emptyset +\!\!+ \ell_2 = \ell_2]\!] \\
\text{L}_2 &= \text{SUBST L}_1 \text{ IN } \ell \rightarrow \text{length}(\ell) && :: [\![\text{length}(\emptyset +\!\!+ \ell_2) = \text{length}(\ell_2)]\!] \\
\text{L}_3 &= \text{DEFN} && :: [\![\text{length}(\emptyset) = 0]\!] \\
\text{L}_4 &= \text{SUBST L}_3 \text{ IN } n \rightarrow n + \text{length}(\ell_2) && :: [\![\text{length}(\emptyset) + \text{length}(\ell_2) = 0 + \text{length}(\ell_2)]\!] \\
\text{L}_5 &= \text{ADD ZERO } [\![\text{length}(\ell_2)]\!] && :: [\![0 + \text{length}(\ell_2) = \text{length}(\ell_2)]\!] \\
\text{L}_6 &= \text{TRANS L}_4\ \text{L}_5 && :: [\![\text{length}(\emptyset) + \text{length}(\ell_2) = \text{length}(\ell_2)]\!] \\
\text{L}_7 &= \text{REFL L}_6 && :: [\![\text{length}(\ell_2) = \text{length}\emptyset + \text{length}(\ell_2)]\!] \\
\text{L}_8 &= \text{TRANS L}_2\ \text{L}_7 && :: [\![\text{length}(\emptyset +\!\!+ \ell_2) = \text{length}\emptyset + \text{length}(\ell_2)]\!]
\end{aligned}$$

in which the fifth lemma L_5 invokes the following well-known axiom of integer arithmetic:

$$\text{ADD ZERO } [\![\cdot]\!] :: \forall\, n :: \textit{integer} \rightarrow [\![0 + n = n]\!]$$

Similarly, the well-known transitivity of integer addition can be captured in the standard Haskell prelude by the following axiom:

$$\text{ADD TRANS } [\![\cdot]\!] :: \forall\, n_1,\, n_2,\, n_3 :: \textit{integer} \rightarrow [\![n_1 + (n_2 + n_3) = (n_1 + n_2) + n_3]\!]$$

Since the foundations of mathematics are not the subject of this work, in the subsequent proofs I will generally assume that such trivial axioms are either attainable implicitly through the theorem-proving ingenuity of our type checker, or else readily available as

part of the standard Haskell libraries and, in both cases, represent them within more interesting proofs using the single keyword "TRIV".

The second case of the THEOREM_6 proof deals with non-empty lists:

$$\text{THEOREM}_6\ [\![\text{WF}_{(:)}\hat{x}\ \hat{\ell}_1]\!]\ [\![\hat{\ell}_2]\!] = \text{L}_{11}$$

where
$$\begin{aligned}
T &:: *\\
x &:: T\\
\hat{x} &:: \text{WF}(x)\\
\ell_1, \ell_2 &:: [T]\\
\hat{\ell}_1 &:: \text{WF}(\ell_1)\\
\hat{\ell}_2 &:: \text{WF}(\ell_2)
\end{aligned}$$

$$\begin{aligned}
\text{L}_1 &= \text{DEFN} && :: [\![(x{:}\ell_1) +\!\!+ \ell_2 = x{:}(\ell_1 +\!\!+ \ell_2)]\!]\\
\text{L}_2 &= \text{DEFN} && :: [\![\text{length}(x{:}(\ell_1 +\!\!+ \ell_2)) = 1 + \text{length}(\ell_1 +\!\!+ \ell_2)]\!]\\
\text{L}_3 &= \text{SUBST L}_1\ \text{IN}\ \ell \to \text{length}(\ell) && :: [\![\text{length}((x{:}\ell_1) +\!\!+ \ell_2) = \text{length}(x{:}(\ell_1 +\!\!+ \ell_2))]\!]\\
\text{L}_4 &= \text{TRANS L}_3\ \text{L}_2 && :: [\![\text{length}((x{:}\ell_1) +\!\!+ \ell_2) = 1 + \text{length}(\ell_1 +\!\!+ \ell_2)]\!]\\
\text{L}_5 &= \text{THEOREM}_6\ \hat{\ell}_1\ \hat{\ell}_2 && :: [\![\text{length}(\ell_1 +\!\!+ \ell_2) = \text{length}(\ell_1) + \text{length}(\ell_2)]\!]\\
\text{L}_6 &= \text{SUBST L}_5\ \text{IN}\ n \to 1 + n && \\
& && :: [\![1 + \text{length}(\ell_1 +\!\!+ \ell_2) = 1 + (\text{length}(\ell_1) + \text{length}(\ell_2))]\!]\\
\text{L}_7 &= \text{TRIV} && :: [\![1 + (\text{length}(\ell_1) + \text{length}(\ell_2)) = (1 + \text{length}(\ell_1)) + \text{length}(\ell_2)]\!]\\
\text{L}_8 &= \text{DEFN} && :: [\![\text{length}(x{:}\ell_1) = 1 + \text{length}(\ell_2)]\!]\\
\text{L}_9 &= \text{REFL L}_8 && :: [\![1 + \text{length}(\ell_2) = \text{length}(x{:}\ell_1)]\!]\\
\text{L}_{10} &= \text{SUBST L}_{10}\ \text{in}\ n \to n + \text{length}(\ell_2) && \\
& && :: [\![(1 + \text{length}(\ell_1)) + \text{length}(\ell_2) = \text{length}(x{:}\ell_1) + \text{length}(\ell_2)]\!]\\
\text{L}_{11} &= \text{TRANS TRANS TRANS L}_4\ \text{L}_6\ \text{L}_7\ \text{L}_{10} && \\
& && :: [\![\text{length}((x{:}\ell_1) +\!\!+ \ell_2) = \text{length}(x{:}\ell_1) + \text{length}(\ell_2)]\!]
\end{aligned}$$

in which the fifth lemma L_5 invokes the induction principle and the entire proof is established from transitivity of the lemmas L_4, L_6, L_7 and L_{10}, while the "TRIV" proof of L_7 bears witness to the well-known transitivity axiom of integer arithmetic. □

3.8 Implementation

Since automated theorem proving is not the subject of this work, Sections 3.5, 3.6 and 3.7 provide only a brief outline of the required extensions to the Haskell type system. It is my strong belief that these extensions could be readily implemented as a very capable proof assistant environment, although I leave such implementation for future work, since it would provide an undue distraction from the primary topic of this thesis. In principle, the extensions to the Haskell type system described in the last three sections could be effected through a translation of Haskell programs into the *calculus of inductive constructions* (CIC) similar to that employed in the highly-successful proof development environment Coq [INRIA 02]. As argued by Coquand, it is possible to admit non-termination at the term level of the calculus without loss of soundness [Coquand 86]. However, in order to deliver such an implementation, many details of the design which have been glossed over in the above discussion must be worked out. For example, the precise criteria for establishment of the DEFN and TRIV proofs must be determined, together with a satisfactory treatment of diverging

expressions. Finally, soundness of the entire theory must be established formally, which, as the past work on automated theorem proving has shown, is a rather non-trivial undertaking.

This final challenge also requires development of a formal semantic model for the entire Haskell language. So far, the glaring absence of such a model was largely compensated by the language's closeness to System F, whose own well-understood semantic interpretation has served the community well for almost two decades [Hudak 07]. However, if an application of Haskell as a program specification and verification framework is to be taken seriously, the entire language must be given a firm formal footing.

In the meantime, all logical deductions presented in this work should be considered to represent "paper proofs" and accepted with a degree of caution appropriate for such reasoning. Nevertheless, it is my hope that, one day, the extensions described above will find their way into an actual Haskell compiler, completing the rigorous proof of compiler correctness presented in the following chapters.

4

LAMBDA CALCULUS

Entia non sunt multiplicanda praeter necessitatem
(Entities should not be multiplied beyond necessity)
— William Ockham

The entire feasibility of linear correctness rests on our ability to find a program representation suitable for simultaneous use as both an intermediate depiction of computations within the compiler and also as a semantic notation in the formal specification of its source and target languages. It is perhaps surprising that a such a construction exists at all, let alone that it is readily available in the form of the *untyped lambda calculus* [Church 36, Church 41], which, as will be demonstrated in Chapters 5 and 6, captures successfully the meanings of typical source and target programming languages and, further, carries the benefits of an extensive body of knowledge about its properties accumulated over the past seventy years since its introduction.

Lambda calculus was originally studied in the context of the foundational work on computability and the nature of mathematics. It made its debut in 1936 in a seminal paper by Alonzo Church, in which Church uses the calculus to provide the famous negative solution to the *Entscheidungsproblem* [Church 36]. In an effort to ensure the system's consistency, Church extended the calculus with a *type system* in 1940 [Church 40] and conducted an extensive study of its properties in his book on *Calculi of Lambda Conversion* published in the following year [Church 41]. In 1943, Kleene demonstrated an equivalence between lambda calculus and the Turing machine [Turing 36], a result that is now known as the Church-Turing thesis [Kleene 43].

The calculus was first employed in a design of a practical programming language by John McCarthy as part of his research on artificial intelligence [McCarthy 58]. Throughout the 1950's, McCarthy gradually developed LISP [McCarthy 60]. This effort culminated in 1959 in the famous presentation of the *eval* function which defined the operational semantics of LISP [McCarthy 59] and, to McCarthy's uneasy surprise, could be implemented in LISP itself [Smith 84]. The first such complete LISP system, executed entirely in LISP, was delivered in 1962 by Timothy Hart and Michael Levin [McCarthy 62]. LISP was also famously used to implement the original FORTRAN-77 compiler for UNIX [Pitman 79], which arguably constitutes the first practical encoding of an imperative language in a declarative system. More recently, the language has been reformulated as Scheme and standardised by the Institute of Electrical and Electronics Engineers [IEEE 91]. In this new form, LISP has survived the test of time and remains in active use to this day.

The correspondence between lambda calculus and imperative programming languages was first established in 1963 by Landin, who made the observation for ALGOL [Landin 63]. Unfortunately, Landin's result remained, for the most part, unexplored in the research community for over thirty years. In 1991, Cytron *et al.* presented a highly influential algorithm for translation of imperative programs into their *static single assignment form* (SSA) [Cytron 91], which was subsequently shown to be equivalent to lambda calculus by Appel [Appel 98, Appel 98ML]. In 1995, Kelsey established the formal equivalence between SSA and the *continuation passing style* of lambda calculus through bidirectional translation between the two forms [Kelsey 95]. In our own prior work on the subject, we have presented a translation from SSA to the *administrative normal form* of lambda calculus and argued for the use of purely-functional representations in optimising compilers for imperative languages [Chakravarty 03]. Moggi's work on monadic semantics of programming languages [Moggi 91] and Danvy's research on the *monadic normal form* of programs [Hatcliff 94, Danvy 03] also contain material related to the subject. However, a direct transformation of a complete imperative programming language in common use into lambda calculus has not been attempted before and is a new contribution of the present work.

4.1 Lambda Calculus

In its simplest form, lambda calculus consists of only three syntactic constructs: *variables*, *applications* and *lambda abstractions*. This form of the calculus, known as the *pure lambda calculus*, can be described using the following BNF grammar:

pure-term:	
pure-variable	*(variables)*
pure-term pure-term	*(application expressions)*
λ *pure-variable . pure-term*	*(lambda abstractions)*

As discussed in Chapter 3, the grammar presented above is actually implemented as a definition of the Haskell data type "PureTermT" with constructors "PureVar" and "PureApp" whose names are suppressed from the presentation for the sake of clarity and "PureLambda", which is typeset simply as "λ". The reader is advised to refer to Chapter 3 for a further discussion of the precise rules governing typesetting of all Haskell material in this work and Appendix D for an exhaustive list of all Haskell types, data constructors and functions mentioned in the presentation, cross-indexed with the page numbers of their respective definitions.

We shall refer to all expressions of this kind as *lambda expressions* or *terms*. Simple lambda expressions consisting of a single identifier are called *variables*. Terms of the form "$\lambda x.\tau$" are known as *lambda abstractions*. Intuitively, they represent a function or a procedure with a single formal parameter x and a body τ which computes the value returned by the function and may contain references to x. Functions of multiple parameters may be constructed using nested lambda abstractions. For example, the term "$\lambda x.\lambda y.\tau$" represents a function of two variables x and y. Finally, terms of the

form "$\tau_1\,\tau_2$" are known as *application expressions* or just *applications*. In such expressions, the first term (which represents an actual function) is said to be *applied* to the second one, which depicts the value of that function's operand. Intuitively, an application of the form "$\tau_1\,\tau_2$" represents a call to the function τ_1 with the argument value specified by τ_2. In particular, a term of the form "$(\lambda x.\tau_1)\tau_2$" is considered equivalent to the result of replacing every occurrence of x in τ_1 with the expression τ_2. For the sake of exposition, application expressions are taken to be right-associative and have a higher precedence than lambda abstractions. In other words, "$\tau_1\,\tau_2\,\tau_3$" is always taken to mean "$(\tau_1\,\tau_2)\tau_3$" rather than "$\tau_1(\tau_2\,\tau_3)$" and "$\lambda x.\tau_1\,\tau_2$" is taken to mean "$\lambda x.(\tau_1\,\tau_2)$" rather than "$(\lambda x.\tau_1)\tau_2$".

Variables, lambda abstractions and application expressions are the only three syntactic elements of the pure lambda calculus; it is a remarkable consequence of the Church-Turing thesis [Kleene 43] that all interesting programs (and many others that are not interesting at all) may be constructed from these three symbolic patterns alone. The set of all computations that are expressible in lambda calculus is precisely that of *computable functions*, that is, functions that may be evaluated by a physical device.

It should be observed that not every conceivable function is computable. For example, in 1936 Church and Turing have independently shown that the *halting problem*, which poses the question of whether any given function terminates, can never be solved in its full generality [Church 36, Turing 36]. Many unsolvable problems, are, however, *partially computable*, in the sense that it is possible to devise a function that either returns the desired property or does not terminate. In the later case, it is usually possible to further categorise the argument values leading to such non-termination. For example, it is quite possible to devise a function that, given an arbitrary lambda expression, returns "1" whenever that expression terminates and fails to terminate itself for all non-terminating input expressions. We will often find it necessary to work with such partially-computable functions when designing a compiler. In particular, every attempt to express a semantics of a lambda calculus as a function that reduces its terms to their values, is by the very nature of computability, doomed to result in a partially-computable construction.

The minimality of lambda calculus is certainly very appealing, since, with only three language constructs, it renders itself readily to detailed scrutiny. Although, in this work, I will refrain from defining a complete formal semantic model of the pure calculus presented above, such semantics are readily available in literature [Barendregt 84] and structures similar to "*pure-term*" continue to prevail as the authoritative representation of computability. Accordingly, a suitable variant of lambda calculus seems to be a highly attractive choice for an intermediate program representation in a linearly-correct compiler. In the remainder of this chapter, I first present a few examples of useful lambda constructions, followed by two simple variants of their operational meanings. The actual Etude program representation utilised in our compiler verification project is then defined in Sections 4.4–4.7, complete with a formal specification of a suitable al-

gebraic semantics for the language. Although Etude represents a substantial departure from the simplicity of its pure cousin, it is the purity and deeply-rooted minimality of Church's invention which makes it possible to verify an entire C compiler within the space of a single manuscript.

4.1.1 Examples

Since, in lambda calculus, abstractions and their applications are the only building blocks available for assembly of the plethora of common programming idioms that occur routinely throughout computer science, every such idiom must, somehow, be formulated as an appropriate function, whose arguments are themselves functions. This minimality may be appealing to a mathematician but can easily prove intimidating to a programmer accustomed to dealing with more intuitive operational constructs such as integers, conditionals and loops.

Fortunately, all such constructs may be readily formulated in lambda calculus as *combinators*, or functions that modify other functions, so that the user is not burdened with inventing a new lambda abstraction for every programming task required in the course of a software design project. As the first example of such combinators, let us consider the two familiar boolean constants "true" and "false", which, in the pure lambda calculus, are usually defined as follows:

$$\text{pure-true} = \lambda x.\lambda y.x$$
$$\text{pure-false} = \lambda x.\lambda y.y$$

Intuitively, both "pure-true" and "pure-false" represent *binary functions* of two arguments x and y, such that "pure-true" always returns the first argument x while "pure-false" returns the second of its operands y. In this way, the two constants allow us to select between two arbitrary lambda expressions, capturing the notion of *choice* central to the concept of a boolean constant. Accordingly, the familiar conditional construct "if p then x else y" may be defined in the pure lambda calculus as the following *ternary function* of three arguments:

$$\text{pure-if} = \lambda p.\lambda x.\lambda y.p\ x\ y$$

The expression "pure-if $p\ f_1\ f_2$" evaluates to f_1 if p is equal to "pure-true" and f_2 if p is "pure-false", by applying its x and y operands to the condition argument p. To further convince ourselves that "pure-true" and "pure-false" do, in fact, represent boolean constants, let us define the three common boolean operations "$\neg$", "$\wedge$" and "$\vee$":

$$\text{pure-not} = \lambda p.\lambda x.\lambda y.p\ y\ x$$
$$\text{pure-and} = \lambda p.\lambda q.\lambda x.\lambda y.p\ (q\ x\ y)\ y$$
$$\text{pure-or} = \lambda p.\lambda q.\lambda x.\lambda y.p\ x\ (q\ x\ y)$$

An application of the "pure-not" combinator to a boolean constant (or binary function) p constructs a new binary function which applies x and y to p in reverse order, effectively inverting its selection process. "pure-and $p\ q$" constructs a binary function that uses q to choose between x and y and then applies p to choose between the result of q and y as the second alternative, effectively implementing the decision process

"if p then (if q then x else y) else y" which, as every programmer knows, captures the notion of a logical conjunction. Similarly, "pure-or p q" implements the decision process "if p then x else (if q then x else y)" representing logical disjunction of p and q.

The constants "pure-true" and "pure-false" were easy to define due to the finite nature of boolean algebra. A more interesting example of combinators is introduced by the concept of natural numbers 0, 1, 2 and so on. Readers may be surprised that such infinite sets can be expressed in the pure lambda calculus, well, naturally:

$$\begin{aligned}
\text{pure-zero} &= \lambda f.\lambda x.x \\
\text{pure-one} &= \lambda f.\lambda x.f\ x \\
\text{pure-two} &= \lambda f.\lambda x.f\ (f\ x) \\
\text{pure-three} &= \lambda f.\lambda x.f\ (f\ (f\ x)) \\
\text{pure-four} &= \lambda f.\lambda x.f\ (f\ (f\ (f\ x)))
\end{aligned}$$

and so on, until every number required by the program has been defined. These lambda expressions are known as *Church numerals*; the number of fs in the body of each numeral encodes its numeric value. Just as the boolean constants "pure-true" and "pure-false" captured the primitive notion of choice, every Church numeral "pure-n" encodes the primitive notion of *counting* successive applications of a function to its argument. Such chained application of the result of one function g directly to another function f is known as *function composition*, commonly written as "$f \circ g$". Because composition can be considered as the multiplication operator of the world of lambda abstractions, the composition "$f \circ f$" of a function f with itself is traditionally written as f^2, the composition "$f \circ f \circ f$" as f^3 and so on. Using this notation, every Church numeral n can be formulated as a combinator which constructs the n-fold composition of its argument function f^n. To see why Church numerals represent natural numbers, let us try to define the common arithmetic operations on these objects:

$$\begin{aligned}
\text{pure-add} &= \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x) \\
\text{pure-mul} &= \lambda m.\lambda n.\lambda f.m\ (n\ f) \\
\text{pure-exp} &= \lambda m.\lambda n.n\ m
\end{aligned}$$

The lambda abstractions "pure-add", "pure-mul" and "pure-exp" implement the arithmetic operations $m + n$, $m \times n$ and m^n, respectively. Notice how the composition of Church numerals implements multiplication of the corresponding natural numbers, making our definitions of "pure-mul" and "pure-exp" trivial. Although superficially more complicated, the above definition of "pure-add" is also quite intuitive: we simply concatenate the two application sequences by inserting the body of n at the end of the chain in m.

Composite objects such as pairs are also representable as lambda combinators. For example, the notion of a pair of lambda terms is conveniently captured as follows:

$$\begin{aligned}
\text{pure-pair} &= \lambda x.\lambda y.\lambda f.f\ x\ y \\
\text{pure-first} &= \lambda z.z\ (\lambda x.\lambda y.x) \\
\text{pure-second} &= \lambda z.z\ (\lambda x.\lambda y.y)
\end{aligned}$$

The expression "pure-pair τ_1 τ_2" produces a combinator that, when, applied to a binary function f, makes the two individual components of the pair available to f as its two operand values. The expressions "pure-first τ" and "pure-second τ" extract these component from the pair τ by applying it to a binary function which returns its first or second parameter, respectively.

Many more common and powerful combinators may be defined using the similar approach of capturing the essence of the underlying concept. However, by now, I hope that the reader has gained sufficient intuition for the behaviour of lambda abstractions, so that, without further ado, we can proceed to formalising the general meaning of lambda expressions.

4.1.2 Free Variables

A lambda abstraction of the form "$\lambda x.\tau$" is said to *bind* all occurrences of the variable x in τ. If a variable is not bound by any lambda abstraction within the expression in which it appears, it is known as a *free variable* and the corresponding expression is said to be an *open expression*. Conversely, an expression that does not contain any free variables is said to be *closed*. The set of all variables appearing free in a lambda expression can be constructed as follows:

$$\mathrm{FV}[\![\cdot]\!] :: \textit{pure-term} \to \{\textit{pure-variable}\}$$

$$\begin{aligned}\mathrm{FV}[\![x]\!] &= \{x\}\\ \mathrm{FV}[\![\lambda x.\tau]\!] &= \mathrm{FV}(\tau)\backslash\{x\}\\ \mathrm{FV}[\![\tau_1\ \tau_2]\!] &= \mathrm{FV}(\tau_1) \cup \mathrm{FV}(\tau_2)\end{aligned}$$

where the notation "$\{$*pure-variable*$\}$" appearing in the signature of "FV" represents the type of all identifier sets. The notation "$\{x\}$" constructs a singleton set consisting of x alone, while the operators "$\cup$" and "$\backslash$" return the union and difference of two sets, respectively. All of these definitions are taken directly from the standard Haskell libraries discussed in Appendix A and are typeset through an appropriate application of the formatting rules described in Chapter 3.

4.1.3 Capture-Avoiding Substitution

Before turning our attention to a semantic study of the pure lambda calculus, we must also formalise the notion of a *capture-avoiding substitution*. Intuitively, a substitution of the form "$\tau_1/\{x{:}\tau_2\}$" systematically replaces all free occurrences of the variable x in τ_1 with the term τ_2. Note that any bound occurrences of x in τ_2 are not replaced, so that the result of substituting τ for x in the expression "$(\lambda x.x)x$" is "$(\lambda x.x)\tau$" rather than "$(\lambda x.\tau)\tau$". Capture-avoiding substitution can be defined in Haskell as a function with the following type signature:

$$[\![\cdot]\!] \,/\, \{[\![\cdot]\!]\} :: \textit{pure-term} \to (\textit{pure-variable}{:}\textit{pure-term}) \to \textit{pure-term}$$

Barring certain cumbersome technicalities, a typical implementation of this function

assumes the following inductive form:

$$\begin{array}{lll}
[\![x]\!] \quad / \{[\![x':\tau']\!]\} & \mid (x = x') & = \tau' \\
 & \mid (x \neq x') & = [\![x]\!] \\
[\![\lambda x.\tau]\!] \ / \{[\![x':\tau']\!]\} & \mid (x = x') & = [\![\lambda y.\tau']\!] \\
 & \mid (x \notin \mathrm{FV}(\tau')) & = [\![\lambda x.[\![\tau / \{x':\tau'\}]\!]]\!] \\
[\![\tau_1\ \tau_2]\!] \ / \{[\![x':\tau']\!]\} & & = [\![[\![\tau_1 / \{x':\tau'\}]\!]\ [\![\tau_2 / \{x':\tau'\}]\!]]\!]
\end{array}$$

in which the notation "$\{a{:}b\}$" stands simply for a fancy typesetting of an ordinary Haskell pair "(a, b)". Appendix A introduces a number of such presentation forms, whose only purpose is to improve exposition of certain important constructs such as variable substitutions, in an attempt to bridge the gap between established mathematical notations and their rendition in the Haskell language. In the actual source code, all of these constructs materialise simply as pairs of ordinary parentheses symbols "()".

A careful reader will observe that the above formulation of substitution represents a partial function, undefined whenever the expression being substituted contains a free occurrence of a variable bound within the expression being modified. Since the terms being substituted into an expression are usually closed, this restriction is usually irrelevant in practice and the above definition of capture-avoiding substitution proves sufficient for the purpose of the present discussion. Nevertheless, the extended Etude calculus defined later in this chapter goes to some lengths in order to lift the restriction for the actual intermediate program representation deployed in the design of our C compiler.

4.1.4 Operational Semantics

A lambda expression of the form "$(\lambda x.\tau_1)\tau_2$" is known as a *beta redex*. It represents an application of the term τ_2 to the function "$\lambda x.\tau_1$". Such expressions are equivalent to the result of evaluating the function, formalised by the substitution of τ_2 for every free occurrence of x in τ_1. This form of substitution corresponds closely to the intuitive notion of calling a function in an imperative programming language, since it renders an application of a function to a particular argument value equivalent to an instance of the function's body with all free occurrences of the formal parameter instantiated to the supplied expression. The process of replacing a beta redex with the corresponding substituted expression is known as *beta reduction*. If we apply beta reduction systematically to eliminate all redexes from an expression, we arrive at the expression's *beta-normal form* which may be viewed as a outcome of *evaluating* that expression. It is a remarkable result known as the *Church-Rosser property* that every lambda expression has at most one beta-normal form. This theorem, proven for the pure lambda calculus in 1936 by Church and Rosser [Rosser 36], allows us to view the beta-normal form of an expression as its *value* and renders beta reduction an attractive basis for an operational interpretation of the pure lambda calculus.

We can formulate an operational semantics of the pure lambda calculus as a function $\mathcal{E}$ which takes lambda expressions to their fully-evaluated beta-normal forms. Observe that multiple implementations of $\mathcal{E}$ are possible, corresponding to the different

viable orders in which beta redexes can be eliminated from some lambda expressions. Perhaps the most obvious evaluation strategy, known as an *applicative order evaluation*, is to follow the intuitive understanding of the function call mechanism, according to which arguments are always substituted into function bodies in their fully-reduced beta-normal forms. In Haskell, this can be specified as follows:

$$\mathcal{E}_{\mathsf{A}}[\![\cdot]\!],\ \mathcal{E}'_{\mathsf{A}}[\![\cdot]\!] :: \textit{pure-term} \rightarrow \textit{pure-term}$$

$$\begin{array}{ll} \mathcal{E}_{\mathsf{A}}\,[\![x]\!] & = [\![x]\!] \\ \mathcal{E}_{\mathsf{A}}\,[\![\lambda x.\tau]\!] & = [\![\lambda x.[\![\mathcal{E}_{\mathsf{A}}(\tau)]\!]]\!] \\ \mathcal{E}_{\mathsf{A}}\,[\![\tau_1\ \tau_2]\!] & = [\![[\![\mathcal{E}'_{\mathsf{A}}(\mathcal{E}_{\mathsf{A}}(\tau_1))]\!]\ [\![\mathcal{E}_{\mathsf{A}}(\tau_2)]\!]]\!] \end{array}$$

where the auxiliary function $\mathcal{E}'_{\mathsf{A}}$ performs the actual beta reduction of an applicative term formed from a pair of lambda expressions already reduced into their respective normal forms:

$$\begin{array}{ll} \mathcal{E}'_{\mathsf{A}}[\![(\lambda x.\tau_1)\ \tau_2]\!] & = \mathcal{E}_{\mathsf{A}}(\tau_1/\{x{:}\tau_2\}) \\ \mathcal{E}'_{\mathsf{A}}[\![\tau_1\ \tau_2]\!] & = [\![\tau_1\ \tau_2]\!] \end{array}$$

It can be shown that, for all closed lambda expressions τ, if $\mathcal{E}_{\mathsf{A}}(\tau)$ terminates, then it produces a beta-normal form, although the proof of this simple theorem is rather involved in practice, due to various technicalities mentioned in our earlier discussion of the capture avoiding substitution operator from Section 4.1.3. Accordingly, I leave verification of this theorem as an exercise for a keen reader.

However, it should be noted that $\mathcal{E}_{\mathsf{A}}$ cannot construct the beta-normal form of every possible lambda expression. Consider, for example, the expression "$(\lambda x.x)(\lambda x.x)$". No matter how hard we try, we cannot eliminate all beta redexes from this expression, since reducing one redex through beta conversion merely gives rise to another one in the reduced expression. In fact, this expression remains unchanged by beta reduction! Such lambda terms are said to be *diverging* and are distinguished by the equivalence of "$\tau_1\,\tau_2$" with the expression τ_1 itself for every argument value τ_2. They are usually represented collectively by the symbol "$\perp$". It would be tempting to disregard diverging expressions as irrelevant and forbid them from ever appearing in lambda terms, but, unfortunately, it turns out that this cannot be done without unduly limiting the computational power of the calculus. In particular, observe that diverging expressions may be passed safely as arguments to some other expressions; for example, the beta-normal form of "$(\lambda x.y)\tau$" is equivalent to the variable y for all lambda terms τ, including the diverging ones. Unfortunately, $\mathcal{E}_{\mathsf{A}}$ fails to construct the beta-normal form of "$(\lambda x.y)((\lambda x.x)(\lambda x.x))$", by attempting to obtain the non-existent beta-normal form of the term "$(\lambda x.x)(\lambda x.x)$". We say that such evaluation functions *diverge* when applied to diverging lambda terms.

4.1.5 Normal Order Evaluation

Fortunately, as shown by Rosser [Rosser 36], it is possible to reformulate $\mathcal{E}$ in such a way that it always finds the beta-normal form of every lambda expression for which the beta-normal form exists. Unsurprisingly, this strategy order is known as the *normal*

order evaluation and it can be modelled in Haskell as the following definition:

$$\mathcal{E}_{\mathrm{N}}[\![\cdot]\!],\ \mathcal{E}'_{\mathrm{N}}[\![\cdot]\!] :: \textit{pure-term} \rightarrow \textit{pure-term}$$

such that:

$$\begin{array}{lll}
\mathcal{E}_{\mathrm{N}}\,[\![x]\!] & = & [\![x]\!] \\
\mathcal{E}_{\mathrm{N}}\,[\![\lambda x.\tau]\!] & = & [\![\lambda x.[\![\mathcal{E}_{\mathrm{N}}(\tau)]\!]]\!] \\
\mathcal{E}_{\mathrm{N}}\,[\![\tau_1\ \tau_2]\!] & = & [\![[\![\mathcal{E}'_{\mathrm{N}}(\mathcal{E}_{\mathrm{N}}(\tau_1))]\!]\ \tau_2]\!] \\
 & & \\
\mathcal{E}'_{\mathrm{N}}[\![(\lambda x.\tau_1)\ \tau_2]\!] & = & \mathcal{E}_{\mathrm{N}}(\tau_1/\{x{:}\tau_2\}) \\
\mathcal{E}'_{\mathrm{N}}[\![\tau_1\ \tau_2]\!] & = & [\![\tau_1\ [\![\mathcal{E}_{\mathrm{N}}(\tau_2)]\!]]\!]
\end{array}$$

Intuitively, this evaluation strategy always reduces the *head* of an application expression before reducing the corresponding function argument. Note that, under the normal order evaluation, the argument expressions are only evaluated if they are actually referenced within the corresponding function bodies. However, normal order evaluation is not entirely about performance. One consequence of admitting this evaluation strategy in a formulation of an operational semantics for the pure lambda calculus is an ability to handle some important cases of diverging expressions. Perhaps the best known of all such constructs is the *fixed point combinator*, sometimes called the *Y combinator* following the nomenclature of Curry. This combinator can be defined in the pure lambda calculus as the following expression "pure-fix":

$$\text{pure-fix} = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

The "pure-fix" function was initially discovered by Moses Schönfinkel [Schönfinkel 24, Bauer-Mengelberg 67] and rediscovered by Haskell Curry after Schönfinkel's work was culled by Stalin's imperialism and the outbreak of World War II [Curry 58, Curry 72]. It takes a single argument f (which we assume to be a lambda abstraction) and constructs a new function that applies f to a somewhat-unintelligible argument with a structure similar to the diverging expression "$(\lambda x.x)(\lambda x.x)$" mentioned earlier, except that, in the fixed point combinator, the body x is replaced with an application expression "$f(xx)$". If we try to reduce the body of the constructed function "pure-fix f", we will end up with the original expression applied to f itself. In other words, "pure-fix" calls f with a an infinite sequence of applications "$f(f(f(f(...))))$", or, put differently, "pure-fix f" is equivalent to "f(pure-fix f)", since adding another f to an already-infinite sequence of fs is no different from the original infinite sequence.

At this point, the reader may wonder why such bizarre expression could ever be useful. We note that, since the argument of f is equivalent to "pure-fix f", the fixed point operator provides a means for the body of f to refer to itself. In other words, "pure-fix" allows us to define self-referencing or *recursive* lambda expressions. As mentioned earlier, such expressions may have normal forms (or *converge*) if, eventually, f is reduced to an abstraction that does not refer to its parameter. Extensive experience with lambda calculus has shown that recursive expressions are of an immense practical importance and that they are required by virtually every useful program ever written.

Nevertheless, for pragmatic reasons applicative order evaluation plays an important rôle in the study of an operational semantics of the pure lambda calculus. Since many common lambda abstractions such as "$\lambda x.(f\,x\,x)$" contain multiple references to their parameters, applying $\mathcal{E}_{\mathsf{N}}$ to such lambda terms, which substitutes an unreduced argument into the function body, results in a separate evaluation of that argument for its every occurrence within the lexical syntax of the program. As terms grow larger, the amount of additional work mandated by normal order evaluation grows exponentially with their complexity. Such operational semantics of the lambda calculus may insinuate that a perfectly-sensible lambda expression "$\lambda x.(f\,x\,x)$" is prohibitively expensive to apply in a real program, while in fact its actual evaluation on a real computational system can be efficient and pleasant. In the interest of relevance, we cannot afford for our evaluation model to disperse such misinformation and, accordingly, we should endeavour to reflect the computational reality of the language within its specification with a greater degree of immediacy. The reader is assured that such formulation will be presented in Section 4.6 for the full-featured variant of the calculus deployed as the intermediate program representation in this work.

4.2 Extending the Calculus

The pure lambda calculus described in the previous section is seldom deployed directly in executable software. Most often, the syntax of the calculus is tailored to fit the specific conventions of a particular application area and its operational semantics is likewise tweaked to reflect the intended use of lambda terms. In particular, in the intermediate program representation deployed within our compiler, both the syntax and semantics of the pure lambda calculus have been adjusted to emphasise the correspondence between lambda terms and the actual computational facilities of a typical modern hardware architecture, while, at all times, taking care to preserve the *referential transparency* of the language, which dictates that every pair of lambda terms rendered equivalent by its semantics be interchangeable anywhere within a valid program. I call the resulting formalism "Etude" for its simplicity and elegance of form.

There are many reasons to extend the pure calculus from Section 4.1. Perhaps the least compelling is the desire to make expressions more readable. It is not at all obvious that the unwieldy term "$(\lambda m.\lambda n.\lambda f.\lambda x.(m\,f\,(n\,f\,x)))(\lambda f.\lambda x.(f\,x))(\lambda f.\lambda x.x)$" represents the addition "$1+0$", even though it has been constructed by GHCi from the Haskell expression "pure-add pure-one pure-zero" using the definitions of "pure-add", "pure-one" and "pure-zero" from Section 4.1.1. However, since, in this work, lambda terms are generally constructed and examined by a compiler rather than a human audience, such verbosity does not, by itself, pose problems sufficient to warrant a sacrifice of its minimality.

Nevertheless, reliance on combinators for representation of basic programming concepts has three important disadvantages, the first of which pertains to the accuracy of an operational semantics of the language. By hiding the distinction between

internal semantics of primitive objects such as integers and the external semantics of control-flow constructs such as function applications, the approach grossly misrepresents the true behaviour of programs. For example, the "pure-add" combinator from Section 4.1.1 performs addition of two Church numerals n and m in $O(n+m)$ individual reduction steps, while, on all modern computational hardware, the same addition would be performed in a constant time, regardless of the values of its operands. The second problematic aspect of the pure lambda calculus relates to the disparity between its representation of common programming constructs and the actual facilities available on a typical computational hardware, which renders many pure lambda terms resilient to an effective translation into a stream of efficient machine instructions as performed by the compiler's back end. In particular, one aspect of pure lambda terms that hinders their execution on modern computational hardware is the fact that, by definition, combinators essentially represent means of generating new functions during program execution, which is difficult to realise efficiently on most modern computers. Finally, since common interactive constructs such as the "printf" function in C do not represent computable functions, they cannot be defined sensibly in the pure lambda calculus without appropriate language extensions.

The most direct means of alleviating these problems is to extend the pure calculus with a set of various *constants* that capture the essence of the primitive computational facilities supported natively by typical computer architectures. For example, Church numerals could be represented by the familiar syntax "0", "1" and so on. To further provision for a smooth transition from Etude into the realm of actual machine instruction sequences described in Chapter 6, we also require every lambda abstraction in the program to be associated by the programmer with an explicitly-specified and globally-unique name, a technique that goes by the name of *defunctionalisation* in literature. Intuitively defunctionalisation translates higher-order functional program such as those devised in the pure lambda calculus from Section 4.1 into their first-order semantic equivalents [Reynolds 82, Danvy 03]. Nevertheless, it should be observed that Etude's approach to defunctionalisation is unusual in that it identifies functions with numeric quantities rather than abstract values and, as we shall discover shortly, such function names may be constructed dynamically by an Etude program using arbitrary arithmetic expressions, which adheres closely to the operational interpretation of these names as the functions' locations within the program's memory-resident image during its execution.

Such use of syntactic extensions allows for an effortless translation of Etude programs into a target language as described in Chapter 6 and, although the resulting language may, at the first glance, seem to bear little resemblance to the orthogonal syntax of the pure calculus, it nevertheless constitutes a purely-functional programming language that retains the essential referential transparency of the original calculus, which renders subsequent manipulation of programs efficient and pleasant, brining us closer the ultimate goal of a successful compiler verification.

Before proceeding with a detailed specification of this program representation, it should be noted that the definition of Etude is split into two largely independent parts, the first of which pertains to that fragment of the language which remains invariant across all possible target architectures, while the second tailors those portable semantics to the specific needs of a particular computational system. In Section 4.3, I outline the general mechanism by which that separation is maintained throughout the project.

At the syntactic level, the structure of Etude is split into the level of *atoms* discussed in Section 4.4, *terms*, which capture the monadic structure of computations whose syntax and portable algebraic semantics are described in Section 4.6 and, finally, *modules* from Section 4.7 that facilitate separate compilation of distinct program fragments. Further, the general issues and semantics of stateful computations are discussed in Section 4.5, while a more concrete operational treatment of the language is postponed until later in Chapter 6, where it is presented together with the balance of Etude's specification tailored to the particular needs of the targeted MMIX architecture [Knuth 05], as well as a rigorous formal proof of the linear correctness property for the entire compiler.

4.3 A Language with a Distinction

It is generally acknowledged that the design of any intermediate program representation should pay some attention to the language's representation of programs destined for execution on a broad range of different computational architectures, so that a compiler originally designed to generate instructions for one computer system may be easily retargeted for another without the need to rewrite large portions of its front end and optimisation algorithms. However, this issue is even more important in a compiler designed under the linear correctness regime, since, in this case, the intermediate program representation's rôle as a semantic calculus essentially precludes tweaking of that representation for individual target architectures. If such adjustments were to be permitted, a source language would have a different definition and meaning on every computer system, an effect that would certainly resonate with the user in an unpleasant fashion. Nevertheless, while the pure lambda calculus from Section 4.1 avoids any issues of portability through its high level of abstraction, in reality many architectural features such as the precise layout of its memory address space cannot be hidden from programs expressed in a typical low-level language such as C without an undue sacrifice of their performance.

The semiformal ANSI/ISO specification of the standard C language [ANSI 89] addresses these challenges by leaving portions of the semantic model *undefined*, *unspecified* or *implementation-defined* and, in general, open to further architecture-specific interpretation. Intuitively, such specifications describe not a single programming language but a whole family of languages, whose members differ in the specific choices of behaviours assigned to these portions of their definition. A similar approach is also pursued in this work, where the three forms of under-specification present in Standard C are modelled as follows:

① *Undefined behaviour* is described in the C Standard as "behaviour, upon use of a non-portable or erroneous program construct, of erroneous data or of indeterminately valued object, for which this International Standard imposes no requirements." A typical example of programs that invoke undefined behaviour are non-terminating C functions without side effects or those which attempt to access the value of a non-existent memory-resident object. In Chapter 5, the meaning of all C constructs whose behaviour is undefined in Standard C is modelled by translation into irreducible Etude terms, i.e., terms for which the evaluation function described later in Section 6.3 is itself undefined.

② *Implementation-defined behaviour* is specified by the ANSI/ISO committee as "behaviour, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document." Typical examples include the semantics of C bit arithmetic operators "**~**", "**&**", "**^**" and "**|**", or the range of numeric values representable exactly by a given scalar object. Like most other work on language semantics, this project models such behaviour with the help of numerous *language parameters*, whose precise values are explicitly excluded from the *generic fragment* of the specification. In particular, the generic fragment of Etude is contained entirely in the present chapter, while the generic fragment of C itself is defined later in Chapter 5. Every language parameter is modelled as a specific Haskell construct whose critical algebraic properties (but not the actual definition) are captured within its type signature and a set of theorems expressed in the dependant type system described earlier in Chapter 3. For Etude, the actual implementation-defined bindings of these parameters, appropriate for the MMIX architecture targeted by the compiler being described are included in Chapter 6, while, for C, a typical implementation of the most important language parameters is specified separately in Appendix C. In both cases, language parameters can assume the form of numeric constants, functions or even theorems as required for a successful rendition of the corresponding linguistic properties.

③ Finally, *unspecified behaviour* is defined in the C Standard as "behaviour, for a correct program construct and correct data, for which this International Standard explicitly imposes no requirements." A typical example includes the order in which the side effects resulting from evaluation of individual function arguments are effected in the program. Since, like implementation-defined behaviours, unspecified behaviours are characteristic of semantically-meaningful programs, the only difference between the two is in their documentation requirements. Although an explicit definition of any unspecified language parameters should be generally excluded from the compiler's description, some of these parameters are critical to a successful semantic treatment of the targeted MMIX architecture and, for that reason, many of them are given concrete definitions in Chapter 6 and distinguished from implementation-defined parameters only in the associated informal discussion.

4.4 Atoms and Their Formats

Perhaps the most pervasive of all language parameters are those which, collectively, abstract over the details of a specific arithmetic model adopted by a particular instruction set architecture. A correct level of abstraction in this model is as critical to a faithful specification of C as it is essential for definition of Etude's semantics itself. Accordingly, I begin the description of Etude with a detailed account of the facilities provided by the language for manipulation of numeric quantities, whose concrete implementation will be completed in Chapter 6 in a manner appropriate for the MMIX instruction set architecture.

In the pure lambda calculus from the previous section, the structure of lambda terms represented the only means of capturing the essence of all mathematical constructs. This is not so in Etude, in which all such objects are depicted by expression-like entities known as *atoms*, intuitively intended to designate simple scalar object such as variables and concrete integer values. To improve readability of the following formalisation of the C language, Etude atoms also admit simple algebraic operations on these objects such as negation "$-_\phi(\alpha)$" and multiplication "$\alpha_1 \times_\phi \alpha_2$". In particular, the abstract structure of all atoms is captured by the following set of four Haskell data type definitions:

atom$_{[\![v]\!]}$*:*

v	*(variable atoms)*
# *rational* $_{format}$	*(constant atoms)*
unary-op $_{format}$ *atom*$_v$	*(unary operations)*
atom$_v$ *binary-op* $_{format}$ *atom*$_v$	*(binary operations)*

atoms$_{[\![v]\!]}$*:* *(lists of atoms)*

[*atom*$_v$]

unary-op: one of

$-$ $\sim$ *format*

binary-op: one of

$+$ $-$ $\times$ $\div$ $\cdot|\cdot$ $\triangle$ $\bar{\triangle}$ ∇ $\ll$ $\gg$ $=$ $\neq$ $<$ $>$ $\leq$ $\geq$

In other words, the syntax of atoms includes variables, rational numbers and a handful of predefined arithmetic operations such as "$-_\phi(\alpha)$" and "$\alpha_1 \times_\phi \alpha_2$", in which α, α_1 and α_2 represent some other arbitrary atomic expressions. The precise meanings of all atomic operators described by the Haskell data types "*unary-op*" and "*binary-op*", as well as the purpose of their *format* operand ϕ are discussed later in this section.

The simplest of all atomic forms are the *variable atoms*, depicted by the variable's identifier as in "v". Unlike the pure lambda calculus from Section 4.1, Etude does not prescribe a predetermined implementation of such terms, allowing us to represent variables by values of an arbitrary Haskell type that is a member of the standard classes "eq", "ord" and "enum" described separately in Appendix A. Given one such type v, the type of Etude atoms in which all variables are represented by well-formed values of v is written as "*atom*$_v$". Individual atom expressions of this type are generally

depicted by the Greek letter "α" or a decorated version thereof, such as "α'", "α_k", etc. Similarly, operator values of the "*unary-op*" and "*binary-op*" types are generally represented by the variable "*op*", while "*format*" entities are, for conciseness, depicted by the Greek letter "ϕ".

Since the syntax of atoms does not include a binding construct such as "$\lambda v.\tau$", every variable appearing anywhere in an atom α is automatically included in its set of free variables $\mathrm{FV}(\alpha)$, which can be constructed trivially by the following induction over the atom's lexical structure:

$$\mathrm{FV}[\![\cdot]\!] :: (\mathrm{ord}\ v) \Rightarrow \mathit{atom}_v \rightarrow \{v\}$$

$$\begin{aligned}
\mathrm{FV}[\![v]\!] &= \{v\} \\
\mathrm{FV}[\![\#x_\phi]\!] &= \varnothing \\
\mathrm{FV}[\![\mathit{op}_\phi(\alpha)]\!] &= \mathrm{FV}(\alpha) \\
\mathrm{FV}[\![\alpha_1\ \mathit{op}_\phi\ \alpha_2]\!] &= \mathrm{FV}(\alpha_1) \cup \mathrm{FV}(\alpha_2)
\end{aligned}$$

and also, for entire lists of atoms:

$$\mathrm{FV}[\![\cdot]\!] :: (\mathrm{ord}\ v) \Rightarrow \mathit{atoms}_v \rightarrow \{v\}$$

$$\mathrm{FV}[\![\bar{\alpha}]\!] = \bigcup[\mathrm{FV}(\alpha_k) \mid \alpha_k \leftarrow \bar{\alpha}]$$

The later definition uses the Haskell list comprehension notation "$[f(x_k) \mid x_k \leftarrow \bar{x}]$" in order to apply the "FV" function to every element of the supplied atomic list $\bar{\alpha}$ and, subsequently, the list union operator "$\bigcup$" described in Appendix A.8 to collapse the resulting list into a single set.

In general, variable atoms are meaningless in isolation, but may be closed by the simple process of variable substitution defined by the following Haskell constructions:

$$[\![\cdot]\!]/[\![\cdot]\!] :: (\mathrm{ord}\ v) \Rightarrow \mathit{atom}_v \rightarrow (v \mapsto \mathit{atom}_v) \rightarrow \mathit{atom}_v$$

such that "v/S" replaces the variable v with its binding in the finite map S whenever such binding exists, leaving all other atoms $v' \notin \mathrm{dom}(S)$ unchanged:

$$\begin{aligned}
[\![v]\!]/[\![S]\!] \mid\ & v \in \mathrm{dom}(S) = S(v) \\
\mid\ & \text{otherwise} = [\![v]\!]
\end{aligned}$$

For all other atomic forms, the substitution is simply applied recursively to every constituent of the construct:

$$\begin{aligned}
[\![\#x_\phi]\!] &/[\![S]\!] = [\![\#x_\phi]\!] \\
[\![\mathit{op}_\phi(\alpha)]\!] &/[\![S]\!] = [\![\mathit{op}_\phi[\![\alpha/S]\!]]\!] \\
[\![\alpha_1\ \mathit{op}_\phi\ \alpha_2]\!]/[\![S]\!] &= [\![[\![\alpha_1/S]\!]\ \mathit{op}_\phi\ [\![\alpha_2/S]\!]]\!]
\end{aligned}$$

In the same vein, a substitution of an entire list of atoms can be described by the following list comprehension:

$$[\![\cdot]\!]/[\![\cdot]\!] :: (\mathrm{ord}\ v) \Rightarrow \mathit{atoms}_v \rightarrow (v \mapsto \mathit{atom}_v) \rightarrow \mathit{atoms}_v$$

$$[\![\bar{\alpha}]\!]/[\![S]\!] = [\alpha_k/S \mid \alpha_k \leftarrow \bar{\alpha}]$$

The second most common atomic form, written as "$\#x_\phi$", represents quantities with a predefined numeric value depicted by the rational constant x. Precisely what values of x

should be permitted in a well-formed atomic construct is, unfortunately, a controversial topic since, after over 50 years of development, designers of instruction set architectures are yet to reach a universal agreement on the precise mapping between the infinite set of all rational numbers known to mathematics and the finite resources available to physical computational machines. In the generic fragment of Etude semantics, this mapping between atomic and numeric values is known as a *format*, so called because, under typical Etude implementations, it is directly related to the encoding of numeric values into bit patterns manipulated by binary computational hardware. The reader should take care to note that the concept of a format is vastly different from that of an expression *type* in the type-theoretic sense of the term. In fact, formats do little to mitigate Etude's untyped nature and, under most modern instruction sets architectures, only a handful of different representations of data is supported directly by the underlying computational hardware (the MMIX architecture described in Chapter 6 supports only one.) The format annotations found in atoms of the form "$\#x_\phi$" are used only to guide the precise binary encoding of the rational quantity x and those associated with all other atomic forms serve merely as a convenient lexical means of grouping the plethora of otherwise-distinct arithmetic operations into a small number of related categories.

Formally, every format is represented by a pair of the form "$\gamma.\varepsilon$", in which the two components γ and ε are known, respectively, as the format's *genre* and *encoding*:

format:
    genre . *encoding*

genre: one of
    N Z R F O

For convenience, these two components may be extracted from a given format with the help of the following Haskell definitions:

$\gamma[\![\cdot]\!] :: \mathit{format} \to \mathit{genre}$
$\gamma[\![\gamma.\varepsilon]\!] = \gamma$

$\varepsilon[\![\cdot]\!] :: \mathit{format} \to \mathit{encoding}$
$\varepsilon[\![\gamma.\varepsilon]\!] = \varepsilon$

Genres are used to subdivide the set of all available formats into six different families according to their various semantic properties. In particular, the "N" genre represents a family of formats capable of representing non-negative integers or *natural numbers*, while the "Z" formats interpret bit patterns as signed integers taken from a suitable range of values as determined by the format's associated encoding component. Formats in these two genres are known as *natural* and *integer formats*, respectively. The "R" genre describes a small set of *rational formats* that enable Etude programs to manipulate a more general class of *rational numbers* represented in one of the popular *floating point encodings* such as the *IEEE 754 Standard for Binary Floating-Point Arithmetic* [IEEE 754]. The remaining two genres "F" and "O" characterise the highly-specialised *function* and *object formats*, collectively known as the *pointer format* fam-

ily. Their individual rôles are discussed later in the current chapter. Collectively, formats from the "N" and "Z" genres are known as *integral formats* and those from "N", "Z" or "R" genres are often referred to as *arithmetic formats* in this work.

The genre also determines the set of encoding values permitted in a format, although, once again, most of the rules that determine the precise relationship between genres and encodings is left unspecified in the generic fragment of the Etude language. Accordingly, in this chapter, the well-formedness property "$\text{WF}[\![\cdot]\!] :: \textit{format} \rightarrow \star$" is described only indirectly through a small set of simple constraints that, collectively, outline the minimal algebraic properties of all well-formed Etude formats.

In particular, every Etude implementation is guaranteed to support at least one format from every genre, whose encoding is represented by some unspecified common value "Φ" and, further, for every well-formed integer format "$\text{Z}.\varepsilon$", the corresponding natural format "$\text{N}.\varepsilon$" is also guaranteed to be well-formed. Formally:

$$\begin{aligned}&\text{WF}_\Phi :: \forall\ \gamma \Rightarrow \text{WF}(\gamma) \rightarrow \text{WF}[\![\gamma.\Phi]\!]\\ &\text{WF}_\text{I} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{Z}.\varepsilon]\!] \rightarrow \text{WF}[\![\text{N}.\varepsilon]\!]\end{aligned}$$

Such distinguished encoding "Φ" is known as the *standard encoding* of the underlying instruction set architecture and is represented formally by an implementation-defined Haskell definition with the following type signature:

$$\Phi :: \textit{encoding}$$

Every well-formed format ϕ is further associated with four numeric properties known as its *size*, *width*, *greatest lower bound* and *least upper bound*, represented by the following Haskell constructions:

$$\begin{aligned}&\mathcal{S}\ [\![\cdot]\!] :: \textit{format} \rightarrow \textit{integer} && \textit{(size)}\\ &\mathcal{W}[\![\cdot]\!] :: \textit{format} \rightarrow \textit{integer} && \textit{(width)}\\ &\text{glb}[\![\cdot]\!] :: \textit{format} \rightarrow \textit{rational} && \textit{(greatest lower bound)}\\ &\text{lub}[\![\cdot]\!] :: \textit{format} \rightarrow \textit{rational} && \textit{(least upper bound)}\end{aligned}$$

Intuitively, a format's size $\mathcal{S}(\phi)$ represents the amount of space (in bytes) occupied by values encoded under that format in memory, its width $\mathcal{W}(\phi)$ represents the number of bits utilised in the underlying binary representation of data and the lower and upper bounds depict the smallest and greatest numeric quantities that are representable precisely under that format. Formally, these properties are subject to the following universal constraints on every Etude implementation:

$$\begin{aligned}&\text{SIZE}_\phi :: \forall\ \phi \Rightarrow \text{WF}(\phi) \rightarrow [\![\mathcal{S}(\phi) > 0]\!]\\ &\text{WIDTH}_\phi :: \forall\ \phi \Rightarrow \text{WF}(\phi) \rightarrow [\![0 < \mathcal{W}(\phi) \leq \omega \times \mathcal{S}(\phi)]\!]\\ &\text{GLB}_\phi :: \forall\ \phi \Rightarrow \text{WF}(\phi) \rightarrow [\![\text{glb}(\phi) \leq 0]\!]\\ &\text{LUB}_\phi :: \forall\ \phi \Rightarrow \text{WF}(\phi) \rightarrow [\![\text{lub}(\phi) > 0]\!]\end{aligned}$$

In other words, the greatest lower bound of every Etude format must always have a negative or zero value, while its size, width and least upper bound are guaranteed to represent positive integer quantities, with the width no greater than the number of bytes required for representation of that format's value in a pure binary notation, i.e., the

product of its size with the architecture's *byte width* parameter. In the remainder of this work, the architecture's byte width is represented universally by the Greek letter "ω", as stipulated by the following Haskell type signature:

$$\omega :: \textit{integer}$$

A further two constraints are imposed on the bounds of all well-formed natural formats as follows:

$$\begin{aligned}
&\text{GLB}_\text{N} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{N}.\varepsilon]\!] \rightarrow [\![\text{glb}[\![\text{N}.\varepsilon]\!] = 0]\!] \\
&\text{LUB}_\text{N} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{N}.\varepsilon]\!] \rightarrow [\![\text{lub}[\![\text{N}.\varepsilon]\!] = 2^{\mathcal{W}[\![\text{N}.\varepsilon]\!]} - 1]\!]
\end{aligned}$$

From theorem WIDTH_ϕ, it should be clear that the byte width must always assume a strictly-positive integer value. Collectively, the above theorems guarantee that, on every Etude implementation, all natural numbers are always represented using *pure binary notation* as required by the ISO C Standard [ANSI 89]. On the other hand, integer formats have a much more relaxed set of properties that are captured as follows:

$$\begin{aligned}
&\text{SIZE}_\text{Z} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{Z}.\varepsilon]\!] \rightarrow [\![\mathcal{S}[\![\text{Z}.\varepsilon]\!] = \mathcal{S}[\![\text{N}.\varepsilon]\!]]\!] \\
&\text{GLB}_\text{Z} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{Z}.\varepsilon]\!] \rightarrow [\![\text{glb}[\![\text{Z}.\varepsilon]\!] \leq -\text{lub}[\![\text{Z}.\varepsilon]\!]]\!] \\
&\text{LUB}_\text{Z} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{Z}.\varepsilon]\!] \rightarrow [\![\text{lub}[\![\text{Z}.\varepsilon]\!] < \text{lub}[\![\text{N}.\varepsilon]\!]]\!]
\end{aligned}$$

The range of quantities representable under the rational arithmetic formats in the "R" genre is somewhat more difficult to characterise. On almost all modern architectures, these formats are associated with the numeric encodings defined by the *IEEE 754 Standard for Binary Floating Point Arithmetic* [IEEE 754], although other similar representations of floating point numbers can be also found in common use. In general, every such representation is characterised completely by four integer quantities known as the format's *radix*, *precision*, *minimum exponent* and *maximum exponent*, respectively. Formally, these language parameters are captured for every well-formed rational format "R.ε" by the following four Haskell functions:

$$\begin{aligned}
&\text{r}[\![\cdot]\!] &&:: \textit{format} \rightarrow \textit{integer} && \textit{(radix)} \\
&\text{p}[\![\cdot]\!] &&:: \textit{format} \rightarrow \textit{integer} && \textit{(precision)} \\
&\text{E}_\text{min}[\![\cdot]\!] &&:: \textit{format} \rightarrow \textit{integer} && \textit{(minimum exponent)} \\
&\text{E}_\text{max}[\![\cdot]\!] &&:: \textit{format} \rightarrow \textit{integer} && \textit{(maximum exponent)}
\end{aligned}$$

In general, these four properties are only ever defined for well-formed rational formats, whereby they are subject to the following value constraints:

$$\begin{aligned}
&\text{RADIX} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{r}[\![\text{R}.\varepsilon]\!] > 1]\!] \\
&\text{PREC}\ \ :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{p}[\![\text{R}.\varepsilon]\!] > 0]\!] \\
&\text{EMIN}\ \ :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{E}_\text{min}[\![\text{R}.\varepsilon]\!] < 0]\!] \\
&\text{EMAX}\ \ :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{E}_\text{max}[\![\text{R}.\varepsilon]\!] > 0]\!]
\end{aligned}$$

Intuitively, a given rational formal ϕ is capable of representing precisely every number of the form $(-1)^s \times m \times \text{r}(\phi)^{e\ -\ \text{p}(\phi)}$, where s, m and e are non-negative integers such that $0 \leq s \leq 1$, $\text{E}_\text{min}(\phi) \leq e \leq \text{E}_\text{max}(\phi)$ and $0 \leq m < \text{r}(\phi)^{\text{p}(\phi)}$. Formally:

$$\begin{aligned}
&\text{GLB}_\text{R} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{glb}[\![\text{R}.\varepsilon]\!] = -\text{r}[\![\text{R}.\varepsilon]\!]^{\text{p}[\![\text{R}.\varepsilon]\!]} - 1 \times \text{r}[\![\text{R}.\varepsilon]\!]^{\text{E}_\text{max}[\![\text{R}.\varepsilon]\!]\ -\ \text{p}[\![\text{R}.\varepsilon]\!]}]\!] \\
&\text{LUB}_\text{R} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{lub}[\![\text{R}.\varepsilon]\!] = -\text{glb}[\![\text{R}.\varepsilon]\!]]\!]
\end{aligned}$$

But back to Etude atoms. In this chapter, their meaning is represented by an algebraic semantics formed from the usual well-formedness property "$\text{WF}[\![\cdot]\!] :: \mathit{atom}_\nu \to \star$" and an atom equivalence relation "$[\![\cdot]\!] \equiv [\![\cdot]\!] :: \mathit{atom}_\nu \to \mathit{atom}_\nu \to \star$". In Chapter 6, both properties are defined precisely in terms of the operational behaviour of Etude atoms and the two semantic formulations are shown to be equivalent by proving all of the theorems outlined in the remainder of this section.

In general, well-formedness can be asserted only for closed atoms, so that variables may never appear in operands of the "WF" and "$\equiv$" properties. A constant atom of the form "$\#x_\phi$", however, has a well-defined meaning provided that the rational number x is equal to 0 or else it is representable precisely under the corresponding format ϕ:

$$\text{WF}_0 :: \forall\ \phi \Rightarrow \text{WF}(\phi) \to \text{WF}[\![\#0_\phi]\!]$$

$$\begin{aligned}\text{WF}_\text{I} :: \forall\ & n :: \mathit{integer},\ \phi \Rightarrow \\ & \text{WF}(\phi) \to \\ & [\![\gamma(\phi) \in \{\mathsf{N}, \mathsf{Z}\} \wedge \text{glb}(\phi) \le n \le \text{lub}(\phi)]\!] \to \\ & \text{WF}[\![\#x_\phi]\!]\end{aligned}$$

$$\begin{aligned}\text{WF}_\text{R} :: \forall\ & s, m, e :: \mathit{integer},\ \phi \Rightarrow \\ & \text{WF}(\phi) \to \\ & [\![\gamma(\phi) = [\![\mathsf{R}]\!] \wedge 0 \le s \le 1 \wedge 0 \le m < \text{r}(\phi)^{\text{p}(\phi)} \wedge \text{E}_{\min}(\phi) \le e \le \text{E}_{\max}(\phi)]\!] \to \\ & \text{WF}[\![\#[\![(-1)^s \times m \times \text{r}(\phi)^{e\,-\,\text{p}(\phi)}]\!]_\phi]\!]\end{aligned}$$

The last of these theorems defines the minimal set of atoms guaranteed to be representable under a rational format from the "R" genre. Intuitively, this set includes at least the *normalised floating point numbers* of the form $(-1)^s \times m \times \text{r}(\phi)^{e\,-\,\text{p}(\phi)}$, where s, m and e are arbitrary integers such that $0 \le s \le 1$, $0 \le m < \text{r}(\phi)^{\text{p}(\phi)}$ and $\text{E}_{\min}(\phi) \le e \le \text{E}_{\max}(\phi)$.

Otherwise, in all arithmetic atoms of the form "$\mathit{op}_\phi(\#x_\phi)$" or "$\#x_\phi\ \mathit{op}_\phi\ \#y_\phi$", every operand "$\#x_\phi$" and "$\#y_\phi$" must represent a well-formed atom. More so, the operations are guaranteed to be meaningful only if their true arithmetic result falls within the numeric bounds established by ϕ, or else if ϕ belongs to the natural genre "N". If *op* represents one of the five arithmetic operators "$+_\phi$", "$-_\phi$", "$\times_\phi$", "$\div_\phi$" or "$\cdot|\cdot_\phi$", then these constraints are captured formally by the following five theorems:

$$\begin{aligned}\text{WF}_\neg :: \forall\ & x, \phi \Rightarrow \\ & \text{WF}[\![\#x_\phi]\!] \to \\ & [\![\gamma(\phi) = [\![\mathsf{N}]\!] \vee (\gamma(\phi) \in \{\mathsf{Z}, \mathsf{R}\} \wedge \text{glb}(\phi) \le -x \le \text{lub}(\phi))]\!] \to \\ & \text{WF}[\![-_\phi \#x_\phi]\!]\end{aligned}$$

$$\begin{aligned}\text{WF}_+ :: \forall\ & x, y, \phi \Rightarrow \\ & \text{WF}[\![\#x_\phi]\!] \to \text{WF}[\![\#y_\phi]\!] \to \\ & [\![\gamma(\phi) = [\![\mathsf{N}]\!] \vee (\gamma(\phi) \in \{\mathsf{Z}, \mathsf{R}\} \wedge \text{glb}(\phi) \le x + y \le \text{lub}(\phi))]\!] \to \\ & \text{WF}[\![\#x_\phi\ +_\phi\ \#y_\phi]\!]\end{aligned}$$

$$\begin{aligned}\text{WF}_- :: \forall\ & x, y, \phi \Rightarrow \\ & \text{WF}[\![\#x_\phi]\!] \to \text{WF}[\![\#y_\phi]\!] \to \\ & [\![\gamma(\phi) = [\![\mathsf{N}]\!] \vee (\gamma(\phi) \in \{\mathsf{Z}, \mathsf{R}\} \wedge \text{glb}(\phi) \le x - y \le \text{lub}(\phi))]\!] \to \\ & \text{WF}[\![\#x_\phi\ -_\phi\ \#y_\phi]\!]\end{aligned}$$

$$\begin{aligned}
&\mathrm{WF}_{\times} :: \forall\, x,\, y,\, \phi \Rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow \mathrm{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) = [\![\mathsf{N}]\!] \vee (\gamma(\phi) \in \{\mathsf{Z},\, \mathsf{R}\} \wedge \mathrm{glb}(\phi) \leq x \times y \leq \mathrm{lub}(\phi))]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi \times_\phi \#y_\phi]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{WF}_{\div} :: \forall\, x,\, y,\, \phi \Rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow \mathrm{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\, \mathsf{Z},\, \mathsf{R}\} \wedge \mathrm{glb}(\phi) \leq x/y \leq \mathrm{lub}(\phi)]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi \div_\phi \#y_\phi]\!]
\end{aligned}$$

More so, in a *remainder operation* of the form "$\#x_\phi \cdot|\cdot_\phi \#y_\phi$", ϕ must always represent an integral format with $\mathrm{glb}(\phi) \leq x/y \leq \mathrm{lub}(\phi)$:

$$\begin{aligned}
&\mathrm{WF}_{\cdot|\cdot} :: \forall\, x,\, y,\, \phi \Rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow \mathrm{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\, \mathsf{Z},\, \mathsf{R}\} \wedge \mathrm{glb}(\phi) \leq x/y \leq \mathrm{lub}(\phi)]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi \cdot|\cdot_\phi \#y_\phi]\!]
\end{aligned}$$

Similarly, in the *bit operations* "$\alpha_1 \triangle_\phi \alpha_2$", "$\alpha_1 \bar{\vee}_\phi \alpha_2$", "$\alpha_1 \vee_\phi \alpha_2$", "$\alpha_1 \ll_\phi \alpha_2$" and "$\alpha_1 \gg_\phi \alpha_2$", ϕ must belong to an integral genre and both operand atoms must have non-negative values. For the *bit shift operators* "$\ll_\phi$" and "$\gg_\phi$", the value of α_2 must be smaller than the width of ϕ and, further, for the *right bit shift* "$\gg_\phi$", the product of α_1 and 2^{α_2} must also fall within the range of values representable by that format. In Haskell:

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{BIT}} :: \forall\, x,\, y,\, \phi,\, op \Rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow \mathrm{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![x \geq 0 \wedge y \geq 0 \wedge op \in \{\triangle, \bar{\vee}, \vee\} \wedge \gamma(\phi) \in \{\mathsf{N}, \mathsf{Z}\}]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi\ op_\phi\ \#y_\phi]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{WF}_{\ll} :: \forall\, x,\, y,\, \phi \Rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow \mathrm{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![x \geq 0 \wedge 0 \leq y < \mathcal{W}(\phi) \wedge (\gamma(\phi) = [\![\mathsf{N}]\!] \vee (\gamma(\phi) = [\![\mathsf{Z}]\!] \wedge x \times 2^y \leq \mathrm{lub}(\phi)))]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi \ll_\phi \#y_\phi]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{WF}_{\gg} :: \forall\, x,\, y,\, \phi \Rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow \mathrm{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![x \geq 0 \wedge 0 \leq y < \mathcal{W}(\phi) \wedge \gamma(\phi) \in \{\mathsf{N}, \mathsf{Z}\}]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi \gg_\phi \#y_\phi]\!]
\end{aligned}$$

Finally, the remaining *bit complement operator* "$\sim_\phi$" is guaranteed to be applicable only under natural formats from the "N" genre:

$$\mathrm{WF}_{\sim} :: \forall\, x,\, \phi \Rightarrow \mathrm{WF}[\![\#x_\phi]\!] \rightarrow [\![\gamma(\phi) = [\![\mathsf{N}]\!]]\!] \rightarrow \mathrm{WF}[\![\sim_\phi \#x_\phi]\!]$$

The addition operator "$+_\phi$" may be also applied under a well-formed format from the object genre "O", provided that its second operand is well-formed under the standard integer format "Z.Φ". However, the precise construction of all such atomic forms depends heavily on the underlying memory architecture and their generic semantics are discussed separately in Sections 4.5 and 4.6 later in this chapter.

The two *equality operators* "$=_\phi$" and "$\neq_\phi$" are well-formed for all pairs of meaningful operand values, while the *relational operators* "$<_\phi$", "$>_\phi$", "$\leq_\phi$" and "$\geq_\phi$" are

meaningful only under the arithmetic and object (but not functional) formats:

$$\begin{aligned}&\mathrm{WF}_{\mathrm{EQL}} :: \forall\, x,\, y,\, \phi,\, op \Rightarrow\\ &\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow \mathrm{WF}[\![\#y_\phi]\!] \rightarrow\\ &\quad [\![op \in \{=, \neq\}]\!] \rightarrow\\ &\quad \mathrm{WF}[\![\#x_\phi\ op_\phi\ \#y_\phi]\!]\end{aligned}$$

$$\begin{aligned}&\mathrm{WF}_{\mathrm{REL}} :: \forall\, x,\, y,\, \phi,\, op \Rightarrow\\ &\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow \mathrm{WF}[\![\#y_\phi]\!] \rightarrow\\ &\quad [\![op \in \{<, >, \leq, \geq\} \wedge \gamma(\phi) \neq [\![\mathsf{F}]\!]]\!] \rightarrow\\ &\quad \mathrm{WF}[\![\#x_\phi\ op_\phi\ \#y_\phi]\!]\end{aligned}$$

Finally, a *conversion operation* of the form "$\phi'_\phi(\#x_\phi)$" is well-formed whenever both of the formats ϕ and ϕ' belong to some arithmetic genre and the numeric value of x falls within the bounds of the target format ϕ', or else if an integral format ϕ is converted into a natural format ϕ', or when the operation converts between natural, integral, function and object representations of the constant 0. Formally:

$$\begin{aligned}&\mathrm{WF}_{\mathrm{CN}} :: \forall\, x,\, \phi,\, \phi' \Rightarrow\\ &\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow\\ &\quad [\![\gamma(\phi) \in \{\mathsf{N}, \mathsf{Z}\} \wedge \gamma(\phi') = [\![\mathsf{N}]\!]]\!] \rightarrow\\ &\quad \mathrm{WF}[\![\phi'_\phi(\#x_\phi)]\!]\end{aligned}$$

$$\begin{aligned}&\mathrm{WF}_{\mathrm{CA}} :: \forall\, x,\, \phi,\, \phi' \Rightarrow\\ &\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow\\ &\quad [\![\{\gamma(\phi), \gamma(\phi')\} \subset \{\mathsf{N}, \mathsf{Z}, \mathsf{R}\} \wedge \mathrm{glb}(\phi') \leq x \leq \mathrm{lub}(\phi')]\!] \rightarrow\\ &\quad \mathrm{WF}[\![\phi'_\phi(\#x_\phi)]\!]\end{aligned}$$

$$\begin{aligned}&\mathrm{WF}_{\mathrm{CP}} :: \forall\, \phi,\, \phi' \Rightarrow\\ &\quad \mathrm{WF}(\phi) \rightarrow \mathrm{WF}(\phi') \rightarrow\\ &\quad [\![\gamma(\phi) \notin [\![\mathsf{R}]\!] \wedge \gamma(\phi) \notin [\![\mathsf{R}]\!]]\!] \rightarrow\\ &\quad \mathrm{WF}[\![\phi'_\phi(\#0_\phi)]\!]\end{aligned}$$

No other form of atoms need be well-formed on ever Etude implementation, except that, in order to reason about more complex inductively-defined atomic structures, one additional theorem is required:

$$\mathrm{WF}_{\equiv} :: \forall\, \alpha_1,\, \alpha_2 \Rightarrow \mathrm{WF}(\alpha_1) \rightarrow (\alpha_1 \equiv \alpha_2) \rightarrow \mathrm{WF}(\alpha_2)$$

Intuitively, this states that well-formedness of a particular atom α is always sufficient to establish well-formedness of all other atoms α' belonging to the same equivalence class. In the generic algebraic semantics of Etude, these equivalence classes are defined by the binary Haskell property "$\equiv$", which, like, "WF", may have various implementation-specific aspects and, accordingly, is only characterised through a set of appropriate constraint theorems in this chapter. For the MMIX instruction set architecture, its complete formulation is given separately in Chapter 6.

Since "$\equiv$" should represent an equivalence relation, it must always satisfy the three equivalence laws of *reflexivity*, *symmetry* and *transitivity*. Formally:

$$\begin{aligned}&\mathrm{REFL}_\alpha \quad :: \forall\, \alpha \rightarrow (\alpha \equiv \alpha)\\ &\mathrm{SYMM}_\alpha :: \forall\, \alpha_1,\, \alpha_2 \Rightarrow (\alpha_1 \equiv \alpha_2) \rightarrow (\alpha_2 \equiv \alpha_1)\\ &\mathrm{TRANS}_\alpha :: \forall\, \alpha_1,\, \alpha_2,\, \alpha_3 \Rightarrow (\alpha_1 \equiv \alpha_2) \rightarrow (\alpha_2 \equiv \alpha_3) \rightarrow (\alpha_1 \equiv \alpha_3)\end{aligned}$$

More so, applying a unary or binary arithmetic operator to pairwise-equivalent arguments always produces a pair of equivalent Etude atoms as depicted by the following two *compatibility* laws:

$$\begin{aligned}
&\text{EQV}_{\alpha 1} :: \forall\ \alpha_1,\ \alpha_2,\ \phi,\ op \Rightarrow (\alpha_1 \equiv \alpha_2) \rightarrow [\![op_\phi(\alpha_1)]\!] \equiv [\![op_\phi(\alpha_2)]\!] \\
&\text{EQV}_{\alpha 2} :: \forall\ \alpha_{11},\ \alpha_{21},\ \alpha_{12},\ \alpha_{22},\ \phi,\ op \Rightarrow \\
&\qquad (\alpha_{11} \equiv \alpha_{12}) \rightarrow (\alpha_{21} \equiv \alpha_{22}) \rightarrow [\![\alpha_{11}\ op_\phi\ \alpha_{21}]\!] \equiv [\![\alpha_{12}\ op_\phi\ \alpha_{22}]\!]
\end{aligned}$$

The remaining theorems in this section capture the precise set of properties guaranteed for arithmetic operations by the C Standard [ANSI 89]. First of all, every well-formed arithmetic operation applied under a natural format "$\text{N}.\varepsilon$" is equivalent to an atom of the form "$\#x_{\text{N}.\varepsilon}$", where x represents the true mathematical result of the operation taken modulo $\text{lub}[\![\text{N}.\varepsilon]\!] + 1$. Formally:

$$\begin{aligned}
&\text{EQV}_{+\text{N}} :: \forall\ x,\ y,\ \varepsilon \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\text{N}.\varepsilon}]\!] \rightarrow \text{WF}[\![\#y_{\text{N}.\varepsilon}]\!] \rightarrow \\
&\qquad [\![\#x_{\text{N}.\varepsilon} +_{\text{N}.\varepsilon} \#y_{\text{N}.\varepsilon}]\!] \equiv [\![\#[\![(x+y) \bmod (\text{lub}[\![\text{N}.\varepsilon]\!]+1)]\!]_{\text{N}.\varepsilon}]\!] \\
&\text{EQV}_{-\text{N}} :: \forall\ x,\ y,\ \varepsilon \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\text{N}.\varepsilon}]\!] \rightarrow \text{WF}[\![\#y_{\text{N}.\varepsilon}]\!] \rightarrow \\
&\qquad [\![\#x_{\text{N}.\varepsilon} -_{\text{N}.\varepsilon} \#y_{\text{N}.\varepsilon}]\!] \equiv [\![\#[\![(x-y) \bmod (\text{lub}[\![\text{N}.\varepsilon]\!]+1)]\!]_{\text{N}.\varepsilon}]\!] \\
&\text{EQV}_{\times\text{N}} :: \forall\ x,\ y,\ \varepsilon \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\text{N}.\varepsilon}]\!] \rightarrow \text{WF}[\![\#y_{\text{N}.\varepsilon}]\!] \rightarrow \\
&\qquad [\![\#x_{\text{N}.\varepsilon} \times_{\text{N}.\varepsilon} \#y_{\text{N}.\varepsilon}]\!] \equiv [\![\#[\![(x \times y) \bmod (\text{lub}[\![\text{N}.\varepsilon]\!]+1)]\!]_{\text{N}.\varepsilon}]\!] \\
&\text{EQV}_{\neg\text{N}} :: \forall\ x,\ \varepsilon \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\text{N}.\varepsilon}]\!] \rightarrow \\
&\qquad [\![-_{\text{N}.\varepsilon}(\#x_{\text{N}.\varepsilon})]\!] \equiv [\![\#[\![-x \bmod (\text{lub}[\![\text{N}.\varepsilon]\!]+1)]\!]_{\text{N}.\varepsilon}]\!] \\
&\text{EQV}_{\sim\text{N}} :: \forall\ x,\ \varepsilon \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\text{N}.\varepsilon}]\!] \rightarrow \\
&\qquad [\![\sim_{\text{N}.\varepsilon}(\#x_{\text{N}.\varepsilon})]\!] \equiv [\![\#[\![\text{lub}[\![\text{N}.\varepsilon]\!] - x]\!]_{\text{N}.\varepsilon}]\!] \\
&\text{EQV}_{\text{IN}} :: \forall\ x,\ \phi,\ \varepsilon \Rightarrow \\
&\qquad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\text{N}.\varepsilon]\!] \rightarrow [\![\gamma(\phi) \in \{\text{N},\ \text{Z}\}]\!] \rightarrow \\
&\qquad [\![\text{N}.\varepsilon_\phi(\#x_\phi)]\!] \equiv [\![\#[\![x \bmod (\text{lub}[\![\text{N}.\varepsilon]\!]+1)]\!]_{\text{N}.\varepsilon}]\!]
\end{aligned}$$

The last two theorems in the above list describe the meanings of *bit complement* and *natural conversion* operations, respectively. Intuitively, a bit complement of the form "$\sim_\phi(\#x_\phi)$" inverts the binary representation of x, producing the value of "$\text{lub}(\phi) - x$" for all natural formats ϕ, while the conversion atom "$\phi'_\phi(\#x_\phi)$" transforms an arbitrary signed or unsigned integer x into the natural number "$x \bmod (\text{lub}(\phi') - 1)$".

On the other hand, under an integer or rational format ϕ, these operations always approximate their true arithmetic result x within some predetermined margin of error $\text{ulp}_\phi(x)$, provided that this result falls within the numeric bounds established by the format. In particular, if the true arithmetic result x of such an operation is approximated by an atom of the form "$\#x'_\phi$", then we can always be sure that $|x' - x| < \text{ulp}_\phi(x)$. Formally, the upper bounds imposed on this approximation error is known as the *unit of least precision*. For formats from the "Z" genre, this value is always equal to 1, so that all arithmetic operations over integer values are always exact in Etude. Under

rational formats, however, the only guarantee provided by the generic fragments of the language is that the operation's result differs from its true arithmetic meaning by less than $\mathrm{r}(\phi)^{\lfloor \log_{\mathrm{r}(\phi)} |x| \rfloor - \mathrm{p}(\phi) + 1}$ or $\mathrm{r}(\phi)^{\mathrm{E}_{\min}(\phi) - 1}$ in magnitude, whichever is greater. Formally:

$$\mathrm{ulp}_{[\![\cdot]\!]}[\![\cdot]\!] :: \mathit{format} \rightarrow \mathit{rational} \rightarrow \mathit{rational}$$

$$\begin{array}{lll} \mathrm{ulp}_{\phi}[\![x]\!] \mid \gamma(\phi) = [\![\mathsf{Z}]\!] & & = 1 \\ \quad \mid \gamma(\phi) = [\![\mathsf{R}]\!] \wedge |x| < \mathrm{r}(\phi)^{\mathrm{E}_{\min}(\phi) - 1} & & = \mathrm{r}(\phi)^{\mathrm{E}_{\min}(\phi) - 1} \\ \quad \mid \gamma(\phi) = [\![\mathsf{R}]\!] \wedge |x| \geq \mathrm{r}(\phi)^{\mathrm{E}_{\min}(\phi) - 1} & & = \mathrm{r}(\phi)^{\lfloor \log_{\mathrm{r}(\phi)} |x| \rfloor - \mathrm{p}(\phi) + 1} \end{array}$$

Using this auxiliary definition, the meanings of all arithmetic operations over signed integers and rational numbers can be formalised as follows:

$$\begin{array}{l} \mathrm{EQV}_{+\mathrm{A}} :: \forall\, x, y, z, \phi \Rightarrow \\ \quad \mathrm{WF}[\![\#x_{\phi}]\!] \rightarrow \mathrm{WF}[\![\#y_{\phi}]\!] \rightarrow \\ \quad [\![\gamma(\phi) \in \{\mathsf{Z}, \mathsf{R}\} \wedge \mathrm{glb}(\phi) \leq x + y \leq \mathrm{lub}(\phi)]\!] \rightarrow \\ \quad [\![\#x_{\phi} +_{\phi} \#y_{\phi}]\!] \equiv [\![\#z_{\phi}]\!] \rightarrow \\ \quad [\![|z - (x + y)| < \mathrm{ulp}_{\phi}(x + y)]\!] \end{array}$$

$$\begin{array}{l} \mathrm{EQV}_{-\mathrm{A}} :: \forall\, x, y, z, \phi \Rightarrow \\ \quad \mathrm{WF}[\![\#x_{\phi}]\!] \rightarrow \mathrm{WF}[\![\#y_{\phi}]\!] \rightarrow \\ \quad [\![\gamma(\phi) \in \{\mathsf{Z}, \mathsf{R}\} \wedge \mathrm{glb}(\phi) \leq x - y \leq \mathrm{lub}(\phi)]\!] \rightarrow \\ \quad [\![\#x_{\phi} -_{\phi} \#y_{\phi}]\!] \equiv [\![\#z_{\phi}]\!] \rightarrow \\ \quad [\![|z - (x - y)| < \mathrm{ulp}_{\phi}(x - y)]\!] \end{array}$$

$$\begin{array}{l} \mathrm{EQV}_{\times\mathrm{A}} :: \forall\, x, y, z, \phi \Rightarrow \\ \quad \mathrm{WF}[\![\#x_{\phi}]\!] \rightarrow \mathrm{WF}[\![\#y_{\phi}]\!] \rightarrow \\ \quad [\![\gamma(\phi) \in \{\mathsf{Z}, \mathsf{R}\} \wedge \mathrm{glb}(\phi) \leq x \times y \leq \mathrm{lub}(\phi)]\!] \rightarrow \\ \quad [\![\#x_{\phi} \times_{\phi} \#y_{\phi}]\!] \equiv [\![\#z_{\phi}]\!] \rightarrow \\ \quad [\![|z - (x \times y)| < \mathrm{ulp}_{\phi}(x \times y)]\!] \end{array}$$

$$\begin{array}{l} \mathrm{EQV}_{\div\mathrm{A}} :: \forall\, x, y, z, \phi \Rightarrow \\ \quad \mathrm{WF}[\![\#x_{\phi}]\!] \rightarrow \mathrm{WF}[\![\#y_{\phi}]\!] \rightarrow \\ \quad [\![\gamma(\phi) \in \{\mathsf{Z}, \mathsf{R}\} \wedge y \neq 0 \wedge \mathrm{glb}(\phi) \leq x/y \leq \mathrm{lub}(\phi)]\!] \rightarrow \\ \quad [\![\#x_{\phi} \div_{\phi} \#y_{\phi}]\!] \equiv [\![\#z_{\phi}]\!] \rightarrow \\ \quad [\![|z - (x/y)| < \mathrm{ulp}_{\phi}(x/y)]\!] \end{array}$$

$$\begin{array}{l} \mathrm{EQV}_{\neg\mathrm{A}} :: \forall\, x, z, \phi \Rightarrow \\ \quad \mathrm{WF}[\![\#x_{\phi}]\!] \rightarrow \\ \quad [\![\gamma(\phi) \in \{\mathsf{Z}, \mathsf{R}\} \wedge \mathrm{glb}(\phi) \leq -x \leq \mathrm{lub}(\phi)]\!] \rightarrow \\ \quad [\![-_{\phi}(\#x_{\phi})]\!] \equiv [\![\#z_{\phi}]\!] \rightarrow \\ \quad [\![|z - (-x)| < \mathrm{ulp}_{\phi}(-x)]\!] \end{array}$$

When applied under an integral format, the two Etude operators "$\div_{\phi}$" and "$\cdot|\cdot_{\phi}$" are further constrained to produce the quotient and remainder from the division of their operands' values, so that, if both of the operands involved in an atom of the form "$\#x_{\phi} \div_{\phi} \#y_{\phi}$" have non-negative values, then the result is always equivalent to the integral portion of the fraction x/y. Further, for every triple of integers x, y and z such that "$\#x_{\phi} \div_{\phi} \#y_{\phi} \equiv \#z_{\phi}$", the corresponding *remainder operation* of the form "$\#x_{\phi} \cdot|\cdot_{\phi} \#y_{\phi}$" is guaranteed to be equivalent to "$\#x - y \times z_{\phi}$", provided that this result

falls within the bounds of the underlying integral format ϕ. Formally:

$$\begin{aligned}
&\text{EQV}_{\div\text{I}} :: \forall\ x,\ y,\ \phi\ \Rightarrow \\
&\quad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\ \mathsf{Z}\} \wedge x \geq 0 \wedge y > 0]\!] \rightarrow \\
&\quad [\![\#x_\phi \div_\phi \#y_\phi]\!] \equiv [\![\#[\![\lfloor x/y \rfloor]\!]_\phi]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\cdot|\cdot\text{I}} :: \forall\ x,\ y,\ z,\ \phi\ \Rightarrow \\
&\quad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\ \mathsf{Z}\} \wedge y \neq 0 \wedge \text{glb}(\phi) \leq x - y \times z \leq \text{lub}(\phi)]\!] \rightarrow \\
&\quad [\![\#x_\phi \div_\phi \#y_\phi]\!] \equiv [\![\#z_\phi]\!] \rightarrow \\
&\quad [\![\#x_\phi \cdot|\cdot_\phi \#y_\phi]\!] \equiv [\![\#[\![x - y \times z]\!]_\phi]\!]
\end{aligned}$$

Besides the earlier natural conversions, Etude also supports translations between arbitrary arithmetic formats, as well as conversions between an integral format and one from the functional or object genre. In particular, if "$\#x_\phi$" represents a well-formed atom from the arithmetic genre "N", "Z" or "R" with $\text{glb}(\phi') \leq x \leq \text{lub}(\phi')$, then a conversion operation of the form "$\phi'_\phi(\#x_\phi)$" is always equivalent to some other atomic form "$\#z_{\phi'}$", in which z approximates x to within one unit of the target format's least precision $\text{ulp}_{\phi'}(x)$. Further, if ϕ' belongs to an integral genre, then z is guaranteed to be equal to the integral part of x as depicted by the notation "$\lfloor x \rfloor$" defined separately in Appendix A. Finally, if x is equal to 0 and both of the two formats involved belong to a non-rational genre, then the resulting atom is simply equivalent to "$\#0_{\phi'}$". Formally:

$$\begin{aligned}
&\text{EQV}_{\text{RA}} :: \forall\ x,\ z,\ \phi,\ \phi'\ \Rightarrow \\
&\quad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\phi']\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\ \mathsf{Z},\ \mathsf{R}\} \wedge \gamma(\phi') = [\![\mathsf{R}]\!] \wedge \text{glb}(\phi') \leq x \leq \text{lub}(\phi')]\!] \rightarrow \\
&\quad [\![\phi'_\phi(\#x_\phi)]\!] \equiv [\![\#z_{\phi'}]\!] \rightarrow \\
&\quad [\![|z - x| < \text{ulp}_{\phi'}(x)]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\text{AI}} :: \forall\ x,\ \phi,\ \phi'\ \Rightarrow \\
&\quad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\phi']\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\ \mathsf{Z},\ \mathsf{R}\} \wedge \gamma(\phi') \in \{\mathsf{N},\ \mathsf{Z}\} \wedge \text{glb}(\phi') \leq [x] \leq \text{lub}(\phi')]\!] \rightarrow \\
&\quad [\![\phi'_\phi(\#x_\phi)]\!] \equiv [\![\#[\![[x]]\!]_{\phi'}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\text{PP}} :: \forall\ \phi,\ \phi'\ \Rightarrow \\
&\quad \text{WF}(\phi) \rightarrow \text{WF}(\phi') \rightarrow \\
&\quad [\![\gamma(\phi) \neq [\![\mathsf{R}]\!] \wedge \gamma(\phi') \neq [\![\mathsf{R}]\!]]\!] \rightarrow \\
&\quad [\![\phi'_\phi(\#0_\phi)]\!] \equiv [\![\#0_{\phi'}]\!]
\end{aligned}$$

The three *bitwise operators* "$\triangle_\phi$", "$\bigtriangledown_\phi$" and "$\bigtriangledown_\phi$" have a universally-defined meaning only if both of their operands have non-negative integer values. When applied to a pair of such well-formed operands under an appropriate integral format, these operations are equivalent to performing, respectively, the boolean operation "$\wedge$", "$\neq$" and "$\vee$" on the pairwise-corresponding bits in the binary representation of their two arguments. In Haskell, these requirements can be captured rather inefficiently as follows:

$$\begin{aligned}
&\text{EQV}_{\triangle} :: \forall\ x,\ y,\ \phi\ \Rightarrow \\
&\quad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\ \mathsf{Z}\} \wedge x \geq 0 \wedge y \geq 0]\!] \rightarrow \\
&\quad [\![\#x_\phi \triangle_\phi \#y_\phi]\!] \equiv [\![\#[\![\textstyle\sum[(x(k) \wedge y(k)) \times 2^i \mid k \leftarrow [0 \ldots \mathcal{W}(\phi) - 1]]]\!]_\phi]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\veebar} :: \forall\, x,\, y,\, \phi \Rightarrow \\
&\quad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\, \mathsf{Z}\} \wedge x \geq 0 \wedge y \geq 0]\!] \rightarrow \\
&\quad [\![\#x_\phi \veebar_\phi \#y_\phi]\!] \equiv [\![\#[\![\textstyle\sum[(x(k) \neq y(k)) \times 2^i \mid k \leftarrow [0 \,..\, \mathcal{W}(\phi) - 1]]]\!]_\phi]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\triangledown} :: \forall\, x,\, y,\, \phi \Rightarrow \\
&\quad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\, \mathsf{Z}\} \wedge x \geq 0 \wedge y \geq 0]\!] \rightarrow \\
&\quad [\![\#x_\phi \triangledown_\phi \#y_\phi]\!] \equiv [\![\#[\![\textstyle\sum[(x(k) \vee y(k)) \times 2^i \mid k \leftarrow [0 \,..\, \mathcal{W}(\phi) - 1]]]\!]_\phi]\!]
\end{aligned}$$

using the bit index notation "$x(k)$" and list summation operator "Σ" whose formal definition is given in Appendix A. Observe that only the least-significant $\mathcal{W}(\phi)$ bits of x and y are ever manipulated by these operations and that the result always represents a non-negative integer in the range of the respective format, provided that the same holds for each of the two operands to the construction.

The remaining two bit operations "$\#x_\phi \ll_\phi \#y_\phi$" and "$\#x_\phi \ll_\phi \#y_\phi$" are used to shift the bit pattern encoded in their first operand x left or right by the number of bits designated by the second argument y, effectively multiplying or dividing x by the integer quantity 2^y and rounding the resulting quotient towards $-\infty$. Both operations are only guaranteed to be defined if x has a non-negative value, y is both non-negative and less than the width of ϕ and, for the left shift operator "$\ll_\phi$", ϕ belongs to a natural genre "N". As expected for natural formats, the arithmetic is performed modulo $\text{lub}(\phi) + 1$, although this point is mute for the "$\gg_\phi$" operator, whose result $\lceil x/2^y \rceil$ is always smaller than x in magnitude. Formally, these provisions are captured by the following pair of theorems:

$$\begin{aligned}
&\text{EQV}_{\ll\mathsf{N}} :: \forall\, x,\, y,\, \varepsilon \Rightarrow \\
&\quad \text{WF}[\![\#x_{\mathsf{N}.\varepsilon}]\!] \rightarrow \text{WF}[\![\#y_{\mathsf{N}.\varepsilon}]\!] \rightarrow \\
&\quad [\![0 \leq y < \mathcal{W}[\![\mathsf{N}.\varepsilon]\!]]\!] \rightarrow \\
&\quad [\![\#x_{\mathsf{N}.\varepsilon} \ll_{\mathsf{N}.\varepsilon} \#y_{\mathsf{N}.\varepsilon}]\!] \equiv [\![\#[\![(x \times 2^y) \bmod (\text{lub}[\![\mathsf{N}.\varepsilon]\!] + 1)]\!]_{\mathsf{N}.\varepsilon}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\ll\mathsf{Z}} :: \forall\, x,\, y,\, \varepsilon \Rightarrow \\
&\quad \text{WF}[\![\#x_{\mathsf{Z}.\varepsilon}]\!] \rightarrow \text{WF}[\![\#y_{\mathsf{Z}.\varepsilon}]\!] \rightarrow \\
&\quad [\![0 \leq x \times 2^y \leq \text{lub}(\mathsf{Z}.\varepsilon) \wedge 0 \leq y < \mathcal{W}[\![\mathsf{Z}.\varepsilon]\!]]\!] \rightarrow \\
&\quad [\![\#x_{\mathsf{Z}.\varepsilon} \ll_{\mathsf{Z}.\varepsilon} \#y_{\mathsf{Z}.\varepsilon}]\!] \equiv [\![\#[\![x \times 2^y]\!]_{\mathsf{Z}.\varepsilon}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\gg} :: \forall\, x,\, y,\, \phi \Rightarrow \\
&\quad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{N},\, \mathsf{Z}\} \wedge x \geq 0 \wedge 0 \leq y < \mathcal{W}(\phi)]\!] \rightarrow \\
&\quad [\![\#x_\phi \gg_\phi \#y_\phi]\!] \equiv [\![\#[\![\lfloor x/2^y \rfloor]\!]_\phi]\!]
\end{aligned}$$

Similarly, the operations "$\#x_\phi <_\phi \#y_\phi$", "$\#x_\phi >_\phi \#y_\phi$", "$\#x_\phi \leq_\phi \#y_\phi$" and "$\#x_\phi \geq_\phi \#y_\phi$" are always equivalent to "$\#z_{\mathsf{Z}.\Phi}$", where z is equal to 1 whenever the corresponding relation holds for its two operands and 0 otherwise, provided that ϕ represents a well-formed format from any genre other than "F":

$$\begin{aligned}
&\text{EQV}_{<} :: \forall\, x,\, y,\, \phi \Rightarrow \\
&\quad \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \neq [\![\mathsf{F}]\!]]\!] \rightarrow \\
&\quad [\![\#x_\phi <_\phi \#y_\phi]\!] \equiv [\![\#[\![x < y]\!]_{\mathsf{Z}.\Phi}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{>} :: \forall\ x,\ y,\ \phi\ \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\phi}]\!] \rightarrow \text{WF}[\![\#y_{\phi}]\!] \rightarrow \\
&\qquad [\![\gamma(\phi) \neq [\![\text{F}]\!]]\!] \rightarrow \\
&\qquad [\![\#x_{\phi} >_{\phi} \#y_{\phi}]\!] \equiv [\![\#[\![x > y]\!]_{\text{Z.}\Phi}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\leq} :: \forall\ x,\ y,\ \phi\ \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\phi}]\!] \rightarrow \text{WF}[\![\#y_{\phi}]\!] \rightarrow \\
&\qquad [\![\gamma(\phi) \neq [\![\text{F}]\!]]\!] \rightarrow \\
&\qquad [\![\#x_{\phi} \leq_{\phi} \#y_{\phi}]\!] \equiv [\![\#[\![x \leq y]\!]_{\text{Z.}\Phi}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\geq} :: \forall\ x,\ y,\ \phi\ \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\phi}]\!] \rightarrow \text{WF}[\![\#y_{\phi}]\!] \rightarrow \\
&\qquad [\![\gamma(\phi) \neq [\![\text{F}]\!]]\!] \rightarrow \\
&\qquad [\![\#x_{\phi} \geq_{\phi} \#y_{\phi}]\!] \equiv [\![\#[\![x \geq y]\!]_{\text{Z.}\Phi}]\!]
\end{aligned}$$

Little can be said in the generic fragment of Etude semantics about the meanings of atoms constructed under object formats from the "O" genre. Nevertheless, every implementation of the language is required to preserve the following four simple equivalence relations:

$$\begin{aligned}
&\text{EQV}_{+0} :: \forall\ x,\ \varepsilon\ \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\text{O.}\varepsilon}]\!] \rightarrow \\
&\qquad [\![\#x_{\text{O.}\varepsilon}]\!] \equiv [\![\#x_{\text{O.}\varepsilon} +_{\text{O.}\varepsilon} \#0_{\text{Z.}\Phi}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{+\text{O}} :: \forall\ x,\ m,\ n,\ \varepsilon\ \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\text{O.}\varepsilon} +_{\text{O.}\varepsilon} \#m_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}[\![\#m_{\text{Z.}\Phi} +_{\text{Z.}\Phi} \#n_{\text{Z.}\Phi}]\!] \rightarrow \\
&\qquad [\![(\#x_{\text{O.}\varepsilon} +_{\text{O.}\varepsilon} \#n_{\text{Z.}\Phi}) +_{\text{O.}\varepsilon} \#m_{\text{Z.}\Phi}]\!] \equiv [\![\#x_{\text{O.}\varepsilon} +_{\text{O.}\varepsilon} (\#m_{\text{Z.}\Phi} +_{\text{Z.}\Phi} \#n_{\text{Z.}\Phi})]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{-\text{O}} :: \forall\ x,\ m,\ n,\ \varepsilon\ \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\text{O.}\varepsilon} +_{\text{O.}\varepsilon} \#m_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}[\![\#x_{\text{O.}\varepsilon} +_{\text{O.}\varepsilon} \#n_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}[\![\#m_{\text{Z.}\Phi} -_{\text{Z.}\Phi} \#n_{\text{Z.}\Phi}]\!] \rightarrow \\
&\qquad [\![(\#x_{\text{O.}\varepsilon} +_{\text{O.}\varepsilon} \#m_{\text{Z.}\Phi}) -_{\text{O.}\varepsilon} (\#x_{\text{O.}\varepsilon} +_{\text{O.}\varepsilon} \#n_{\text{Z.}\Phi})]\!] \equiv [\![\#m_{\text{Z.}\Phi} -_{\text{Z.}\Phi} \#n_{\text{Z.}\Phi}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EQV}_{\text{OO}} :: \forall\ x,\ \varepsilon\ \Rightarrow \\
&\qquad \text{WF}[\![\#x_{\text{O.}\varepsilon}]\!] \rightarrow \\
&\qquad [\![\#x_{\text{O.}\varepsilon}]\!] \equiv [\![\text{O.}\varepsilon_{\text{O.}\Phi}(\text{O.}\Phi_{\text{O.}\varepsilon}(\#x_{\text{O.}\varepsilon}))]\!]
\end{aligned}$$

Intuitively, these state that every valid atomic form "$\#x_{\text{O.}\varepsilon}$" can be also interpreted as an object sum "$\#y_{\text{O.}\varepsilon} +_{\text{O.}\varepsilon} \#n_{\text{Z.}\Phi}$", in which y and n are known as the atom's *base* and *offset*, respectively. Under this interpretation, every constant object atom "$\#x_{\text{O.}\varepsilon}$" has the base x with an implicit offset of 0. Further, the sum of two object atoms with the same base is always equivalent to an atom with the same base and the sum of the operands' individual offsets. More so, the difference of two atoms with equal base components is always equal to the difference of their offset values, provided that, in all cases, every intermediate step of these computations represents a well-formed Etude entity. Finally, every well-formed object atom retains its meaning after round-trip conversion through the standard object format "O.Φ". A motivation for such interpretation of these constructs can be found in Section 4.5 later in this chapter.

Last but not least, the portable fragment of Etude's atomic semantics always assigns a very specific meaning to the four comparison operators "$=_{\phi}$", "$\neq_{\phi}$", "$<_{\phi}$", "$>_{\phi}$", "$\leq_{\phi}$" and "$\geq_{\phi}$". In particular, for all well-formed values of "$\#x_{\phi}$" and "$\#y_{\phi}$", the atom "$\#x_{\phi} =_{\phi} \#y_{\phi}$" is always equivalent to "$\#z_{\text{Z.}\Phi}$", where z has the value of 1 if x and y have

identical numeric values and 0 otherwise. Conversely, under the same circumstances, the "$\neq_\phi$" operator delivers 0 if its operands have equal values and 1 otherwise. Both constraints can be expressed concisely by the following pair of theorems:

$$\text{EQV}_{=} :: \forall\, x,\, y,\, \phi \Rightarrow \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\#y_\phi]\!] \rightarrow [\![\#x_\phi =_\phi \#y_\phi]\!] \equiv [\![\#[\![x = y]\!]_{\text{Z.}\Phi}]\!]$$
$$\text{EQV}_{\neq} :: \forall\, x,\, y,\, \phi \Rightarrow \text{WF}[\![\#x_\phi]\!] \rightarrow \text{WF}[\![\#y_\phi]\!] \rightarrow [\![\#x_\phi \neq_\phi \#y_\phi]\!] \equiv [\![\#[\![x \neq y]\!]_{\text{Z.}\Phi}]\!]$$

An astute reader will rightly observe that, in most of the above constraint theorems, well-formed atoms of the form "$\#x_\phi$" are singled out as particularly interesting. In this work, such atoms are said to represent *immediate constants* and are identified formally by the following Haskell predicate function:

$$\text{IMM}[\![\cdot]\!] :: (\text{ord}\ \nu) \Rightarrow \mathit{atom}_\nu \rightarrow \mathit{bool}$$

$$\begin{aligned}\text{IMM}[\![\#x_\phi]\!] &= \text{true}\\ \text{IMM}[\![\mathit{other}]\!] &= \text{false}\end{aligned}$$

On all sensible Etude implementations, every well-formed atom α is always equivalent to some immediate constant α' that can be viewed as the atom's fully-reduced normal form. In this work, this constant is represented by the notation "$\mathcal{E}(\alpha)$", using the *evaluation function* $\mathcal{E}$ with the following Haskell signature:

$$\mathcal{E}[\![\cdot]\!] :: (\text{ord}\ \nu) \Rightarrow \mathit{atom}_\nu \rightarrow \mathit{atom}_\nu$$

and the obvious trio of semantic properties:

$$\begin{aligned}\text{WF}_{\mathcal{E}} &:: \forall\, \alpha \Rightarrow \text{WF}(\alpha) \rightarrow \text{WF}[\![\mathcal{E}(\alpha)]\!]\\ \text{IMM}_{\mathcal{E}} &:: \forall\, \alpha \Rightarrow \text{WF}(\alpha) \rightarrow [\![\text{IMM}(\mathcal{E}(\alpha))]\!]\\ \text{EQV}_{\mathcal{E}} &:: \forall\, \alpha_1,\, \alpha_2 \Rightarrow \text{WF}(\alpha_1) \rightarrow \text{WF}(\alpha_2) \rightarrow [\![\mathcal{E}(\alpha_1) = \mathcal{E}(\alpha_2)]\!] \rightarrow (\alpha_1 \equiv \alpha_2)\end{aligned}$$

Intuitively, $\mathcal{E}$ represents the *operational semantics* of Etude atoms, which must be defined individually by every complete Etude implementation. In Chapter 6, one such definition is provided for the MMIX-specific incarnation of the language. In contrast, the above constraints on the atomic well-formedness and equivalence properties constitutes the *algebraic semantics* of the portable Etude fragment, whose consistency with the operational formulation is asserted by appropriate proofs of these properties given in Section 6.3.

Although the above algebraic semantics may, at first sight, appear somewhat arbitrary, a closer scrutiny will quickly reveal that they describe precisely the minimal set of requirements imposed on arithmetic operations by the ISO C Standard [ANSI 89], whose accurate formalisation is the ultimate purpose of the Etude program representation. If we were to apply the same methodology in a description of some other programming language, these algebraic semantics would, in all likelihood require a substantial reworking, so that, in its present format, the calculus remains unsuitable for a broader deployment in other compiler verification projects.

An astute reader may also observe that the resulting arithmetic model is incomplete, in that it readily supports a rather unhelpful implementation of Etude that places all atoms in the same single equivalence class. However, Chapter 6 bears witness to the

fact that at least one sensible operational model consistent with the above constraints can be formulated for the language. Since all Etude programs that aspire for complete portability must conform to every admissible operational interpretation of their syntax, they must, by definition, also retain their validity on MMIX. Accordingly, the generic Etude algebra presented in this section remains sufficient for a successful depiction of the meaning of all programs conforming to the C Standard, while its rudimentary nature readily explains the difficulty of constructing such programs within the Spartan framework of ANSI C.

4.5 State of the Matter

Atoms are not the only kind of primitive entities manipulate by Etude. For better or worse, most modern computational hardware is explicitly geared towards stateful operations on mutable objects, whose individual meanings may, contrary to all sensibilities of the pure lambda calculus, change gradually throughout execution of a program.

The simplest and most popular way of modelling the dynamic semantics of such rogue primitives is through the notion of an *execution state*, an abstract object that is implicitly received, modified and returned by every function in the program. In Etude, the execution state consists of two distinct components known individually as the *function* and *object environments* which, for conciseness, are generally represented by the Greek letters Λ and Δ, or collectively as a tuple of the form "(Λ, Δ)".

Intuitively, a function environment Λ represents an oracle that encodes a potentially unbounded set of Etude functions using a finite number of distinct atomic values, by mapping every abstraction λ that actually appears within a program's definition to a predetermined atom $\Lambda(\lambda)$, so that every every first-class entity in the Etude language is always identified with a suitable atomic value, which greatly simplifies definition of a bijective mapping between Etude programs and executable machine instruction sequences exemplified later in Chapter 6. Formally, the function environment constitutes a mapping of atoms to syntactic representations of lambda abstractions as described by the following Haskell data types:

$$\begin{aligned}
&\textit{f-env}_{[\![v]\!]}: \\
&\qquad \textit{integer} \mapsto \textit{function}_v \\
&\textit{function}_{[\![v]\!]}: \\
&\qquad \lambda\ \textit{parameters}_v\ .\ \textit{term}_v \\
&\textit{parameters}_{[\![v]\!]}: \\
&\qquad [v]
\end{aligned}$$

in which the notation "$A \mapsto B$" represents a polymorphic Haskell type that describes an extendible partial function from A to B, or a mapping from a finite set of Haskell values of type A to values of type B. It is implemented separately in Appendix A with the help of the existing standard Haskell libraries. In particular, the above specification of $\textit{f-env}_v$ defines the function encoding oracle as a mapping of integers to Etude abstractions of

the form "$\lambda x_1, x_2 \ldots x_n.\tau$", where the structure of the abstraction body τ is depicted by the data type "$term_\nu$" scrutinised later in Section 4.6. In general, the Λ oracle is constructed by the system's linker in an unspecified manner by enumerating all functions defined within the individual program modules, although we could easily extend Etude to accommodate more elaborate instruction set architectures which permit dynamic introduction of new functions during the actual evaluation of a program.

In a well-formed function environment Λ, every integer $n \in \mathrm{dom}(\Lambda)$ is mapped to a well-formed closed Etude function, with 0 generally excluded from Λ's domain. Formally, these requirements can be captured concisely by the following property type:

$$\begin{aligned}&\text{data } \mathrm{WF}[\![\cdot]\!] :: \forall\ \nu \Rightarrow \textit{f-env}_\nu \to \star\\ &\quad \text{where } \mathrm{WF}_\Lambda :: \forall\ \Lambda \Rightarrow [\![0 \notin \mathrm{dom}(\Lambda)]\!] \to\\ &\qquad\qquad (\forall\ n \to [\![n \in \mathrm{dom}(\Lambda)]\!] \to \mathrm{WF}[\![\Lambda(n)]\!]) \to\\ &\qquad\qquad (\forall\ n \to [\![n \in \mathrm{dom}(\Lambda)]\!] \to [\![\mathrm{FV}(\Lambda(n)) = \varnothing]\!]) \to\\ &\qquad\qquad \mathrm{WF}(\Lambda)\end{aligned}$$

where the well-formedness of individual functions is determined inductively from well-formedness of their parameter lists as follows:

$$\begin{aligned}&\text{data } \mathrm{WF}[\![\cdot]\!] :: \forall\ \nu \Rightarrow \textit{function}_\nu \to \star\\ &\quad \text{where } \mathrm{WF}_\lambda :: \forall\ \bar{\nu},\ \tau \Rightarrow \mathrm{WF}(\bar{\nu}) \to \mathrm{WF}[\![\lambda\bar{\nu}.\tau]\!]\end{aligned}$$

observing that the actual body term τ need not always be well-formed in such entities. A list of variables represents a well-formed parameter list if and only if it actually constitutes a well-formed set, i.e., if its length is equal to the cardinality of a set constructed from these variable names. Formally:

$$\begin{aligned}&\text{data } \mathrm{WF}[\![\cdot]\!] :: \forall\ \nu \Rightarrow \textit{parameters}_\nu \to \star\\ &\quad \text{where } \mathrm{WF}_\nu :: \forall\ \bar{\nu} \Rightarrow [\![\mathrm{length}(\bar{\nu}) = |\bar{\nu}|]\!] \to \mathrm{WF}[\![\bar{\nu}]\!]\end{aligned}$$

Finally, the set of variables appearing free within an Etude function includes every free variable from its body term, excluding any identifiers found directly in the function's parameter list. In Haskell, this is captured by the following natural definition:

$$\begin{aligned}&\mathrm{FV}[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \textit{function}_\nu \to \{\nu\}\\ &\mathrm{FV}[\![\lambda\bar{\nu}.\tau]\!] = \mathrm{FV}(\tau)\backslash\bar{\nu}\end{aligned}$$

The structure and algebraic properties of Etude terms appearing in function bodies is discussed later in Section 4.6. For now, we will only characterise them by the simple substitution construct defined on all well-formed Etude functions as follows:

$$\begin{aligned}&[\![\cdot]\!]/[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \textit{function}_\nu \to (\nu \mapsto \textit{atom}_\nu) \to \textit{function}_\nu\\ &[\![\lambda\bar{\nu}.\tau]\!]/[\![S]\!] = [\![\lambda\bar{\nu}.[\![\tau/(S\backslash\bar{\nu})]\!]]\!]\end{aligned}$$

Unlike the function environment described above, the structure and properties of the *object environment* Δ threaded throughout the program's execution as part of its evaluation state are left mostly-unspecified in the generic fragment of Etude's algebraic semantics, since their complete implementation requires a great deal of knowledge about

the precise operational behaviour of all memory access operations supported by the underlying instruction set architecture. Intuitively, an object environment describes the address space layout of an Etude program under execution and, further, depicts the current value of every memory-resident *mutable object* manipulated by monadic Etude terms in accordance with the semantic rules defined later in Section 4.6.

Regardless of any particular implementation-specific memory organisation regime, a generic abstract model of an object environment Δ can be described by a set $\bar{\xi}(\Delta)$ known as the environment's *envelope*, a list $\bar{\psi}(\Delta)$ known as its *data stack* and a partial function $\bar{\alpha}(\Delta)$ that depicts bindings of the individual memory-resident objects to their current contents. Formally, every envelope $\bar{\xi}$ consists of a set of tuples $(\sigma_k,\ \phi_k,\ \bar{\mu}_k)$, in which the integer σ_k represents some concrete memory location, the format ϕ_k describes the desired interpretation of data stored at that address and $\bar{\mu}_k$ depicts a set of zero or more *memory access attributes* that further constrain the range of operations permitted on that address space region. In Haskell, the type of every Etude envelope can be defined as follows:

envelope:
    {*envelope-element*}

envelope-element:
    (*integer*, *format*, {*mode*})

mode: one of
    `c` `v`

Intuitively, each envelope element $(\sigma_k,\ \phi_k,\ \bar{\mu}_k)$ characterises all operational properties of the address space region $[\sigma_k \ldots \sigma_k + \mathcal{S}(\phi_k) - 1]$, observing that entries found in distinct envelopes may, on occasions, refer to overlapping regions in the program's memory image.

The set of access attributes $\bar{\mu}$ associated with each envelope element provides additional information about the precise semantics of the corresponding memory location. In particular, portable Etude programs are universally prohibited from modifying values of objects tagged with the *constant attribute* "`c`" and, further, may never assume any particular behaviour for those object which, in the program's address space, are tagged with the *volatile attribute* "`v`", since any such addresses may correspond to *shared memory regions* associated with memory-mapped registers and buffers of various hardware devices, whose precise semantics remain outside of the program's explicit control.

In every well-formed envelope, each element $(\sigma,\ \phi,\ \bar{\mu})$ must entail the validity of the format ϕ, the set of attributes $\bar{\mu}$ and the object atom "$\#\sigma_{[\![\mathcal{O}(\phi)]\!]}$", so that σ must, in effect, represent a well-formed pointer to some ϕ-formatted memory-resident object. Formally:

$\text{data } \mathrm{WF}[\![\cdot]\!] :: \textit{envelope-element} \rightarrow \star$

$\text{where } \mathrm{WF} :: \forall\ \sigma,\ \phi,\ \bar{\mu} \Rightarrow \mathrm{WF}(\phi) \rightarrow \mathrm{WF}(\bar{\mu}) \rightarrow \mathrm{WF}[\![\#\sigma_{[\![\mathcal{O}(\phi)]\!]}]\!] \rightarrow \mathrm{WF}[\![(\sigma,\ \phi,\ \bar{\mu})]\!]$

in which the notation "$O(\phi)$" provides a mapping of each Etude format ϕ to a designated object format suitable for depiction of address values for ϕ-formatted objects:

$$O[\![\cdot]\!] :: \mathit{format} \rightarrow \mathit{format}$$

The precise size $\mathcal{S}(\bar{\xi})$ of an entire memory region described by a given envelope $\bar{\xi}$ is defined as a difference between the greatest and smallest address value corresponding to any of its individual envelope elements, or zero if $\bar{\xi}$ represents an empty set. Formally:

$$\mathcal{S}[\![\cdot]\!] :: \mathit{envelope} \rightarrow \mathit{integer}$$

$$\begin{aligned}\mathcal{S}[\![\bar{\xi}]\!] \;&|\; (\bar{\xi} = \varnothing) = 0\\ &|\; (\bar{\xi} \neq \varnothing) = \max[\sigma_k + \mathcal{S}(\phi_k) \;|\; (\sigma_k,\, \phi_k,\, \bar{\mu}_k) \leftarrow \bar{\xi}] - \min[\sigma_k \;|\; (\sigma_k,\, \phi_k,\, \bar{\mu}_k) \leftarrow \bar{\xi}]\end{aligned}$$

A given envelope $\bar{\xi}$ may be also *shifted* into a particular memory location σ by adding $n - \sigma_0$ to the the offset σ_k of every envelope element $(\sigma_k,\, \phi_k,\, \bar{\mu}_k) \in \bar{\xi}$, where σ_0 represents the least address of any element in $\bar{\xi}$. In Haskell, this simple construction can be executed as follows:

$$[\![\cdot]\!] \oplus [\![\cdot]\!] :: \mathit{envelope} \rightarrow \mathit{integer} \rightarrow \mathit{envelope}$$

$$\begin{aligned}[\![\bar{\xi}]\!] \oplus [\![n]\!] \;&= \{(\sigma_k + n - \sigma_0,\, \phi_k,\, \bar{\mu}_k) \;|\; (\sigma_k,\, \phi_k,\, \bar{\mu}_k) \leftarrow \bar{\xi}\}\\ &\text{where } \sigma_0 = \min[\sigma_k \;|\; (\sigma_k,\, \phi_k,\, \bar{\mu}_k) \leftarrow \bar{\xi}]\end{aligned}$$

Finally, an envelope $\bar{\xi}$ may be purged of any elements that overlap a given memory region $[\sigma_i \,...\, \sigma_f - 1]$ as follows:

$$[\![\cdot]\!] \backslash [\![\cdot]\!] :: \mathit{envelope} \rightarrow (\mathit{integer}, \mathit{integer}) \rightarrow \mathit{envelope}$$

$$[\![\bar{\xi}]\!] \backslash [\![\sigma_i,\, \sigma_f]\!] = \{(\sigma_k,\, \phi_k,\, \bar{\mu}_k) \;|\; (\sigma_k,\, \phi_k,\, \bar{\mu}_k) \leftarrow \bar{\xi},\, \sigma_k + \mathcal{S}(\phi_k) \leq \sigma_i \vee \sigma_k \geq \sigma_f\}$$

The use of access attribute tags in individual envelope elements introduces a number of challenges that are best met by formalising the notion of an *element accessibility* as the binary relation "$\xi_k \in_{\mathrm{A}} \bar{\xi}$". In particular, a given element $(\sigma,\, \phi,\, \bar{\mu})$ is said to be *accessible* in $\bar{\xi}$ if and only if that envelope contains at least one element of the form $(\sigma,\, \phi,\, \bar{\mu}')$ such that $\bar{\mu}' \subseteq \bar{\mu}$. Intuitively, this definition ensures that every memory access operation performed by an Etude program must always specify at least the complete attribute set associated with the targeted object in the program's address space. Formally, this operator is defined in Haskell as follows:

$$[\![\cdot]\!] \in_{\mathrm{A}} [\![\cdot]\!] :: \mathit{envelope\text{-}element} \rightarrow \mathit{envelope} \rightarrow \mathit{bool}$$

$$[\![\sigma,\, \phi,\, \bar{\mu}]\!] \in_{\mathrm{A}} [\![\bar{\xi}]\!] = \bigvee[\sigma_k = \sigma \wedge \phi_k = \phi \wedge \bar{\mu}_k \subseteq \bar{\mu} \;|\; (\sigma_k,\, \phi_k,\, \bar{\mu}_k) \leftarrow \bar{\xi}]$$

Besides the address space envelope $\bar{\xi}(\Delta)$ mentioned earlier, every well-formed object environment also includes a second envelope $\bar{\xi}_i(\Delta)$ whose elements describe all currently uninitialised memory objects introduced by the program. Formally, both notations are given a formal mandate by the following pair of implementation-defined Haskell functions:

$$\bar{\xi}[\![\cdot]\!],\, \bar{\xi}_i[\![\cdot]\!] :: \mathit{o\text{-}env} \rightarrow \mathit{envelope}$$

Further, in order to facilitate establishment of the linear correctness property for our translation of Etude into the MMIX architecture in Chapter 6, it is necessary to restrict well-formed Etude programs to certain predictable patterns of memory allocations. To this end, our abstract model of object environments includes an additional *stack* component $\bar{\psi}(\Delta)$ that, intuitively, contains the complete history of all individual envelopes introduced into $\bar{\xi}(\Delta)$ by the program as a list of *stack frames*, every one of which is represented simply by an envelope tagged with its allotted location within $\bar{\xi}(\Delta)$. Formally, the structure of such a stack can be depicted by the following pair of Haskell data types:

stack :
    [*stack-frame*]

stack-frame :
    (*integer*, *envelope*)

so that the notation "$\bar{\psi}(\Delta)$" can be formalised by some implementation-defined construction with the following type signature:

$\bar{\psi}[\![\cdot]\!] :: \textit{o-env} \rightarrow \textit{stack}$

Finally, the actual content of all initialised address space regions within a given object environment Δ can be depicted by a partial function $\bar{\alpha}(\Delta \triangleright \sigma, \phi)$ that maps every valid pair of the form "(σ, ϕ)" to its concrete atomic value as follows:

$\bar{\alpha}[\![\cdot]\!] :: (\text{ord } v) \Rightarrow (\textit{o-env} \triangleright \textit{integer}, \textit{format}) \rightarrow \textit{atom}_v$

The precise behaviour of this function will be, for most part, left unspecified in the generic fragment of the Etude language. Intuitively, a given memory object (σ, ϕ) is said to be *populated* in Δ if and only if $\bar{\alpha}(\Delta \triangleright \sigma, \phi)$ represents a well-formed Etude atom, in which case that construct also determines the object's current content in Δ.

Although, in general, we leave formulation of the precise validity criteria for object environments at the mercy of individual Etude implementations, every well-formed Δ must always specify a valid envelope $\bar{\xi}(\Delta)$ that includes at least all of the envelope elements from $\bar{\xi}_i(\Delta)$, together with any envelopes that are mentioned by the individual stack frames of $\bar{\psi}(\Delta)$. Formally:

$\text{ENV}_\text{E} :: \forall\, \Delta \Rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\bar{\xi}(\Delta)]\!]$
$\text{ENV}_\text{I} :: \forall\, \Delta \Rightarrow \text{WF}(\Delta) \rightarrow [\![\bar{\xi}_i(\Delta) \subseteq \bar{\xi}(\Delta)]\!]$
$\text{ENV}_\text{S} :: \forall\, \Delta \Rightarrow \text{WF}(\Delta) \rightarrow [\![\bigcup[\bar{\xi}_k \oplus \sigma_k \mid (\sigma_k, \bar{\xi}_k) \leftarrow \bar{\psi}(\Delta)] \subseteq \bar{\xi}(\Delta)]\!]$

The actual introduction of a new envelope $\bar{\xi}$ into an existing object environment Δ is modelled by an *environment extension operation* of the form "$\Delta/\bar{\xi}$", while the dual *environment contraction* "$\Delta\backslash\bar{\xi}$" can be used to purge such an envelope from Δ. In the algebraic semantics of Etude, the precise behaviour of these operations is left open to further specification by individual Etude implementations, which should always assign concrete definition to the following pair of Haskell functions:

$[\![\cdot]\!]/[\![\cdot]\!] :: \textit{o-env} \rightarrow \textit{envelope} \rightarrow \textit{o-env}$
$[\![\cdot]\!]\backslash[\![\cdot]\!] :: \textit{o-env} \rightarrow \textit{envelope} \rightarrow \textit{o-env}$

In all cases, however, a well-formed extension Δ' of some existing environment Δ by an envelope $\bar{\xi}$ introduces a new stack frame $(\sigma_c,\ \bar{\xi})$ into $\bar{\psi}(\Delta')$, whereby it is inserted at the front of the existing stack frame list $\bar{\psi}(\Delta)$. Further, the new envelope is included in both $\bar{\xi}(\Delta')$ and $\bar{\xi}_i(\Delta')$, after shifting it into place by its concrete location σ_c chosen in some unspecified manner by the underlying instruction set architecture. Formally, these constraints on the behaviours of the environment extension operator are captured by the following three theorems:

$$\begin{aligned}
&\text{EXT}_\text{S} :: \forall\ \Delta,\ \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow \text{WF}(\Delta/\bar{\xi}) \rightarrow [\![\bar{\psi}(\Delta/\bar{\xi}) = (\sigma_c(\Delta \triangleright \bar{\xi}),\ \bar{\xi}) +\!\!+ \bar{\psi}(\Delta)]\!] \\
&\text{EXT}_\text{E} :: \forall\ \Delta,\ \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow \text{WF}(\Delta/\bar{\xi}) \rightarrow [\![\bar{\xi}(\Delta/\bar{\xi}) = \bar{\xi}(\Delta) \cup (\bar{\xi} \oplus \sigma_c(\Delta \triangleright \bar{\xi}))]\!] \\
&\text{EXT}_\text{I} :: \forall\ \Delta,\ \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow \text{WF}(\Delta/\bar{\xi}) \rightarrow [\![\bar{\xi}_i(\Delta/\bar{\xi}) = \bar{\xi}_i(\Delta) \cup (\bar{\xi} \oplus \sigma_c(\Delta \triangleright \bar{\xi}))]\!]
\end{aligned}$$

in which the actual location of $\bar{\xi}$ within the resulting address space of $\Delta/\bar{\xi}$ is represented by the notation "$\sigma_c(\Delta \triangleright \bar{\xi})$", mandated by the following implementation-defined Haskell function:

$$\sigma_c[\![\cdot]\!] :: (\textit{o-env} \triangleright \textit{envelope}) \rightarrow \textit{integer}$$

In every other respects, the environment extension preserves all other formal properties of the original environment Δ. In particular, all atomic bindings from Δ that do not overlap the newly-introduced envelope $\bar{\xi}$ always remain equivalent to their corresponding meanings in the extended environment $\Delta/\bar{\xi}$, provided that both Δ and $\Delta/\bar{\xi}$ are deemed well-formed according to the implementation-defined rules outlined earlier:

$$\begin{aligned}
&\text{EXT}_\text{A} :: \forall\ \Delta,\ \bar{\xi},\ \sigma_k,\ \phi_k,\ \bar{\mu}_k \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \text{WF}(\Delta/\bar{\xi}) \rightarrow [\![(\sigma_k,\ \phi_k,\ \bar{\mu}_k) \in \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad [\![\bar{\alpha}(\Delta/\bar{\xi} \triangleright \sigma_k,\ \phi_k)]\!] \equiv [\![\bar{\alpha}(\Delta \triangleright \sigma_k,\ \phi_k)]\!]
\end{aligned}$$

Conversely, the dual operation $\Delta\backslash\bar{\xi}$ purges the most recently allocated envelope $\bar{\xi}$ from the address space of Δ. In fact, such operations are always guaranteed to be well-formed if the specified envelope actually appears at the top of the existing stack $\bar{\psi}(\Delta)$, a property that can be captured in Haskell by the following theorem:

$$\text{CON}_\Delta :: \forall\ \Delta,\ \sigma_c,\ \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\ \bar{\xi}))]\!] \rightarrow \text{WF}[\![\Delta\backslash\bar{\xi}]\!]$$

After every such well-formed contraction operation, the specified stack frame is popped or removed from the top of $\bar{\psi}(\Delta)$ and the associated envelope is purged from the environment's two envelopes $\bar{\xi}(\Delta)$ and $\bar{\xi}_i(\Delta)$:

$$\begin{aligned}
&\text{CON}_\text{S} :: \forall\ \Delta,\ \sigma_c,\ \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\ \bar{\xi}))]\!] \rightarrow [\![\bar{\psi}(\Delta\backslash\bar{\xi}) = \text{tail}(\bar{\psi}(\Delta))]\!] \\
&\text{CON}_\text{E} :: \forall\ \Delta,\ \sigma_c,\ \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\ \bar{\xi}))]\!] \rightarrow [\![\bar{\xi}(\Delta\backslash\bar{\xi}) = \bar{\xi}(\Delta)\backslash(\bar{\xi} \oplus \sigma_c)]\!] \\
&\text{CON}_\text{I} :: \forall\ \Delta,\ \sigma_c,\ \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\ \bar{\xi}))]\!] \rightarrow [\![\bar{\xi}_i(\Delta\backslash\bar{\xi}) = \bar{\xi}_i(\Delta)\backslash(\bar{\xi} \oplus \sigma_c)]\!]
\end{aligned}$$

Most readers will readily recognise a correspondence between the above theorem and the actual data stack regime imposed on the computational models of most modern instruction set architectures. Such detailed formalisation of this model proves critical to a successful treatment of certain rogue machine instruction sequences in a definition of their semantics through a mapping of assembly programs onto Etude functions, as mandated by the linear correctness principle from Chapter 2. Nevertheless, the

reader should observe that, in most cases, neither the envelope $\bar{\xi}(\Delta)$ nor the stack $\bar{\psi}(\Delta)$ feature in the actual hardware implementation of the program's address space, since, strictly speaking, these components are not required for a successful evaluation of well-formed Etude programs. Nevertheless, they are included in the above format model, in order to restrain meaningful Etude programs from adopting certain patterns of memory allocation that would render verification of the linear correctness principle all but impossible in practice.

A well-formed contraction of the form "$\Delta\backslash\bar{\xi}$" is also guaranteed to preserve the content of every memory-resident object that, in Δ, is stored in a previously-allocated region $(\sigma_k,\ \phi_k,\ \bar{\mu}_k) \in \bar{\xi}(\Delta)$ of the program's address space, provided that this region does not overlap any of the elements in the envelope $\bar{\xi}$ that is being exterminated by the contraction operation:

$$\begin{aligned}
&\text{CON}_\text{A} :: \forall\ \Delta,\ \sigma_c,\ \bar{\xi},\ \sigma_k,\ \phi_k,\ \bar{\mu}_k \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \\
&\qquad [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\ \bar{\xi}))]\!] \rightarrow \\
&\qquad [\![(\sigma_k,\ \phi_k,\ \bar{\mu}_k) \in \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad [\![\sigma_k + \mathcal{S}(\phi_k) \leq \sigma_c \vee \sigma_k \geq \sigma_c + \mathcal{S}(\bar{\xi})]\!] \rightarrow \\
&\qquad [\![\bar{\alpha}(\Delta\backslash\bar{\xi} \triangleright \sigma_k,\ \phi_k)]\!] \equiv [\![\bar{\alpha}(\Delta \triangleright \sigma_k,\ \phi_k)]\!]
\end{aligned}$$

Finally, an Etude program may populate the individual objects in its address space with concrete atomic values using an *environment update* operation of the general form "$\Delta/(\sigma,\ \phi,\ \alpha)$", as represented by the following implementation-defined function:

$$[\![\cdot]\!]/[\![\cdot]\!] :: (\text{ord } v) \Rightarrow \textit{o-env} \rightarrow (\textit{integer}, \textit{format}, \textit{atom}_v) \rightarrow \textit{o-env}$$

In plain terms, $\Delta/(\sigma,\ \phi,\ \alpha)$ constructs a new environment Δ' in which the given object $(\sigma,\ \phi)$ is populated with the specified atom α and the associated envelope element is removed from $\bar{\xi}_i(\Delta')$, provided that the resulting object environment is valid and that $\bar{\xi}(\Delta)$ includes some element of the form "$(\sigma,\ \phi,\ \bar{\mu})$". Formally:

$$\begin{aligned}
&\text{SET}_\alpha :: \forall\ \Delta,\ \sigma,\ \phi,\ \alpha,\ \bar{\mu} \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma,\ \phi,\ \alpha)]\!] \rightarrow [\![(\sigma,\ \phi,\ \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad [\![\bar{\alpha}(\Delta/(\sigma,\ \phi,\ \alpha) \triangleright \sigma,\ \phi)]\!] \equiv [\![\alpha]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{SET}_\text{I} :: \forall\ \Delta,\ \sigma,\ \phi,\ \alpha,\ \bar{\mu} \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma,\ \phi,\ \alpha)]\!] \rightarrow [\![(\sigma,\ \phi,\ \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad [\![\bar{\xi}_i(\Delta/(\sigma,\ \phi,\ \alpha) \triangleright \sigma,\ \phi) = \bar{\xi}_i(\Delta)\backslash(\sigma,\ \sigma + \mathcal{S}(\phi))]\!]
\end{aligned}$$

Such well-formed object update operations will never affect the contents of any non-overlapping memory-resident objects currently allocated in the same address space:

$$\begin{aligned}
&\text{SET}_{\bar{\alpha}} :: \forall\ \Delta,\ \sigma,\ \phi,\ \alpha,\ \bar{\xi},\ \sigma_k,\ \phi_k,\ \bar{\mu}_k \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma,\ \phi,\ \alpha)]\!] \rightarrow \\
&\qquad [\![(\sigma,\ \ \phi,\ \ \bar{\mu})\ \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad [\![(\sigma_k,\ \phi_k,\ \bar{\mu}_k) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad [\![\sigma_k + \mathcal{S}(\phi_k) \leq \sigma \vee \sigma_k \geq \sigma + \mathcal{S}(\phi)]\!] \rightarrow \\
&\qquad [\![\bar{\alpha}(\Delta/(\sigma,\ \phi,\ \alpha) \triangleright \sigma_k,\ \phi_k)]\!] \equiv [\![\bar{\alpha}(\Delta \triangleright \sigma_k,\ \phi_k)]\!]
\end{aligned}$$

and, further, they always preserve all other properties of the original environment as follows:

$$\begin{aligned}
&\mathrm{SET_E} :: \forall\, \Delta,\, \sigma,\, \phi,\, \alpha,\, \bar{\mu} \Rightarrow \\
&\quad \mathrm{WF}(\Delta) \to \mathrm{WF}[\![\Delta/(\sigma,\, \phi,\, \alpha)]\!] \to [\![(\sigma,\, \phi,\, \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \to \\
&\quad [\![\bar{\xi}(\Delta/(\sigma,\, \phi,\, \alpha)) = \bar{\xi}(\Delta)]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{SET_S} :: \forall\, \Delta,\, \sigma,\, \phi,\, \alpha,\, \bar{\mu} \Rightarrow \\
&\quad \mathrm{WF}(\Delta) \to \mathrm{WF}[\![\Delta/(\sigma,\, \phi,\, \alpha)]\!] \to [\![(\sigma,\, \phi,\, \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \to \\
&\quad [\![\bar{\psi}(\Delta/(\sigma,\, \phi,\, \alpha)) = \bar{\psi}(\Delta)]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{SET_X} :: \forall\, \Delta,\, \sigma,\, \phi,\, \alpha,\, \bar{\mu},\, \bar{\xi} \Rightarrow \\
&\quad \mathrm{WF}(\Delta) \to \mathrm{WF}[\![\Delta/(\sigma,\, \phi,\, \alpha)]\!] \to [\![(\sigma,\, \phi,\, \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \to \mathrm{WF}[\![\Delta/\bar{\xi}]\!] \to \\
&\quad \mathrm{WF}[\![(\Delta/(\sigma,\, \phi,\, \alpha))/\bar{\xi}]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{SET_N} :: \forall\, \Delta,\, \sigma,\, \phi,\, \alpha,\, \bar{\mu},\, \bar{\xi} \Rightarrow \\
&\quad \mathrm{WF}(\Delta) \to \mathrm{WF}[\![\Delta/(\sigma,\, \phi,\, \alpha)]\!] \to [\![(\sigma,\, \phi,\, \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \to \mathrm{WF}[\![\Delta/\bar{\xi}]\!] \to \\
&\quad [\![\sigma_c(\Delta/(\sigma,\, \phi,\, \alpha) \triangleright \bar{\xi}) = \sigma_c(\Delta \triangleright \bar{\xi})]\!]
\end{aligned}$$

However, the resulting object environment is guaranteed to be well-formed only if α represents a meaningful constant akin to "$\#x_\phi$" for some rational number x, and under assumption that an appropriate element $(\sigma,\, \phi,\, \bar{\mu})$ has been already allocated in the address space of Δ:

$$\begin{aligned}
&\mathrm{SET_\Delta} :: \forall\, \Delta,\, \sigma,\, \phi,\, x,\, \bar{\mu} \Rightarrow \\
&\quad \mathrm{WF}(\Delta) \to \mathrm{WF}[\![\#x_\phi]\!] \to [\![(\sigma,\, \phi,\, \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \to \\
&\quad \mathrm{WF}[\![\Delta/(\sigma,\, \phi,\, [\![\#x_\phi]\!])]\!]
\end{aligned}$$

Finally, all updates of the same memory-resident object $(\sigma,\, \phi)$ with equivalent atomic forms always result in identical object environments, whether the operation itself is well-formed or not:

$$\mathrm{SET_\equiv} :: \forall\, \Delta,\, \sigma,\, \phi,\, \alpha_1,\, \alpha_2 \Rightarrow [\![\alpha_1]\!] \equiv [\![\alpha_2]\!] \to [\![\Delta/(\sigma,\, \phi,\, \alpha_1) = \Delta/(\sigma,\, \phi,\, \alpha_2)]\!]$$

The reader should observe that the above definition of environment update operations does not scrutinise any access attributes associated with the targeted object. However, these attributes will soon become relevant to the semantics of the actual Etude term forms that empower programs with a direct access to the environment data during their execution.

4.6 Terms of the Game

Armed with the notions of atoms and the evaluation state, we are now ready to describe the actual computational framework of the language, depicted by a family of syntactic entities known as *terms*. Besides delivering their desired result values, Etude terms may be also used to manipulate the program's evaluation state and interact with other aspects of its surrounding environment, such as the plethora of input and output devices supported by modern computers. Although a number of different approaches towards

modelling of such interactive programs in a purely-functional language have been proposed in literature, the most satisfactory one depicts interactive computations by structures known as *monads* [Moggi 89, Moggi 91, Wadler 92, Peyton Jones 93]. Without dwelling into the somewhat arcane and, in our case, mostly irrelevant category-theoretic foundations of these constructions, it will suffice to observe that monads in Etude follow the same principles as their use in Haskell, with an additional provision made for the absence of first-class functions in our intermediate program representation. In short, the technique is based on a realisation that, in principle, it should be possible to detach the semantic model of an interactive program from that of the devices being interacted with, in the same way as atoms provide for a separation between the meanings of Etude terms and the arithmetic objects that these terms manipulate. To this end, an interactive Etude program is described as a *monadic sequence* of abstract entities depicting invocations of the individual operating system primitives, every one of which represents a single incident in the program's life, generally without any attempt to model the perceivable effects of these incidents on the surrounding computational environment. In this, the monadic approach abstracts over the precise behaviour of individual input and output devices, focusing instead on the manner in which these devices are utilised by the program. In particular, given their interactive nature, the meanings of Etude terms are most naturally expressed as state transformation functions instead of mere scalar values. For example, the denotation of the C "printf" function is captured not by its rather uninteresting return value, or by the collection of side effects resulting from its evaluation such as the sequence of characters printed on the user's terminal, but rather by a function that applies a sequence of character-producing primitive operations to an abstract object that represents the world in which the function is being evaluated.

Formally, the abstract syntax of all recognised Etude term forms is depicted by the following set of Haskell data type definitions:

$term_{[\![v]\!]}$:

RET ($atoms_v$)	*(lifted atoms)*
$atom_v$ ($atoms_v$)	*(function application)*
prim($atoms_v$)	*(primitive operation)*
IF $atom_v$ THEN $term_v$ ELSE $term_v$	*(conditional expressions)*
NEW (*envelope*)	*(object introduction)*
DEL (*envelope*)	*(object destruction)*
GET [$atom_v$, {*mode*}] *format*	*(object inspection)*
SET [$atom_v$, {*mode*}] *format* TO $atom_v$	*(object adjustment)*
SET_I [$atom_v$, {*mode*}] *format* TO $atom_v$	*(object initialisation)*
LET $bindings_v$; $term_v$	*(monadic bindings)*

$bindings_{[\![v]\!]}$:

$[binding_v]$	*(set of term bindings)*

$binding_{[\![v]\!]}$:

$parameters_v$ = $term_v$	*(individual term binding)*

The algebraic semantics of terms are modelled by a set of Haskell functions and proper-

ties analogous to the well-formedness and equivalence of Etude atoms described earlier in Section 4.4. In particular, every Etude term τ is associated with a set of free variables $\mathrm{FV}(\tau)$ derived from its lexical syntax as follows:

$$\mathrm{FV}[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \mathit{term}_\nu \to \{\nu\}$$

$$\begin{array}{ll}
\mathrm{FV}[\![\textsf{RET}\ (\bar{\alpha})]\!] & = \mathrm{FV}(\bar{\alpha}) \\
\mathrm{FV}[\![\alpha(\bar{\alpha})]\!] & = \mathrm{FV}(\alpha) \cup \mathrm{FV}(\bar{\alpha}) \\
\mathrm{FV}[\![\pi(\bar{\alpha})]\!] & = \mathrm{FV}(\bar{\alpha}) \\
\mathrm{FV}[\![\textsf{IF}\ \alpha\ \textsf{THEN}\ \tau_1\ \textsf{ELSE}\ \tau_2]\!] & = \mathrm{FV}(\alpha) \cup \mathrm{FV}(\tau_1) \cup \mathrm{FV}(\tau_2) \\
\mathrm{FV}[\![\textsf{NEW}\ (\bar{\xi})]\!] & = \varnothing \\
\mathrm{FV}[\![\textsf{DEL}\ \ (\bar{\xi})]\!] & = \varnothing \\
\mathrm{FV}[\![\textsf{GET}\ \ [\alpha,\ \ \bar{\mu}]_\phi]\!] & = \mathrm{FV}(\alpha) \\
\mathrm{FV}[\![\textsf{SET}\ \ [\alpha_1,\ \bar{\mu}]_\phi\ \textsf{TO}\ \alpha_2]\!] & = \mathrm{FV}(\alpha_1) \cup \mathrm{FV}(\alpha_2) \\
\mathrm{FV}[\![\textsf{SET}_\mathsf{I}\ \ [\alpha_1,\ \bar{\mu}]_\phi\ \textsf{TO}\ \alpha_2]\!] & = \mathrm{FV}(\alpha_1) \cup \mathrm{FV}(\alpha_2) \\
\mathrm{FV}[\![\textsf{LET}\ \ \bar{\beta};\ \tau]\!] & = \mathrm{FV}(\bar{\beta}) \cup (\mathrm{FV}(\tau) \backslash \mathrm{BV}(\bar{\beta}))
\end{array}$$

and, for the actual list of bindings found in every "LET" construct:

$$\mathrm{FV}[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \mathit{bindings}_\nu \to \{\nu\}$$

$$\mathrm{FV}[\![\bar{\beta}]\!] = \bigcup[\mathrm{FV}(\tau_k)\ \mid\ [\![\nu_k = \tau_k]\!] \leftarrow \bar{\beta}]$$

In general, the above construction collects all variable names appearing anywhere within the term's structure, except that any variables bound by the definition list $\bar{\beta}$ in a term of the form "$\textsf{LET}\ \bar{\beta};\ \tau$" are always excluded from the set of variables appearing free in such term's body τ. Formally, the set of variables bound by a list of Etude definitions is depicted by the notation "$\mathrm{BV}(\bar{\beta})$" and constructed from the union of all variables appearing anywhere to the left of the "=" sign in the list as follows:

$$\mathrm{BV}[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \mathit{bindings}_\nu \to \{\nu\}$$

$$\mathrm{BV}[\![\bar{\beta}]\!] = \bigcup[\nu_k\ \mid\ [\![\nu_k = \tau_k]\!] \leftarrow \bar{\beta}]$$

Further, we can also subject terms to the usual substitution function "τ/S" where S represents a mapping of variable names to atoms intended as a replacement for these variables within the term τ. Assuming that all atoms appearing in the finite map S are closed, the definition of "/" is simple, provided that we take care to purge from S any variables bound by every "LET" term before applying it to that term's body:

$$[\![\cdot]\!]/[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \mathit{term}_\nu \to (\nu \mapsto \mathit{atom}_\nu) \to \mathit{term}_\nu$$

$$\begin{array}{lll}
[\![\textsf{RET}\ (\bar{\alpha})]\!] & /[\![S]\!] & = [\![\textsf{RET}\ ([\![\bar{\alpha}/S]\!])]\!] \\
[\![\alpha(\bar{\alpha})]\!] & /[\![S]\!] & = [\![[\![\alpha/S]\!]([\![\bar{\alpha}/S]\!])]\!] \\
[\![\pi(\bar{\alpha})]\!] & /[\![S]\!] & = [\![\pi([\![\bar{\alpha}/S]\!])]\!] \\
[\![\textsf{IF}\ \alpha\ \textsf{THEN}\ \tau_1\ \textsf{ELSE}\ \tau_2]\!] & /[\![S]\!] & = [\![\textsf{IF}\ [\![\alpha/S]\!]\ \textsf{THEN}\ [\![\tau_1/S]\!]\ \textsf{ELSE}\ [\![\tau_2/S]\!]]\!] \\
[\![\textsf{NEW}\ (\bar{\xi})]\!] & /[\![S]\!] & = [\![\textsf{NEW}\ (\bar{\xi})]\!] \\
[\![\textsf{DEL}\ \ (\bar{\xi})]\!] & /[\![S]\!] & = [\![\textsf{DEL}\ \ (\bar{\xi})]\!] \\
[\![\textsf{GET}\ \ [\alpha,\ \ \bar{\mu}]_\phi]\!] & /[\![S]\!] & = [\![\textsf{GET}\ \ [[\![\alpha/S,\ \bar{\mu}]\!]]_\phi]\!] \\
[\![\textsf{SET}\ \ [\alpha_1,\ \bar{\mu}]_\phi\ \textsf{TO}\ \alpha_2]\!] & /[\![S]\!] & = [\![\textsf{SET}\ \ [[\![\alpha_1/S,\ \bar{\mu}]\!]]_\phi\ \textsf{TO}\ [\![\alpha_2/S]\!]]\!] \\
[\![\textsf{SET}_\mathsf{I}\ \ [\alpha_1,\ \bar{\mu}]_\phi\ \textsf{TO}\ \alpha_2]\!] & /[\![S]\!] & = [\![\textsf{SET}_\mathsf{I}\ \ [[\![\alpha_1/S,\ \bar{\mu}]\!]]_\phi\ \textsf{TO}\ [\![\alpha_2/S]\!]]\!] \\
[\![\textsf{LET}\ \ \bar{\beta};\ \tau]\!] & /[\![S]\!] & = [\![\textsf{LET}\ \ [\![\bar{\beta}/S]\!];\ \tau/(S\backslash\mathrm{BV}(\bar{\beta}))]\!]
\end{array}$$

and, for "LET" bindings:

$$[\![\cdot]\!]/[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \mathit{bindings}_\nu \rightarrow (\nu \mapsto \mathit{atom}_\nu) \rightarrow \mathit{bindings}_\nu$$

$$[\![\bar{\beta}]\!]/[\![S]\!] = [[\![\bar{\nu}_k = [\![\tau_k/S]\!]]\!] \mid [\![\bar{\nu}_k = \tau_k]\!] \leftarrow \bar{\beta}]$$

Like atoms, terms are generally considered to be meaningful only if they are closed, i.e., if all of their free variables have been substituted for some well-formed atomic constants. However, in order to formalise the impact of monadic Etude terms on their evaluation state, every such term τ must be always interpreted in the context of an appropriate function environment Λ and object environment Δ, so that its well-formedness is asserted by a property of the form "$\text{WF}[\![\Lambda, \Delta \triangleright \tau]\!]$" rather than a mere "$\text{WF}[\![\tau]\!]$" as done previously for Etude atoms. Similarly, the equivalence of two terms τ_1 and τ_2, each taken in the context of its respective evaluation state, is always depicted as a binary property of the form "$[\![\Lambda_1, \Delta_1 \triangleright \tau_1]\!] \equiv [\![\Lambda_2, \Delta_2 \triangleright \tau_2]\!]$".

In particular, a monadic unit term of the form "RET ($\bar{\alpha}$)" permits Etude programs to *lift* a list of atoms into the term monad. For conciseness, we will usually omit the parentheses surrounding the atom list $\bar{\alpha}$ whenever that list consists of precisely one atom. Conceptually, such lifted terms depict the *unit* action of the term monad, which always delivers a predetermined set of results described precisely by the specified list of atoms without affecting the program's environment in any other way. Accordingly, well-formedness of lifted terms is determined solely by the well-formedness of its constituents, as captured by the following simple theorem:

$$\text{WF}_{\text{RET}} :: \forall\ \Lambda, \Delta, \bar{\alpha} \Rightarrow \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}(\bar{\alpha}) \rightarrow \text{WF}[\![\Lambda, \Delta \triangleright \text{RET } (\bar{\alpha})]\!]$$

and a pair of such terms should be always considered equivalent whenever it is formed from pairwise-equivalent atomic lists:

$$\text{EQV}_{\text{RET}} :: \forall\ \Lambda, \Delta, \bar{\alpha}_1, \bar{\alpha}_2 \Rightarrow (\bar{\alpha}_1 \equiv \bar{\alpha}_2) \rightarrow [\![\Lambda, \Delta \triangleright \text{RET } (\bar{\alpha}_1)]\!] \equiv [\![\Lambda, \Delta \triangleright \text{RET } (\bar{\alpha}_2)]\!]$$

In Etude, an application of a given function atom α to a list of zero or more arguments $\bar{\alpha}$ is depicted by a term of the form "$\alpha(\bar{\alpha})$". Both the function itself and its arguments are represented by atomic values. While Etude places no specific requirements on the argument list $\bar{\alpha}$ beyond its general validity, the function value α must always represent an equivalent of a well-formed atomic form "$\#x_\phi$", in which ϕ is a valid format from the function genre and the integer x is mapped in the associated function environment Λ to a well-formed Etude function of the form "$\lambda\bar{\nu}.\tau$". Further, for such an application term to be considered well-formed, the function's parameter list $\bar{\nu}$ must have the same length as the argument list $\bar{\alpha}$ and the body term τ must be well-formed after substitution of all parameter variables from $\bar{\nu}$ for the corresponding atomic values from $\bar{\alpha}$. Formally:

$$\begin{aligned}
&\text{WF}_{\text{APP}} :: \forall\ \Lambda, \Delta, x, \bar{\alpha}, \bar{\nu}, \tau \Rightarrow \\
&\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{F.}\Phi}]\!] \rightarrow \text{WF}(\bar{\alpha}) \rightarrow \\
&\quad [\![\Lambda(x) = [\![\lambda\bar{\nu}.\tau]\!] \wedge \text{length}(\bar{\nu}) = \text{length}(\bar{\alpha})]\!] \rightarrow \\
&\quad \text{WF}(\Lambda, \Delta \triangleright \tau/(\bar{\nu}|\bar{\alpha})) \rightarrow \\
&\quad \text{WF}[\![\Lambda, \Delta \triangleright \#x_{\text{F.}\Phi}(\bar{\alpha})]\!]
\end{aligned}$$

Every such application term is equivalent precisely to a term formed from the substitution of $\bar{\alpha}$ for the variable list $\bar{\nu}$ in the corresponding function's body τ:

$$\begin{aligned}&\text{EQV}_{\beta} :: \forall\ \Lambda,\ \Delta,\ x,\ \bar{\alpha},\ \bar{\nu},\ \tau \Rightarrow\\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{F.}\Phi}]\!] \rightarrow \text{WF}(\bar{\alpha}) \rightarrow\\ &\quad [\![\Lambda(x) = [\![\lambda\bar{\nu}.\tau]\!] \wedge \text{length}(\bar{\nu}) = \text{length}(\bar{\alpha})]\!] \rightarrow\\ &\quad \text{WF}[\![\Lambda,\ \Delta \triangleright \tau/(\bar{\nu}|\bar{\alpha})]\!] \rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \#x_{\text{F.}\Phi}(\bar{\alpha})]\!] \equiv [\![\Lambda,\ \Delta \triangleright \tau/(\bar{\nu}|\bar{\alpha})]\!]\end{aligned}$$

Readers will observe that the above theorem provides a direct Etude analogue of the beta reduction rule from Section 4.1. To aid reasoning about unreduced function applications, we also insist on the following structural equivalence between pairs of such terms in which all corresponding components represent pairwise-equivalent atomic values:

$$\begin{aligned}&\text{EQV}_{\text{APP}} :: \forall\ \Lambda,\ \Delta,\ \alpha_1,\ \bar{\alpha}_1,\ \alpha_2,\ \bar{\alpha}_2 \Rightarrow\\ &\quad (\alpha_1 \equiv \alpha_2) \rightarrow (\bar{\alpha}_1 \equiv \bar{\alpha}_2) \rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \alpha_1(\bar{\alpha}_1)]\!] \equiv [\![\Lambda,\ \Delta \triangleright \alpha_2(\bar{\alpha}_2)]\!]\end{aligned}$$

Although superficially-similar, terms of the form "$\pi(\bar{\alpha})$", in which π represents a *primitive operation* taken from an implementation-defined set of values selected individually by every system architecture have a very different set of formal properties. For one, the generic fragment of Etude's semantics remains silent on any well-formedness criteria applicable to these terms, since, in general, the range of concrete values acceptable for a given primitive π is determined solely by the murky details of the underlying operating system. Nevertheless, if the same primitive is applied to a pair of equivalent atomic lists, then we can always be sure about the equivalence of the resulting terms:

$$\text{EQV}_{\text{SYS}} :: \forall\ \Lambda,\ \Delta,\ \pi,\ \bar{\alpha}_1,\ \bar{\alpha}_2 \Rightarrow (\bar{\alpha}_1 \equiv \bar{\alpha}_2) \rightarrow [\![\Lambda,\ \Delta \triangleright \pi(\bar{\alpha}_1)]\!] \equiv [\![\Lambda,\ \Delta \triangleright \pi(\bar{\alpha}_2)]\!]$$

Theorem EQV_{SYS} makes it possible to detach the notion of term equivalence from the semantic specifics of the underlying operating system. To complete this detachment, conditional terms of the form "IF α THEN τ_1 ELSE τ_2" allow for embedding of arbitrary, even malformed operands within well-formed Etude programs. In such constructs, only one of the two branches τ_1 and τ_2 needs to represent a valid computation:

$$\begin{aligned}&\text{WF}_{\text{TT}} :: \forall\ \Lambda,\ \Delta,\ x,\ \tau_1,\ \tau_2 \Rightarrow\\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}(\tau_1) \rightarrow [\![x \neq 0]\!] \rightarrow\\ &\quad \text{WF}[\![\text{IF } \#x_{\text{Z.}\Phi} \text{ THEN } \tau_1 \text{ ELSE } \tau_2]\!]\end{aligned}$$

$$\begin{aligned}&\text{WF}_{\text{FF}} :: \forall\ \Lambda,\ \Delta,\ x,\ \tau_1,\ \tau_2 \Rightarrow\\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}(\tau_2) \rightarrow [\![x = 0]\!] \rightarrow\\ &\quad \text{WF}[\![\text{IF } \#x_{\text{Z.}\Phi} \text{ THEN } \tau_1 \text{ ELSE } \tau_2]\!]\end{aligned}$$

In particular, if α represents any natural number other than zero, then the Etude term "IF α THEN τ_1 ELSE τ_2" is always equivalent to τ_1:

$$\begin{aligned}&\text{EQV}_{\text{TT}} :: \forall\ \Lambda,\ \Delta,\ x,\ \tau_1,\ \tau_2 \Rightarrow\\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}(\tau_1) \rightarrow [\![x \neq 0]\!] \rightarrow\\ &\quad [\![\text{IF } \#x_{\text{Z.}\Phi} \text{ THEN } \tau_1 \text{ ELSE } \tau_2]\!] \equiv [\![\Lambda,\ \Delta \triangleright \tau_1]\!]\end{aligned}$$

Conversely, if α has a zero value, then the conditional term serves as a mere synonym for its later branch τ_2. Formally:

$$\begin{aligned}&\text{EQV}_{\text{FF}} :: \forall\ \Lambda,\ \Delta,\ x,\ \tau_1,\ \tau_2 \Rightarrow \\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}(\tau_2) \rightarrow [\![x = 0]\!] \rightarrow \\ &\quad [\![\textsf{IF}\ \#x_{\text{Z.}\Phi}\ \textsf{THEN}\ \tau_1\ \textsf{ELSE}\ \tau_2]\!] \equiv [\![\Lambda,\ \Delta \triangleright \tau_2]\!]\end{aligned}$$

As for the earlier term form, two such terms are also equivalent when assembled from pairwise-equivalent components:

$$\begin{aligned}&\text{EQV}_{\text{IF}} :: \forall\ \Lambda_1,\ \Delta_1,\ \alpha_1,\ \tau_{11},\ \tau_{21},\ \Lambda_2,\ \Delta_2,\ \alpha_2,\ \tau_{12},\ \tau_{22} \Rightarrow \\ &\quad (\alpha_1 \equiv \alpha_2) \rightarrow \\ &\quad (\Lambda_1,\ \Delta_1 \triangleright \tau_{11}) \equiv (\Lambda_2,\ \Delta_2 \triangleright \tau_{12}) \rightarrow (\Lambda_1,\ \Delta_1 \triangleright \tau_{12}) \equiv (\Lambda_2,\ \Delta_2 \triangleright \tau_{22}) \rightarrow \\ &\quad [\![\Lambda_1,\ \Delta_1 \triangleright \textsf{IF}\ \alpha_1\ \textsf{THEN}\ \tau_{11}\ \textsf{ELSE}\ \tau_{21}]\!] \equiv [\![\Lambda_2,\ \Delta_2 \triangleright \textsf{IF}\ \alpha_2\ \textsf{THEN}\ \tau_{12}\ \textsf{ELSE}\ \tau_{22}]\!]\end{aligned}$$

Terms of the form "NEW ($\bar{\xi}$)" and "DEL ($\bar{\xi}$)" provide Etude programs with explicit facilities for extending and contracting the address space of their environment by the envelope $\bar{\xi}$. Formally, such terms are considered to be well-formed only in the context of an object environment Δ to which the corresponding extension and contraction operations $\Delta/\bar{\xi}$ and $\Delta\backslash\bar{\xi}$ can be applied within the constraints of the underlying language implementation outlined earlier in Section 4.5. Specifically:

$$\begin{aligned}&\text{WF}_{\text{NEW}} :: \forall\ \Lambda,\ \Delta,\ \bar{\xi} \Rightarrow \text{WF}(\Lambda) \rightarrow \text{WF}[\![\Delta/\bar{\xi}]\!] \rightarrow \text{WF}[\![\Lambda,\ \Delta \triangleright \textsf{NEW}\ (\bar{\xi})]\!] \\ &\text{WF}_{\text{DEL}} :: \forall\ \Lambda,\ \Delta,\ \bar{\xi} \Rightarrow \text{WF}(\Lambda) \rightarrow \text{WF}[\![\Delta\backslash\bar{\xi}]\!] \rightarrow \text{WF}[\![\Lambda,\ \Delta \triangleright \textsf{DEL}\ \ (\bar{\xi})]\!]\end{aligned}$$

Under these conditions, "NEW ($\bar{\xi}$)" is always equivalent to a lifted term of the form "RET ($\#x_{\text{O.}\Phi}$)" in which the integer x has the value of $\sigma_c(\Delta \triangleright \bar{\xi})$, while "DEL ($\bar{\xi}$)" stands for an empty list of atoms "RET ()" in the context of an object environment Δ contracted by the supplied envelope $\bar{\xi}$. Specifically:

$$\begin{aligned}&\text{EQV}_{\text{NEW}} :: \forall\ \Lambda,\ \Delta,\ \bar{\xi} \Rightarrow \\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}[\![\Delta/\bar{\xi}]\!] \rightarrow \\ &\quad [\![\Lambda,\ \Delta \triangleright \textsf{NEW}\ (\bar{\xi})]\!] \equiv [\![\Lambda,\ \Delta/\bar{\xi} \triangleright \textsf{RET}\ (\#[\![\sigma_c(\Delta \triangleright \bar{\xi})]\!]_{\text{O.}\Phi})]\!]\end{aligned}$$

$$\begin{aligned}&\text{EQV}_{\text{DEL}} :: \forall\ \Lambda,\ \Delta,\ \bar{\xi} \Rightarrow \\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}[\![\Delta\backslash\bar{\xi}]\!] \rightarrow \\ &\quad [\![\Lambda,\ \Delta \triangleright \textsf{DEL}\ \ (\bar{\xi})]\!] \equiv [\![\Lambda,\ \Delta\backslash\bar{\xi} \triangleright \textsf{RET}\ ()]\!]\end{aligned}$$

Further, each valid construction of the form "NEW ($\bar{\xi}$)" guarantees well-formedness of every object atom "$\alpha +_{[\![O(\phi)]\!]} \#n_{\text{Z.}\Phi}$", in which α is equivalent to an appropriately converted value of "$\#[\![\sigma_c(\Delta \triangleright \bar{\xi})]\!]_{\text{O.}\Phi}$" and n represents an offset into one of the envelope elements in $\bar{\xi}$, as captured by the following Haskell definition:

$$\begin{aligned}&\text{WF}_{\text{OBJ}} :: \forall\ \Lambda,\ \Delta,\ \bar{\xi} \Rightarrow \\ &\quad \text{WF}[\![\Lambda,\ \Delta \triangleright \textsf{NEW}\ (\bar{\xi})]\!] \rightarrow \\ &\quad (\forall\ n,\ \phi,\ \bar{\mu} \Rightarrow [\![(n,\ \phi,\ \bar{\mu}) \in \bar{\xi}]\!] \rightarrow \text{WF}[\![([\![O(\phi)]\!]_{\text{O.}\Phi}(\#[\![\sigma_c(\Delta \triangleright \bar{\xi})]\!]_{\text{O.}\Phi})) +_{[\![O(\phi)]\!]} \#n_{\text{Z.}\Phi}]\!])\end{aligned}$$

Similarly, the "GET $[\alpha,\ \bar{\mu}]_\phi$" term form makes it possible for Etude programs to access individual atomic bindings from the associated object environment Δ. As is to be expected, the α argument of "GET" should be equivalent to some well-formed object

atom "$\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}$", such that $\bar{\alpha}(\Delta \triangleright x+n,\ \phi)$ represents a valid atomic value whenever $(x+n,\ \phi,\ \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)$. Formally:

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{GET}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \phi,\ \bar{\mu} \Rightarrow \\
&\quad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}]\!] \rightarrow \mathrm{WF}[\![\bar{\alpha}(\Delta \triangleright x+n,\ \phi)]\!] \rightarrow \\
&\quad [\![(x+n,\ \phi,\ \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\Lambda,\ \Delta \triangleright \mathsf{GET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_{\phi}]\!]
\end{aligned}$$

In all cases, a pair of "GET" terms that access equivalent address atoms under identical access modes and formats are always considered to be equivalent:

$$\begin{aligned}
&\mathrm{EQV}_{\mathrm{GET}} :: \forall\ \Lambda,\ \Delta,\ \alpha_1,\ \alpha_2,\ \phi,\ \bar{\mu} \Rightarrow \\
&\quad (\alpha_1 \equiv \alpha_2) \rightarrow \\
&\quad [\![\Lambda,\ \Delta \triangleright \mathsf{GET}\ [\alpha_1,\ \bar{\mu}]_{\phi}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \mathsf{GET}\ [\alpha_2,\ \bar{\mu}]_{\phi}]\!]
\end{aligned}$$

Further, if the address $x+n$ is actually mapped in the term's evaluation state to an atom α' without the volatile access attribute "V", then "$\mathsf{GET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_{\phi}$" is also equivalent to a simple lifted term of the form "$\mathsf{RET}\ (\alpha)$". Formally:

$$\begin{aligned}
&\mathrm{EQV}_{\mathrm{GETO}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \phi,\ \bar{\mu} \Rightarrow \\
&\quad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}]\!] \rightarrow \mathrm{WF}[\![\bar{\alpha}(\Delta \triangleright x+n,\ \phi)]\!] \rightarrow \\
&\quad [\![(x+n,\ \phi,\ \bar{\mu}\backslash\{\mathrm{V}\}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\quad [\![\Lambda,\ \Delta \triangleright \mathsf{GET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_{\phi}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \mathsf{RET}\ [\![\bar{\alpha}(\Delta \triangleright x+n,\ \phi)]\!]]\!]
\end{aligned}$$

Conversely, both the "$\mathsf{SET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_{\phi}\ \mathsf{TO}\ \alpha$" term form and its "$\mathsf{SET_I}$" variant perform the environment update operation $\Delta/(x+n,\ \phi,\ \alpha)$. In other words, their well-formedness is mandated by the following pair of theorems:

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{SET}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \phi,\ \bar{\mu},\ \alpha \Rightarrow \\
&\quad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}(\Delta/(x+n,\ \phi,\ \alpha)) \rightarrow \mathrm{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}]\!] \rightarrow \\
&\quad [\![(x+n,\ \phi,\ \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\Lambda,\ \Delta \triangleright \mathsf{SET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_{\phi}\ \mathsf{TO}\ \alpha]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{INI}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \phi,\ \bar{\mu},\ \alpha \Rightarrow \\
&\quad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}(\Delta/(x+n,\ \phi,\ \alpha)) \rightarrow \mathrm{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}]\!] \rightarrow \\
&\quad [\![(x+n,\ \phi,\ \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}_i(\Delta)]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\Lambda,\ \Delta \triangleright \mathsf{SET_I}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_{\phi}\ \mathsf{TO}\ \alpha]\!]
\end{aligned}$$

Intuitively, the "SET" variant of an environment update preserves the conventional semantics of the constant memory access attribute "C", while "$\mathsf{SET_I}$" ignores that attribute for objects located within the uninitialised region of the address space $\bar{\xi}_i(\Delta)$, in order to facilitate a once-off initialisation of otherwise-immutable memory-resident objects within a well-formed Etude program. In particular, both forms of the term "$\mathsf{SET}\ [\alpha_1,\ \bar{\mu}]_{\phi}\ \mathsf{TO}\ \alpha_2$" are reducible into the trivial monadic sequence "$\mathsf{RET}\ ()$", taken in the context of an object environment updated with the binding of the address α_1 to the value of α_2, provided that such an update is deemed well-formed by the above validity criteria:

$$\begin{aligned}
&\mathrm{EQV}_{\mathrm{SETO}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \bar{\mu},\ \phi,\ \alpha \Rightarrow \\
&\quad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}(\Delta/(x+n,\ \phi,\ \alpha)) \rightarrow \mathrm{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}]\!] \rightarrow \\
&\quad [\![(x+n,\ \phi,\ \varnothing) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\quad [\![\Lambda,\ \Delta \triangleright \mathsf{SET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_{\phi}\ \mathsf{TO}\ \alpha]\!] \equiv [\![\Lambda,\ \Delta/(x+n,\ \phi,\ \alpha) \triangleright \mathsf{RET}\ ()]\!]
\end{aligned}$$

$$\begin{aligned}&\text{EQV}_{\text{INIO}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \bar{\mu},\ \phi,\ \alpha \Rightarrow \\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta/(x+n,\ \phi,\ \alpha)) \rightarrow \text{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\text{Z.}\Phi}]\!] \rightarrow \\ &\quad [\![(x+n,\ \phi,\ \{\text{C}\}) \in_{\text{A}} \bar{\xi}_i(\Delta)]\!] \rightarrow \\ &\quad [\![\Lambda,\ \Delta \triangleright \text{SET}_{\text{I}}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\text{Z.}\Phi},\ \bar{\mu}]_{\phi}\ \text{TO}\ \alpha]\!] \equiv [\![\Lambda,\ \Delta/(x+n,\ \phi,\ \alpha) \triangleright \text{RET}\ ()]\!]\end{aligned}$$

Finally, both "SET" and "SET_{I}" are subject to the usual structural equivalence rules as follows:

$$\begin{aligned}&\text{EQV}_{\text{SET}} :: \forall\ \Lambda,\ \Delta,\ \alpha_{11},\ \alpha_{21},\ \alpha_{12},\ \alpha_{22},\ \bar{\mu},\ \phi \Rightarrow \\ &\quad (\alpha_{11} \equiv \alpha_{12}) \rightarrow (\alpha_{21} \equiv \alpha_{22}) \rightarrow \\ &\quad [\![\Lambda,\ \Delta \triangleright \text{SET}\ [\alpha_{11},\ \bar{\mu}]_{\phi}\ \text{TO}\ \alpha_{21}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \text{SET}\ [\alpha_{12},\ \bar{\mu}]_{\phi}\ \text{TO}\ \alpha_{22}]\!]\end{aligned}$$

$$\begin{aligned}&\text{EQV}_{\text{INI}} :: \forall\ \Lambda,\ \Delta,\ \alpha_{11},\ \alpha_{21},\ \alpha_{12},\ \alpha_{22},\ \bar{\mu},\ \phi \Rightarrow \\ &\quad (\alpha_{11} \equiv \alpha_{12}) \rightarrow (\alpha_{21} \equiv \alpha_{22}) \rightarrow \\ &\quad [\![\Lambda,\ \Delta \triangleright \text{SET}_{\text{I}}\ [\alpha_{11},\ \bar{\mu}]_{\phi}\ \text{TO}\ \alpha_{21}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \text{SET}\ [\alpha_{12},\ \bar{\mu}]_{\phi}\ \text{TO}\ \alpha_{22}]\!]\end{aligned}$$

Last but not least, a term of the form "LET $\bar{\nu}_1$ = τ_1, $\bar{\nu}_2$ = τ_2 ... $\bar{\nu}_n$ = τ_n; τ" permits Etude programs to capture the results delivered by the monadic terms τ_1 ... τ_n and makes them available within the body term τ as the corresponding variables from the parameter lists $\bar{\nu}_1$ ... $\bar{\nu}_n$. Accordingly, a degenerate form of the "LET $\bar{\beta}$; τ" term that specifies an empty binding list $\bar{\beta}$ is always equivalent to its body expression τ and well-formed whenever τ happens to be valid. Formally:

$$\text{EQV}_{\text{LETE}} :: \forall\ \Lambda,\ \Delta,\ \tau \Rightarrow [\![\Lambda,\ \Delta \triangleright \text{LET};\ \tau]\!] \equiv [\![\Lambda,\ \Delta \triangleright \tau]\!]$$

Further, a term of the form "LET $\bar{\nu}$ = RET $(\bar{\alpha})$; τ" constitutes a simple syntactic equivalent of the beta redex "$(\lambda\bar{\nu}.\tau)(\bar{\alpha})$" from the pure lambda calculus described in Section 4.1. Accordingly, such terms are deemed valid only if $\bar{\nu}$ and $\bar{\alpha}$ represent well-formed lists of equal lengths. Their generic properties are determined in a manner similar to that described earlier for function applications:

$$\begin{aligned}&\text{EQV}_{\text{LETA}} :: \forall\ \Lambda,\ \Delta,\ \bar{\nu},\ \bar{\alpha} \Rightarrow \\ &\quad \text{WF}(\bar{\nu}) \rightarrow \text{WF}(\bar{\alpha}) \rightarrow \\ &\quad [\![\text{length}(\bar{\nu}) = \text{length}(\bar{\alpha})]\!] \rightarrow \\ &\quad [\![\Lambda,\ \Delta \triangleright \text{LET}\ \bar{\nu} = \text{RET}\ (\bar{\alpha});\ \tau]\!] \equiv (\Lambda,\ \Delta \triangleright \tau/(\bar{\nu}|\bar{\alpha}))\end{aligned}$$

For conciseness, in the remainder of this work I will usually drop the "RET" keyword in such trivial "LET" expressions, writing the above term simply as "LET $\bar{\nu}$ =$(\bar{\alpha})$; τ" for readability. This style of programming, in which all subexpressions are systematically bound to variable names and all beta redexes appear as explicit "LET" bindings is well known in literature under the title of the *monadic normal form* used to study the operational semantics of call-by-value lambda calculi [Moggi 91, Hatcliff 94, Danvy 03]. It is also known as the *administrative normal form* to some authors, since it eliminates the so-called *administrative redexes* introduced by translation of a program into the *continuation passing style* [Sabry 92, Flanagan 93].

The semantics of "LET" bindings get interesting when two such terms are nested into a binding of the form "LET $\bar{\nu}$ = (LET $\bar{\beta}$; τ_1); τ_2". The next theorem states that all such constructions can be always flattened into "LET $\bar{\beta}$; (LET $\bar{\nu}$ = τ_1; τ_2)", provided that

none of the variables bound in $\bar{\beta}$, other than those found in the associated parameter list $\bar{\nu}$, appear free in τ_2. Formally:

$$\begin{aligned}&\text{EQV}_{\text{LETN}} :: \forall\ \Lambda,\ \Delta,\ \bar{\nu},\ \bar{\beta},\ \tau_1,\ \tau_2 \Rightarrow\\ &\quad [\![\text{BV}(\bar{\beta}) \cap (\text{FV}(\tau_2) \backslash \bar{\nu}) = \varnothing]\!] \rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ (\bar{\nu} = \texttt{LET}\ \bar{\beta};\ \tau_1);\ \tau_2]\!] \equiv [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\beta};\ (\texttt{LET}\ \bar{\nu} = \tau_1;\ \tau_2)]\!]\end{aligned}$$

The reader should observe that the free variable constraint in the above theorem conveniently circumvents the variable capture problem inherent to the pure calculus formulation of the substitution function from Section 4.1.

Unfortunately, little else can be said about validity of any other singular term form "LET $\bar{\nu} = \tau_1$; τ_2" without dwelling into the underlying implementation details of Etude's operational model, since the potential presence of primitive applications in τ_1 makes it impossible to narrow down the set of evaluation states that may be encountered by the body term τ_2. However, a rather relaxed set of requirements for these remaining constructs may be captured eloquently without divulging any details of the underlying operating system implementation with the help of the following theorem:

$$\begin{aligned}&\text{EQV}_{\text{LETS}} :: \forall\ \Lambda,\ \Delta,\ \bar{\nu}_1,\ \tau_{11},\ \tau_{21},\ \bar{\nu}_2,\ \tau_{12},\ \tau_{22} \Rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \tau_{11}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \tau_{12}]\!] \rightarrow\\ &\quad [\![\text{length}(\bar{\nu}_1) = \text{length}(\bar{\nu}_2)]\!] \rightarrow\\ &\quad (\forall\ \Lambda',\ \Delta',\ \bar{\alpha}' \Rightarrow\\ &\qquad \text{WF}(\bar{\alpha}') \rightarrow\\ &\qquad [\![\text{length}(\bar{\alpha}') = \text{length}(\bar{\nu}_1)]\!] \rightarrow\\ &\qquad [\![\Lambda',\ \Delta' \triangleright \tau_{21}/(\bar{\nu}_1|\bar{\alpha}')]\!] \equiv [\![\Lambda',\ \Delta' \triangleright \tau_{22}/(\bar{\nu}_2|\bar{\alpha}')]\!]) \rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\nu}_1 = \tau_{11};\ \tau_{21}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\nu}_2 = \tau_{12};\ \tau_{22}]\!]\end{aligned}$$

Intuitively, theorem EQV_{LETS} states that we can always assert equivalence between a pair of terms of the form "LET $\bar{\nu}_1 = \tau_{11}$; τ_{21}" and "LET $\bar{\nu}_2 = \tau_{12}$; τ_{22}", provided that τ_{11} is equivalent to τ_{12}, $\bar{\nu}_1$ and $\bar{\nu}_2$ represent variable lists of equal lengths and, no matter what well-formed atomic values are substituted for these variables into the body terms τ_{21} and τ_{22}, these bodies remain equivalent under every conceivable evaluation state.

A reader may observe that, so far, we have not mentioned anything about "LET" terms that specify multiple bindings in the list $\bar{\beta}$. Intuitively, such constructs are intended to capture the unspecified aspects of certain expression forms in the C language, whose precise evaluation order is left open to individual interpretation by every implementation of a C compiler. In principle, such terms are essentially equivalent to a sequential application of each binding from $\bar{\beta}$:

$$\begin{aligned}&\text{EQV}_{\text{LETM}} :: \forall\ \Lambda,\ \Delta,\ \bar{\nu},\ \bar{\beta},\ \tau \Rightarrow\\ &\quad [\![\text{length}(\bar{\beta}) > 1]\!] \rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\beta};\ \tau]\!] \equiv [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ [\![\text{head}(\bar{\beta})]\!];\ \texttt{LET}\ [\![\text{tail}(\bar{\beta})]\!];\ \tau]\!]\end{aligned}$$

However, they are only considered to be well-formed if all permutations of $\bar{\beta}$ are equivalent to each other and, further, if none of the variables found to the left of one of the "=" signs appears free in any other binding from the list. Using the standard mathematical definition of the factorial function $x!$, together with $\mathcal{P}(k,\ \bar{\beta})$, which returns the

canonical k-th permutation of the list $\bar{\beta}$, this constraint can be formalised in Haskell as the following theorem:

$$\begin{aligned}
&\text{WF}_{\text{LETM}} :: \forall\ \Lambda,\ \Delta,\ \bar{\beta},\ \tau \Rightarrow \\
&\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \\
&\quad \text{WF}[\![\Lambda,\ \Delta \triangleright \texttt{LET}\ [\![\text{head}(\bar{\beta})]\!];\ \texttt{LET}\ [\![\text{tail}(\bar{\beta})]\!];\ \tau]\!] \rightarrow \\
&\quad [\![[\text{FV}(\tau_k) \backslash \bar{v}_k \mid [\![\bar{v}_k = \tau_k]\!] \leftarrow \bar{\beta}] \cup \cap\ \text{BV}(\bar{\beta}) = \varnothing]\!] \rightarrow \\
&\quad (\forall\ k \Rightarrow [\![0 \leq k < \text{length}(\bar{\beta})!]\!] \rightarrow [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\beta};\ \tau]\!] \equiv [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ [\![\mathcal{P}(k,\ \bar{\beta})]\!];\ \tau]\!]) \rightarrow \\
&\quad \text{WF}[\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\beta};\ \tau]\!]
\end{aligned}$$

To wrap up our algebraic treatment of Etude terms, we must specify four more administrative theorems. The first of these states that, if one term in a given equivalence class happens to represent a meaningful computation, then all other terms equivalent to it are likewise well-formed:

$$\begin{aligned}
&\text{WF}_{\text{EQV}} :: \forall\ \Lambda_1,\ \Delta_1,\ \tau_1,\ \Lambda_2,\ \Delta_2,\ \tau_2 \Rightarrow \\
&\quad \text{WF}[\![\Lambda_1,\ \Delta_1 \triangleright \tau_1]\!] \rightarrow \\
&\quad [\![\Lambda_1,\ \Delta_1 \triangleright \tau_1]\!] \equiv [\![\Lambda_2,\ \Delta_2 \triangleright \tau_2]\!] \rightarrow \\
&\quad \text{WF}[\![\Lambda_2,\ \Delta_2 \triangleright \tau_2]\!]
\end{aligned}$$

Finally, the "$\equiv$" property of terms must always represent a true equivalence relation, in that the following reflexivity, symmetry and transitivity laws must be satisfied:

$$\begin{aligned}
&\text{REFL}_\tau \quad :: \forall\ \Lambda,\ \Delta,\ \tau \Rightarrow \\
&\quad [\![\Lambda,\ \ \Delta\ \ \triangleright \tau\ ]\!] \equiv [\![\Lambda,\ \ \Delta\ \ \triangleright \tau]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{SYMM}_\tau\ :: \forall\ \Lambda_1,\ \Delta_1,\ \tau_1,\ \Lambda_2,\ \Delta_2,\ \tau_2 \Rightarrow \\
&\quad [\![\Lambda_1,\ \Delta_1 \triangleright \tau_1]\!] \equiv [\![\Lambda_2,\ \Delta_2 \triangleright \tau_2]\!] \rightarrow \\
&\quad [\![\Lambda_2,\ \Delta_2 \triangleright \tau_2]\!] \equiv [\![\Lambda_1,\ \Delta_1 \triangleright \tau_1]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{TRANS}_\tau :: \forall\ \Lambda_1,\ \Delta_1,\ \tau_1,\ \Lambda_2,\ \Delta_2,\ \tau_2,\ \Lambda_3,\ \Delta_3,\ \tau_3 \Rightarrow \\
&\quad [\![\Lambda_1,\ \Delta_1 \triangleright \tau_1]\!] \equiv [\![\Lambda_2,\ \Delta_2 \triangleright \tau_2]\!] \rightarrow \\
&\quad [\![\Lambda_2,\ \Delta_2 \triangleright \tau_2]\!] \equiv [\![\Lambda_3,\ \Delta_3 \triangleright \tau_3]\!] \rightarrow \\
&\quad [\![\Lambda_1,\ \Delta_1 \triangleright \tau_1]\!] \equiv [\![\Lambda_3,\ \Delta_3 \triangleright \tau_3]\!]
\end{aligned}$$

In the following chapters, incomplete expression of the form "$\texttt{LET}\ \bar{\beta};$" that depict Haskell functions of the type "$term_\nu \rightarrow term_\nu$" are often composed together and eventually applied to some tail τ in order to construct a proper Etude term entity. In Haskell, such structures would be normally written as "$(\mathring{\tau}_1 \circ \mathring{\tau}_2 \circ ... \circ \mathring{\tau}_k)\tau$", but their semantic meaning is more obvious when the composition operator "$\circ$" is replaced with the ";" symbol, resulting in the expected term sequence notation "$\mathring{\tau}_1;\ \mathring{\tau}_2;\ ...\ \mathring{\tau}_k;\ \tau$". To give this notation a formal mandate, it is useful to introduce the following two Haskell combinators:

$$[\![\cdot]\!];\ [\![\cdot]\!] :: (term_\nu \rightarrow term_\nu) \rightarrow (term_\nu \rightarrow term_\nu) \rightarrow (term_\nu \rightarrow term_\nu)$$

$$[\![\mathring{\tau}_1]\!];\ [\![\mathring{\tau}_2]\!] = \mathring{\tau}_1 \circ \mathring{\tau}_2$$

$$[\![\cdot]\!];\ [\![\cdot]\!] :: (term_\nu \rightarrow term_\nu) \rightarrow term_\nu \rightarrow term_\nu$$

$$[\![\mathring{\tau}]\!];\ [\![\tau]\!] \quad = \mathring{\tau}\ \tau$$

In other words, in this work the ";" symbol may stand for either the function composition operator "$\circ$" or a function application, as dictated by the expression's context. Either way, the Etude syntax and the associated informal discussion should always make the intended meaning clear, so that, in practice, this overloading of the ";" operator should never impair clarity of the following presentation.

An astute reader will rightly observe that the above set of theorems is, by itself, insufficient to establish confluence of the rewrite system implied by the stated well-formedness and term equivalence laws. However, in Chapter 6, confluence of the MMIX Etude variant is proven constructively by the virtue of its *evaluation function*, which performs a single-step functional reduction of a monadic term into its appropriate normal form. In the meantime, however, the above Spartan system of algebraic relations will have to suffice for all reasoning about generic Etude constructs and the portable C programs from which they have been derived.

4.7 Modules and Programs

Usually, Etude programs are not constructed as a single holistic entity, but rather assembled from one or more discrete *modules* that are designed and compiled as autonomous work units. Any realistic translation of a language such as C must recognise this common software engineering practice and, accordingly, Etude provides the following Haskell structures intended to capture the essential idea of module separation:

$module_{[\![v]\!]}$:
    MODULE $term_v$ EXPORT $exports_v$ WHERE $items_v$

$exports_{[\![v]\!]}$:
    $string \mapsto v$

$items_{[\![v]\!]}$:
    $v \mapsto item_v$

$item_{[\![v]\!]}$:
    $function_v$
    OBJ ($data_v$) OF ($envelope$)
    IMP $string$

$data_{[\![v]\!]}$:
    $\{integer, format, \{mode\}, atom_v\}$

A single module consists of a term known as *module initialiser* and two sets known as *module exports* and *module definitions*, respectively. The initialiser represents an expression evaluated at the beginning of every execution of the program and it is these terms, irrespective of any environments current upon their termination, that dictate all algebraic properties of all software written in Etude. Module exports specify those variables defined within the module that are to be made visible to all other entities within the program. Any such variables are annotated in the module's syntax with their globally-unique names depicted by Haskell string values. Finally, every module

includes a set of zero or more bindings of Etude variables to *module items*, which, intuitively, describe the initial evaluation environment current prior to every invocation of the program. An individual item can represent an Etude function, an *object specification* or a reference to a named declaration imported from some other module of the entire program. Every object specification entity consists, in turn, of an envelope associated with a *data specification set* whose structure resembles another envelope, in which every *datum* has been annotated with a concrete atomic value, intended to represent the initial content of the corresponding address space region within the object. Intuitively, the collection of all object specifications included in an Etude program represents a concise depiction of its initial object environment defined statically prior to the execution of any initialiser terms and corresponds directly to the notion of static variables in the C programming language.

For convenience, the three components of an Etude module may be accessed individually by the following Haskell functions:

$$\tau[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \mathit{module}_\nu \to \mathit{term}_\nu$$
$$\bar{\xi}[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \mathit{module}_\nu \to \mathit{exports}_\nu$$
$$\bar{\iota}\,[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \mathit{module}_\nu \to \mathit{items}_\nu$$

$$\tau[\![\textsf{MODULE } \tau \textsf{ EXPORT } \bar{\xi} \textsf{ WHERE } \bar{\iota}]\!] = \tau$$
$$\bar{\xi}[\![\textsf{MODULE } \tau \textsf{ EXPORT } \bar{\xi} \textsf{ WHERE } \bar{\iota}]\!] = \bar{\xi}$$
$$\bar{\iota}\,[\![\textsf{MODULE } \tau \textsf{ EXPORT } \bar{\xi} \textsf{ WHERE } \bar{\iota}]\!] = \bar{\iota}$$

Although a module represents an irreducible entity that, in isolation, does not possess any specific semantic meaning, we can still refer to the free variable sets of its export and item lists using the following predictable Haskell constructions:

$$\text{FV}[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \mathit{exports}_\nu \to \{\nu\}$$
$$\text{FV}[\![\bar{\xi}]\!] = \{\nu_k \mid [\![x_k = \nu_k]\!] \leftarrow \bar{\xi}\}$$

$$\text{FV}[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \mathit{items}_\nu \to \{\nu\}$$
$$\text{FV}[\![\bar{\iota}]\!] = \bigcup[\text{FV}(\iota_k) \mid [\![\nu_k = \iota_k]\!] \leftarrow \bar{\iota}]$$

$$\text{FV}[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \mathit{item}_\nu \to \{\nu\}$$
$$\begin{aligned}
\text{FV}[\![\lambda\bar{\nu}.\tau]\!] &= \text{FV}(\tau)\backslash\bar{\nu} \\
\text{FV}[\![\textsf{OBJ } (\bar{\delta}) \textsf{ OF } (\bar{\xi})]\!] &= \bigcup[\text{FV}(\alpha_k) \mid [\![n_k, \phi_k, \bar{\mu}_k, \alpha_k]\!] \leftarrow \bar{\delta}] \\
\text{FV}[\![\textsf{IMP } x]\!] &= \varnothing
\end{aligned}$$

Further, we can also formulate the usual substitution function over the list of module items as follows:

$$[\![\cdot]\!]/[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \mathit{items}_\nu \to (\nu \mapsto \mathit{atom}_\nu) \to \mathit{items}_\nu$$
$$[\![\bar{\iota}]\!]/[\![S]\!] = \{[\![\nu_k = [\![\iota_k/S]\!]]\!] \mid [\![\nu_k = \iota_k]\!] \leftarrow \bar{\iota}\}$$

$$[\![\cdot]\!]/[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \mathit{item}_\nu \to (\nu \mapsto \mathit{atom}_\nu) \to \mathit{item}_\nu$$
$$\begin{aligned}
[\![\lambda\bar{\nu}.\tau]\!] &/[\![S]\!] = [\![\lambda\bar{\nu}.[\![\tau/(S\backslash\bar{\nu})]\!]]\!] \\
[\![\textsf{OBJ } (\bar{\delta}) \textsf{ OF } (\bar{\xi})]\!]&/[\![S]\!] = [\![\textsf{OBJ } ([\![\{[\![n_k, \phi_k, \bar{\mu}_k, [\![\alpha_k/S]\!]]\!] \mid [\![n_k, \phi_k, \bar{\mu}_k, \alpha_k]\!] \leftarrow \bar{\delta}\}]\!]) \textsf{ OF } (\bar{\xi})]\!] \\
[\![\textsf{IMP } x]\!] &/[\![S]\!] = [\![\textsf{IMP } x]\!]
\end{aligned}$$

Although all practical Etude implementations, including the one from Chapter 6, are typically defined to perform translations of only a single module at a time, conceptually, the well-formedness and equivalence properties can be only asserted over entire Etude programs. Formally, such programs are formed by annotating every one of its modules m_k with a suitable substitution function S_k, so that the set of modules that constitutes a complete description of an entire self-contained Etude program can be represented by the following Haskell data structure:

$$\begin{aligned} &\mathit{modules}_{[\![\nu]\!]}\,: \\ &\qquad \mathit{module}_\nu \mapsto (\nu \mapsto \mathit{atom}_\nu) \end{aligned}$$

In general, the required substitution functions are constructed by an unspecified *linking process* that is defined individually for each operating system targeted by our compiler and therefore remains outside of the present work's scope. However, on every implementation of the Etude language, the resulting set of modules $\bar{m}$ must always satisfy a number of simple constraints described in the remainder of this section.

First of all, for every module m_k in $\mathrm{dom}(\bar{m})$, the set of all free variables appearing in the module's initialiser term $\tau(m_k)$, exports $\bar{\xi}(m_k)$ and item definitions $\bar{\iota}(m_k)$ must represent a subset of the variables mapped in $\bar{\iota}(m_k)$, which, in turn, must be identical to the domain of the corresponding substitution function $\bar{m}(m_k)$:

$$\begin{aligned} &\mathrm{WF}_{\mathrm{M1}} :: \forall\, \bar{m},\, m_k \Rightarrow \\ &\qquad \mathrm{WF}(\bar{m}) \rightarrow \\ &\qquad [\![m_k \in \mathrm{dom}(\bar{m})]\!] \rightarrow \\ &\qquad [\![(\mathrm{FV}(\tau(m_k)) \cup \mathrm{FV}(\bar{\xi}(m_k)) \cup \mathrm{FV}(\bar{\iota}(m_k))) \subseteq \mathrm{dom}(\bar{\iota}(m_k)) = \mathrm{dom}(\bar{m}(m_k))]\!] \end{aligned}$$

Further, for every variable ν_j bound in this set to a definition of an Etude function item "$\nu_j = \lambda\bar{\nu}_j.\tau_j$", the corresponding substitution function $\bar{m}(m_k)$ must map ν_j to a well-formed atom of the form "$\#[\![n_j]\!]_{\mathsf{F}.\Phi}$" for some integer value n_j:

$$\begin{aligned} &\mathrm{WF}_{\mathrm{M2}} :: \forall\, \bar{m},\, m_k,\, \nu_j,\, \bar{\nu}_j,\, \tau_j \Rightarrow \\ &\qquad \mathrm{WF}(\bar{m}) \rightarrow \\ &\qquad [\![m_k \in \mathrm{dom}(\bar{m}) \wedge \bar{\iota}(m_k)(\nu_j) = [\![\lambda\bar{\nu}_j.\tau_j]\!]]\!] \rightarrow \\ &\qquad \mathrm{WF}[\![\#[\![\mathcal{L}_{\mathsf{F}}(\bar{m}(m_k)(\nu_j))]\!]_{\mathsf{F}.\Phi}]\!] \end{aligned}$$

in which $\mathcal{L}_{\mathsf{F}}(\bar{m}(m_k)(\nu_j))$ abstracts over the concrete value of n_j using the following implementation-defined Haskell construction:

$$\mathcal{L}_{\mathsf{F}}[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \mathit{atom}_\nu \rightarrow \mathit{integer}$$

More so, every variable ν_j that is bound in one of the supplied modules m_k to a function definition of the form "$\nu_j = \lambda\bar{\nu}_j.\tau_j$" must be renamed by $\bar{m}(m_k)$ to an integer value n_j unique across the entire module collection $\bar{m}$:

$$\begin{aligned} &\mathrm{WF}_{\mathrm{M3}} :: \forall\, \bar{m},\, \bar{n} \Rightarrow \\ &\qquad \mathrm{WF}(\bar{m}) \rightarrow \\ &\qquad [\![\bar{n} = [\mathcal{L}_{\mathsf{F}}(\bar{m}(m_k)(\nu_j)) \mid m_k \leftarrow \mathrm{dom}(\bar{m}),\, [\![\nu_j = \lambda\bar{\nu}_j.\tau_j]\!] \leftarrow \bar{\iota}(m_k)]]\!] \rightarrow \\ &\qquad [\![\mathrm{length}(\bar{n}) = |\bar{n}|]\!] \end{aligned}$$

Similarly, for every variable ν_j bound in this set to a definition of an Etude object specification "ν_j = OBJ ($\bar{\delta}_j$) OF ($\bar{\xi}_j$)", the corresponding substitution function $\bar{m}(m_k)$ must map ν_j to a well-formed atom of the form "$\#[\![n_j]\!]_{\mathrm{O}.\Phi}$" for some integer value n_j:

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{M4}} :: \forall\ \bar{m},\ m_k,\ \nu_j,\ \bar{\delta}_j \Rightarrow \\
&\quad \mathrm{WF}(\bar{m}) \rightarrow \\
&\quad [\![m_k \in \mathrm{dom}(\bar{m}) \wedge \bar{\imath}(m_k)(\nu_j) = [\![\mathsf{OBJ}\ (\bar{\delta}_j)\ \mathsf{OF}\ (\bar{\xi}_j)]\!]]\!] \rightarrow \\
&\quad \mathrm{WF}[\![\#[\![\mathcal{L}_{\mathrm{O}}(\bar{m}(m_k)(\nu_j))]\!]_{\mathrm{O}.\Phi}]\!]
\end{aligned}$$

in which, once again, $\mathcal{L}_{\mathrm{O}}(\bar{m}(m_k)(\nu_j))$ abstracts over the concrete value of n_j using the following implementation-defined Haskell construction:

$$\mathcal{L}_{\mathrm{O}}[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \mathit{atom}_{\nu} \rightarrow \mathit{integer}$$

As for function definitions, the list of all such Etude variables must constitute a set of integer values unique across the entire module collection:

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{M5}} :: \forall\ \bar{m} \Rightarrow \\
&\quad \mathrm{WF}(\bar{m}) \rightarrow \\
&\quad [\![\bar{n} = [\mathcal{L}_{\mathrm{O}}(\bar{m}(m_k)(\nu_j)) \mid m_k \leftarrow \mathrm{dom}(\bar{m}),\ [\![\nu_j = \mathsf{OBJ}\ (\bar{\delta}_j)\ \mathsf{OF}\ (\bar{\xi}_j)]\!] \leftarrow \bar{\imath}(m_k)]]\!] \rightarrow \\
&\quad [\![\mathrm{length}(\bar{n}) = |\bar{n}|]\!]
\end{aligned}$$

More so, for every integer n_j associated with an object item "ν_j = OBJ ($\bar{\delta}_j$) OF ($\bar{\xi}_j$)", the precise location of every envelope datum $(n_i,\ \phi_i,\ \bar{\mu}_i,\ \alpha_i) \in \bar{\delta}_j$ must be representable as an atom of the form "$(O(\phi_i)_{\mathrm{O}.\Phi}(\#[\![n_j]\!]_{\mathrm{O}.\Phi})) +_{O(\phi_i)} \#[\![n_i]\!]_{\mathrm{Z}.\Phi}$":

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{M6}} :: \forall\ \bar{m},\ m_k,\ \nu_j,\ \bar{\delta}_j,\ n_i,\ \phi_i,\ \bar{\mu}_i,\ \alpha_i \Rightarrow \\
&\quad \mathrm{WF}(\bar{m}) \rightarrow \\
&\quad [\![m_k \in \mathrm{dom}(\bar{m}) \wedge \bar{\imath}(m_k)(\nu_j) = [\![\mathsf{OBJ}\ (\bar{\delta}_j)\ \mathsf{OF}\ (\bar{\xi}_j)]\!] \wedge (n_i,\ \phi_i,\ \bar{\mu}_i,\ \alpha_i) \in \bar{\delta}_j]\!] \rightarrow \\
&\quad \mathrm{WF}[\![([\![O(\phi_i)]\!]_{\mathrm{O}.\Phi}(\#[\![\mathcal{L}_{\mathrm{O}}(\bar{m}(m_k)(\nu_j))]\!]_{\mathrm{O}.\Phi})) +_{O(\phi_i)} \#[\![n_i]\!]_{\mathrm{Z}.\Phi}]\!]
\end{aligned}$$

Finally, across the entire module collection, every variable ν_j bound to an imported item of the form "IMP x_j" must be renamed to the same value as that associated with the corresponding export definition found in some other module in the program. Formally:

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{M7}} :: \forall\ \bar{m},\ m_k,\ \nu_j,\ x_j \Rightarrow \\
&\quad \mathrm{WF}(\bar{m}) \rightarrow \\
&\quad [\![m_k \in \mathrm{dom}(\bar{m}) \wedge \bar{\imath}(m_k)(\nu_j) = [\![\mathsf{IMP}\ x_j]\!]]\!] \rightarrow \\
&\quad [\![\bar{m}(m_k)(\nu_j) = \bar{\xi}(\bar{m})(x_j)]\!]
\end{aligned}$$

where the notation "$\bar{\xi}(\bar{m})$" constructs a finite map from exported definition names to their respective atomic values as follows:

$$\bar{\xi}[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow (\mathit{module}_{\nu} \mapsto (\nu \mapsto \mathit{atom}_{\nu})) \rightarrow (\mathit{string} \mapsto \mathit{atom}_{\nu})$$

$$\bar{\xi}[\![\bar{m}]\!] = \{x_k = [\![S_k(\nu_k)]\!] \mid m_k{:}S_k \leftarrow \bar{m},\ x_k = \nu_k \leftarrow \bar{\xi}(m_k)\}$$

Intuitively, for every module m_k that is associated in $\bar{m}$ with the substitution function S_k and for every variable ν_k exported from that module as string x_k, the above $\bar{\xi}(\bar{m})$ construction includes a binding of the string x_k to the atom associated with ν_k by the substitution function S_k.

Last but not least, every well-formed collection of Etude modules $\bar{m}$ must represent a valid Etude program:

$$\text{WF}_{\text{M8}} :: \forall\ \bar{m} \Rightarrow \text{WF}(\bar{m}) \rightarrow \text{WF}(\mathcal{L}(\bar{m}))$$

derived from the included module list by the following simple process:

$$\mathcal{L}[\![\cdot]\!] :: (\text{ord}\ \nu) \Rightarrow (\mathit{module}_\nu \mapsto (\nu \mapsto \mathit{atom}_\nu)) \rightarrow (\mathit{f\text{-}env}_\nu,\ \mathit{o\text{-}env} \triangleright \mathit{term}_\nu)$$

$$\begin{aligned}
\mathcal{L}[\![\bar{m}]\!] &= [\![\Lambda_0,\ \Delta_0 \triangleright \texttt{LET}\ \bar{\beta}_0;\ \texttt{RET}\ ()]\!] \\
\text{where}\ \Lambda_0 &= \{[\![[\![\mathcal{L}_\text{O}(S_k(\nu_j))]\!] = \lambda\bar{\nu}_j.[\![\tau_j/S_k]\!]]\!] \\
&\qquad\qquad \mid m_k{:}S_k \leftarrow \bar{m},\ [\![\nu_j = \lambda\bar{\nu}_j.\tau_j]\!] \leftarrow \bar{\iota}(m_k)\} \\
\Delta_0 &= \mathcal{L}_\Delta\{\mathcal{L}_\text{F}(S_k(\nu_j)){:}[\![\texttt{OBJ}\ (\bar{\delta}_j)\ \texttt{OF}\ (\bar{\xi}_j)]\!]/S_k \\
&\qquad\qquad \mid m_k{:}S_k \leftarrow \bar{m},\ [\![\nu_j = \texttt{OBJ}\ (\bar{\delta}_j)\ \texttt{OF}\ (\bar{\xi}_j)]\!] \leftarrow \bar{\iota}(m_k)\} \\
\bar{\beta}_0 &= [[\![() = [\![\tau(m_k)/S_k]\!]]\!] \mid m_k{:}S_k \leftarrow \bar{m}]
\end{aligned}$$

In other words, the program constructed by the linking process evaluates, in an unspecified order, the initialiser terms of all the modules in the collection. Every one of these initialiser terms $\tau(m_k)$ must represent a monadic construct that delivers an empty set of Etude atoms "RET ()", so that the entire program constitutes a simple term of the form "LET $\bar{\beta}_0$; RET ()". Its initial function environment Λ_0 is constructed from all the function definitions found anywhere across the entire collection, while the initial object environment Δ_0 is obtained from the set of suitably-renamed versions of all object specifications in the individual item lists $\bar{\iota}(m_k)$ using the following unspecified construction:

$$\mathcal{L}_\Delta[\![\cdot]\!] :: (\text{ord}\ \nu) \Rightarrow (\mathit{integer} \mapsto \mathit{item}_\nu) \rightarrow \mathit{o\text{-}env}$$

such that, for every $(n_k,\ \phi_k,\ \bar{\mu}_k)$ and $(n_k,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k)$ included in the specification of an object mapped in $\bar{\iota}$ at some address k, the initial object environment $\mathcal{L}_\Delta(\bar{\iota})$ includes a binding of $(k + n_k,\ \phi_k)$ to the atom α_k under the set of access attributes $\bar{\mu}_k$. Formally:

$$\begin{aligned}
&\text{LINK}_1 :: \forall\ \bar{\iota},\ k,\ n_j,\ \phi_j,\ \bar{\mu}_j,\ \alpha_j \Rightarrow \\
&\qquad \text{WF}(\mathcal{L}_\Delta(\bar{\iota})) \rightarrow \\
&\qquad [\![\bar{\iota}(k) = [\![\texttt{OBJ}\ (\bar{\delta}_k)\ \texttt{OF}\ (\bar{\xi}_k)]\!]]\!] \rightarrow \\
&\qquad [\![(n_j,\ \phi_j,\ \bar{\mu}_j) \in \bar{\xi}_k \vee (n_j,\ \phi_j,\ \bar{\mu}_j,\ \alpha_j) \in \bar{\delta}_k]\!] \rightarrow \\
&\qquad [\![(k + n_j,\ \phi_j,\ \bar{\mu}_j) \in_\text{A} \bar{\xi}(\mathcal{L}_\Delta(\bar{\iota}))]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{LINK}_2 :: \forall\ \bar{\iota},\ k,\ n_j,\ \phi_j,\ \bar{\mu}_j,\ \alpha_j \Rightarrow \\
&\qquad \text{WF}(\mathcal{L}_\Delta(\bar{\iota})) \rightarrow \\
&\qquad [\![\bar{\iota}(k) = [\![\texttt{OBJ}\ (\bar{\delta}_k)\ \texttt{OF}\ (\bar{\xi}_k)]\!]]\!] \rightarrow \\
&\qquad [\![(n_j,\ \phi_j,\ \bar{\mu}_j,\ \alpha_j) \in \bar{\delta}_k]\!] \rightarrow \\
&\qquad [\![\bar{\alpha}(\mathcal{L}_\Delta(\bar{\iota}) \triangleright k + n_j,\ \phi_j)]\!] \equiv [\![\alpha_j]\!])
\end{aligned}$$

The reader should observe that the structure of Etude data specification sets allows programs to place multiple data elements at overlapping address space regions within a single memory-resident object, with an understanding that the meaning of such constructions is well-defined only if the binary representations of the respective atomic values assigned to any such elements are in agreement as to the object's initial memory image.

For a representative sample of typical Etude programs, the reader is referred to the number of examples featured later in Sections 5.1, 5.11 and 6.5.

5

THE C PROGRAMMING LANGUAGE

Philosophy is a battle against the bewitchment
of our intelligence by means of language.
— Ludwig Wittgenstein

The C programming language was originally developed by Kernighan and Ritchie at Bell Labs in 1970s, as part of their work on the UNIX operating system for the PDP-7 architecture [Kernighan 78]. It provides a quintessential imperative design environment in the ALGOL tradition, whose wide availability, simplicity and access to low-level programming features has rendered it quickly into the favourite tool of the UNIX community from the 1980s. Since its introduction, the user base of C has been increasing steadily across the whole spectrum of the software development industry and, today, it arguably remains the single most popular programming language in existence. To meet the growing demand for portability, C has been ratified into an international standard ANSI/ISO 9899:1990 and further refined in 2000 by the same regulatory group [ANSI 89, ANSI 99].

In recent years, C saw a growing deployment in software written for embedded devices and, with these developments, a pressing need for a formal verification of C programs. Such verification necessarily involves a mathematically rigorous definition of the language. In the past, numerous authors have undertaken the task of devising a formal semantic model of C, with the most recent, successful and complete treatments given by Norrish and Papaspyrou [Norrish 97, Papaspyrou 98]. Nevertheless, to the best of my knowledge, none of these models have been ever incorporated into a fully featured compiler for the language and, accordingly, in this chapter I revisit the task of formalising Standard C by presenting a complete and detailed translation of its programs into Etude, the variant of lambda calculus defined in Chapter 4. To make the work more specific, I focus the project on the original version of the Standard described in the ANSI/ISO 9899:1990 document [ANSI 89]. The formal definition presented in this chapter covers all aspects of C programs with the sole exception of their lexical and syntactic analysis, which are omitted in recognition of the already-extensive treatment given to these subjects in literature [Aho 86, Appel 98ML]. Coupled with the partial Etude semantics presented earlier in Chapter 4, the translation formalises precisely the set of requirements described in the C Standard [ANSI 89]. Further, in combination with the mapping of Etude programs to executable machine instructions presented in Chapter 6, it also forms a prototype compiler implementation, that, although inefficient as given (preferring clarity of presentation over optimality of the generated code), could

be easily improved into an industrial-quality development environment, while ensuring that the result always remains provably equivalent to its present canonical definition.

Throughout this chapter, I carefully avoid any over-specification of those aspects of the source language that the C Standard leaves either unspecified or open to further specification by individual compiler vendors. Following the principles discussed earlier in Section 4.3, all such linguistic features are described without a concrete implementation in terms of explicit *language parameters*, whose minimal semantic requirements are captured in appropriate algebraic theorems. For example, in Section 5.4.5, I formalise the precise representation of C types in terms of a function ϕ, together with a set of 42 theorems that describe the minimal constraints on its definition which must be satisfied by every instance of the language. A typical implementation of ϕ is given separately in Appendix C, together with definitions of all other implementation-specific language parameters that are introduced throughout this chapter.

The reader should also keep in mind that the resulting formal semantics of C rely fundamentally on the Etude program representation given in Chapter 4, and especially on the generic fragment of its semantics presented in Sections 4.4, 4.5 and 4.6. In particular, it should be emphasised that many aspects of C are rendered implementation-defined, unspecified or undefined indirectly through translation into Etude LET group bindings, for which theorems WF_{LETM} and EQV_{LETM} from Section 4.6 provide only a very rudimentary set of operational guarantees. The minimal documentation requirements mandated by the C Standard for all such indirectly-defined aspects of the language are presented in Section 6.3, in the form of a concrete operational semantics for Etude on the MMIX instruction set architecture.

5.1 Overview

Under the formal semantic model of C presented in this chapter, every well-formed translation unit is represented naturally by an Etude module. Global and static variables are mapped to mutable object items in the constructed program, while all C functions and statement labels are rendered as Etude functions. To illustrate the intended behaviour of this translation, let us consider the following simple C program that computes the factorial of an integer x:

```
int fac(int x) {
        int r, i;
        r = 1;
        i = 1;
        while (i <= x) {
                r *= i;
                i += 1;
        }
        return r;
}
```

This program could be mapped into the following set of three Etude functions:

$fac = \lambda x.$
LET r = $\#1_{\mathsf{Z.\Phi}}$;
LET i = $\#1_{\mathsf{Z.\Phi}}$;
$fac_1(x, r, i)$

$fac_1 = \lambda x, r, i.$
IF $i \leq_{\mathsf{Z.\Phi}} x$ THEN
  LET r = $r \times_{\mathsf{Z.\Phi}} i$;
  LET i = $i +_{\mathsf{Z.\Phi}} \#1_{\mathsf{Z.\Phi}}$;
  $fac_1(x, r, i)$
ELSE
  $fac_2(x, r, i)$

$fac_2 = \lambda x, r, i.$
RET (r)

Under this mapping, every C function and statement label (including the implicit "**`continue`**" and "**`break`**" labels defined by every iteration statement) is represented by an Etude function which acccepts as its parameters the list of all local C variables that are visible within its scope. Jumps are translated into Etude tail calls and function calls are translated into monadic bindings of appropriate Etude application expressions.

While this natural translation works for the current elementary example, in most cases Etude variables are not a very good match for their C counterparts, since they cannot be accessed indirectly through pointer references in the way that C variables can. In a more accurate translation, every C variable should denote a mutable Etude object whose content corresponds to the variable's value. Accordingly, the translation defined in this chapter produces a representation of "*fac*" akin to the following Etude program:

$fac' = \lambda x.$
LET r = NEW $(0, \mathsf{Z.\Phi})$;
LET i = NEW $(0, \mathsf{Z.\Phi})$;
LET $()$ = SET $[r]_{\mathsf{Z.\Phi}}$ TO $\#1_{\mathsf{Z.\Phi}}$;
LET $()$ = SET $[i]_{\mathsf{Z.\Phi}}$ TO $\#1_{\mathsf{Z.\Phi}}$;
$fac_2'(x, r, i)$

$fac_1' = \lambda x, r, i.$
LET t_1 = GET $[i]_{\mathsf{Z.\Phi}}$,
  t_2 = GET $[x]_{\mathsf{Z.\Phi}}$;
IF $t_1 \leq_{\mathsf{Z.\Phi}} t_2$ THEN
  LET t_1 = GET $[r]_{\mathsf{Z.\Phi}}$,
    t_2 = GET $[i]_{\mathsf{Z.\Phi}}$;
  LET $()$ = SET $[r]_{\mathsf{Z.\Phi}}$ TO $t_1 \times_{\mathsf{Z.\Phi}} t_2$;
  LET t_1 = GET $[i]_{\mathsf{Z.\Phi}}$;
  LET $()$ = SET $[i]_{\mathsf{Z.\Phi}}$ TO $t_1 +_{\mathsf{Z.\Phi}} \#1_{\mathsf{Z.\Phi}}$;
  $fac_2'(x, r, i)$
ELSE
  $fac_2'(x, r, i)$

$fac_2' = \lambda x, r, i.$
LET t_1 = GET $[r]_{\mathsf{Z.\Phi}}$;
LET $()$ = DEL $(0, \mathsf{Z.\Phi})$;
LET $()$ = DEL $(0, \mathsf{Z.\Phi})$;
RET (t_1)

Under this mapping, every C variable is represented by a memory-resident object with an envelope derived from that variable's C type. For example, in the above code

fragment, *r* and *i* represent integer variables, so that all of the corresponding envelopes contain a single tuple of the form "(0, Z.Φ)", which binds the envelope offset 0 to the standard integer format "Z.Φ" with an empty set of access attributes. All such objects are deallocated at the end of their C scope by appropriate "DEL" terms, invoked in the reverse order of the objects' introduction. Further, since the C Standard does not prescribe a specific order of evaluation for most subexpressions in the program, the corresponding Etude terms are always evaluated within a single Etude group, such as that used to bind the temporary variables t_1 and t_2 in the above example.

Although this version of the program is arguably less pleasant than the earlier translation, it carries the benefit of a uniform treatment of all C expressions and scopes. Since the Etude term equivalence relation defined in Chapter 4 considers only the sequence of actual system calls invoked during a program's evaluation and not the resulting memory image, in the context of an entire program both of the above representations of "*fac*" may, in principle, be deemed synonymous, which a smarter implementation of the compiler could easily utilise to produce an even more efficient representation of that function, either immediately during its translation into Etude terms, or else through a set of generic term transformations applied to the constructed Etude modules in an optimising back end of the compiler. The linear approach to compiler design specifically promotes such performance improvements through its formulation of the linear correctness criteria.

5.2 Notation and Lexical Syntax

For conciseness, the translation presented in this chapter omits details of a textual analysis of C programs, effectively operating entirely on a parse tree of a single translation unit. Nevertheless, the structural resemblance of that representation to the formal BNF grammar defined in the C Standard should provide sufficient clues for a future implementation and verification efforts of these remaining stages of the compilation process, using any one of a number of well-established techniques described extensively in literature [Aho 86, Appel 98ML].

The parse tree of a C program is an inductive data structure pieced together from various nodes that represent *keywords*, *tokens* and *non-terminals*. Keywords carry no information and are therefore omitted from the implementation. In typesetting, they are written in bold monospaced font to distinguish them visually from other Haskell constructs; "**`int`**", "**`while`**" and "**`==`**" are three examples of such keywords. Keywords introduced by entities that do not appear within the concrete syntax of C programs are typeset in an italic font such as "***`int`***" to distinguish them from any occurrences of similar lexemes in the actual source files.

The remaining nodes are collectively known as *syntactic entities*, every one of which is defined by a distinct Haskell data type in a manner similar to the earlier treatment of Etude's syntax in Chapter 4. Although, in a formal description of the language,

syntactic entities represent ordinary Haskell terms, for the sake of exposition every occurrence of such an entity within a general Haskell expression is surrounded by the semantic brackets "$[\![]\!]$" as described earlier in Chapter 3. Further, in order to provide a visual separation between the syntactic and semantic aspects of the language, every occurrence of a non-trivial general Haskell expression within an entity's syntax is itself surrounded by the same semantic brackets as in "$[\![[\![\mathcal{B}(t)]\!]\star]\!]$", in which the subexpression "$\mathcal{B}(t)$" represents a single element of a larger syntactic form "$[\![t'\star]\!]$". If a given syntactic entity has a context-sensitive interpretation, a suitable formal representation of that context is also inserted into the semantic brackets, where it appears ahead of the entity itself and is separated from it by the "$\triangleright$" symbol as in "$\mathcal{D}[\![S, T, D, I \triangleright e]\!]$", in which e depicts a syntactic entity that is be interpreted in the context of four elements S, T, D and I.

In general, the meaning of every syntactic C entity e in its lexical context Γ is represented by an appropriate denotation written as "$\mathcal{D}[\![\Gamma \triangleright e]\!]$". In order to meet the minimal diagnostic requirements mandated by the C Standard, most of such denotations are represented by monadic Haskell constructions obtained in the context of some unspecified monad M from the Haskell class "monad-fix", so that any violations of semantic constraints whose identification is required by the language definition can be detected as a monadic exception raised using the standard Haskell function "fail", written as "reject" in this presentation. For clarity, most of such exceptions are raised by assertions of the form "require P", in which P is an appropriate boolean Haskell expression, whose value must be true in every valid C program. Further, I rely heavily on a monadic combinator "||", defined in Appendix A in such a way that "$M_1 \,||\, M_2$" reduces to the first of the two monadic actions M_1 and M_2 which does not represent a failure. In effect, in the common pattern:

```
  do require P1
     M1
|| do require P2
     M2
     ⋮
|| do require Pn
     Mn
```

the boolean expressions P_1 ... P_n split the analysis into n distinct cases, scrutinised by the associated monadic actions M_1 ... M_n, so that, if such a Haskell expression was to be used to describe some syntactic entity e, the entity's ultimate meaning would be determined by the first successful action M_i for which the corresponding condition P_i holds. In general, I usually formulate the n predicates in such a way as to ensure that at most one of them will ever hold for any given entity, so that the resulting semantics should remain unaffected by the order in which the cases are analysed in the presentation. As a result, our compiler could be easily coerced into producing a more meaningful diagnostic messages through a suitable refinement of the monad M.

The reader should observe however, that, in this work, the *M* monad serves no other purpose beyond error detection and that the semantic context of every entity is always propagated explicitly throughout the translation process, so that *M* is not expected to provide any specific state management facilities.

5.3 Identifiers and Variables

In the concrete syntax of C, tokens known as *identifiers* are used to designate various semantic entities such as variables, functions and members of structure or union types. In the present formal definition of its semantics, these identifiers are represented simply as Haskell string values, in order to render them usable as names of exported Etude items. Nevertheless, to improve the clarity of our discussion, it is useful to define a distinct Haskell type intended solely for depiction of C identifiers. Formally, this simple type is introduced by the following Haskell synonym declaration:

identifier :
    string

Although the discussion in Section 5.1 suggests that names such as "*fac*", "*x*" and "*i*" assigned by a programmer to the individual C entities are retained under their translation into Etude programs, in reality C identifiers are not a very good match for Etude variables. In particular, the C programming language has its own set of idiosyncratic rules that define visibility of names within the program, which are often orthogonal to the actual life spans of objects associated with these entities by the translation. While the match is perfect for global individuals such as C functions, in most cases, the entity designated by an identifier may persist long after the identifier itself ceased to be visible within the program and, in some cases, certain entities such as string literals are not associated with any C identifier at all. In recognition of these difficulties, all Etude variables that appear in the translated program are represented uniformly by the following Haskell data type "*v*":

v :

identifier	*(external variables)*
$\mathrm{V}_{integer}$	*(internal variables)*
$\mathrm{T}_{integer}$	*(temporary variable)*

The "*identifier*" form of these Etude variables is reserved solely for representation of C entities such as functions and named global objects whose declarations appear outside of any function definitions, while all other constructs are bound systematically to unique variable names of the form "V_n", in which the integer *n* represents a globally-unique index introduced by a simple algorithm described later in Section 5.5. More so, temporary variables of the form "T_n" are reserved for internal use within individual Etude terms constructed by the translation and, as a rule, will never appear free within a denotation of a complete C entity, thus ensuring that they are always available for an unrestricted use by any other closed term in the program.

5.4 Types

During translation, many C entities such as function declarations and members of a structure or union object are annotated with various auxiliary information used to provide further information about the entity's semantics. In standard C nomenclature, these annotations are known as *types*, although they bear little resemblance to their type-theoretic namesakes. In the actual source files, such types are usually depicted using the "*type-name*" syntax defined later in Section 5.8.9. However, for historical reasons, this syntax is unsuitable for a direct deployment during translation and, accordingly, most C compilers resort to an abstracted denotation of these entities during analysis of C programs. In the following presentation, these abstract entities are always referred to as *C types* in order to distinguish them from their Haskell cousins.

5.4.1 Abstract Syntax

The language defines fifteen distinct C type forms, described collectively by the following Haskell constructions:

type:

`signed` *integer-specifier*	*(signed integer types)*
`unsigned` *integer-specifier*	*(unsigned integer types)*
`char`	*(plain character type)*
`int`	*(plain integer type)*
`enum` *type* ν	*(enumeration types)*
`float`	*(single precision floating type)*
`double`	*(double precision floating type)*
`long double`	*(extended floating point type)*
struct-or-union ν *members*	*(structure and union types)*
type **`*`**	*(pointer types)*
type **`[`** $integer_{opt}$ **`]`**	*(array types)*
type **`(`** $prototype_{opt}$ **`)`**	*(function types)*
type **`:`** *integer* **`.`** *integer*	*(bit field types)*
`void`	*(the void type)*
type-qualifier type	*(variable types)*

types:

[*type*]	*(a list of types)*

integer-specifier: one of

`char short int long`	*(integral type variants)*

member:

type $identifier_{opt}$ **`@`** *integer*	*(structure and union members)*

members:

[*member*]	*(member lists)*

prototype:

types	
types **`...`**	*(function prototypes)*

where the two Haskell types "*struct-or-union*" and "*type-qualifier*" used in the above definition are discussed separately in Sections 5.8.4 and5.8.7 later in this chapter.

As described later in Section 5.8, in the concrete syntax of the language most of these types are usually depicted by sequences of C tokens similar to those found in their abstract names given above. In particular, structure, union and enumeration types of the form "**struct** *v*", "**union** *v*" and "***enum*** *t v*" are introduced into C programs using the "*struct-or-union-specifier*" and "*enum-specifier*" entities defined in Sections 5.8.4 and 5.8.5. On the other hand, pointer, array and function types are represented by the concrete syntax of C *declarators* discussed in Section 5.8.8, whose enormous complexity is the main reason for introduction of the above abstract structures into the present formal specification of the language. Finally, *bit field types* of the form "*t* **:** *n* **.** *m*" can be introduced only as part of a larger structure or union declaration as described in Section 5.8.4. Within such bit fields, the "***int***" type assumes a slightly unusual semantics, whereas, in every other context, it is precisely equivalent to "***signed int***".

5.4.2 Corresponding Types

Many type properties are left unspecified in the C Standard, with an intention that a suitable choice of their values should be made in consideration of various architecture-specific factors. In the following presentation, such properties of C types are usually described using Haskell signatures and theorems instead of complete definitions. Nevertheless, most of the partially-specified type forms are guaranteed to share certain properties with some other predetermined entity known as that form's *corresponding type*. Five such designated C types are introduced by the following Haskell signature:

char_t, ***int_t***, ***wchar_t***, ***ptrdiff_t***, ***size_t*** :: *type*

Informally, "***char_t***" and "***int_t***" depict the corresponding types of their "***char***" and "***int***" variants, with "***int_t***" used only during analysis of the bit field type form "***int*** *:m .n*". Both of these types are always guaranteed to be equal to a "***signed***" or "***unsigned***" version of "***char***" and "***int***", respectively:

$\mathrm{WF}_{\texttt{char\_t}} :: [\![\texttt{char\_t} \in \{\texttt{signed char}, \texttt{unsigned char}\}]\!]$
$\mathrm{WF}_{\texttt{int\_t}} :: [\![\texttt{int\_t} \in \{\texttt{signed int}, \texttt{unsigned int}\}]\!]$

Of the remaining three types, "***wchar_t***" is used solely in formalisation of certain *initialiser* forms described in Section 5.8.12, while "***ptrdiff_t***" and "***size_t***" are required to capture various aspects of pointer arithmetic discussed throughout Section 5.7, without committing to its details in the generic specification of the language. Each of these types is guaranteed to be equal to one of the ten *basic integral types*, as captured by the following three theorems:

$\mathrm{WF}_{\texttt{wchar\_t}} :: [\![\texttt{wchar\_t} \in \mathit{BIT}]\!]$
$\mathrm{WF}_{\texttt{ptrdiff\_t}} :: [\![\texttt{ptrdiff\_t} \in \mathit{BIT}]\!]$
$\mathrm{WF}_{\texttt{size\_t}} :: [\![\texttt{size\_t} \in \mathit{BIT}]\!]$

where "*BIT*" represents the complete set of all basic integral type forms. Formally, this set consists of the eight "***signed***" and "***unsigned***" type variants together with their

plain "***char***" and "***int***" cousins, as specified by the following Haskell definition:

```
BIT = {
    signed char, signed short, signed int, signed long,
    unsigned char, unsigned short, unsigned int, unsigned long,
    char, int
}
```

An example of the most popular implementation of the five corresponding types described above, also suitable for the MMIX architecture from Chapter 6, is presented separately in Appendix C.

The reader should also observe that every enumeration type form "***enum*** *t v*" corresponds to another basic integral type *t*. That type, however, is always specified explicitly by every instance of such an entity, since, in some C compilers, its choice may vary between enumerations. For a further discussion of this issue, an interested reader is referred to the definition of enumeration specifiers presented later in Section 5.8.5.

5.4.3 Properties of Types

For convenience, the above syntactic forms of C types are grouped into a number of *type families*, every one of which is identified by an appropriate Haskell predicate function. In particular, types of the form "***enum*** *t v*", and all "**const**" and "**volatile**"-qualified versions thereof, are known as *enumeration types*. Such types are meaningful only if *t* represents one of the ten basic integral types described previously in Section 5.4.2 and if no type qualifier is applied to the enumeration more than once in its syntax. Formally, this group of types is identified by the following predicate "ET":

$$\begin{aligned}
&\mathrm{ET}[\![\cdot]\!] :: \mathit{type} \to \mathit{bool} \\
&\\
&\mathrm{ET}[\![\textbf{\textit{enum}}\ t\ v]\!] = t \in \mathit{BIT} \\
&\mathrm{ET}[\![q\ t]\!] \quad\ = q \notin \mathrm{tq}(t) \wedge \mathrm{ET}(t) \\
&\mathrm{ET}[\![\mathit{other}]\!] \quad = \mathrm{false}
\end{aligned}$$

in which the notation "$q \notin \mathrm{tq}(t)$" is discussed later in Section 5.4.4.

Similarly, the family of *bit field types* is identified by the predicate "BF". It consists of all qualified or unqualified versions of the type forms "***signed int*** *:m . n*", "***unsigned int*** *:m . n*" and "***int*** *:m . n*", provided that both *m* and *n* represent non-negative integers and *m* has a value no greater than $\omega \times \mathcal{S}(t) - n$, where ω is the byte width parameters from Section 4.4 and $\mathcal{S}(t)$ represents the *size* of the C type *t*, as described later in Section 5.4.6. Formally:

$$\begin{aligned}
&\mathrm{BF}[\![\cdot]\!] :: \mathit{type} \to \mathit{bool} \\
&\\
&\mathrm{BF}[\![t\,\textbf{\textit{:m . n}}]\!] = t \in \{\textbf{\textit{signed int}}, \textbf{\textit{unsigned int}}, \textbf{\textit{int}}\} \wedge \\
&\qquad\qquad\qquad n \geq 0 \wedge 0 \leq m \leq \omega \times \mathcal{S}(t) - n \\
&\mathrm{BF}[\![q\ t]\!] \quad\ = q \notin \mathrm{tq}(t) \wedge \mathrm{BF}(t) \\
&\mathrm{BF}[\![\mathit{other}]\!] \quad = \mathrm{false}
\end{aligned}$$

In such types, the integers *m* and *n* are known as the *bit width* and *offset*, respectively.

For convenience, they can be retrieved from an arbitrary bit field type using the following pair of Haskell functions:

$\mathcal{W}[\![\cdot]\!],\ O[\![\cdot]\!] :: \mathit{type} \to \mathit{integer}$

$\mathcal{W}[\![t\,\texttt{:}m\,\texttt{.}\,n]\!] = m$
$\mathcal{W}[\![q\ t]\!] \quad = \mathcal{W}(t)$

$O\ [\![t\,\texttt{:}m\,\texttt{.}\,n]\!] = n$
$O\ [\![q\ t]\!] \quad = O(t)$

Intuitively, the offset of a bit field type determines its location within a larger object described by the associated base type, while its width represents the number of actual bits occupied by the object.

Collectively, the four "***signed***" type forms, "***int***", "***signed int*** *:m . n*" and all qualified versions thereof are known as *signed integral types*. Further, the plain "***char***" type, "***int*** *:m . n*" and enumeration types "***enum*** *t v*" are included in this family if their corresponding types "***char_t***", "***int_t*** *:m . n*" and *t* are themselves signed. In Haskell, these criteria can be captured concisely by the following Haskell predicate "ST":

$\mathrm{ST}[\![\cdot]\!] :: \mathit{type} \to \mathit{bool}$

$\mathrm{ST}[\![\texttt{signed char}]\!] = \mathrm{true}$
$\mathrm{ST}[\![\texttt{signed short}]\!] = \mathrm{true}$
$\mathrm{ST}[\![\texttt{signed int}]\!] = \mathrm{true}$
$\mathrm{ST}[\![\texttt{signed long}]\!] = \mathrm{true}$
$\mathrm{ST}[\![\texttt{int}]\!] = \mathrm{true}$
$\mathrm{ST}[\![\texttt{char}]\!] = \mathrm{ST}[\![\texttt{char\_t}]\!]$
$\mathrm{ST}[\![\texttt{enum}\ t\ v]\!] = \mathrm{ET}[\![\texttt{enum}\ t\ v]\!] \wedge \mathrm{ST}(t)$
$\mathrm{ST}[\![\texttt{signed int}\,\texttt{:}m\,\texttt{.}\,n]\!] = \mathrm{BF}[\![\texttt{signed int}\,\texttt{:}m\,\texttt{.}\,n]\!]$
$\mathrm{ST}[\![\texttt{int}\,\texttt{:}m\,\texttt{.}\,n]\!] = \mathrm{ST}[\![\texttt{int\_t}\,\texttt{:}m\,\texttt{.}\,n]\!]$
$\mathrm{ST}[\![q\ t]\!] = q \notin \mathrm{tq}(t) \wedge \mathrm{ST}(t)$
$\mathrm{ST}[\![\mathit{other}]\!] = \mathrm{false}$

Similarly, the family of *unsigned integral types*, identified with the predicate "UT", is defined trivially as follows:

$\mathrm{UT}[\![\cdot]\!] :: \mathit{type} \to \mathit{bool}$

$\mathrm{UT}[\![\texttt{unsigned char}]\!] = \mathrm{true}$
$\mathrm{UT}[\![\texttt{unsigned short}]\!] = \mathrm{true}$
$\mathrm{UT}[\![\texttt{unsigned int}]\!] = \mathrm{true}$
$\mathrm{UT}[\![\texttt{unsigned long}]\!] = \mathrm{true}$
$\mathrm{UT}[\![\texttt{char}]\!] = \mathrm{UT}[\![\texttt{char\_t}]\!]$
$\mathrm{UT}[\![\texttt{enum}\ t\ v]\!] = \mathrm{ET}[\![\texttt{enum}\ t\ v]\!] \wedge \mathrm{UT}(t)$
$\mathrm{UT}[\![\texttt{unsigned int}\,\texttt{:}m\,\texttt{.}\,n]\!] = \mathrm{BF}[\![\texttt{signed int}\,\texttt{:}m\,\texttt{.}\,n]\!]$
$\mathrm{UT}[\![\texttt{int}\,\texttt{:}m\,\texttt{.}\,n]\!] = \mathrm{UT}[\![\texttt{int\_t}\,\texttt{:}m\,\texttt{.}\,n]\!]$
$\mathrm{UT}[\![q\ t]\!] = q \notin \mathrm{tq}(t) \wedge \mathrm{UT}(t)$
$\mathrm{UT}[\![\mathit{other}]\!] = \mathrm{false}$

Further, the three basic types "***char***", "***signed char***" and "***unsigned char***" are often referred to as *character types*. They are identified by the following Haskell

predicate, observing that the wide character type "**`wchar_t`**" is excluded from this type family, even though, in a sense, it is also used to depict character values:

$\mathrm{CHR}[\![\cdot]\!] :: \mathit{type} \to \mathit{bool}$

$$\begin{array}{ll}
\mathrm{CHR}[\![\texttt{char}]\!] & = \mathrm{true} \\
\mathrm{CHR}[\![\texttt{signed char}]\!] & = \mathrm{true} \\
\mathrm{CHR}[\![\texttt{unsigned char}]\!] & = \mathrm{true} \\
\mathrm{CHR}[\![q\ t]\!] & = q \notin \mathrm{tq}(t) \wedge \mathrm{CHR}(t) \\
\mathrm{CHR}[\![\mathit{other}]\!] & = \mathrm{false}
\end{array}$$

More so, the family of *floating types* consists of the three syntactic forms "**`float`**", "**`double`**" and "**`long double`**", together with all properly-qualified versions thereof. Intuitively, these types describe *floating point numbers*, usually of the kind defined in the *IEEE Standard for Binary Floating-Point Arithmetic* [IEEE 754]. Formally, they are identified as follows:

$\mathrm{FLT}[\![\cdot]\!] :: \mathit{type} \to \mathit{bool}$

$$\begin{array}{ll}
\mathrm{FLT}[\![\texttt{float}]\!] & = \mathrm{true} \\
\mathrm{FLT}[\![\texttt{double}]\!] & = \mathrm{true} \\
\mathrm{FLT}[\![\texttt{long double}]\!] & = \mathrm{true} \\
\mathrm{FLT}[\![q\ t]\!] & = q \notin \mathrm{tq}(t) \wedge \mathrm{FLT}(t) \\
\mathrm{FLT}[\![\mathit{other}]\!] & = \mathrm{false}
\end{array}$$

In C, qualified and unqualified types of the form "t **`*`**" are known as *pointer types*. In this work, they are associated with the type family predicate "PTR". The type t from which such pointers are derived must represent a valid object, function or incomplete type, as depicted by the "OBJ", "FUN" and "INC" predicates defined shortly:

$\mathrm{PTR}[\![\cdot]\!] :: \mathit{type} \to \mathit{bool}$

$$\begin{array}{ll}
\mathrm{PTR}[\![t\,\texttt{*}]\!] & = \mathrm{OBJ}(t) \vee \mathrm{FUN}(t) \vee \mathrm{INC}(t) \\
\mathrm{PTR}[\![q\ t]\!] & = q \notin \mathrm{tq}(t) \wedge \mathrm{PTR}(t) \\
\mathrm{PTR}[\![\mathit{other}]\!] & = \mathrm{false}
\end{array}$$

Collectively, all signed and unsigned types are known simply as *integral types*, which, in combination with the floating type family, also form the set of all *arithmetic types* supported by the C language. More so, arithmetic and pointer types constitute the family of *scalar types* which, intuitively, characterise all C variables whose values are translated into simple Etude atoms. Formally, these three auxiliary type families are depicted by the following predicates "INT", "AT" and "SCR":

$\mathrm{INT}\ [\![\cdot]\!], \mathrm{AT}[\![\cdot]\!], \mathrm{SCR}[\![\cdot]\!] :: \mathit{type} \to \mathit{bool}$

$$\begin{array}{ll}
\mathrm{INT}\ [\![t]\!] & = \mathrm{ST}\ (t) \vee \mathrm{UT}\ (t) \\
\mathrm{AT}\ \ [\![t]\!] & = \mathrm{INT}(t) \vee \mathrm{FLT}\,(t) \\
\mathrm{SCR}[\![t]\!] & = \mathrm{AT}\ (t) \vee \mathrm{PTR}(t)
\end{array}$$

Further, the family of *structure and union types* consists of all correctly-qualified and unqualified versions of constructs with the abstract syntax of "**`struct`** $v\ \bar{m}$" and

"**`union`** $v\ \bar{m}$", in which $\bar{m}$ represents a list of *structure or union members*. Formally, such types are identified by the Haskell predicate "SU", which can be defined as follows:

$$\mathrm{SU}[\![\cdot]\!] :: \mathit{type} \to \mathit{bool}$$

$$\begin{aligned}
\mathrm{SU}[\![\texttt{struct}\ v\ \bar{m}]\!] &= \mathrm{true} \\
\mathrm{SU}[\![\texttt{union}\ \ v\ \bar{m}]\!] &= \mathrm{true} \\
\mathrm{SU}[\![q\ t]\!] &= q \notin \mathrm{tq}(t) \wedge \mathrm{SU}(t) \\
\mathrm{SU}[\![\mathit{other}]\!] &= \mathrm{false}
\end{aligned}$$

As exposed by the earlier BNF grammar, every member of such a type consists of a *member type*, an optional identifier and a non-negative integer known as the member's *offset*, which, informally, specifies the precise location of the member within an entire structure or union as a byte offset into the corresponding Etude object. The three components of every member entity can be extracted individually from its abstract syntax using the following trio of auxiliary Haskell definitions:

$$\begin{aligned}
&\mathcal{T}\ [\![\cdot]\!] :: \mathit{member} \to \mathit{type} && \textit{(type)} \\
&\mathcal{N}[\![\cdot]\!] :: \mathit{member} \to \mathit{identifier}_{opt} && \textit{(name)} \\
&O\ [\![\cdot]\!] :: \mathit{member} \to \mathit{integer} && \textit{(offset)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}\ [\![t\ x_{opt}\ \texttt{@}\ n]\!] &= t \\
\mathcal{N}[\![t\ x_{opt}\ \texttt{@}\ n]\!] &= x_{opt} \\
O\ [\![t\ x_{opt}\ \texttt{@}\ n]\!] &= n
\end{aligned}$$

In practice, every member list constitutes a unique name space for its member names and, accordingly, such lists are usually viewed as sets of members indexed by their associated identifiers. This natural interpretation can be formalised in Haskell using the following implicit coercion function:

$$[\![\cdot]\!] :: \mathit{members} \to (\mathit{identifier} \mapsto \mathit{member})$$

$$[\![\bar{m}]\!] = \{\mathcal{N}(m_k){:}m_k \mid m_k \leftarrow \bar{m},\ \mathcal{N}(m_k) \neq \epsilon\}$$

In other words, the name space introduced by a given structure or union type consists of all named members of that structure or union, with each identifier appearing in such a member mapped to the member entity itself. Anonymous members without an identifier are explicitly excluded from this name space. Observe that the above conversion function is treated as an implicit coercion whose name is always omitted from the presentation, since its existence can be inferred trivially from the context of the surrounding Haskell expression.

The Etude variable associated with every enumeration, structure and union type is known as that type's *tag* and may be obtained from its abstract syntax using the notation "$\mathrm{tag}(t)$", whose obvious implementation can be captured in Haskell as follows:

$$\mathrm{tag}[\![\cdot]\!] :: \mathit{type} \to v$$

$$\begin{aligned}
\mathrm{tag}[\![\texttt{enum}\ t\ v]\!] &= v \\
\mathrm{tag}[\![\mathit{struct\text{-}or\text{-}union}\ v\ \bar{m}]\!] &= v \\
\mathrm{tag}[\![q\ t]\!] &= \mathrm{tag}(t)
\end{aligned}$$

Such tags are used solely to formalise the equivalence relation for structure, union and enumeration types. In this work, they are associated with Etude variables only in order to simplify generation of globally-unique type tags in Sections 5.5 and 5.8.

Similarly, the "*struct-or-union*" and member list components of a structure or union type can be extracted from its abstract syntax using the following simple constructions:

$$\begin{aligned}
&\mathrm{su}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{struct\text{-}or\text{-}union} \\
&\mathrm{su}[\![\mathit{struct\text{-}or\text{-}union}\ \mathsf{v}\ \bar{m}]\!] = \mathit{struct\text{-}or\text{-}union} \\
&\mathrm{su}[\![q\ t]\!] \qquad\qquad\quad = \mathrm{su}(t) \\
\\
&\bar{m}\,[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{members} \\
&\bar{m}\,[\![\mathit{struct\text{-}or\text{-}union}\ \mathsf{v}\ \bar{m}]\!] = \bar{m} \\
&\bar{m}\,[\![q\ t]\!] \qquad\qquad\quad = \bar{m}(t)
\end{aligned}$$

Unqualified types of the form "t **[**n_{opt}**]**" are known as *array types*. In all cases, such types must be derived from a valid member of the object type family, as identified by the "OBJ" predicate described shortly, together with an optional number n, that, when specified, must represent a strictly-positive integer no greater than $\mathrm{lub}(\texttt{ptrdiff\_t})/\mathcal{S}(t)$ in magnitude. Formally:

$$\begin{aligned}
&\mathrm{ARR}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{bool} \\
&\mathrm{ARR}[\![t\ \texttt{[}n_{opt}\texttt{]}]\!] = \mathrm{OBJ}(t) \wedge (n_{opt} = \epsilon \vee 0 < n_{opt} \leq \mathrm{lub}(\texttt{ptrdiff_t})/\mathcal{S}(t)) \\
&\mathrm{ARR}[\![\mathit{other}]\!] \quad = \mathrm{false}
\end{aligned}$$

using the "$\mathcal{S}(t)$" and "$\mathrm{lub}(t)$" notations discussed separately in Sections 5.4.6 and 5.4.5. The optional integer component of an array type is known as the array's *length*. Intuitively, it specifies the number of discrete objects stored in that array. For convenience, the length of every array type can be determined by the following function:

$$\begin{aligned}
&\mathrm{length}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{integer}_{opt} \\
&\mathrm{length}[\![t\ \texttt{[}n_{opt}\texttt{]}]\!] = n_{opt}
\end{aligned}$$

Collectively, all of the above types with the exception of bit fields, arrays of unspecified length and structure or union types with an empty member list form an important family of *object types*, which, intuitively, can be used to describe C variables with predictable storage requirements. In Haskell, this family can be identified as follows:

$$\begin{aligned}
&\mathrm{OBJ}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{bool} \\
&\mathrm{OBJ}[\![t]\!] = \mathrm{unq}(t) \in \mathit{BIT} \vee \\
&\qquad\qquad \mathrm{FLT}(t) \vee \mathrm{ET}(t) \vee \mathrm{PTR}(t) \vee \\
&\qquad\qquad (\mathrm{ARR}(t) \wedge \mathrm{length}(t) \neq \epsilon) \vee \\
&\qquad\qquad (\mathrm{SU}(t) \wedge \bar{m}(t) \neq \varnothing)
\end{aligned}$$

More so, types of the form "t **(**$\mathit{prototype}_{opt}$**)**" represent *function types*, which are never qualified and must be derived from an object or incomplete type other than an array. In Haskell, these types are identified by the following predicate:

$$\begin{aligned}
&\mathrm{FUN}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{bool} \\
&\mathrm{FUN}[\![t\ \texttt{(}p_{opt}\texttt{)}]\!] = (\mathrm{OBJ}(t) \vee \mathrm{INC}(t)) \wedge \neg\mathrm{ARR}(t) \wedge \mathrm{WF}(p_{opt}) \\
&\mathrm{FUN}[\![\mathit{other}]\!] \quad = \mathrm{false}
\end{aligned}$$

and their optional prototype p_{opt} can be obtained by the following function:

$$\text{prot}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{prototype}_{opt}$$

$$\text{prot}[\![t\ \texttt{(}p_{opt}\texttt{)}]\!] = p_{opt}$$

Informally, the prototype of a C function characterises any argument values expected by every call to that function as described in Section 5.7. An empty prototype indicates that no information about the arguments' types is available to the compiler, while a prototype of the form "$\bar{t}$`...`" signals that the function may accept additional arguments beyond those specified by its prototype's type list. When present, a prototype without the "`...`" suffix must be formed only from valid object, incomplete and function types, while prototypes with the "`...`", are considered to be well-formed if they specify a non-empty list of types and if the corresponding prototype without that suffix is also deemed valid. Formally:

$$\text{WF}[\![\cdot]\!] :: (\mathit{prototype}_{opt}) \rightarrow \mathit{bool}$$

$$\begin{aligned}
\text{WF}[\![\epsilon]\!] &= \text{true} \\
\text{WF}[\![\bar{t}]\!] &= \bigwedge[\text{OBJ}(t_k) \vee \text{FUN}(t_k) \vee \text{INC}(t_k) \mid t_k \leftarrow \bar{t}] \wedge \bar{t} \neq \varnothing \\
\text{WF}[\![\bar{t}\texttt{...}]\!] &= \text{WF}(\bar{t}) \wedge \bar{t} \neq \varnothing
\end{aligned}$$

Although a well-formed C program may never define a function with an incomplete parameter type, a careful perusal of the C standard reveals that nonsensical constructs such as "`int(void, void, void)`" are still valid within "`sizeof`" expressions and cast operations. Accordingly, they are supported in this work, although virtually no other C compiler provides a correct implementations of these types.

The special type "`void`" is known as the *void type*. All of its qualified and unqualified variants are identified collectively by the following predicate "VT":

$$\text{VT}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{bool}$$

$$\begin{aligned}
\text{VT}[\![\texttt{void}]\!] &= \text{true} \\
\text{VT}[\![q\ t]\!] &= q \notin \text{tq}(t) \wedge \text{VT}(t) \\
\text{VT}[\![\mathit{other}]\!] &= \text{false}
\end{aligned}$$

The "`void`" type represents a distinguished member of the *incomplete type family*, which, intuitively, describes objects with unknown storage requirements. Other examples of incomplete types include arrays of unspecified length and structures or unions with empty member lists. Formally, these rather unusual entities are identified by the collective predicate "INC":

$$\text{INC}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{bool}$$

$$\text{INC}[\![t]\!] = \text{VT}(t) \vee (\text{ARR}(t) \wedge \text{length}(t) = \epsilon) \vee (\text{SU}(t) \wedge \bar{m}(t) = \varnothing)$$

A reader will observer that the structure of many C type forms defined at the beginning of this section is *derived* from some other type entity, known as that type's *base*. In particular, the base of an enumeration or bit field type defines that type's corresponding basic integral derivation, while the base of a pointer type represents the type of an object addressed by the pointer. Similarly, the base of an array type describes the array's

individual elements, while, for functions, it characterises the entity's returned value. For convenience, the notion of the base type is captured collectively by the following Haskell definition:

$$\mathcal{B}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{type}$$

$$\begin{aligned}
\mathcal{B}[\![\texttt{enum}\ t\ v]\!] &= t \\
\mathcal{B}[\![t\ \texttt{*}]\!] &= t \\
\mathcal{B}[\![t\ \texttt{[}n_{opt}\texttt{]}]\!] &= t \\
\mathcal{B}[\![t\ \texttt{(}p_{opt}\texttt{)}]\!] &= t \\
\mathcal{B}[\![t\ \texttt{:}m\ \texttt{.}\ n]\!] &= t \\
\mathcal{B}[\![q\ t]\!] &= \mathcal{B}(t)
\end{aligned}$$

For clarity, in the C Standard the base type of a pointer, array or function type is usually referred to as the entity's *referenced*, *element* or *returned type*, respectively.

5.4.4 Type Qualifiers

As already hinted in the previous section, most C types can be *qualified* with one or both of the keywords "**`const`**" and "**`volatile`**". Intuitively, a type whose qualification includes the "**`const`**" keyword is said to be *constant-qualified*. Such types describe objects whose values remain immutable throughout execution of a well-formed C program. Similarly, a type that includes the "**`volatile`**" qualifier is said to be *volatile-qualified*. Intuitively, these types describe objects such as memory-mapped hardware device registers, whose values may be modified by means outside of the program's explicit control. The complete set of all such qualifiers is derived from a type's syntax as follows:

$$\mathrm{tq}[\![\cdot]\!] :: \mathit{type} \rightarrow \{\mathit{type\text{-}qualifier}\}$$

$$\begin{aligned}
\mathrm{tq}[\![q\ t]\!] &= \{q\} \cup \mathrm{tq}(t) \\
\mathrm{tq}[\![\mathit{other}]\!] &= \varnothing
\end{aligned}$$

It should be pointed out that, since the above function returns a set of type qualifiers rather than a list, the precise order in which individual qualifiers appear within the syntax of a C type does not affect that type's core meaning, so that both of the syntactic forms "**`const volatile`** t" and "**`volatile const`** t" remain semantically indistinguishable everywhere within a C program.

Informally, every C type corresponds to a unique *unqualified type*, obtained by stripping any type qualifier keywords from its syntax. Formally, this type is depicted by the notation "$\mathrm{unq}(t)$", which can be implemented in Haskell as follows:

$$\mathrm{unq}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{type}$$

$$\begin{aligned}
\mathrm{unq}[\![q\ t]\!] &= \mathrm{unq}(t) \\
\mathrm{unq}[\![\mathit{other}]\!] &= \mathit{other}
\end{aligned}$$

Finally, any type may be qualified with an arbitrary list of type qualifiers $\bar{q}$, which, in this work, is paraphrased as a list concatenation operation "$\bar{q} +\!\!+ t$". When this notation is used to qualify an array type of the form "$t\ \texttt{[}n_{opt}\texttt{]}$", the result is an unqualified type of the same length, whose element type describes the qualified version of the original

array's elements. Otherwise, the qualified type is obtained by composing the supplied list of type qualifiers $\bar{q}$ into a single function of the Haskell type "*type* → *type*" using the list composition operator "⊙" described separately in Appendix A and applying that function to the original type t. Formally:

$$[\![\cdot]\!] \mathbin{+\!\!+} [\![\cdot]\!] :: [\textit{type-qualifier}] \to \textit{type} \to \textit{type}$$

$$\begin{aligned} [\![\bar{q}]\!] \mathbin{+\!\!+} [\![t\ \texttt{[}n_{opt}\texttt{]}]\!] &= [\![[\![\bar{q} \mathbin{+\!\!+} t]\!]\ \texttt{[}n_{opt}\texttt{]}]\!] \\ [\![\bar{q}]\!] \mathbin{+\!\!+} [\![t]\!] &= (\odot\bar{q})(t) \end{aligned}$$

In certain contexts, it is also useful to *requalify* an already-qualified C type, in which case a union of the supplied type qualifiers $\bar{q}$ with the existing qualification of t is obtained and applied to the unqualified version of t using the above "+" operator. In this work, such construction is paraphrased as the following set operation:

$$[\![\cdot]\!] \cup [\![\cdot]\!] :: \{\textit{type-qualifier}\} \to \textit{type} \to \textit{type}$$

$$\begin{aligned} [\![\bar{q}]\!] \cup [\![t\ \texttt{[}n_{opt}\texttt{]}]\!] &= [\![[\![\bar{q} \cup t]\!]\ \texttt{[}n_{opt}\texttt{]}]\!] \\ [\![\bar{q}]\!] \cup [\![t]\!] &= (\bar{q} \cup \mathrm{tq}(t)) \mathbin{+\!\!+} (\mathrm{unq}(t)) \end{aligned}$$

Finally, in Section 5.7.1.3, requalification can be also applied to the list of members of a structure or union type, as implemented by the following simple list composition:

$$[\![\cdot]\!] \cup [\![\cdot]\!] :: \{\textit{type-qualifier}\} \to \textit{members} \to \textit{members}$$

$$[\![\bar{q}]\!] \cup [\![\bar{m}]\!] = [[\![[\![\bar{q} \cup t_k]\!]\ x_k\ \texttt{@}\ n_k]\!] \mid [\![t_k\ x_k\ \texttt{@}\ n_k]\!] \leftarrow \bar{m}]$$

In certain contexts, we must also consider the *recursive qualification* of a structure or union type t, which extends the set $\mathrm{tq}(t)$ with any qualifiers hidden among the construct's individual members, together with the qualification of any array element types found in its syntax. Formally, the recursive qualification of a C type is defined by induction over the entity's structure as follows:

$$\mathrm{rtq}[\![\cdot]\!] :: \textit{type} \to \{\textit{type-qualifier}\}$$

$$\begin{aligned} \mathrm{rtq}[\![t]\!] \mid\ & \mathrm{SU}(t) && = \mathrm{tq}(t) \cup \bigcup[\mathrm{rtq}(\mathcal{T}(m_k)) \mid m_k \leftarrow \bar{m}(t)] \\ \mid\ & \mathrm{ARR}(t) && = \mathrm{tq}(t) \cup \mathrm{rtq}(\mathcal{B}(t)) \\ \mid\ & \text{otherwise} && = \mathrm{tq}(t) \end{aligned}$$

Observe that the result of this construction is well-formed only if the operand does not represent a function type, or else if the specified set of type qualifiers is empty, since function types cannot be qualified in a well-formed C program.

The two "**`const`**" and "**`volatile`**" type qualifiers correspond naturally to the Etude access attributes "C" and "V", a fact that is easily exploited by the following Haskell mapping of these constructs:

$$\mu[\![\cdot]\!] :: \textit{type-qualifier} \to \textit{mode}$$

$$\begin{aligned} \mu[\![\texttt{const}]\!] &= [\![\mathrm{C}]\!] \\ \mu[\![\texttt{volatile}]\!] &= [\![\mathrm{V}]\!] \end{aligned}$$

In Section 5.4.6, this correspondence is utilised to construct Etude envelopes suitable for depiction of C variable objects in an address space of the translated program.

5.4.5 Representation of Values

As discussed in Chapter 4, many unspecified aspects of the various binary encoding schemes used in Etude for representation of its numeric quantities is captured by the notion of an atomic *format*. Likewise, every C object and bit field type is naturally associated with one of these formats by an unspecified function $\phi(t)$, as exposed by the following Haskell signature:

$\phi[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{format}$

In particular, under every implementation of the C language, the format of all signed, unsigned and floating types belongs to the "Z", "N" and "R" genre, respectively:

$$\begin{aligned}
&\text{REPR}_1 :: \forall\, t \Rightarrow [\![\text{ST}\ \ (t)]\!] \rightarrow (\text{WF}[\![\phi(t)]\!],\ [\![\gamma(\phi(t)) = [\![\texttt{Z}]\!]]\!]) \\
&\text{REPR}_2 :: \forall\, t \Rightarrow [\![\text{UT}\ \ (t)]\!] \rightarrow (\text{WF}[\![\phi(t)]\!],\ [\![\gamma(\phi(t)) = [\![\texttt{N}]\!]]\!]) \\
&\text{REPR}_3 :: \forall\, t \Rightarrow [\![\text{FLT}(t)]\!] \rightarrow (\text{WF}[\![\phi(t)]\!],\ [\![\gamma(\phi(t)) = [\![\texttt{R}]\!]]\!])
\end{aligned}$$

Further, the format of all function types and all pointers to such types belongs to the Etude function genre "F", while all structure, union and array types, together with any pointers to an object or incomplete type are formatted in the object genre "O":

$$\begin{aligned}
&\text{REPR}_4 :: \forall\, t \Rightarrow \\
&\qquad [\![\text{FUN}(t) \vee (\text{PTR}(t) \wedge \text{FUN}(\mathcal{B}(t)))]\!] \rightarrow \\
&\qquad (\text{WF}[\![\phi(t)]\!],\ [\![\gamma(\phi(t)) = [\![\texttt{F}]\!]]\!])
\end{aligned}$$

$$\begin{aligned}
&\text{REPR}_5 :: \forall\, t \Rightarrow \\
&\qquad [\![\text{SU}(t) \vee \text{ARR}(t) \vee (\text{PTR}(t) \wedge (\text{OBJ}(\mathcal{B}(t)) \vee \text{INC}(\mathcal{B}(t))))]\!] \rightarrow \\
&\qquad (\text{WF}[\![\phi(t)]\!],\ [\![\gamma(\phi(t)) = [\![\texttt{O}]\!]]\!])
\end{aligned}$$

More so, on every C implementation, pointers to all character and void types must share the same format:

$$\begin{aligned}
&\text{REPR}_6 :: \forall\, t_1, t_2 \Rightarrow \\
&\qquad [\![\text{PTR}(t_1) \wedge (\text{CHR}(\mathcal{B}(t_1)) \vee \text{VT}(\mathcal{B}(t_1)))]\!] \rightarrow \\
&\qquad [\![\text{PTR}(t_2) \wedge (\text{CHR}(\mathcal{B}(t_2)) \vee \text{VT}(\mathcal{B}(t_2)))]\!] \rightarrow \\
&\qquad [\![\phi(t_1) = \phi(t_2)]\!]
\end{aligned}$$

Likewise, all pairs of *compatible object types* described later in Section 5.4.11 always assume a common representation:

$$\text{REPR}_7 :: \forall\, t_1, t_2 \Rightarrow [\![\text{OBJ}(t_1) \wedge \text{OBJ}(t_2) \wedge t_1 \approx t_2]\!] \rightarrow [\![\phi(t_1) = \phi(t_2)]\!]$$

The plain "***char***" type must have the same format as "***char_t***", every well-formed array type must share its format with a pointer to its element type and all function types must have the same representation as a pointer to that function:

$$\begin{aligned}
&\text{REPR}_8\ \ :: [\![\phi(\textbf{\textit{char}}) = \phi(\textbf{\textit{char_t}})]\!] \\
&\text{REPR}_9\ \ :: \forall\, t \Rightarrow [\![\text{ARR}(t)]\!] \rightarrow [\![\phi(t) = \phi[\![[\![\mathcal{B}(t)]\!]\textbf{\textit{*}}]\!]]\!] \\
&\text{REPR}_{10} :: \forall\, t \Rightarrow [\![\text{FUN}(t)]\!] \rightarrow [\![\phi(t) = \phi[\![t\textbf{\textit{*}}]\!]]\!]
\end{aligned}$$

More so, a well-formed bit field type of the form "***int***`:`m`.`n" must be represented identically to its corresponding type "***int_t***`:`m`.`n":

$$\text{REPR}_{11} :: \forall\, m, n \Rightarrow [\![\text{BF}[\![\textbf{\textit{int}}\texttt{:}m\texttt{.}n]\!]]\!] \rightarrow [\![\phi[\![\textbf{\textit{int}}\texttt{:}m\texttt{.}n]\!] = \phi[\![\textbf{\textit{int\_t}}\texttt{:}m\texttt{.}n]\!]]\!]$$

Values of a qualified object or bit field type must have the same representation as those of the corresponding unqualified type:

$$\text{REPR}_{12} :: \forall\, T, t \Rightarrow [\![\text{OBJ}(t) \vee \text{BF}(t)]\!] \rightarrow [\![\phi(t) = \phi(\text{unq}(t))]\!]$$

Last but not least, all well-formed pointers to every qualified and unqualified version of a given C type must share the same format:

$$\text{REPR}_{13} :: \forall\, t \Rightarrow [\![\text{PTR}(t)]\!]\ [\![\phi(t) = \phi[\![[\![\text{unq}(\mathcal{B}(t))]\!]\,\star]\!]]\!]$$

As described in Chapter 4, every supported format ϕ is also associated with a pair of numeric quantities "glb(ϕ)" and "lub(ϕ)", known as the format's *greatest lower bound* and *least upper bound*, respectively. For integral formats, the set of all integers n in the range $\text{glb}(\phi) \leq n \leq \text{lub}(\phi)$ is known as the format's *set of representable values*. Inevitably, every arithmetic C type is also associated with such a set, which can be derived directly from the type itself using the following two Haskell constructions:

$$\text{glb}[\![\cdot]\!], \text{lub}[\![\cdot]\!] :: \textit{type} \rightarrow \textit{rational}$$

$$\begin{aligned}\text{glb}[\![t]\!] &= \text{glb}(\phi(t))\\ \text{lub}[\![t]\!] &= \text{lub}(\phi(t))\end{aligned}$$

We have already discussed many of the constraints placed on Etude formats in Chapter 4. However, the C programming language imposes a number of further requirements on the range of values associated with the individual integral C types. In particular, every basic integral type must be capable of representing at least the following range of values:

$$\begin{aligned}
\text{INT}_1 &:: [\![\text{glb}[\![\textbf{\textit{signed char}}]\!] &&\leq && -127]\!]\\
\text{INT}_2 &:: [\![\text{glb}[\![\textbf{\textit{signed short}}]\!] &&\leq && -32767]\!]\\
\text{INT}_3 &:: [\![\text{glb}[\![\textbf{\textit{signed int}}]\!] &&\leq && -32767]\!]\\
\text{INT}_4 &:: [\![\text{glb}[\![\textbf{\textit{signed long}}]\!] &&\leq && -2147483647]\!]\\
\text{INT}_5 &:: [\![\text{lub}[\![\textbf{\textit{signed char}}]\!] &&\geq && 127]\!]\\
\text{INT}_6 &:: [\![\text{lub}[\![\textbf{\textit{signed short}}]\!] &&\geq && 32767]\!]\\
\text{INT}_7 &:: [\![\text{lub}[\![\textbf{\textit{signed int}}]\!] &&\geq && 32767]\!]\\
\text{INT}_8 &:: [\![\text{lub}[\![\textbf{\textit{signed long}}]\!] &&\geq && 2147483647]\!]\\
\text{INT}_9 &:: [\![\text{lub}[\![\textbf{\textit{unsigned char}}]\!] &&\geq && 255]\!]\\
\text{INT}_{10} &:: [\![\text{lub}[\![\textbf{\textit{unsigned short}}]\!] &&\geq && 65535]\!]\\
\text{INT}_{11} &:: [\![\text{lub}[\![\textbf{\textit{unsigned int}}]\!] &&\geq && 65535]\!]\\
\text{INT}_{12} &:: [\![\text{lub}[\![\textbf{\textit{unsigned long}}]\!] &&\geq && 4294967295]\!]
\end{aligned}$$

Further, the four "***signed***" and "***unsigned***" versions of the "***char***", "***short***", "***int***" and "***long***" types must be capable of encoding all values that are representable by all previous types in that list:

$$\begin{aligned}
\text{INT}_{13} :: [\![&\text{glb}[\![\textbf{\textit{signed char}}]\!] \geq \text{glb}[\![\textbf{\textit{signed short}}]\!]\\
&\geq \text{glb}[\![\textbf{\textit{signed int}}]\!] \geq \text{glb}[\![\textbf{\textit{signed long}}]\!]]\!]
\end{aligned}$$

$$\begin{aligned}
\text{INT}_{14} :: [\![&\text{lub}[\![\textbf{\textit{signed char}}]\!] \leq \text{lub}[\![\textbf{\textit{signed short}}]\!]\\
&\leq \text{lub}[\![\textbf{\textit{signed int}}]\!] \leq \text{lub}[\![\textbf{\textit{signed long}}]\!]]\!]
\end{aligned}$$

$$\begin{aligned}
\text{INT}_{15} :: [\![&\text{lub}[\![\textbf{\textit{unsigned char}}]\!] \leq \text{lub}[\![\textbf{\textit{unsigned short}}]\!]\\
&\leq \text{lub}[\![\textbf{\textit{unsigned int}}]\!] \leq \text{lub}[\![\textbf{\textit{unsigned long}}]\!]]\!]
\end{aligned}$$

Likewise, the set of all non-negative values representable by every "***signed***" integral type form must be a subset of those representable by its "***unsigned***" variant:

$$\text{INT}_{16} :: \forall\, is \Rightarrow \text{WF}(is) \rightarrow [\![\text{lub}[\![\textbf{\textit{signed}}\ is]\!] \leq \text{lub}[\![\textbf{\textit{unsigned}}\ is]\!]]\!]$$

Finally, every well-formed bit field type must support a subset of the values representable by its base type:

$$\text{INT}_{17} :: \forall\, t, m, n \Rightarrow [\![\text{BF}[\![t\,\texttt{:}m\,\texttt{.}n]\!]]\!] \rightarrow [\![\text{glb}(t) \leq \text{glb}[\![t\,\texttt{:}m\,\texttt{.}n]\!] \leq \text{lub}[\![t\,\texttt{:}m\,\texttt{.}n]\!] \leq \text{lub}(t)]\!]$$

In Chapter 4, formats from the "R" genre are further associated with a set of four *floating point parameters* known as the *radix*, *precision*, *minimum exponent* and *maximum exponent*, respectively. These properties can be naturally propagated to floating C types, observing, however, that the C Standard requires all floating C types to share a single radix value:

$$\text{FLT}_1 :: \forall\, t_1, t_2 \Rightarrow [\![\text{FLT}(t_1) \wedge \text{FLT}(t_2)]\!] \rightarrow [\![\text{r}(\phi(t_1)) = \text{r}(\phi(t_2))]\!]$$

Further, the range of values representable exactly by the "***float***" type, as indicated by its precision and exponent bounds, must be a subset of those meaningful under the "***double***" representation and, for "***double***", it must be a subset of "***long double***":

$$\begin{array}{lll}
\text{FLT}_2 :: [\![\text{p}(\phi[\![\textbf{\textit{float}}]\!]) & \leq \text{p}(\phi[\![\textbf{\textit{double}}]\!]) & \leq \text{p}(\phi[\![\textbf{\textit{long double}}]\!])]\!] \\
\text{FLT}_3 :: [\![\text{E}_{\max}(\phi[\![\textbf{\textit{float}}]\!]) & \leq \text{E}_{\max}(\phi[\![\textbf{\textit{double}}]\!]) & \leq \text{E}_{\max}(\phi[\![\textbf{\textit{long double}}]\!])]\!] \\
\text{FLT}_4 :: [\![\text{E}_{\min}(\phi[\![\textbf{\textit{float}}]\!]) & \geq \text{E}_{\min}(\phi[\![\textbf{\textit{double}}]\!]) & \geq \text{E}_{\min}(\phi[\![\textbf{\textit{long double}}]\!])]\!]
\end{array}$$

Finally, each of these three types must be capable of representing at least the values described by the following choices of its four floating point parameters:

$$\begin{array}{lll}
\text{FLT}_5 :: \forall\, n :: \textit{integer} \Rightarrow & [\![(b = 10^n) + \lfloor (p-1) \times \log_{10}(b) \rfloor \geq & 6]\!] \\
\text{FLT}_6 :: \forall\, n :: \textit{integer} \Rightarrow & [\![(b = 10^n) + \lfloor (q-1) \times \log_{10}(b) \rfloor \geq & 10]\!] \\
\text{FLT}_7 :: & [\![\lceil \log_{10}(b^{E_{min}\,-\,1}) \rceil & \leq -37]\!] \\
\text{FLT}_8 :: & [\![\lfloor \log_{10}((1 - b^{-p}) \times b^{E_{max}}) \rfloor & \geq 37]\!] \\
\text{FLT}_9 :: & [\![(1 - b^{-p}) \times b^{E_{max}} & \geq 10^{+37}]\!] \\
\text{FLT}_{10} :: & [\![b^{1\,-\,p} & \leq 10^{-5}]\!] \\
\text{FLT}_{11} :: & [\![b^{1\,-\,q} & \leq 10^{-9}]\!] \\
\text{FLT}_{12} :: & [\![b^{E_{min}\,-\,1} & \leq 10^{-37}]\!]
\end{array}$$

where b, E_{min} and E_{max} represent the radix and exponent bounds of the "***float***" type, p is equal to that type's precision and q determines the precision of "***double***":

$$\begin{array}{lll}
b & = \text{r} & (\phi[\![\textbf{\textit{float}}]\!]) \\
E_{min} & = \text{E}_{\min} & (\phi[\![\textbf{\textit{float}}]\!]) \\
E_{max} & = \text{E}_{\max} & (\phi[\![\textbf{\textit{float}}]\!]) \\
p & = \text{p} & (\phi[\![\textbf{\textit{float}}]\!]) \\
q & = \text{p} & (\phi[\![\textbf{\textit{double}}]\!])
\end{array}$$

For a more thorough treatment of the generic floating point arithmetic in Etude, an interested reader is referred to the earlier discussion of the topic in Section 4.4. A typical implementation of the $\phi(t)$ function, suitable for the MMIX architecture from Chapter 6, is included in Appendix C.

5.4.6 Storage Requirements

Ultimately, every object type defined in the C Standard is intended to characterise some collection of C *variables* or address space regions reserved for storage of mutable C values attributed with that type. In Etude, such regions are described concisely by the notion of an *envelope* from Section 4.5, so that, in effect, every C type t describes an Etude envelope constructed naturally by the following function $\bar{\xi}(t)$:

$$\bar{\xi}[\![\cdot]\!] :: \mathit{type} \to \mathit{envelope}$$

$$\begin{aligned}
\bar{\xi}[\![t]\!] \;\mid\; & \mathrm{VT}\ (t) = \varnothing \\
\mid\; & \mathrm{BF}\ (t) = [\![(0,\ [\![\phi[\![\texttt{unsigned int}]\!]]\!],\ [\![\bar{\mu}(t)]\!])]\!] \\
\mid\; & \mathrm{SCR}\,(t) = [\![(0,\ [\![\phi(t)]\!],\ [\![\bar{\mu}(t)]\!])]\!] \\
\mid\; & \mathrm{ARR}(t) = \bigcup[\bar{\xi}(\mathcal{B}(t)) \oplus (k \times \mathcal{S}(\mathcal{B}(t)))\ \mid\ k \leftarrow [0\ ...\ \mathrm{length}(t) - 1]] \\
\mid\; & \mathrm{SU}\ \ (t) = \bigcup[\bar{\xi}(\mathcal{T}(m_k)) \oplus \mathcal{O}(m_k)\ \mid\ m_k \leftarrow \mathrm{tq}(t) \cup \bar{m}(t)]
\end{aligned}$$

In other words, every scalar type t describes a singular envelope of the form "$(0,\ \phi,\ \bar{\mu})$", where ϕ is either the format of the "**`unsigned int`**" type if t represents a bit field, or else the format of t itself for all other scalar type forms. The set of access attributes $\bar{\mu}$ associated with the type's envelope element is obtained from its recursive qualification using the following simple Haskell construction:

$$\bar{\mu}[\![\cdot]\!] :: \mathit{type} \to \{\mathit{mode}\}$$

$$\bar{\mu}[\![t]\!] = \{\mu(\mu_k)\ \mid\ \mu_k \leftarrow \mathrm{rtq}(t)\}$$

On the other hand, the envelope of a complete structure or union type is defined as a union of the individual envelopes of its member types, each shifted by the member's offset within the object. Finally, the envelope of a complete array type (i.e., an array of some predetermined length n) is formed from n copies of the envelope associated with the array's element type t, each shifted by the offset $k \times \mathcal{S}(t)$ for some k in the range $0 \leq k < n$. Formally, the size $\mathcal{S}(t)$ of every complete object type t is defined simply as the size of t's envelope, as epitomised by the following trivial Haskell function:

$$\mathcal{S}[\![\cdot]\!] :: \mathit{type} \to \mathit{integer}$$

$$\mathcal{S}[\![t]\!] = \mathcal{S}(\bar{\xi}(t))$$

Since the definition of $\bar{\xi}(t)$ rests on the implementation-defined mapping between C types and Etude formats, the size of most object types is left unspecified in the portable fragment of the C language. However, every implementation of a C compiler must ensure that the "**`char`**" type has the predetermined size of 1 and that all "**`signed`**" and "**`unsigned`**" variants of the same integral type have identical storage requirements:

$$\begin{aligned}
&\mathrm{SIZE}_1 :: [\![\mathcal{S}(\texttt{char}) = 1]\!] \\
&\mathrm{SIZE}_2 :: \forall\ \mathit{is} \Rightarrow \mathrm{WF}(\mathit{is}) \to [\![\mathcal{S}[\![\texttt{signed}\ \mathit{is}]\!] = \mathcal{S}[\![\texttt{unsigned}\ \mathit{is}]\!]]\!]
\end{aligned}$$

Further restrictions on C type sizes can be inferred indirectly from the requirements placed upon their corresponding Etude formats described earlier in Section 5.4.5. For example, since all pointers to qualified and unqualified versions of compatible types must share a single binary representation, they must likewise agree in the storage requirements derived from their common Etude format.

5.4.7 Integral Promotion

Most arithmetic operations defined on integer values by the concrete syntax of C expressions described in Section 5.7 can be applied directly only to operands of the "***signed***" and "***unsigned***" versions of the "***int***" or "***long***" type. Such types are said to be *invariant* and are identified by the Haskell predicate "INV". Specifically, the family of invariant types includes "***unsigned int***", the "***signed***" and "***unsigned***" versions of the "***long***" type, as well as all non-integral types. Formally:

$$\text{INV}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{bool}$$

$$\text{INV}[\![t]\!] = \text{unq}(t) \in \{\textbf{\textit{unsigned int}}, \textbf{\textit{signed long}}, \textbf{\textit{unsigned long}}\} \vee \neg\text{INT}(t)$$

In most cases, operands of every non-invariant C type are implicitly converted to the "***signed***" or "***unsigned***" version of the "***int***" type, with the "***signed***" variant chosen whenever it is capable of representing every numeric value of the unconverted operand. This process, known as *integral promotion*, also drops all type qualifiers associated with any invariably-typed operands, except that, for subtle technical reasons described later in Section 5.7.1.3, type qualification is always preserved for structure and union objects. Formally, this construction is modelled in Haskell as follows:

$$\text{ip}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{type}$$

$$\begin{array}{lll}
\text{ip}[\![t]\!] \mid \text{SU}(t) & & = t \\
\quad \mid \text{INV}(t) & & = \text{unq}(t) \\
\quad \mid \text{glb}[\![\textbf{\textit{int}}]\!] \leq \text{glb}(t) \leq \text{lub}(t) \leq \text{lub}[\![\textbf{\textit{int}}]\!] & & = [\![\textbf{\textit{signed int}}]\!] \\
\quad \mid \text{otherwise} & & = [\![\textbf{\textit{unsigned int}}]\!]
\end{array}$$

It should be observed that integral promotion never reduces the set of values representable by its operand. It is applied in C as a convenient means of culling the number of viable operator forms supported by the expression syntax described in Section 5.7.

5.4.8 Pointer Promotion

In most contexts, functions and arrays do not constitute true first-class objects in the C language. Instead, they are usually manipulated indirectly through pointer references, which can be constructed either explicitly by the programmer using the "**&**" operator described in Section 5.7, or implicitly by the compiler through the process known as *pointer promotion*. Formally, pointer promotion converts every function type into a pointer to that function and every array into a pointer to its element type, leaving all other types unchanged. In Haskell, this operation can be modelled as follows:

$$\text{pp}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{type}$$

$$\begin{array}{lll}
\text{pp}[\![t]\!] \mid \text{FUN}(t) & = [\![t\,\textbf{\textit{*}}]\!] \\
\quad \mid \text{ARR}(t) & = [\![[\![\mathcal{B}(t)]\!]\,\textbf{\textit{*}}]\!] \\
\quad \mid \text{SU}\ \ (t) & = t \\
\quad \mid \text{otherwise} & = \text{unq}(t)
\end{array}$$

Like integral promotion, pointer promotion adjusts the types of C objects without affecting their Etude representations, in order to reduce the number of distinct expression forms supported by the language.

5.4.9 Default Argument Promotion

The language also defines a similar conversion known as *default argument promotion*, intended to normalise the types of arguments supplied to functions declared without an explicit prototype. Formally, default argument promotion converts the "**`float`**" type into a "**`double`**" and, for all other argument types, returns the integral and pointer promoted version of its operand:

$$\begin{aligned}
&\mathrm{ap}[\![\cdot]\!] :: \mathit{type} \to \mathit{type} \\
&\mathrm{ap}[\![t]\!] \mid (\mathrm{unq}(t) = [\![\texttt{float}]\!]) = [\![\texttt{double}]\!] \\
&\qquad\quad \mid \text{otherwise} \qquad\qquad\quad = \mathrm{ip}(\mathrm{pp}(t))
\end{aligned}$$

The language always applies this promotion in absence of a precise information about the type of a given function argument. Informally, it makes it possible for such functions to be called with almost arbitrary numeric values, without regard for any differences in the specific storage requirements of the individual arithmetic types.

5.4.10 Usual Arithmetic Conversions

When two C expressions are involved in a single binary operation, their C types are usually unified into a *common arithmetic type* by a process known as the *usual arithmetic conversion*. In all cases, any type qualifiers are first discarded by the operation and integral promotion is applied to both operands. Then, if one of the resulting operands has the "**`signed long`**" type, the other has the "**`unsigned int`**" type and if the set of values representable by "**`unsigned int`**" is not a subset of those representable by "**`signed long`**", then the entire operation assumes the "**`unsigned long`**" type. Otherwise, the common arithmetic type is equal to the first type from the following list that matches one of the two operands after integral promotion:

`long double`, **`double`**, **`float`**,
`unsigned long`, **`signed long`**, **`unsigned int`**, **`signed int`**

Observe that, after integral promotion, every arithmetic type is guaranteed to be converted into one of these seven type forms. Formally, the usual arithmetic conversion is represented by the type operator "$\boxtimes$" and is formulated in Haskell as follows:

$$\begin{aligned}
&[\![\cdot]\!] \boxtimes [\![\cdot]\!] :: \mathit{type} \to \mathit{type} \to \mathit{type} \\
&[\![t_1]\!] \boxtimes [\![t_2]\!] \\
&\quad \mid ((\mathrm{ip}(t_1) = [\![\texttt{signed long}]\!] \wedge \mathrm{ip}(t_2) = [\![\texttt{unsigned int}]\!]) \vee \\
&\qquad (\mathrm{ip}(t_2) = [\![\texttt{signed long}]\!] \wedge \mathrm{ip}(t_1) = [\![\texttt{unsigned int}]\!])) \wedge \\
&\qquad \mathrm{lub}[\![\texttt{unsigned int}]\!] > \mathrm{lub}[\![\texttt{signed long}]\!] \qquad\qquad\qquad\qquad = [\![\texttt{unsigned long}]\!] \\
&\quad \mid \mathrm{ip}(t_1) = [\![\texttt{long double}]\!] \quad \vee \mathrm{ip}(t_2) = [\![\texttt{long double}]\!] \quad = [\![\texttt{long double}]\!] \\
&\quad \mid \mathrm{ip}(t_1) = [\![\texttt{double}]\!] \quad\;\; \vee \mathrm{ip}(t_2) = [\![\texttt{double}]\!] \quad\;\; = [\![\texttt{double}]\!] \\
&\quad \mid \mathrm{ip}(t_1) = [\![\texttt{float}]\!] \quad\;\; \vee \mathrm{ip}(t_2) = [\![\texttt{float}]\!] \quad\;\; = [\![\texttt{float}]\!] \\
&\quad \mid \mathrm{ip}(t_1) = [\![\texttt{unsigned long}]\!] \vee \mathrm{ip}(t_2) = [\![\texttt{unsigned long}]\!] = [\![\texttt{unsigned long}]\!] \\
&\quad \mid \mathrm{ip}(t_1) = [\![\texttt{signed long}]\!] \quad \vee \mathrm{ip}(t_2) = [\![\texttt{signed long}]\!] \quad = [\![\texttt{signed long}]\!] \\
&\quad \mid \mathrm{ip}(t_1) = [\![\texttt{unsigned int}]\!] \quad \vee \mathrm{ip}(t_2) = [\![\texttt{unsigned int}]\!] \quad = [\![\texttt{unsigned int}]\!] \\
&\quad \mid \mathrm{ip}(t_1) = [\![\texttt{unsigned int}]\!] \quad \vee \mathrm{ip}(t_2) = [\![\texttt{unsigned int}]\!] \quad = [\![\texttt{unsigned int}]\!] \\
&\quad \mid \mathrm{ip}(t_1) = [\![\texttt{signed int}]\!] \quad\;\; \vee \mathrm{ip}(t_2) = [\![\texttt{signed int}]\!] \quad\;\; = [\![\texttt{signed int}]\!] \\
&\quad \mid \text{otherwise} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad = t_1 \sqcup t_2
\end{aligned}$$

The final rule in this definition proclaims the common arithmetic type of non-arithmetic operands equal to their *composite type*, using an algorithm depicted by the notation "$t_1 \sqcup t_2$" and described later in Section 5.4.12. Although this extension does not follow the letter of the C Standard, it provides for a very concise formalisation of certain C expression forms described in Section 5.7, while retaining the published semantics of all usual arithmetic conversions under their standard application to pairs of arithmetic types.

The usual arithmetic conversion affects the operands' actual values only if one of them represents a negative quantity of the "***signed long***" type, the other has the "***unsigned int***" type and if both of these are converted into "***unsigned long***" by the above algorithm. In all other cases, only the types of the operands are adjusted by the above conversion and their actual values remain unchanged.

5.4.11 Type Compatibility

In the course of translation, qualified types are usually compared using a relation known as *type compatibility*. By definition, two types are said to be compatible if they have identical qualifications and if their unqualified variants satisfy one of the following criteria:

① both entities represent structurally-identical arithmetic types, or
② one of the two operands represents an enumeration type and the other is compatible with that enumeration's corresponding type, or
③ both operands represent structure or union types with the same "*struct-or-union*" component and type tag, or
④ both represent pointers to compatible referenced types, or
⑤ both represent arrays of compatible element types and either both arrays have equal lengths, or else at least one of them does not include a length component, or
⑥ both constitute functions with compatible returned types and prototypes, or
⑦ both types represent identically-qualified versions of the "***void***" type, or
⑧ one of the two operands has the "***int***" form and the other is a "***signed int***".

In this work, the type compatibility relation is written as "$t_1 \approx t_2$". Formally, the "$\approx$" operator is defined by the following protracted Haskell predicate:

$$
\begin{aligned}
&[\![\cdot]\!] \approx [\![\cdot]\!] :: \mathit{type} \to \mathit{type} \to \mathit{bool} \\
&[\![t_1]\!] \approx [\![t_2]\!] = (\mathrm{tq}(t_1) = \mathrm{tq}(t_2)) \wedge \\
&\qquad ((\mathrm{AT}\ \ (t_1) \wedge \mathrm{unq}(t_1) = \mathrm{unq}(t_2)) \vee \\
&\qquad\ (\mathrm{ET}\ \ (t_1) \wedge \mathcal{B}(t_1) \approx \mathrm{unq}(t_2)) \vee \\
&\qquad\ (\mathrm{ET}\ \ (t_2) \wedge \mathcal{B}(t_2) \approx \mathrm{unq}(t_1)) \vee \\
&\qquad\ (\mathrm{SU}\ \ (t_1) \wedge \mathrm{SU}\ \ (t_2) \wedge \mathrm{su}(t_1) = \mathrm{su}(t_2) \wedge \mathrm{tag}(t_1) = \mathrm{tag}(t_2)) \vee \\
&\qquad\ (\mathrm{PTR}\ (t_1) \wedge \mathrm{PTR}\ (t_2) \wedge \mathcal{B}(t_1) \approx \mathcal{B}(t_2)) \vee \\
&\qquad\ (\mathrm{ARR}(t_1) \wedge \mathrm{ARR}(t_2) \wedge \mathcal{B}(t_1) \approx \mathcal{B}(t_2) \wedge \\
&\qquad\quad (\mathrm{length}(t_1) = \mathrm{length}(t_2) \vee \mathrm{length}(t_1) = \epsilon \vee \mathrm{length}(t_2) = \epsilon)) \vee \\
&\qquad\ (\mathrm{FUN}(t_1) \wedge \mathrm{FUN}(t_2) \wedge \mathcal{B}(t_1) \approx \mathcal{B}(t_2) \wedge \mathrm{prot}(t_1) \approx \mathrm{prot}(t_2)) \vee \\
&\qquad\ (\mathrm{VT}\ \ (t_1) \wedge \mathrm{VT}\ \ (t_2)) \vee \\
&\qquad\ (\{\mathrm{unq}(t_1), \mathrm{unq}(t_2)\} = \{[\![\texttt{int}]\!], [\![\texttt{signed int}]\!]\}))
\end{aligned}
$$

where the compatibility relation for function prototypes is formalised as follows:

$$[\![\cdot]\!] \approx [\![\cdot]\!] :: \mathit{prototype}_{opt} \to \mathit{prototype}_{opt} \to \mathit{bool}$$

$$\begin{array}{lcll}
[\![]\!] & \approx & [\![]\!] & = \text{true} \\
[\![]\!] & \approx & [\![\bar{t}]\!] & = [\![\bar{t}]\!] \approx [\![[\text{ap}(t_k) \mid t_k \leftarrow \bar{t}]]\!] \\
[\![\bar{t}]\!] & \approx & [\![]\!] & = [\![\bar{t}]\!] \approx [\![[\text{ap}(t_k) \mid t_k \leftarrow \bar{t}]]\!] \\
[\![\bar{t}_1]\!] & \approx & [\![\bar{t}_2]\!] & = \text{length}(\bar{t}_1) = \text{length}(\bar{t}_2) \wedge \\
 & & & \quad \bigwedge[\text{unq}(\text{pp}(t_{k1})) \approx \text{unq}(\text{pp}(t_{k2})) \mid (t_{k1}, t_{k2}) \leftarrow \bar{t}_1 | \bar{t}_2] \\
[\![\bar{t}_1]\!] & \approx & [\![\bar{t}_2]\!] & = \bar{t}_1 \approx \bar{t}_2 \\
[\![\mathit{other}_1]\!] & \approx & [\![\mathit{other}_2]\!] & = \text{false}
\end{array}$$

In other words, two omitted function prototypes are always compatible with each other and, further, a non-empty prototype without the "`...`" suffix is deemed compatible with an omitted prototype if and only if it is compatible with a prototype formed by applying default argument promotion to every element of its type list. On the other hand, a compatible pair of prototypes without the "`...`" suffix must specify type lists of equal lengths, whose corresponding elements constitute qualified or unqualified versions of compatible types after pointer promotion. In all other cases, a pair of function prototypes is compatible if and only if both operands include the "`...`" suffix and type lists that are pairwise-compatible with each other.

Two compatible types are considered to be "essentially the same" by the language and are generally interchangeable anywhere within a C program, unless one of these types represents an incomplete array type or a function without a prototype, in which case the under-specified variant of the type may remain inadmissible in certain semantic contexts described later in Sections 5.7 and 5.8.

5.4.12 Composite Types

A pair of compatible types may be used to derive a single *composite type*, which is always guaranteed to be compatible with both of the original candidates for such a composition. Intuitively, the composition algorithm combines all information depicted by its two type operands. Formally, it is represented by the binary type operator "$\sqcup$" and implemented by the following Haskell function:

$$[\![\cdot]\!] \sqcup [\![\cdot]\!] :: \mathit{type} \to \mathit{type} \to \mathit{type}$$

$$\begin{array}{l}
[\![t_1]\!] \sqcup [\![t_2]\!] \\
\quad \mid \text{AT} \quad (t_1) \wedge \text{unq}(t_1) = \text{unq}(t_2) = \text{tq}(t_1) \cup \text{tq}(t_2) \cup \text{unq}(t_1) \\
\quad \mid \text{ET} \quad (t_1) \wedge \mathcal{B}(t_1) \approx \text{unq}(t_2) \quad = \text{tq}(t_2) \cup t_1 \\
\quad \mid \text{ET} \quad (t_2) \wedge \mathcal{B}(t_2) \approx \text{unq}(t_1) \quad = \text{tq}(t_1) \cup t_2 \\
\quad \mid \text{SU} \quad (t_1) \wedge \text{SU} \quad (t_2) \wedge \text{su}(t_1) = \text{su}(t_2) \wedge \text{tag}(t_1) = \text{tag}(t_2) \\
\qquad\qquad = \text{tq}(t_1) \cup \text{tq}(t_2) \cup [\![[\![\text{su}(t_1)]\!]\ [\![\text{tag}(t_1)]\!]\ [\![\bar{m}(t_1) \sqcup \bar{m}(t_2)]\!]]\!] \\
\quad \mid \text{PTR}\ (t_1) \wedge \text{PTR}\ (t_2) = \text{tq}(t_1) \cup \text{tq}(t_2) \cup [\![[\![\mathcal{B}(t_1) \sqcup \mathcal{B}(t_2)]\!]\ \texttt{*}]\!] \\
\quad \mid \text{PTR}\ (t_1) \wedge \text{INT}\ (t_2) = \text{tq}(t_2) \cup (t_1) \\
\quad \mid \text{INT}\ (t_1) \wedge \text{PTR}\ (t_2) = \text{tq}(t_1) \cup (t_2) \\
\quad \mid \text{ARR}(t_1) \wedge \text{ARR}(t_2) = \text{tq}(t_1) \cup \text{tq}(t_2) \cup [\![[\![\mathcal{B}(t_1) \sqcup \mathcal{B}(t_2)]\!]\ \texttt{[}[\![\text{length}(t_1) \sqcup \text{length}(t_2)]\!]\ \texttt{]}]\!] \\
\quad \mid \text{FUN}(t_1) \wedge \text{FUN}(t_2) = \text{tq}(t_1) \cup \text{tq}(t_2) \cup [\![[\![\mathcal{B}(t_1) \sqcup \mathcal{B}(t_2)]\!]\ \texttt{(}[\![\text{prot}(t_1) \sqcup \text{prot}(t_2)]\!]\ \texttt{)}]\!] \\
\quad \mid \text{VT} \quad (t_1) \vee \text{VT} \quad (t_2) = \text{tq}(t_1) \cup \text{tq}(t_2) \cup [\![\texttt{void}]\!] \\
\quad \mid \{\text{unq}(t_1), \text{unq}(t_2)\} = \{[\![\texttt{int}]\!], [\![\texttt{signed int}]\!]\} = \text{tq}(t_1) \cup \text{tq}(t_2) \cup [\![\texttt{signed int}]\!]
\end{array}$$

Intuitively, this construction merges two compatible types by obtaining a composition of their individual components. If one of the operands represents an enumeration type and the other is compatible with that enumeration's corresponding integral type, then their composition returns the enumeration operand unchanged. Similarly, if one of the two types represents a qualified or unqualified version of the plain "***int***" type and the other is an explicit "***signed int***", then their composite type is, by definition, equal to the "***signed int***" type. Further, in order to simplify formalisation of certain C expressions described later in Section 5.7, the above definition extends the standard type composition rules as follows:

① If the two entities represent differently-qualified versions of otherwise compatible types, then their composite type is qualified with all the type qualifiers of both operands.

② If one of the two entities represents a void type, then the result is always equal to an appropriately-qualified version of that type, regardless of the nature of the other operand.

③ Finally, if one of the two entities represents a pointer and the other is an integral type, then the result is equal to the pointer operand.

If, in a composition of two array types, one of the operands has an incomplete length, then the result always assumes the length of the other operand. Otherwise, the result has a length identical to that of both types. Formally:

$$[\![\cdot]\!] \sqcup [\![\cdot]\!] :: \mathit{integer}_{opt} \to \mathit{integer}_{opt} \to \mathit{integer}_{opt}$$

$$\begin{array}{lll} [\![]\!] & \sqcup [\![n_{opt}]\!] & = n_{opt} \\ [\![n_{opt}]\!] & \sqcup [\![]\!] & = n_{opt} \\ [\![n_1]\!] & \sqcup [\![n_2]\!] \mid (n_1 = n_2) & = n_1 \end{array}$$

Similarly, when two structure or union types are composed by the above algorithm and one of these has an empty member list, then the result has the member list of the other operand. Otherwise, both member lists must have identical lengths and their composition is performed pairwise on the corresponding list elements as follows:

$$[\![\cdot]\!] \sqcup [\![\cdot]\!] :: \mathit{members} \to \mathit{members} \to \mathit{members}$$

$$\begin{array}{lll} [\![\bar{m}_1]\!] \sqcup [\![\bar{m}_2]\!] & \mid \bar{m}_1 = \varnothing & = \bar{m}_2 \\ & \mid \bar{m}_2 = \varnothing & = \bar{m}_1 \\ & \mid \mathrm{length}(\bar{m}_1) = \mathrm{length}(\bar{m}_2) & = [m_{k1} \sqcup m_{k2} \mid (m_{k1}, m_{k2}) \leftarrow \bar{m}_1 | \bar{m}_2] \end{array}$$

In particular, a pair of compatible members must always assume identical names and offset values, with their respective types composed in the result:

$$[\![\cdot]\!] \sqcup [\![\cdot]\!] :: \mathit{member} \to \mathit{member} \to \mathit{member}$$

$$[\![t_1\ x_1\ \texttt{@}\ n_1]\!] \sqcup [\![t_2\ x_2\ \texttt{@}\ n_2]\!] \mid (x_1 = x_2 \wedge n_1 = n_2) = [\![[\![t_1 \sqcup t_2]\!]\ x_1\ \texttt{@}\ n_1]\!]$$

Finally, the composition of two optional function prototypes produces an omitted prototype if and only if neither operand includes any prototype information. Otherwise, if only one of the operands has a prototype and that prototype does not include the

"*...*" suffix, then that prototype is preserved under composition. In all other cases, both prototypes must have identical lengths and agree in their use of the "*...*" suffix, in which case their composition produces a similarly-structured prototype, whose elements are obtained by a pairwise composition of the corresponding elements in their type lists after pointer promotion. In Haskell:

$$[\![\cdot]\!] \sqcup [\![\cdot]\!] :: \mathit{prototype}_{opt} \to \mathit{prototype}_{opt} \to \mathit{prototype}_{opt}$$

$$\begin{array}{lll}
[\![]\!] & \sqcup [\![]\!] & = [\![]\!] \\
[\![]\!] & \sqcup [\![\bar{t}]\!] & = \bar{t} \\
[\![\bar{t}]\!] & \sqcup [\![]\!] & = \bar{t} \\
[\![\bar{t}_1]\!] & \sqcup [\![\bar{t}_2]\!] \quad \mid \mathrm{length}(\bar{t}_1) = \mathrm{length}(\bar{t}_2) & = [\![[\mathrm{pp}(t_{k1}) \sqcup \mathrm{pp}(t_{k2}) \mid (t_{k1}, t_{k2}) \leftarrow \bar{t}_1 | \bar{t}_2]]\!] \\
[\![\bar{t}_1 \ldots]\!] & \sqcup [\![\bar{t}_2 \ldots]\!] \mid \mathrm{length}(\bar{t}_1) = \mathrm{length}(\bar{t}_2) & = [\![[\mathrm{pp}(t_{k1}) \sqcup \mathrm{pp}(t_{k2}) \mid (t_{k1}, t_{k2}) \leftarrow \bar{t}_1 | \bar{t}_2] \ \ldots]\!]
\end{array}$$

In Sections 5.8 and 5.10, composition is used to merge the typing information for multiple declarations of a single logical entity such as C function. Further, in Section 5.7, type composition is used to compute the types of most C expressions constructed from multiple operands with distinct typing properties. Although the later use of this operation is not recognised by the C Standard, it allows us to refactor the semantics of most C expression forms into a small number of orthogonal rules, which proves invaluable to a successful formalisation of these entities in Section 5.7.

5.5 Name Spaces and Scopes

In C, identifiers are used to designate a number of different kinds of entities, such as *functions*, *variables*, *type names*, *type tags*, *statement labels* and finally *members* of structure and union types. When two or more entities in a C program are designated by the same identifier, they are said to *share* that identifier with each other. A single identifier may be always shared by entities of a distinct kind, with the actual entity being denoted determined by its syntactic context. Accordingly, the language is said to define a number of *name spaces* for its identifiers, one for functions, variables and type names (known collectively as *ordinary identifiers*), one for type tags, one for statement labels and one each for every structure and union type defined by the program. The present section contains a detailed discussion of the ordinary identifier, tag and member name spaces, while the the rather specialised label identifiers are scrutinised later in Section 5.9 as part of a semantic analysis of C statements.

A single identifier may be also shared by multiple entities belonging to the same name space. However, unlike identifiers shared across name spaces, at most one of the entities that share such an identifier can be *visible* to any given portion of a C program. The collection of all entities visible to some program fragment is known as a *scope* and, in most cases, corresponds loosely to a set of constructs enclosed in a single compound statement "**{**...**}**".

The scoping structure of C programs differs substantially from that of typical purely functional languages such as Etude. In the Etude term "LET $\nu = \tau_1; \tau_2$", the scope of ν is contained cleanly within the body τ_2, as dictated by the lexical structure

of the underlying parse tree, whereas, in the C statement "**{int** *x*; s_1; s_2; ... s_n; **}**", the scope of *x* extends to the entire statement list "s_1; s_2; ... s_n;" and concludes only at the end of the complete compound statement that surrounds its declaration.

This effect is even more pronounced for tag names, whose declarations may be buried deeply within a C expression, often transcending the program's lexical syntax in a highly unpredictable manner. For example, the following code fragment constitutes a perfectly valid C program:

```
int x      = sizeof (struct s { int a, b, c });
struct s y = { 1, 2, 3 };
int z      = y.a + y.b + y.c;
```

The structure type *s* is introduced within the initialiser of *x* and remains visible to the following declarations of *y* and *z*, so that its scope extends outside of the nesting structure within which *s* has been defined.

In light of such examples, most of the entities encountered within a C program cannot be analysed in isolation and must be generally interpreted in the context of the surrounding program in which they appear. In the following discussion, this linguistic feature is modelled by a *translation context* explicitly associated with most syntactic entities of the program. Any adjustments to this context introduced by a given syntactic entity are included as part of that entity's formal denotation and, in almost all cases, the updated context is propagated to all subsequent entities in the depth-first left-to-right traversal of the program's parse tree. For example, in the above code fragment, the structure type *s* is introduced into the current tag scope *T* by its definition "**struct** *x* **{int** *a, b, c;* **}**" and remains visible to the following two statements.

For most entities, the translation context includes at least four components *S*, *T*, *D* and *I*, known as the *current variable* and *tag scopes*, *set of item definitions* and the *scope index*, respectively. Their structures are described in Haskell by the following three type synonyms:

S :	*identifier* $\mapsto$ *designator*	*(current variable or tag scope)*
D:	$v \mapsto item_{v\,opt}$	*(variable definitions)*
I :	*integer*	*(scope index)*

In particular, the current scopes *S* and *T* define the semantic significance of all visible identifiers whose declarations are bound to a C variable, typedef name, function designator, structure, union or enumeration type. Further, the set of item definitions *D* depicts the actual bindings of all Etude variables associated with any C functions and data objects defined in the course of translation. Finally, the scope index *I* represents the nesting depth of the current scope with the program's lexical structure.

The actual binding of a given entity in its current scope associates its identifier with a *name designator* formed from a C type, an entity known as *linkage* and a scope index, as captured by the following data type definition:

```
designator:
        linkage type @ I
```

whose three components can be accessed individually with help of the following trivial Haskell functions:

$\mathcal{L}[\![\cdot]\!]$ :: *designator* $\rightarrow$ *linkage* *(linkage)*
$\mathcal{T}[\![\cdot]\!]$ :: *designator* $\rightarrow$ *type* *(type)*
$I[\![\cdot]\!]$ :: *designator* $\rightarrow$ *integer* *(identifier)*

$\mathcal{L}[\![\ell\ t\ \texttt{@}\ I]\!] = \ell$
$\mathcal{T}[\![\ell\ t\ \texttt{@}\ I]\!] = t$
$I[\![\ell\ t\ \texttt{@}\ I]\!] = I$

A designator's scope index $I(d)$ identifies the precise scope into which the binding of the corresponding C name has been originally introduced. As we shall soon discover, this index is critical to a correct formalisation of the C declaration semantics in Section 5.8. The linkage of a designator has the following structure:

linkage:

`extern` *v*	*(external linkage)*
`intern` *v*	*(internal linkage)*
`private` *v*	*(private linkage)*
`auto` *v*	*(automatic linkage)*
`register` *v*	*(register linkage)*
`const` *integer*	*(constant linkage)*
`type`	*(type linkage)*

Collectively, all of the "**`extern`**" and "**`intern`**" linkage forms are said to represent the family of *globally-linked* entities, which are identified formally by the notation "GL(ℓ)". Further, the set of all "**`extern`**", "**`intern`**" and "**`private`**" entities constitutes the family of *static linkage forms*, identified by the Haskell predicate "SL" as follows:

GL$[\![\cdot]\!]$, SL$[\![\cdot]\!]$ :: *linkage* $\rightarrow$ *bool*

GL$[\![$**`extern`** *v*$]\!]$ = true
GL$[\![$**`intern`** *v*$]\!]$ = true
GL$[\![$*other*$]\!]$ = false

SL$[\![$**`extern`** *v*$]\!]$ = true
SL$[\![$**`intern`** *v*$]\!]$ = true
SL$[\![$**`private`** *v*$]\!]$ = true
SL$[\![$*other*$]\!]$ = false

The linkage of a designator serves the dual rôle of associating certain C identifiers with Etude variables, while also providing additional information about the precise usage of these variables within a C program. In particular, an identifier declared with a linkage of the form "**`extern`** *v*", "**`intern`** *v*" or "**`private`** *v*" always designates a single Etude object, whose declaration is included in the set of item definitions of the eventual Etude module constructed from the entire translation unit, while identifiers associated with a linkage of the form "**`auto`** *v*" or "**`register`** *v*" represent temporal variables defined locally within a C function, with the later form reserved specifically for identifiers declared with the "**`register`**" storage class specifier as described later

in Section 5.7. Collectively, these five constructions are known as the *l-value linkage* forms and are singled out by the following predicate "LV":

$$\mathrm{LV}[\![\cdot]\!] :: \mathit{linkage} \to \mathit{bool}$$

$$\begin{aligned}
\mathrm{LV}[\![&\texttt{extern}\ v]\!] &&= \mathrm{true}\\
\mathrm{LV}[\![&\texttt{intern}\ v]\!] &&= \mathrm{true}\\
\mathrm{LV}[\![&\texttt{private}\ v]\!] &&= \mathrm{true}\\
\mathrm{LV}[\![&\texttt{auto}\ v]\!] &&= \mathrm{true}\\
\mathrm{LV}[\![&\texttt{register}\ v]\!] &&= \mathrm{true}\\
\mathrm{LV}[\![&\mathit{other}]\!] &&= \mathrm{false}
\end{aligned}$$

A further discussion of l-values is included later in Section 5.7.1.2. Intuitively the Etude variable v associated with every such linkage form denotes a mutable object designated by that identifier. In this work, this variable is represented uniformly by the notation "$v(\ell)$", as formalised by the following Haskell definition:

$$v[\![\cdot]\!] :: \mathit{linkage} \to v$$

$$\begin{aligned}
v[\![&\texttt{extern}\ v]\!] &&= v\\
v[\![&\texttt{intern}\ v]\!] &&= v\\
v[\![&\texttt{private}\ v]\!] &&= v\\
v[\![&\texttt{auto}\ v]\!] &&= v\\
v[\![&\texttt{register}\ v]\!] &&= v
\end{aligned}$$

The actual bindings of these Etude variables are maintained in the associated set of variable definitions D, whose codomain will contain, by the end of an entire translation process, the precise set of items declared in the Etude module constructed from the supplied C program. Since local C variables are never associated with a static Etude item, they are represented in D by an omitted entity "ϵ". Their bindings are only recorded in D for the sake of presentation, in order to ensure that the denotation of every logically-distinct C entity retains a unique syntax after its translation into Etude. In particular, whenever a new local object is introduced into the C program, its linkage is generally bound to a unique Etude variable of the form "v_n", in which the integer index n renders the variable distinct from all others already present in D. Such a unique variable can be obtained naturally by the following Haskell construction:

$$\mathrm{v}[\![\cdot]\!] :: D \to v$$

$$\mathrm{v}[\![D]\!] = \mathrm{succ}(\max(\{[\![\mathrm{v}_k]\!] \mid [\![\mathrm{v}_k]\!] \leftarrow \mathrm{dom}(D)\} \cup \{\mathrm{v}_0\}))$$

which relies on the standard Haskell successor function "succ" to construct a new variable index greater than any others already present in D. Before obtaining the successor of the maximum element from the domain of D, the above construction ensures that this domain includes at least one internal variable of the "v_n" form by extending this set with an otherwise-unused variable "v_0". We will discuss the precise syntax and semantics of variable introduction separately in Section 5.8.

A linkage of the form "**const** n" indicates that the corresponding identifier represents an *enumeration constant* with the integer value specified by the integer n. Since

this value can be viewed as an intuitive denotation of the constant, its retrieval is depicted in the following presentation by the following notation "$n(d)$":

$n[\![\cdot]\!] :: \mathit{linkage} \to \mathit{integer}$

$n[\![\texttt{const}\ n]\!] = n$

Finally, a variable name bound to an object declaration with a "**`type`**" linkage is said to represent a *typedef name*. Such identifiers denote a C type rather than a variable and are introduced into the concrete syntax of the language by the "**`typedef`**" declarations scrutinised later in Section 5.8. Nevertheless, since typedef names share a single name space with other ordinary identifiers used to designate C variables and functions, they are recorded in the current scope as part of its object declaration set.

The "**`type`**" linkage form is also associated with the designator of every tag name binding present in a current tag scope T. The type of such a designator always represents a C structure, union or enumeration type, the syntax of which includes a tag variable used to differentiate between structurally identical but logically distinct types introduced by the program. To ensure uniqueness of these entities, every new structure, union and enumeration type is always labelled with a unique tag variable upon its introduction into the program. For convenience, these tags are generated by the following function "tag(D)" similar to the earlier definition of "$\nu(D)$":

$\mathrm{tag}[\![\cdot]\!] :: D \to \nu$

$\mathrm{tag}[\![D]\!] = \mathrm{succ}(\max(\{[\![\mathrm{T}_k]\!] \mid [\![\mathrm{T}_k]\!] \leftarrow \mathrm{dom}(D)\} \cup \{\mathrm{T}_0\}))$

Besides its use in type identification, these temporary variables serve no other useful purpose in the program and are always bound in the set of its item definitions D to an omitted Etude item "ϵ".

When a new structure or union type is initially introduced into the program's syntax, it always begins as an incomplete type with an empty list of members, subject to a later completion by a subsequent declaration of that type within the same scope. Formally, this type completion process is depicted by the notation "$\mathcal{C}(t \triangleright S)$", which adjusts the types of all designators found in the supplied scope S as follows:

$\mathcal{C}[\![\cdot]\!] :: (\mathit{type} \triangleright S) \to S$

$\mathcal{C}[\![t \triangleright S]\!] = \{x_k{:}[\![\ell_k\ [\![\mathcal{C}_\mathrm{T}(t' \triangleright t_k)]\!]\ \texttt{@}\ I_k]\!] \mid x_k{:}[\![\ell_k\ t_k\ \texttt{@}\ I_k]\!] \leftarrow S\}$
where $t' = [\![[\![\mathrm{su}(t)]\!]\ [\![\mathrm{tag}(t)]\!]\ [\![\mathcal{C}_\mathrm{M}(t' \triangleright \bar{m}(t))]\!]]\!]$

assuming, of course, that t represents a complete structure or union type.

In particular the construction "$\mathcal{C}_\mathrm{T}(t' \triangleright t)$" substitutes the supplied complete type t' for every occurrence of a compatible structure or union type in t. Formally:

$\mathcal{C}_\mathrm{T}[\![\cdot]\!] :: (\mathit{type} \triangleright \mathit{type}) \to \mathit{type}$

$$
\begin{array}{lll}
\mathcal{C}_\mathrm{T}[\![t' \triangleright t]\!] & \mid \mathrm{unq}(t) \approx t' & = \mathrm{tq}(t) \cup t' \\
& \mid \mathrm{SU}\ \ (t) & = \mathrm{tq}(t) \cup [\![[\![\mathrm{su}(t)]\!]\ [\![\mathrm{tag}(t)]\!]\ [\![\mathcal{C}_\mathrm{M}(t' \triangleright \bar{m}(t))]\!]]\!] \\
& \mid \mathrm{PTR}\ (t) & = \mathrm{tq}(t) \cup [\![[\![\mathcal{C}_\mathrm{T}(t' \triangleright \mathcal{B}(t))]\!]\ \texttt{*}]\!] \\
& \mid \mathrm{FUN}(t) & = \mathrm{tq}(t) \cup [\![[\![\mathcal{C}_\mathrm{T}(t' \triangleright \mathcal{B}(t))]\!]\ \texttt{(}[\![\mathcal{C}_\mathrm{P}(t' \triangleright \mathrm{prot}(t))]\!]\texttt{)}]\!] \\
& \mid \mathrm{ARR}(t) & = \mathrm{tq}(t) \cup [\![[\![\mathcal{C}_\mathrm{T}(t' \triangleright \mathcal{B}(t))]\!]\ \texttt{[}[\![\mathrm{length}(t)]\!]\texttt{]}]\!] \\
& \mid \mathrm{otherwise} & = t
\end{array}
$$

in which $\mathcal{C}_{\mathrm{M}}(t \triangleright \bar{m})$ projects the same algorithm onto the type of every member found in the specified list $\bar{m}$:

$$\mathcal{C}_{\mathrm{M}}[\![\cdot]\!] :: (\mathit{type} \triangleright \mathit{members}) \rightarrow \mathit{members}$$

$$\mathcal{C}_{\mathrm{M}}[\![t \triangleright \bar{m}]\!] = [[\![[\![\mathcal{C}_{\mathrm{T}}(t \triangleright t_k)]\!]\ x_k\ \texttt{@}\ n_k]\!] \mid [\![t_k\ x_k\ \texttt{@}\ n_k]\!] \leftarrow \bar{m}]$$

and, similarly, $\mathcal{C}_{\mathrm{P}}(t \triangleright p_{opt})$ completes every type found in a given optional function prototype p_{opt} as follows:

$$\mathcal{C}_{\mathrm{P}}[\![\cdot]\!] :: (\mathit{type} \triangleright \mathit{prototype}_{opt}) \rightarrow \mathit{prototype}_{opt}$$

$$\begin{aligned}
\mathcal{C}_{\mathrm{P}}[\![t \triangleright \epsilon]\!] &= [\![\epsilon]\!] \\
\mathcal{C}_{\mathrm{P}}[\![t \triangleright \bar{t}]\!] &= [\![[\![[\mathcal{C}_{\mathrm{T}}(t \triangleright t_k) \mid t_k \leftarrow \bar{t}]]\!]\!] \\
\mathcal{C}_{\mathrm{P}}[\![t \triangleright \bar{t} \ldots]\!] &= [\![[\![[\mathcal{C}_{\mathrm{T}}(t \triangleright t_k) \mid t_k \leftarrow \bar{t}]]\!] \ldots]\!]
\end{aligned}$$

A careful reader will observe that scope completion may, in fact, produce an infinite Haskell object, whenever the structure or union being completed is itself referenced in the syntax of a pointer entity embedded within the supplied type, member or function prototype operand. Fortunately, with its non-strict evaluation semantics, Haskell is well-equipped to deal with such constructions, provided that we refrain from excessive examination of the resulting member types. For example, in Section 5.4.11, the type compatibility algorithm for structures and unions was defined solely on the basic of equality between their "*struct-or-union*" and tag components, avoiding any reference to the associated member lists.

5.6 Constants

In the concrete syntax of the C programming language, most predetermined numeric values are represented by lexical tokens known as *constants*. Their collective structure is defined by the following trivial data type definitions:

constant :
floating-constant
integer-constant
enumeration-constant
character-constant

Since, in this work, we do not concern ourselves with details of a textual representation of C programs, the "*floating-constant*", "*integer-constant*" and "*character-constant*" constructs, as well as a similar syntactic entity "*string-literal*" described later in Section 5.7, are treated as abstract data types with an unspecified implementation. Their precise meanings are derived from the entity's lexical syntax and, in this work, are represented by the following four unspecified Haskell functions:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{FC}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \mathit{floating\text{-}constant} \rightarrow M(\mathit{type}, \mathit{rational}) \\
&\mathcal{D}_{\mathrm{IC}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \mathit{integer\text{-}constant} \rightarrow M(\mathit{type}, \mathit{integer}) \\
&\mathcal{D}_{\mathrm{CC}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \mathit{character\text{-}constant} \rightarrow M(\mathit{type}, \mathit{integer}) \\
&\mathcal{D}_{\mathrm{SC}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \mathit{string\text{-}literal} \rightarrow M(\mathit{type}, \mathit{data}_{\mathrm{v}})
\end{aligned}$$

observing that every such entity denotes a C type paired with the entity's actual value, as represented by a rational number for floating constants, an integer quantity for integer and character constants and an Etude object data specification for string literals. In all cases, the entity's denotation is derived within some unspecified exception monad M, in order to provision for any error diagnostics as stipulated earlier in Section 5.2.

On the other hand, every *enumeration constant* is represented simply by an identifier with a "**`const`**" linkage, so that its concrete syntax is defined in Haskell as follows:

enumeration-constant:
    identifier

Using these definitions, the type and rational value of every C constant can be derived in the context of its current scope S by the following construction:

$$\mathcal{D}_{\mathrm{C}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S \triangleright \mathit{constant}) \rightarrow M(\mathit{type}, \mathit{rational})$$

$$\begin{aligned}
\mathcal{D}_{\mathrm{C}}[\![S \triangleright \mathit{floating\text{-}constant}]\!] &= \mathcal{D}_{\mathrm{FC}}(\mathit{floating\text{-}constant}) \\
\mathcal{D}_{\mathrm{C}}[\![S \triangleright \mathit{integer\text{-}constant}]\!] &= \mathcal{D}_{\mathrm{IC}}(\mathit{integer\text{-}constant}) \\
\mathcal{D}_{\mathrm{C}}[\![S \triangleright \mathit{character\text{-}constant}]\!] &= \mathcal{D}_{\mathrm{CC}}(\mathit{character\text{-}constant})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\mathrm{C}}[\![S \triangleright \mathit{enumeration\text{-}constant}]\!] = \text{do} \\
&\quad \text{require } (\mathit{enumeration\text{-}constant} \in \mathrm{dom}(S) \wedge \mathcal{L}(d) = [\![\texttt{const}\ [\![n(\mathcal{L}(d))]\!]]\!]) \\
&\quad \text{return } (\mathcal{T}(d), n(\mathcal{L}(d))) \\
&\text{where } d = S(\mathit{enumeration\text{-}constant})
\end{aligned}$$

so that every well-formed enumeration constant must always represent a variable identifier that is bound in the current variable scope S to a designator with a constant linkage.

5.7 Expressions

Most of the computational complexity associated with C programs is captured by entities known as *expressions*. In the concrete syntax of the language, expressions can assume a number of different forms, most of which consist of an *operator* depicted by a lexical token such as "**`+`**" or "**`&&`**", together with one, two or three *operand expressions*.

In order to capture the relative binding strengths of the various arithmetic operators provided by the language, the concrete syntax of C expression has been stratified into sixteen distinct *precedence levels*, each associated with a unique non-terminal symbol in the BNF grammar and, consequently, a separate Haskell type in this presentation. In particular, every C expression form is uniquely classified as a *primary*, *postfix*, *unary*, *cast*, *multiplicative*, *additive*, *shift*, *relational*, *equality*, *bitwise AND*, *bitwise exclusive OR*, *bitwise inclusive OR*, *logical AND*, *logical OR*, *conditional*, *assignment* or plain *expression*. The precedence level of a C operator is used to disambiguate its application in a larger expression that also incorporates references to other operation forms. For example, since the "**`+`**" operator has a lower precedence than "**`*`**", the composite expression form "e_1 **`+`** e_2 **`*`** e_3" is interpreted as "e_1 **`+ (`** e_2 **`*`** e_3**`)`**" rather than "**`(`** e_1 **`+`** e_2**`) *`** e_3". Besides this simple syntactic rôle, precedence levels serve no other purpose in the language and do not influence a semantic interpretation of the individual expression forms in any other way.

The syntax of all valid C expression forms is captured by the following set of nineteen Haskell data types:

primary-expression:
- *identifier*
- *constant*
- *string-literal*
- `(` *expression* `)`

postfix-expression:
- *primary-expression*
- *postfix-expression* `[` *expression* `]`
- *postfix-expression* `(` $argument\text{-}expression\text{-}list_{opt}$ `)`
- *postfix-expression* `.` *identifier*
- *postfix-expression* `->` *identifier*
- *postfix-expression* `++`
- *postfix-expression* `--`

unary-expression:
- *postfix-expression*
- `++` *unary-expression*
- `--` *unary-expression*
- *unary-operator cast-expression*
- `sizeof` *unary-expression*
- `sizeof (` *type-name* `)`

unary-operator: one of
- `&` `*` `+` `-` `~` `!`

cast-expression:
- *unary-expression*
- `(` *type-name* `)` *cast-expression*

multiplicative-expression:
- *cast-expression*
- *multiplicative-expression* `*` *cast-expression*
- *multiplicative-expression* `/` *cast-expression*
- *multiplicative-expression* `%` *cast-expression*

additive-expression:
- *multiplicative-expression*
- *additive-expression* `+` *multiplicative-expression*
- *additive-expression* `-` *multiplicative-expression*

shift-expression:
- *additive-expression*
- *shift-expression* `<<` *additive-expression*
- *shift-expression* `>>` *additive-expression*

relational-expression:
- *shift-expression*
- *relational-expression* `<` *shift-expression*
- *relational-expression* `>` *shift-expression*
- *relational-expression* `<=` *shift-expression*
- *relational-expression* `>=` *shift-expression*

equality-expression:
    relational-expression
    equality-expression **==** *relational-expression*
    equality-expression **!=** *relational-expression*

AND-expression:
    equality-expression
    AND-expression **&** *equality-expression*

exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression **^** *AND-expression*

inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression **|** *exclusive-OR-expression*

logical-AND-expression:
    inclusive-OR-expression
    logical-AND-expression **&&** *inclusive-OR-expression*

logical-OR-expression:
    logical-AND-expression
    logical-OR-expression **||** *logical-AND-expression*

conditional-expression:
    logical-OR-expression
    logical-OR-expression **?** *expression* **:** *conditional-expression*

assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression

assignment-operator: one of
    = *= /= %= += -= <<= >>= &= ^= |=

expression:
    assignment-expression
    expression **,** *assignment-expression*

The user should observe that every Haskell type used to describe a given expression precedence level explicitly includes all data constructors from levels higher than itself. For example, "*additive-expression*" includes all constructors of the "*multiplicative-expression*" type. In the actual Haskell implementation, the initial data constructors of the "*multiplicative-expression*" and "*additive-expression*" types are actually written as "MultiplicativeExpressionT (CastExpressionT)" and "AdditiveExpression (MultiplicativeExpressionT)", respectively, although, for conciseness, their names are always hidden in the following presentation using a liberal application of appropriate implicit coercion functions. However, in most contexts, this segregation of C expressions into sixteen distinct precedence levels serves little useful purpose and it is convenient to think of all C expressions simply as values of the Haskell type "*expression*", which implicitly includes all other expression forms. Since most functions presented in this section are defined only on plain "*expression*" arguments, the reader should always

assume that every operand to these functions has been promoted into that type using an appropriate implicit coercion function, whose trivial definition and application have been excluded from the presentation in the interest of readability.

5.7.1 Meanings of Expressions

In the C language, the above expression syntax is utilised by a number of related, but conceptually distinct logical constructs known as *function designators*, *l-values*, *values*, *void*, *constant* and *static initialiser expressions*. The formal denotations of these entities are scrutinised individually by five separate Haskell functions "$\mathcal{D}_{\mathrm{FD}}$", "$\mathcal{D}_{\mathrm{LV}}$", "$\mathcal{D}_{\mathrm{V}}$", "$\mathcal{D}_{\mathrm{VE}}$", "$\mathcal{D}_{\mathrm{ICE}}$" and "$\mathcal{D}_{\mathrm{SIE}}$" that are defined independently in Sections 5.7.1.1, 5.7.1.2, 5.7.1.3, 5.7.1.4, 5.7.3.1 and 5.7.3.3 below.

In all cases, however, these denotations are derived in the context of a current variable and tag scopes S and T, set of item definitions D and the current scope index I. While no expression introduces any bindings into its current scope by itself, any type tag declarations found in the syntax of casts and "`sizeof`" operations may extend the scope with new declarations of structure, union and enumeration types. As already mentioned, such declarations persist outside of the expression's own syntax and, accordingly must be always propagated throughout the translation in the order of a depth-first left-to-right traversal of the program's parse tree. Further, the translation of string literals and certain forms of function call operations may result in an introduction of new bindings into the set of variable definitions D, whose envelopes may be also recorded in the *temporary variable set* V, which, intuitively, collects all variables local to a given C expression. The precise purpose of this set is discussed later in Section 5.9. Formally, the structure of V is defined as the following trivial Haskell type synonym:

$$V: \qquad \nu \mapsto \textit{envelope} \qquad\qquad \textit{(local variable definitions)}$$

Accordingly, a complete denotation of every C expression form consists of the updated scopes S' and T', item definition set D', the set V of all temporary variables introduced by the entity, as well as the expression's C type t and a monadic Etude term τ, whose reduction produces the sequence of side effects and an atomic value corresponding to the expression's computational meaning, except that, for constant expressions, τ can be variously replaced by the entity's numeric or atomic value, as described later in Section 5.7.3.

5.7.1.1 Function Designators

As suggested by its name, a *function designator* describes a reference to a C function. The set of all admissible function designator expressions consists precisely of those variables which, in the expression's current scope S, are bound to a well-formed l-value designator of a function type, together with every unary *indirection operation* of the form "$\star e$" in which e represents a well-formed value of a function pointer, as well as all parenthesised versions of one of these two expression forms. The meaning of every such function designator e is described by an appropriate C function type and

a monadic Etude term whose evaluation produces an atom from the function genre "F", with a value equal to the atomic encoding of the corresponding Etude function. Formally:

$$\mathcal{D}_{\text{FD}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \textit{expression}) \rightarrow M(S,\, S,\, D,\, V,\, \textit{type},\, \textit{term}_V)$$

$$\begin{aligned}
\mathcal{D}_{\text{FD}}[\![S,\, T,\, D,\, I \triangleright x]\!] &= \text{do require } (x \in \text{dom}(S) \wedge \text{LV}(\mathcal{L}(S(x))) \wedge \text{FUN}(\mathcal{T}(S(x)))) \\
&\qquad \text{return } (S,\, T,\, D,\, \varnothing,\, \mathcal{T}(S(x)),\, [\![\textsf{RET}\ [\![\textsf{v}(\mathcal{L}(S(x)))]\!]]\!]) \\
\mathcal{D}_{\text{FD}}[\![S,\, T,\, D,\, I \triangleright \star e]\!] &= \text{do } (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{V}}(S,\, T,\, D,\, I \triangleright e) \\
&\qquad \text{require } (\text{PTR}(t_e) \wedge \text{FUN}(\mathcal{B}(t_e))) \\
&\qquad \text{return } (S_e,\, T_e,\, D_e,\, V_e,\, \mathcal{B}(t_e),\, \tau_e) \\
\mathcal{D}_{\text{FD}}[\![S,\, T,\, D,\, I \triangleright \texttt{(}e\texttt{)}]\!] &= \text{do } \mathcal{D}_{\text{FD}}(S,\, T,\, D,\, I \triangleright e) \\
\mathcal{D}_{\text{FD}}[\![S,\, T,\, D,\, I \triangleright \textit{other}]\!] &= \text{reject}
\end{aligned}$$

Intuitively, expressions of the form "$\star e$" allow C programs to convert a value denotation of any well-formed function pointer e into a function designator, although, in reality, this operation is rarely required in practice, since, as described later in Section 5.7.1.3, the compiler is usually able to perform it implicitly without any aid from the programmer.

In the context of a *function call operation* discussed later in Section 5.7.1.3, an undeclared identifier also behaves like a designator of a function with the type "**`int()`**". Formally, the denotation of such a degenerate designator is obtained by the following trivial derivation:

$$\mathcal{D}_{\text{UND}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \textit{expression}) \rightarrow M(S,\, S,\, D,\, V,\, \textit{type},\, \textit{term}_V)$$

$$\begin{aligned}
&\mathcal{D}_{\text{UND}}[\![S,\, T,\, D,\, I \triangleright x]\!] = \text{do} \\
&\quad \text{require } (x \notin \text{dom}(S)) \\
&\quad \text{return } (S,\, T,\, \{x{:}[\![\textsf{IMP}\ x]\!]\}/D,\, \varnothing,\, [\![\texttt{int()}]\!],\, [\![\textsf{RET}\ (x)]\!])
\end{aligned}$$

$$\mathcal{D}_{\text{UND}}[\![S,\, T,\, D,\, I \triangleright \textit{other}]\!] = \text{reject}$$

observing that all such identifiers are implicitly bound in the resulting set of item definitions D to an imported Etude item "IMP x", provided that no previous binding of that identifier exists in the expression's context. In the above Haskell definition, this effect is achieved by extending a singular finite map "$\{x{:}[\![\textsf{IMP}\ x]\!]\}$" with all the bindings already present in D, using the map extension operator "$/$" described in Appendix A. Since "A/B" always replaces any keys mapped in both of its operands A and B with their B values, this useful idiom ensures that D is affected by the operation only if it contains no binding of x at the beginning of the construct.

5.7.1.2 L-Values

In C, an *l-value* represents an actual memory-resident object whose content may be manipulated by the program. Its denotation always depicts an Etude term which delivers a single atom from a function or object genre, whose reduced value describes an actual location of the denoted C object in the program's address space. Formally, the meanings of all such expressions are characterised by the following Haskell function:

$$\mathcal{D}_{\text{LV}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \textit{expression}) \rightarrow M(S,\, S,\, D,\, V,\, \textit{type},\, \textit{term}_V)$$

In particular, every variable name intended as a depiction of an l-value must be bound in its current scope to a designator with an l-value linkage and an object or incomplete type. Predictably, such identifiers denote the designator's type and a trivial monadic term that delivers the value of an Etude variable associated with that designator:

$$\begin{aligned}&\mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright x]\!] = \text{do}\\ &\quad \text{require } (x \in \text{dom}(S) \wedge \text{LV}(\mathcal{L}(S(x))) \wedge (\text{OBJ}(\mathcal{T}(S(x))) \vee \text{INC}(\mathcal{T}(S(x)))))\\ &\quad \text{return } (S,\, T,\, D,\, \varnothing,\, \mathcal{T}(S(x)),\, [\![\textsf{RET}\ [\![\nu(\mathcal{L}(S(x)))]\!]]\!])\end{aligned}$$

On the other hand, every well-formed string literal introduces a new unique variable $\nu(D)$ into its set of variable definitions D, binding that variable to a new Etude object item whose data specification is obtained from the string literal's denotation. An entire l-value of this form has the string literal's type and a denotation of the trivial Etude term reducible to its new variable value:

$$\begin{aligned}&\mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright \textit{string-literal}]\!] = \text{do}\\ &\quad (t,\, \bar{\delta}) \leftarrow \mathcal{D}_{\text{SC}}(\textit{string-literal})\\ &\quad \text{return } (S,\, T,\, D \cup \{\nu(D){:}[\![\textsf{OBJ}\ (\bar{\delta})\ \textsf{OF}\ ([\![\bar{\xi}(t)]\!])]\!]\},\, \varnothing,\, t,\, [\![\textsf{RET}\ [\![\nu(D)]\!]]\!])\end{aligned}$$

Every *direct member operation* of the form "$e.x$" also constitutes a well-formed l-value, provided that its expression operand e is itself an l-value of a complete structure or union type and that x represents some named member m of that type. The entire operation has the member's type $\mathcal{T}(m)$ and a denotation that computes a sum of its offset $O(m)$ within the entire structure or union object with the member's l-value converted into the corresponding object format $O(\phi(\mathcal{T}(m)))$. Formally:

$$\begin{aligned}&\mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright e.x]\!] = \text{do}\\ &\quad (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{LV}}(S,\, T,\, D,\, I \triangleright e)\\ &\quad (m) \leftarrow \bar{m}(t_e)(x)\\ &\quad \text{require } (\text{SU}(t_e) \wedge x \in \text{dom}(\bar{m}(t_e)))\\ &\quad \text{return } (S_e,\, T_e,\, D_e,\, V_e,\, \mathcal{T}(m),\\ &\qquad\qquad [\![\textsf{LET}\ \text{T}_1 = \tau_e;\ \textsf{RET}\ ([\![O(\phi(\mathcal{T}(m)))]\!]_{\text{O}.\Phi}(\text{T}_1) +_{[\![O(\phi(\mathcal{T}(m)))]\!]}\ \#[\![O(m)]\!]_{\text{Z}.\Phi})]\!])\end{aligned}$$

Similarly, in every *indirect member operation* of the form "$e\texttt{->}x$", the expression operand e must represent a well-formed value of a pointer to a structure or union type that includes x as a named member, with the entire construct assuming the member's type and a denotation that delivers the sum of the pointer's own value with the member's offset within the entire structure or union object. Formally:

$$\begin{aligned}&\mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright e\texttt{->}x]\!] = \text{do}\\ &\quad (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{V}}(S,\, T,\, D,\, I \triangleright e)\\ &\quad (m) \leftarrow \bar{m}(\mathcal{B}(t_e))(x)\\ &\quad \text{require } (\text{PTR}(t_e) \wedge \text{SU}(\mathcal{B}(t_e)) \wedge x \in \text{dom}(\bar{m}(t_e)))\\ &\quad \text{return } (S_e,\, T_e,\, D_e,\, V_e,\, \mathcal{T}(m),\\ &\qquad\qquad [\![\textsf{LET}\ \text{T}_1 = \tau_e;\ \textsf{RET}\ ([\![O(\phi(\mathcal{T}(m)))]\!]_{\text{O}.\Phi}(\text{T}_1) +_{[\![O(\phi(\mathcal{T}(m)))]\!]}\ \#[\![O(m)]\!]_{\text{Z}.\Phi})]\!])\end{aligned}$$

Intuitively, such operations are equivalent to an expression of the form "`(*`e`).`x", in which the *indirection operation* "`*`e" converts a value e of a pointer type "t`*`" into an l-value of type t that designates an object located at the memory address depicted by the

pointer. As described in Section 5.7.1.3, the denotation of every C pointer represents a monadic Etude term reducible precisely to such an address, so that its value is directly equivalent to that of the corresponding indirect l-value:

$$\begin{aligned}&\mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright \texttt{*}e]\!] = \text{do}\\ &\quad (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{V}}(S,\, T,\, D,\, I \triangleright e)\\ &\quad \text{require } (\text{PTR}(t_e))\\ &\quad \text{return } (S_e,\, T_e,\, D_e,\, V_e,\, \mathcal{B}(t_e),\, \tau_e)\end{aligned}$$

Last but not least, every parenthesised version of an l-value "`(`e`)`" has the same denotation as its constituent e, while an *array subscript operations* with the syntax of "e_1 `[`e_2`]`" is semantically equivalent to "`*(`e_1 `+ (`e_2`))`", which completes the entire collection of all l-value forms supported by the language:

$$\begin{aligned}\mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright \texttt{(}e\texttt{)}]\!] &= \mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright e]\!]\\ \mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright e_1\,\texttt{[}e_2\texttt{]}]\!] &= \mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright \texttt{*(}e_1\texttt{ + (}e_2\texttt{))}]\!]\\ \mathcal{D}_{\text{LV}}[\![S,\, T,\, D,\, I \triangleright \textit{other}]\!] &= \text{reject}\end{aligned}$$

An l-value is said to be *modifiable* if it represents an object whose value can be used as a target of an assignment operation. Formally, such l-values must have a complete object type other than an array and neither the object itself nor any of its members may include the "`const`" type qualifier. In all other respects, modifiable l-values behave like their ordinary l-value cousins, so that their meanings can be formalised concisely by the following construction:

$$\mathcal{D}_{\text{MLV}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \textit{expression}) \rightarrow M(S,\, S,\, D,\, V,\, \textit{type},\, \textit{term}_{\text{V}})$$

$$\begin{aligned}&\mathcal{D}_{\text{MLV}}[\![S,\, T,\, D,\, I \triangleright e]\!] = \text{do}\\ &\quad (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{LV}}(S,\, T,\, D,\, I \triangleright e)\\ &\quad \text{require } (\text{OBJ}(t_e) \wedge \neg(\text{ARR}(t_e)) \wedge [\![\texttt{const}]\!] \notin \text{rtq}(t_e))\\ &\quad \text{return } (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e)\end{aligned}$$

Finally, a well-formed l-value is said to be *addressable* if it does not describe an object of a C bit field type, or one declared with a register linkage form. Formally, the denotations of such constructs are derived in Haskell as follows:

$$\mathcal{D}_{\text{ALV}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \textit{expression}) \rightarrow M(S,\, S,\, D,\, V,\, \textit{type},\, \textit{term}_{\text{V}})$$

$$\begin{aligned}&\mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright x]\!] = \text{do}\\ &\quad \text{require } (x \in \text{dom}(S) \wedge \text{LV}(\mathcal{L}(S(x))) \wedge \mathcal{L}(S(x)) \neq [\![\texttt{register}\ [\![\nu(\mathcal{L}(S(x)))]\!]]\!]) \wedge\\ &\qquad\quad (\text{OBJ}(\mathcal{T}(S(x))) \vee \text{INC}(\mathcal{T}(S(x)))) \wedge \neg\text{BF}(\mathcal{T}(S(x)))\\ &\quad \text{return } (S,\, T,\, D,\, \varnothing,\, \mathcal{T}(S(x)),\, [\![\textsf{RET}\ [\![\nu(\mathcal{L}(S(x)))]\!]]\!])\end{aligned}$$

$$\begin{aligned}&\mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright \textit{string-literal}]\!] = \text{do}\\ &\quad (t,\, \bar{\delta}) \leftarrow \mathcal{D}_{\text{SC}}(\textit{string-literal})\\ &\quad \text{return } (S,\, T,\, D \cup \{\nu(D){:}[\![\textsf{OBJ}\ (\bar{\delta})\ \textsf{OF}\ ([\![\bar{\xi}(t)]\!])]\!]\},\, \varnothing,\, t,\, [\![\textsf{RET}\ [\![\nu(D)]\!]]\!])\end{aligned}$$

$$\begin{aligned}&\mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright e.x]\!] = \text{do}\\ &\quad (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{ALV}}(S,\, T,\, D,\, I \triangleright e)\\ &\quad (m) \leftarrow \bar{m}(t_e)(x)\\ &\quad \text{require } (\text{SU}(t_e) \wedge x \in \text{dom}(\bar{m}(t_e)))\\ &\quad \text{return } (S_e,\, T_e,\, D_e,\, V_e,\, \mathcal{T}(m),\\ &\qquad\quad [\![\textsf{LET T}_1 = \tau_e\textsf{; RET } ([\![O(\phi(\mathcal{T}(m)))]\!]_{\text{O}.\Phi}(\text{T}_1) +_{[\![O(\phi(\mathcal{T}(m)))]\!]} \#[\![O(m)]\!]_{\text{Z}.\Phi})]\!])\end{aligned}$$

$$\mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright e\texttt{->}x]\!] = \text{do}$$
$$\quad (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{V}}(S,\, T,\, D,\, I \triangleright e)$$
$$\quad (m) \leftarrow \bar{m}(\mathcal{B}(t_e))(x)$$
$$\quad \text{require } (\text{PTR}(t_e) \wedge \text{SU}(\mathcal{B}(t_e)) \wedge x \in \text{dom}(\bar{m}(t_e)))$$
$$\quad \text{return } (S_e,\, T_e,\, D_e,\, V_e,\, \mathcal{T}(m),$$
$$\qquad [\![\texttt{LET T}_1\texttt{ = }\tau_e\texttt{; RET (}[\![O(\phi(\mathcal{T}(m)))]\!]_{\texttt{O.}\Phi}(\texttt{T}_1) +_{[\![O(\phi(\mathcal{T}(m)))]\!]} \texttt{\#}[\![O(m)]\!]_{\texttt{z.}\Phi}\texttt{)}]\!])$$

$$\mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright \texttt{*}e]\!] = \text{do}$$
$$\quad (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{V}}(S,\, T,\, D,\, I \triangleright e)$$
$$\quad \text{require } (\text{PTR}(t_e))$$
$$\quad \text{return } (S_e,\, T_e,\, D_e,\, V_e,\, \mathcal{B}(t_e),\, \tau_e)$$

$$\mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright \texttt{(}e\texttt{)}]\!] = \mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright e]\!]$$
$$\mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright e_1\texttt{[}e_2\texttt{]}]\!] = \mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright \texttt{*(}e_1\texttt{ + (}e_2\texttt{))}]\!]$$
$$\mathcal{D}_{\text{ALV}}[\![S,\, T,\, D,\, I \triangleright \textit{other}]\!] = \text{reject}$$

Intuitively, such l-values describe well-formed operands of a unary *address operation* of the form "**&***e*" but, in all other contexts, are treated identically to any other l-value form described earlier.

5.7.1.3 Values

Besides function designators and l-values, most well-formed expressions belong to the semantic family of *values* which, intuitively, depict the actual Etude atoms resulting from such expressions' evaluation. In the translated C program, the meaning of a value with a scalar type is represented uniformly by a monadic Etude term that delivers an atom depicting the actual content of some memory-resident l-value, or else a result of an appropriate arithmetic operation on such atoms. On the other hand, values of a structure or union type are represented directly by their location within the program's address space, since the content of the corresponding object cannot, in general, be depicted by a single scalar quantity. Formally, the denotations of all well-formed C values are represented by the following Haskell construction:

$$\mathcal{D}_{\text{V}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \textit{expression}) \rightarrow M(S,\, S,\, D,\, V,\, \textit{type},\, \textit{term}_{\text{V}})$$

In particular, every parenthesised version of a value expression "**(***e***)**" has the same meaning as its constituent *e*, while a well-formed constant also constitutes a value of a C type derived from its lexical syntax, with a denotation that delivers an atomic rendition of the constant's numeric value under an appropriate Etude format:

$$\mathcal{D}_{\text{V}}[\![S,\, T,\, D,\, I \triangleright \textit{constant}]\!] = \text{do } (t,\, x) \leftarrow \mathcal{D}_{\text{C}}(S \triangleright \textit{constant})$$
$$\qquad\qquad\qquad\qquad\qquad\qquad \text{return } (S,\, T,\, D,\, \varnothing,\, t,\, [\![\texttt{RET (\#}x_{[\![\phi(t)]\!]}\texttt{)}]\!])$$
$$\mathcal{D}_{\text{V}}[\![S,\, T,\, D,\, I \triangleright \texttt{(}e\texttt{)}]\!] = \text{do } \mathcal{D}_{\text{V}}(S,\, T,\, D,\, I \triangleright e)$$

Further, every application of the "**sizeof**" operator to a type name that depicts an object type, or to an l-value or value expression of such a type, denotes a trivial term of the implementation-defined "***size_t***" type described earlier in Section 5.4.2, whose evaluation delivers the size of the operand's type as an integer atom. The reader should observe that any C expressions appearing within a "**sizeof**" operation are never evaluated by the program, so that all associated temporary variables V_e introduced

by these expressions' denotations are quietly discarded from the resulting set of item definitions D:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright \texttt{sizeof (}\textit{type-name}\texttt{)}]\!] = \mathrm{do}\\
&\quad (S_t, T_t, D_t, t_t) \leftarrow \mathcal{D}_{\mathrm{TN}}(S, T, D, I \triangleright \textit{type-name})\\
&\quad \mathrm{require}\ (\mathrm{OBJ}(t_t))\\
&\quad \mathrm{return}\ (S_t, T_t, D_t, \varnothing, [\![\texttt{size_t}]\!], [\![\mathsf{RET}\ (\#[\![\mathcal{S}(t_t)]\!]_{[\![\phi[\![\texttt{size_t}]\!]]\!]})]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright \texttt{sizeof}\ e]\!] = \mathrm{do}\\
&\quad (S_e, T_e, D_e, V_e, t_e, \tau_e) \leftarrow \mathcal{D}_{\mathrm{LV}}(S, T, D, I \triangleright e)\ ||\ \mathcal{D}_{\mathrm{V}}(S, T, D, I \triangleright e)\\
&\quad \mathrm{require}\ (\mathrm{OBJ}(t_e))\\
&\quad \mathrm{return}\ (S_e, T_e, D\backslash\mathrm{dom}(V_e), \varnothing, [\![\texttt{size_t}]\!], [\![\mathsf{RET}\ (\#[\![\mathcal{S}(t_e)]\!]_{[\![\phi[\![\texttt{size_t}]\!]]\!]})]\!])
\end{aligned}$$

in which the "||" symbol represents a generic monadic choice operator described in Appendix A. Given a list of n monadic Haskell terms $M_1, M_2 \ldots M_n$, the construction "$M_1 \,||\, M_2 \,||\, \ldots \,||\, M_n$" returns the first of its operands M_i which does not represent the monadic zero (or failure) action. In other words, in the above definition, the expression "$\mathcal{D}_{\mathrm{LV}}(S, T, D, I \triangleright e)\ ||\ \mathcal{D}_{\mathrm{V}}(S, T, D, I \triangleright e)$" is equivalent to $\mathcal{D}_{\mathrm{LV}}(S, T, D, I \triangleright e)$ if e represents a well-formed l-value and to $\mathcal{D}_{\mathrm{V}}(S, T, D, I \triangleright e)$ if it assumes a valid value form.

In a value context, a direct member operation of the form "*e.x*" obtains a location of the member x within the memory-resident representation of the value operand e and delivers the current content of the underlying Etude object as a result of the entire construct. Formally, its meaning is derived similarly to the one of an l-value with an analogous structure, except that the value, rather than l-value denotation of the operand e is used in the construction. Formally:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e.x]\!] = \mathrm{do}\\
&\quad (S_e, T_e, D_e, V_e, t_e, \tau_e) \leftarrow \mathcal{D}_{\mathrm{V}}(S, T, D, I \triangleright e)\\
&\quad (m) \leftarrow \bar{m}(t_e)(x)\\
&\quad (\tau) \leftarrow [\![\mathsf{LET}\ \mathrm{T}_1 = \tau_e;\ \mathsf{RET}\ ([\![O(\phi(\mathcal{T}(m)))]\!]_{\mathrm{O}.\Phi}(\mathrm{T}_1) +_{[\![O(\phi(\mathcal{T}(m)))]\!]}\ \#[\![O(m)]\!]_{\mathrm{Z}.\Phi})]\!]\\
&\quad \mathrm{require}\ (\mathrm{SU}(t_e) \wedge x \in \mathrm{dom}(\bar{m}(t_e)))\\
&\quad \mathrm{return}\ (S_e, T_e, D_e, V_e, \mathcal{T}(m), \mathcal{V}(\mathcal{T}(m) \triangleright \tau))
\end{aligned}$$

Once an address τ of the required memory-resident object has been obtained by the above definition, the actual value denotation of "*e.x*" is obtained from it by constructing a new Etude term which retrieves a current content of that object from the program's address space. In this work, such terms are depicted by the notation "$\mathcal{V}(t \triangleright \tau)$", in which τ represents an Etude term which delivers an address of a C variable or object with the type t. Specifically, if t represents a bit field type, then the constructed term retrieves the content of the bit field's *container object* using an Etude operation of the form "$\mathsf{GET}\ [\mathrm{T}_1, \bar{\mu}]_{\Psi}$", in which the variable "T_1" is bound to the container's location, $\bar{\mu}$ represents the set of all access attributes associated with t and Ψ depicts a designated Etude format assigned to all bit field containers. By definition, Ψ is always equal to the format of the "***unsigned long***" type, as captured by the following Haskell function:

$$\begin{aligned}
&\Psi :: \textit{format}\\
&\Psi = \phi[\![\texttt{unsigned int}]\!]
\end{aligned}$$

The actual bit field value is then extracted from its container in an unspecified manner discussed separately in Appendix C. On the other hand, if t represents any other scalar type, then the Etude term constructed by $\mathcal{V}$ always retrieves an entire content of the specified memory-resident object and returns it as an atom of a format appropriate for t. In all other cases, the constructed denotation is identical to τ itself, since, as already mentioned earlier in this section, all values of a function, structure, union or array type are depicted directly by the location of the corresponding Etude entity with the address space of a program. In Haskell, the $\mathcal{V}$ combinator can be implemented as follows:

$$\mathcal{V}[\![\cdot]\!] :: (type \triangleright term_{\nu}) \rightarrow term_{\nu}$$

$$\begin{aligned}\mathcal{V}[\![t \triangleright \tau]\!] \;\mid\; \mathrm{BF}(t) \quad &= [\![\texttt{LET T}_1 = \tau\texttt{; LET T}_2 = \texttt{GET [T}_1\texttt{, } [\![\bar{\mu}(t)]\!]]_{\Psi}\texttt{; RET } [\![\mathcal{U}(t \triangleright \texttt{T}_2)]\!]]\!] \\ \mid\; \mathrm{SCR}(t) \quad &= [\![\texttt{LET T}_1 = \tau\texttt{; GET [T}_1\texttt{, } [\![\bar{\mu}(t)]\!]]_{[\![\phi(t)]\!]}]\!] \\ \mid\; \text{otherwise} &= \tau\end{aligned}$$

The notation "$\mathcal{U}(t \triangleright \alpha)$" represents some unspecified Etude term that extracts the bit field t from its container value α. The precise semantics of $\mathcal{U}$ are left open to interpretation by the individual implementations of a C compiler but, on every system, the function is guaranteed to be accompanied by the reverse operation $\mathcal{P}$, such that $\mathcal{U}(t \triangleright \mathcal{P}(t \triangleright \alpha', \alpha)) \equiv \alpha$ for all bit field types t, whenever α and α' represent Etude atoms that are well-formed under the formats of t and the base type of t, respectively. Formally, these two functions are described by the following pair of Haskell type signatures:

$$\mathcal{U}[\![\cdot]\!] :: (type \triangleright atom_{\nu}) \rightarrow atom_{\nu}$$
$$\mathcal{P}[\![\cdot]\!] :: (type \triangleright atom_{\nu}, atom_{\nu}) \rightarrow atom_{\nu}$$

and the above-mentioned constraint on their definition is captured concisely by the following theorem:

$$\begin{aligned}&\textsc{pack} :: \forall\, x, y, t \Rightarrow \\ &\quad \mathrm{WF}[\![\#x_{[\![\phi(t)]\!]}]\!] \rightarrow \mathrm{WF}[\![\#y_{\Psi}]\!] \rightarrow [\![\mathrm{BF}(t)]\!] \rightarrow \\ &\quad [\![\mathcal{U}(t \triangleright \mathcal{P}(t \triangleright [\![\#y_{[\![\Psi]\!]}]\!], [\![\#x_{[\![\phi(t)]\!]}]\!]))]\!] \equiv [\![\#x_{[\![\phi(t)]\!]}]\!]\end{aligned}$$

Intuitively, this theorem guarantees that every numeric quantity within the range of values representable by a given bit field type t can be inserted into a container atom of the bit field format Ψ and subsequently extracted from that container without affecting the bit field's original value. Accordingly, under typical implementations of the C language, adjacent bit field members of structure type are often "packed" into a single "***`unsigned long`***" object in order to reduce the amount of space required for their storage within the program's address space.

In a *function call operation* of the form "e **(**ael_{opt}**)**", the first operand e must represent either a valid pointer value whose referenced type describes a function returning an object type, or else a function designator formed from an undeclared identifier x as described earlier in Section 5.7.1.1. In both cases, the entire function call assumes the returned type of the corresponding function designator. Every such operation introduces a new temporary l-value $\nu(D_a)$ of that returned type into the resulting set of local

variable definitions V, which, intuitively, is used to store the result computed by the targeted C function. Its actual denotation evaluates, in an unspecified order, the value of e and any expressions in the optional argument list ael_{opt}, then applies the function atom designated by e to the list of Etude variables derived from the supplied list of argument expressions and, finally, returns the content of the temporary l-value $\nu(D_a)$ as the result of the entire operation. Formally, the complete translation of such constructs is defined in Haskell as follows:

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S,\, T,\, D,\, I \triangleright e\ \texttt{(}ael_{opt}\texttt{)}\,]\!] = \mathrm{do}\\
&\quad (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\mathrm{V}}(S,\, T,\, D,\, I \triangleright e) \,||\, \mathcal{D}_{\mathrm{UND}}(S,\, T,\, D,\, I \triangleright e)\\
&\quad \mathrm{require}\ (\mathrm{PTR}(t_e) \wedge \mathrm{FUN}(\mathcal{B}(t_e)) \wedge \mathrm{OBJ}(\mathcal{B}(\mathcal{B}(t_e))))\\
&\quad (S_a,\, T_a,\, D_a,\, V_a,\, W_a,\, \bar{\beta}_a) \leftarrow \mathcal{D}_{\mathrm{AEL}}(S_e,\, T_e,\, D_e,\, I,\, \mathrm{prot}(\mathcal{B}(t_e)) \triangleright ael_{opt})\\
&\quad \mathrm{return}\ (S_a,\, T_a,\, D_a \cup \{\nu(D_a){:}\epsilon\},\, V_e \cup V_a \cup W_a \cup \{\nu(D_a){:}\bar{\xi}(\mathcal{B}(\mathcal{B}(t_e)))\},\, \mathcal{B}(\mathcal{B}(t_e)),\\
&\qquad\qquad [\![\mathtt{LET}\ [\![\mathtt{T}_1 = \tau_e]\!] +\!\!+ \bar{\beta}_a]\!];\\
&\qquad\qquad \mathtt{LET}\ () = \mathtt{T}_1([\![[\![\nu(D_a)]\!] +\!\!+ \mathrm{dom}(W_a)]\!]);\\
&\qquad\qquad [\![\mathcal{V}(\mathcal{B}(\mathcal{B}(t_e))) \triangleright \mathtt{RET}\ [\![\nu(D_a)]\!]]\!]\,])
\end{aligned}
$$

where the precise meaning of argument expression lists, represented above by the notation "$\mathcal{D}_{\mathrm{AEL}}(S,\, T,\, D,\, I,\, p_{opt} \triangleright ael_{opt})$", is discussed separately in Section 5.7.2.

Further, all values formed from a *cast operation* of the form "**(***type-name***)***e*" convert the value of their expression operand into the C type denoted by the specified type name, using an appropriate Etude operation "$\phi'_{\phi}(\alpha)$", in which α represents the atom delivered from evaluation of the value e, ϕ is its Etude format and ϕ' is the format corresponding to the targeted C type. In all cases, both the type name operand and the specified expression value must represent scalar quantity. Formally:

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S,\, T,\, D,\, I \triangleright\ \texttt{(}\textit{type-name}\texttt{)}e]\!] = \mathrm{do}\\
&\quad (S_t,\, T_t,\, D_t,\, t_t) \leftarrow \mathcal{D}_{\mathrm{TN}}(S,\, T,\, D,\, I \triangleright \textit{type-name})\\
&\quad (S_e,\, T_e,\, D_e,\, V_e,\, t_e,\, \tau_e) \leftarrow \mathcal{D}_{\mathrm{V}}(S_t,\, T_t,\, D_t,\, I \triangleright e)\\
&\quad \mathrm{require}\ (\mathrm{SCR}(t_t) \wedge \mathrm{SCR}(t_e))\\
&\quad \mathrm{return}\ (S_e,\, T_e,\, D_e,\, V_e,\, t_t,\, [\![\mathtt{LET}\ \mathtt{T}_1 = \tau_e;\ \mathtt{RET}\ ([\![\phi(t_t)]\!]_{[\![\phi(t_e)]\!]}(\mathtt{T}_1))]\!])
\end{aligned}
$$

The reader should, however, recall from Chapter 4 that the Etude format conversion operator is only partially specified in the generic fragment of Etude's semantics and therefore, indirectly, also in standard C programs. At the very least, this operator is always capable of translating atoms between two arithmetic formats, in which case the object's numeric value is retained whenever possible. Further, conversions between objects and integral types, objects of two different kinds and pairs of distinct function encodings are also possible on every instruction set architecture, although the precise meanings of all such *pointer conversions* are always left open for further specification by the individual implementations of the language.

Most often, pointer values are introduced into a C program using an *address operation* of the form "**&***e*". In every such operation, the operand must represent a function designator or an addressable l-value. In all cases, the entire construct has the type of a pointer to its expression operand and a denotation identical to that of the function

designator or l-value e. Formally:

$$\begin{aligned}&\mathcal{D}_{\mathrm{V}}[\![S,\,T,\,D,\,I \triangleright \texttt{\&}e]\!] = \mathrm{do}\\&\quad (S_e,\,T_e,\,D_e,\,V_e,\,t_e,\,\tau_e) \leftarrow \mathcal{D}_{\mathrm{FD}}(S,\,T,\,D,\,I \triangleright e) \;||\; \mathcal{D}_{\mathrm{ALV}}(S,\,T,\,D,\,I \triangleright e)\\&\quad \mathrm{return}\ (S_e,\,T_e,\,D_e,\,V_e,\,[\![t_e\,\texttt{*}]\!],\,\tau_e)\end{aligned}$$

The reader should recall from Chapter 4 that all details of a binary encoding of object values are left unspecified in the generic fragment of Etude semantics, so that no portable C program can ever rely on any particular numeric properties of the value produced by the above "**&**" operator.

Every unary C expression of the form "**+***e*" is equivalent to its value operand e converted into that operand's integral-promoted type. Similarly, "**−***e*" and "**~***e*" deliver the negation and bit complement of e after integral promotion. All three of these operations assume the type of their converted operands, which must belong to the arithmetic family for "**+**" and "**−**", or represent an integral type for "**~**". On the other hand, a *logical not operation* of the form "**!***e*" always adopts the plain "***int***" type and its operand e may represent an arbitrary scalar value, whose result is compared in the constructed denotation for equality with the integer constant "**0**", using an application of the binary Etude operator "$=_\phi$" under an appropriate format ϕ. Formally, the denotations of all such operations are expressed in Haskell as follows:

$$\begin{aligned}&\mathcal{D}_{\mathrm{V}}[\![S,\,T,\,D,\,I \triangleright \texttt{+}e]\!] = \mathrm{do}\\&\quad (S_e,\,T_e,\,D_e,\,V_e,\,t_e,\,\tau_e) \leftarrow \mathcal{D}_{\mathrm{V}}(S,\,T,\,D,\,I \triangleright e)\\&\quad \mathrm{require}\ (\mathrm{AT}(t_e))\\&\quad \mathrm{return}\ (S_e,\,T_e,\,D_e,\,V_e,\,\mathrm{ip}(t_e),\,[\![\mathsf{LET}\ \mathrm{T}_1 = \tau_e;\ \mathsf{RET}\ ([\![\phi(\mathrm{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\mathrm{T}_1))]\!])\end{aligned}$$

$$\begin{aligned}&\mathcal{D}_{\mathrm{V}}[\![S,\,T,\,D,\,I \triangleright \texttt{−}e]\!] = \mathrm{do}\\&\quad (S_e,\,T_e,\,D_e,\,V_e,\,t_e,\,\tau_e) \leftarrow \mathcal{D}_{\mathrm{V}}(S,\,T,\,D,\,I \triangleright e)\\&\quad \mathrm{require}\ (\mathrm{AT}(t_e))\\&\quad \mathrm{return}\ (S_e,\,T_e,\,D_e,\,V_e,\,\mathrm{ip}(t_e),\,[\![\mathsf{LET}\ \mathrm{T}_1 = \tau_e;\ \mathsf{RET}\ (-_{[\![\phi(\mathrm{ip}(t_e))]\!]}([\![\phi(\mathrm{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\mathrm{T}_1)))]\!])\end{aligned}$$

$$\begin{aligned}&\mathcal{D}_{\mathrm{V}}[\![S,\,T,\,D,\,I \triangleright \texttt{\~{}}e]\!] = \mathrm{do}\\&\quad (S_e,\,T_e,\,D_e,\,V_e,\,t_e,\,\tau_e) \leftarrow \mathcal{D}_{\mathrm{V}}(S,\,T,\,D,\,I \triangleright e)\\&\quad \mathrm{require}\ (\mathrm{INT}(t_e))\\&\quad \mathrm{return}\ (S_e,\,T_e,\,D_e,\,V_e,\,\mathrm{ip}(t_e),\,[\![\mathsf{LET}\ \mathrm{T}_1 = \tau_e;\ \mathsf{RET}\ (\sim_{[\![\phi(\mathrm{ip}(t_e))]\!]}([\![\phi(\mathrm{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\mathrm{T}_1)))]\!])\end{aligned}$$

$$\begin{aligned}&\mathcal{D}_{\mathrm{V}}[\![S,\,T,\,D,\,I \triangleright \texttt{!}e]\!] = \mathrm{do}\\&\quad (S_e,\,T_e,\,D_e,\,V_e,\,t_e,\,\tau_e) \leftarrow \mathcal{D}_{\mathrm{V}}(S,\,T,\,D,\,I \triangleright e)\\&\quad \mathrm{require}\ (\mathrm{SCR}(t_e))\\&\quad \mathrm{return}\ (S_e,\,T_e,\,D_e,\,V_e,\,[\![\texttt{int}]\!],\,[\![\mathsf{LET}\ \mathrm{T}_1 = \tau_e;\ \mathsf{RET}\ (\mathrm{T}_1 =_{[\![\phi(t_e)]\!]} \#0_{[\![\phi(t_e)]\!]})]\!])\end{aligned}$$

The two binary operators "*****" and "**/**" form a family of C expressions known as the *binary arithmetic operations*. A denotation of every such entity can be characterised completely by some binary Etude operator *op*, such as "$\times_\phi$" and "$\div_\phi$" for "*****" and "**/**", respectively. Each binary arithmetic operation must be supplied with a pair of values of some arithmetic C types. It first computes the common arithmetic type of these operands using the usual arithmetic conversion algorithm described earlier in Section 5.4.10 and converts both operands' values into that type, which will be also assumed by the entire operation. In all cases, both values are evaluated within a single Etude

group, thus leaving the relative order of their execution unspecified. The resulting Etude atoms are then applied to the provided Etude operator "op_ϕ", where ϕ represents an Etude format derived from the C type of the entire operation. Formally:

$$\begin{aligned}
&\mathcal{D}_{\text{AO}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\quad (S, S, D, I, \textit{binary-op} \triangleright \textit{expression}, \textit{expression}) \to M(S, S, D, V, \textit{type}, \textit{term}_{\text{V}}) \\
&\mathcal{D}_{\text{AO}}[\![S, T, D, I, op \triangleright e_1, e_2]\!] = \text{do} \\
&\quad (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_{\text{V}}(S, T, D, I \triangleright e_1) \\
&\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_{\text{V}}(S_1, T_1, D_1, I \triangleright e_2) \\
&\quad \text{require } (\text{AT}(t_1) \wedge \text{AT}(t_2)) \\
&\quad \text{return } (S_2, T_2, D_2, V_1 \cup V_2, t_1 \boxplus t_2, \\
&\qquad [\![\textsf{LET } \text{T}_1 = \tau_1, \text{T}_2 = \tau_2; \\
&\qquad\ \textsf{RET } (([\![\phi(t_1 \boxplus t_2)]\!]_{[\![\phi(t_1)]\!]}(\text{T}_1))\ op_{[\![\phi(t_1 \boxplus t_2)]\!]}\ ([\![\phi(t_1 \boxplus t_2)]\!]_{[\![\phi(t_2)]\!]}(\text{T}_2)))]\!])
\end{aligned}$$

In particular, the two C operators "`*`" and "`/`" are characterised by the respective binary arithmetic operations "$\times_\phi$" and "$\div_\phi$" as follows:

$$\begin{aligned}
\mathcal{D}_{\text{V}}[\![S, T, D, I \triangleright e_1 \texttt{ * } e_2]\!] &= \mathcal{D}_{\text{AO}}(S, T, D, I, [\![\times]\!] \triangleright e_1, e_2) \\
\mathcal{D}_{\text{V}}[\![S, T, D, I \triangleright e_1 \texttt{ / } e_2]\!] &= \mathcal{D}_{\text{AO}}(S, T, D, I, [\![\div]\!] \triangleright e_1, e_2)
\end{aligned}$$

Further, *addition* and *subtraction operations* of the form "e_1 `+` e_2" or "e_1 `-` e_2" represent binary arithmetic operations "$+_\phi$" and "$-_\phi$" if and only if both of their operands have arithmetic types. Otherwise, one of the two operands in an expression of the form "e_1 `+` e_2" must represent a pointer to an object type and the other must have an integral type, with the entire construct having the pointer operand's type and a denotation that evaluates both operands in an unspecified order, producing the sum of the pointer's value with the product of the integer operand's result converted into the standard integer format "z.Φ" and multiplied by the size of the pointer's referenced type. Formally:

$$\begin{aligned}
&\mathcal{D}_{\text{V}}[\![S, T, D, I \triangleright e_1 \texttt{ + } e_2]\!] \\
&\quad = \mathcal{D}_{\text{AO}}(S, T, D, I, [\![+]\!] \triangleright e_1, e_2) \\
&\quad \mid\mid \text{do } (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_{\text{V}}(S, T, D, I \triangleright e_1) \\
&\qquad\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_{\text{V}}(S_1, T_1, D_1, I \triangleright e_2) \\
&\qquad\quad \text{require } (\text{PTR}(t_1) \wedge \text{OBJ}(\mathcal{B}(t_1)) \wedge \text{INT}(t_2)) \\
&\qquad\quad \text{return } (S_2, T_2, D_2, V_1 \cup V_2, t_1, \\
&\qquad\qquad [\![\textsf{LET } \text{T}_1 = \tau_1, \text{T}_2 = \tau_2; \\
&\qquad\qquad\ \textsf{RET } (\text{T}_1 +_{[\![\phi(t_1)]\!]} (\text{z.}\Phi_{[\![\phi(t_2)]\!]}(\text{T}_2) \times_{\text{z.}\Phi} \#[\![\mathcal{S}(\mathcal{B}(t_1))]\!]_{\text{z.}\Phi}))]\!]) \\
&\quad \mid\mid \text{do } (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_{\text{V}}(S, T, D, I \triangleright e_1) \\
&\qquad\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_{\text{V}}(S_1, T_1, D_1, I \triangleright e_2) \\
&\qquad\quad \text{require } (\text{PTR}(t_2) \wedge \text{OBJ}(\mathcal{B}(t_2)) \wedge \text{INT}(t_1)) \\
&\qquad\quad \text{return } (S_2, T_2, D_2, V_1 \cup V_2, t_2, \\
&\qquad\qquad [\![\textsf{LET } \text{T}_1 = \tau_1, \text{T}_2 = \tau_2; \\
&\qquad\qquad\ \textsf{RET } (\text{T}_2 +_{[\![\phi(t_2)]\!]} (\text{z.}\Phi_{[\![\phi(t_1)]\!]}(\text{T}_1) \times_{\text{z.}\Phi} \#[\![\mathcal{S}(\mathcal{B}(t_2))]\!]_{\text{z.}\Phi}))]\!])
\end{aligned}$$

Similarly, if the first operand to a subtraction operation "e_1 `-` e_2" represents a pointer to an object type and the second has an integral type, then the entire expression has the pointer's type and a denotation that delivers the sum of the pointer's value with the product of the appropriately-converted integer operand multiplied by the negated size of

the pointer's referenced type. More so, if both operands represent pointers to qualified or unqualified versions of compatible object types, then the entire operation has the implementation-defined "**`ptrdiff_t`**" type described in Section 5.4.2 and denotes a difference of the pointers' values, divided by the size of their common referenced type. Formally:

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S,\, T,\, D,\, I \triangleright e_1 \texttt{ - } e_2]\!] \\
&\quad = \mathcal{D}_{\mathrm{AO}}(S,\, T,\, D,\, I,\, [\![-]\!] \triangleright e_1,\, e_2) \\
&\quad \| \text{ do } (S_1,\, T_1,\, D_1,\, V_1,\, t_1,\, \tau_1) \leftarrow \mathcal{D}_{\mathrm{V}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad\quad (S_2,\, T_2,\, D_2,\, V_2,\, t_2,\, \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}(S_1,\, T_1,\, D_1,\, I \triangleright e_2) \\
&\qquad\quad \text{require } (\mathrm{PTR}(t_1) \wedge \mathrm{OBJ}(\mathcal{B}(t_1)) \wedge \mathrm{INT}(t_2)) \\
&\qquad\quad \text{return } (S_2,\, T_2,\, D_2,\, V_1 \cup V_2,\, t_1, \\
&\qquad\qquad\qquad [\![\mathsf{LET}\ \mathrm{T}_1 = \tau_1,\, \mathrm{T}_2 = \tau_2; \\
&\qquad\qquad\qquad\ \mathsf{RET}\ (\mathrm{T}_1 +_{[\![\phi(t_1)]\!]} (\mathrm{Z}.\Phi_{[\![\phi(t_2)]\!]}(\mathrm{T}_2) \times_{\mathrm{Z}.\Phi} \#[\![-\mathcal{S}(\mathcal{B}(t_1))]\!]_{\mathrm{Z}.\Phi}))]\!]) \\
&\quad \| \text{ do } (S_1,\, T_1,\, D_1,\, V_1,\, t_1,\, \tau_1) \leftarrow \mathcal{D}_{\mathrm{V}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad\quad (S_2,\, T_2,\, D_2,\, V_2,\, t_2,\, \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}(S_1,\, T_1,\, D_1,\, I \triangleright e_2) \\
&\qquad\quad \text{require } (\mathrm{PTR}(t_1) \wedge \mathrm{PTR}(t_2) \wedge \mathrm{OBJ}(\mathcal{B}(t_2)) \wedge \mathrm{unq}(\mathcal{B}(t_1)) \approx \mathrm{unq}(\mathcal{B}(t_2))) \\
&\qquad\quad \text{return } (S_2,\, T_2,\, D_2,\, V_1 \cup V_2,\, [\![\texttt{ptrdiff\_t}]\!], \\
&\qquad\qquad\qquad [\![\mathsf{LET}\ \mathrm{T}_1 = \tau_1,\, \mathrm{T}_2 = \tau_2; \\
&\qquad\qquad\qquad\ \mathsf{RET}\ ([\![\phi[\![\texttt{ptrdiff\_t}]\!]]\!]_{\mathrm{Z}.\Phi}((\mathrm{T}_1 -_{[\![\phi(t_1)]\!]} \mathrm{T}_2) \div_{\mathrm{Z}.\Phi} \#[\![\mathcal{S}(\mathcal{B}(t_1 \sqcup t_2))]\!]_{\mathrm{Z}.\Phi}))]\!])
\end{aligned}
$$

Intuitively, if p is a pointer to the n-th element of some C array object and i is an integer, then the operations "p **+** i", "i **+** p" and "i **−** p" construct pointers to the element at the location $n + i$ or $n - i$ in the same array, provided that such an element exists. Further, if p_1 and p_2 represent two pointers into the same C array, then "p_1 **−** p_2" determines the number of elements in the array segment delimited by these two locations.

The next group of six expression forms, constructed from applications of the binary C operators "**%**", "**<<**", "**>>**", "**&**", "**^**" and "**|**" to pairs of value expressions, represents the family of *binary integral operations*. The semantics of such expressions differ from their earlier binary arithmetic cousins only in that their operands are restricted to integral, rather than arbitrary arithmetic types. The precise denotation of every well-formed binary integral operation is described completely by the following Haskell construction:

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{IO}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\qquad (S,\, S,\, D,\, I,\, \textit{binary-op} \triangleright \textit{expression},\, \textit{expression}) \rightarrow M(S,\, S,\, D,\, V,\, \textit{type},\, \textit{term}_{\mathrm{V}}) \\
&\mathcal{D}_{\mathrm{IO}}[\![S,\, T,\, D,\, I,\, op \triangleright e_1,\, e_2]\!] = \text{do} \\
&\qquad (S_1,\, T_1,\, D_1,\, V_1,\, t_1,\, \tau_1) \leftarrow \mathcal{D}_{\mathrm{V}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad (S_2,\, T_2,\, D_2,\, V_2,\, t_2,\, \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}(S_1,\, T_1,\, D_1,\, I \triangleright e_2) \\
&\qquad \text{require } (\mathrm{INT}(t_1) \wedge \mathrm{INT}(t_2)) \\
&\qquad \text{return } (S_2,\, T_2,\, D_2,\, V_1 \cup V_2,\, t_1 \mathbin{\blacksquare} t_2, \\
&\qquad\qquad [\![\mathsf{LET}\ \mathrm{T}_1 = \tau_1,\, \mathrm{T}_2 = \tau_2; \\
&\qquad\qquad\ \mathsf{RET}\ ((\![\phi(t_1 \mathbin{\blacksquare} t_2)]\!]_{[\![\phi(t_1)]\!]}(\mathrm{T}_1))\ op_{[\![\phi(t_1 \mathbin{\blacksquare} t_2)]\!]}\ ([\![\phi(t_1 \mathbin{\blacksquare} t_2)]\!]_{[\![\phi(t_2)]\!]}(\mathrm{T}_2)))]\!])
\end{aligned}
$$

In particular, the six C operators "**%**", "**<<**", "**>>**", "**&**", "**^**" and "**|**" represent binary integral operations characterised by the Etude operators "$\cdot|\cdot_\phi$", "$\ll_\phi$", "$\gg_\phi$", "$\triangle_\phi$",

"$\overline{\vee}_\phi$" and "∇_ϕ", respectively:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{\%}\ e_2]\!] = \mathcal{D}_{\mathrm{IO}}(S, T, D, I, [\![\cdot\%\cdot]\!] \triangleright e_1, e_2)\\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{<<}\ e_2]\!] = \mathcal{D}_{\mathrm{IO}}(S, T, D, I, [\![\ll]\!] \triangleright e_1, e_2)\\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{>>}\ e_2]\!] = \mathcal{D}_{\mathrm{IO}}(S, T, D, I, [\![\gg]\!] \triangleright e_1, e_2)\\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{\&}\ e_2]\!] = \mathcal{D}_{\mathrm{IO}}(S, T, D, I, [\![\overline{\wedge}]\!] \triangleright e_1, e_2)\\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{\^{}}\ e_2]\!] = \mathcal{D}_{\mathrm{IO}}(S, T, D, I, [\![\overline{\vee}]\!] \triangleright e_1, e_2)\\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{|}\ e_2]\!] = \mathcal{D}_{\mathrm{IO}}(S, T, D, I, [\![\nabla]\!] \triangleright e_1, e_2)
\end{aligned}$$

For a further information about the resulting Etude constructions, the reader is referred to the earlier discussion of their properties in Section 4.4.

Similarly, each of the four *relational operations* "e_1 **<** e_2", "e_1 **>** e_2", "e_1 **<=** e_2" and "e_1 **>=** e_2" can be also described by one of the binary Etude operators "$<_\phi$", "$>_\phi$", "$\leq_\phi$" or "$\geq_\phi$", using the following common derivation of its meaning:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{RO}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow\\
&\quad (S, S, D, I, \textit{binary-op} \triangleright \textit{expression}, \textit{expression}) \to M(S, S, D, V, \textit{type}, \textit{term}_{\mathrm{V}})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\mathrm{RO}}[\![S, T, D, I, \textit{op} \triangleright e_1, e_2]\!] = \text{do}\\
&\quad (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_{\mathrm{V}}(S, \ T, \ D, \ I \triangleright e_1)\\
&\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}(S_1, T_1, D_1, I \triangleright e_2)\\
&\quad \text{require } (\mathrm{AT}(t_1) \wedge \mathrm{AT}(t_2)) \vee\\
&\qquad\quad (\mathrm{PTR}(t_1) \wedge (\mathrm{OBJ}(\mathcal{B}(t_1)) \vee \mathrm{INC}(\mathcal{B}(t_1))) \wedge\\
&\qquad\quad \ \mathrm{PTR}(t_2) \wedge (\mathrm{unq}(\mathcal{B}(t_1)) \approx \mathrm{unq}(\mathcal{B}(t_2))))\\
&\quad \text{return } (S_2, T_2, D_2, V_1 \cup V_2, [\![\textbf{\textit{int}}]\!],\\
&\qquad\quad [\![\mathsf{LET}\ \mathrm{T}_1 = \tau_1,\ \mathrm{T}_2 = \tau_2;\\
&\qquad\quad\ \mathsf{RET}\ ([\![\phi(t_1 \ \textcircled{*}\ t_2)]\!]_{[\![\phi(t_1)]\!]}(\mathrm{T}_1))\ op_{[\![\phi(t_1 \ \textcircled{*}\ t_2)]\!]}\ ([\![\phi(t_1 \ \textcircled{*}\ t_2)]\!]_{[\![\phi(t_2)]\!]}(\mathrm{T}_2)))]\!])
\end{aligned}$$

Intuitively, in every such operation, either both of the operands must have arithmetic types, or else both operands must represent pointers to qualified or unqualified versions of compatible object or incomplete types. Either way, the result has the plain "***int***" type and a denotation that applies the corresponding Etude operator *op* to the values of both operands, converted into their common arithmetic type. Specifically, the four C operators "**<**", "**>**", "**<=**" and "**>=**" are typified by their respective Etude equivalents "$<_\phi$", "$>_\phi$", "$\leq_\phi$" and "$\geq_\phi$" as follows:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{<}\ e_2]\!] = \mathcal{D}_{\mathrm{RO}}(S, T, D, I, [\![<]\!] \triangleright e_1, e_2)\\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{>}\ e_2]\!] = \mathcal{D}_{\mathrm{RO}}(S, T, D, I, [\![>]\!] \triangleright e_1, e_2)\\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{<=}\ e_2]\!] = \mathcal{D}_{\mathrm{RO}}(S, T, D, I, [\![\leq]\!] \triangleright e_1, e_2)\\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{>=}\ e_2]\!] = \mathcal{D}_{\mathrm{RO}}(S, T, D, I, [\![\geq]\!] \triangleright e_1, e_2)
\end{aligned}$$

Further, a closely-related pair of *equality operations* "e_1 **==** e_2" and "e_1 **!=** e_2" corresponds to the binary Etude operators "$=_\phi$" and "$\neq_\phi$" as follows:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{==}\ e_2]\!] = \mathcal{D}_{\mathrm{EO}}(S, T, D, I, [\![=]\!] \triangleright e_1, e_2)\\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \ \texttt{!=}\ e_2]\!] = \mathcal{D}_{\mathrm{EO}}(S, T, D, I, [\![\neq]\!] \triangleright e_1, e_2)
\end{aligned}$$

However, equality operators accept a broader range of operands then those supported by the earlier relational operation forms. In particular, they can be also applied in comparison of two function pointers, a pointer to an object against a "***void***" pointer and for comparison of arbitrary pointer values against the integer constant "**0**", optionally

cast into the "***void****" type as described in Section 5.7.3.2. Formally, every application of the C operator "**==**" and "**!=**" must satisfy one of the following conditions:

① both operands must constitute values of arithmetic types, or
② both operands must represent pointers to qualified or unqualified versions of compatible types, or
③ one operand must represent a pointer to an object or incomplete type and the other must describe a pointer to a qualified or unqualified version of "***void***", or
④ one operand must be a pointer and the other must constitute a *null pointer constant*, of the form described later in Section 5.7.3.2.

In Haskell:

$$\mathcal{D}_{\text{EO}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I, \textit{binary-op} \triangleright \textit{expression}, \textit{expression}) \rightarrow M(S, S, D, V, \textit{type}, \textit{term}_v)$$

$$\begin{aligned}
&\mathcal{D}_{\text{EO}}[\![S, T, D, I, op \triangleright e_1, e_2]\!] = \text{do}\\
&\quad (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_{\text{V}}(S,\ T,\ D,\ I \triangleright e_1)\\
&\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_{\text{V}}(S_1, T_1, D_1, I \triangleright e_2)\\
&\quad \text{require } (\text{AT}(t_1) \wedge \text{AT}(t_2)) \vee\\
&\qquad (\text{PTR}(t_1) \wedge \text{PTR}(t_2) \wedge (\text{unq}(\mathcal{B}(t_1)) \approx \text{unq}(\mathcal{B}(t_2)) \vee\\
&\qquad\qquad ((\text{OBJ}(\mathcal{B}(t_1)) \vee \text{INC}(\mathcal{B}(t_1))) \wedge \text{VT}(\mathcal{B}(t_2))) \vee\\
&\qquad\qquad ((\text{OBJ}(\mathcal{B}(t_2)) \vee \text{INC}(\mathcal{B}(t_2))) \wedge \text{VT}(\mathcal{B}(t_1))))) \vee\\
&\qquad (\text{PTR}(t_1) \wedge \text{NULL}(S_1, T_1, D_1, I \triangleright e_2)) \vee\\
&\qquad (\text{PTR}(t_2) \wedge \text{NULL}(S, T, D, I \triangleright e_1))\\
&\quad \text{return } (S_2, T_2, D_2, V_1 \cup V_2, [\![\textbf{\textit{int}}]\!],\\
&\qquad\quad [\![\textsf{LET } \text{T}_1 = \tau_1, \text{T}_2 = \tau_2;\\
&\qquad\qquad \textsf{RET } ((([\![\phi(t_1 \mathbin{\boxtimes} t_2)]\!]_{[\![\phi(t_1)]\!]}(\text{T}_1))\ op_{[\![\phi(t_1 \mathbin{\boxtimes} t_2)]\!]}\ ([\![\phi(t_1 \mathbin{\boxtimes} t_2)]\!]_{[\![\phi(t_2)]\!]}(\text{T}_2)))]\!])
\end{aligned}$$

On the other hand, in all *logical operations* of the form "e_1 **&&** e_2" and "e_1 **||** e_2", both operands may represent arbitrary scalar values and the entire construction always assumes the plain "***int***" type. In particular, a *logical AND operation* of the form "e_1 **&&** e_2" returns the integer constant "**1**" if both of its operands have non-zero values and "**0**" otherwise. Specifically, it begins by evaluating the first operand e_1 and comparing its atomic value for inequality with the constant "**0**" of an appropriate C type. If e_1 has a non-zero value, then the operation evaluates its second operand e_2, delivering the comparison of its atomic value against "**0**". Otherwise, e_2 is never evaluated and the entire expression delivers the constant value of "**0**" as a result of the entire operation:

$$\begin{aligned}
&\mathcal{D}_{\text{V}}[\![S, T, D, I \triangleright e_1 \texttt{ \&\& } e_2]\!] = \text{do}\\
&\quad (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_{\text{V}}(S,\ T,\ D,\ I \triangleright e_1)\\
&\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_{\text{V}}(S_1, T_1, D_1, I \triangleright e_2)\\
&\quad \text{require } (\text{SCR}(t_1) \wedge \text{SCR}(t_2))\\
&\quad \text{return } (S_2, T_2, D_2, V_1 \cup V_2, [\![\textbf{\textit{int}}]\!],\\
&\qquad\quad [\![\textsf{LET } \text{T}_1 = \tau_1;\\
&\qquad\qquad \textsf{IF } \text{T}_1 \neq_{[\![\phi(t_1)]\!]} \#0_{[\![\phi(t_1)]\!]} \textsf{ THEN}\\
&\qquad\qquad\quad \textsf{LET } \text{T}_2 = \tau_2;\\
&\qquad\qquad\quad \textsf{RET } (\text{T}_2 \neq_{[\![\phi(t_2)]\!]} \#0_{[\![\phi(t_2)]\!]})\\
&\qquad\qquad \textsf{ELSE}\\
&\qquad\qquad\quad \textsf{RET } (\#0_{[\![\phi(t_1)]\!]})]\!])
\end{aligned}$$

Similarly, a *logical OR operation* of the form "e_1 `||` e_2" returns "`1`" if and only if at least one of its operands has a non-zero value. First, it compares the operand e_1 against "`0`" and always delivers "`1`" if e_1 has a non-zero value, without ever examining the denotation of its second operand e_2. Otherwise, the entire operation delivers the result of comparing e_2 against "`1`" as follows:

$$
\begin{aligned}
&\mathcal{D}_\mathrm{V}[\![S, T, D, I \triangleright e_1 \texttt{ || } e_2]\!] = \mathrm{do}\\
&\quad (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_\mathrm{V}(S,\ T,\ D,\ I \triangleright e_1)\\
&\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_\mathrm{V}(S_1, T_1, D_1, I \triangleright e_2)\\
&\quad \mathrm{require}\ (\mathrm{SCR}(t_1) \wedge \mathrm{SCR}(t_2))\\
&\quad \mathrm{return}\ (S_2, T_2, D_2, V_1 \cup V_2, [\![\texttt{int}]\!],\\
&\qquad\qquad [\![\mathsf{LET}\ \mathrm{T}_1 = \tau_1;\\
&\qquad\qquad\ \mathsf{IF}\ \mathrm{T}_1 \neq_{[\![\phi(t_1)]\!]} \#0_{[\![\phi(t_1)]\!]}\ \mathsf{THEN}\\
&\qquad\qquad\qquad \mathsf{RET}\ (\#1_{[\![\phi(t_1)]\!]})\\
&\qquad\qquad\ \mathsf{ELSE}\\
&\qquad\qquad\qquad \mathsf{LET}\ \mathrm{T}_2 = \tau_2;\\
&\qquad\qquad\qquad \mathsf{RET}\ (\mathrm{T}_2 \neq_{[\![\phi(t_2)]\!]} \#0_{[\![\phi(t_2)]\!]})]\!])
\end{aligned}
$$

Finally, the meaning of all *conditional operations* of the form "e_1 `?` e_2 `:` e_3" can be formalised in Haskell as follows:

$$
\begin{aligned}
&\mathcal{D}_\mathrm{V}[\![S, T, D, I \triangleright e_1 \texttt{ ? } e_2 \texttt{ : } e_3]\!] = \mathrm{do}\\
&\quad (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_\mathrm{V}(S,\ T,\ D,\ I \triangleright e_1)\\
&\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_\mathrm{V}(S_1, T_1, D_1, I \triangleright e_2)\\
&\quad (S_3, T_3, D_3, V_3, t_3, \tau_3) \leftarrow \mathcal{D}_\mathrm{V}(S_2, T_2, D_2, I \triangleright e_3)\\
&\quad \mathrm{require}\ (\mathrm{SCR}(t_1)) \wedge\\
&\qquad ((\mathrm{AT}(t_2) \ \wedge \mathrm{AT}(t_3)) \vee\\
&\qquad\ (\mathrm{SU}(t_2) \ \wedge \mathrm{unq}(t_2) \approx \mathrm{unq}(t_3)) \vee\\
&\qquad\ (\mathrm{PTR}(t_2) \wedge \mathrm{PTR}(t_3) \wedge \mathrm{unq}(\mathcal{B}(t_2)) \approx \mathrm{unq}(\mathcal{B}(t_3))) \vee\\
&\qquad\ (\mathrm{PTR}(t_2) \wedge \mathrm{NULL}(S_2, T_2, D_2, I \triangleright e_3)) \vee\\
&\qquad\ (\mathrm{PTR}(t_3) \wedge \mathrm{NULL}(S_1, T_1, D_1, I \triangleright e_2)) \vee\\
&\qquad\ (\mathrm{PTR}(t_2) \wedge \mathrm{PTR}(t_3) \wedge (((\mathrm{OBJ}(\mathcal{B}(t_2)) \vee \mathrm{INC}(\mathcal{B}(t_2))) \wedge \mathrm{VT}(\mathcal{B}(t_3))) \vee\\
&\qquad\qquad\qquad\qquad ((\mathrm{OBJ}(\mathcal{B}(t_3)) \vee \mathrm{INC}(\mathcal{B}(t_3))) \wedge \mathrm{VT}(\mathcal{B}(t_2))))))\\
&\quad \mathrm{return}\ (S_3, T_3, D_3, V_1 \cup V_2 \cup V_3, t_2 \boxtimes t_3,\\
&\qquad\qquad [\![\mathsf{LET}\ \mathrm{T}_1 = \tau_1;\\
&\qquad\qquad\ \mathsf{IF}\ \mathrm{T}_1 \neq_{[\![\phi(t_1)]\!]} \#0_{[\![\phi(t_1)]\!]}\ \mathsf{THEN}\\
&\qquad\qquad\qquad \mathsf{LET}\ \mathrm{T}_2 = \tau_2;\\
&\qquad\qquad\qquad \mathsf{RET}\ ([\![\phi(t_2 \boxtimes t_3)]\!]_{[\![\phi(t_2)]\!]}(\mathrm{T}_2))\\
&\qquad\qquad\ \mathsf{ELSE}\\
&\qquad\qquad\qquad \mathsf{LET}\ \mathrm{T}_3 = \tau_3;\\
&\qquad\qquad\qquad \mathsf{RET}\ ([\![\phi(t_2 \boxtimes t_3)]\!]_{[\![\phi(t_3)]\!]}(\mathrm{T}_3))]\!])
\end{aligned}
$$

In other words, such constructs always assume the common arithmetic type of their second and third operand. Their denotations represent monadic Etude terms that evaluate e_1 and return the appropriately-converted denotation of either e_2 or e_3, whenever the first operand has a non-zero or zero value, respectively. The reader should observe that, in all cases, at most one of the two operands e_2 or e_3 is ever reduced on any given execution path through the program. At any rate, in every such conditional operation, the first operand e_1 must have a scalar type and the other two expressions must represent

values that satisfy one of the following five conditions:

① both of these operands represent values of an arithmetic type, or

② both of these operands represent qualified or unqualified versions of compatible structure or union object types, or

③ both represent pointers to qualified or unqualified versions of compatible types, or

④ either operand represents is a pointer and the other is a null pointer constant, or

⑤ one operand represents a pointer to an object or incomplete type and the other represents a pointer to a qualified or unqualified version of the "**`void`**" type.

The denotation of a comma operation "e_1`,`e_2" is rather simpler. Intuitively, in such expressions the first operand e_1 may represent an arbitrary function designator, l-value, value or void expression and the second operand must represent a value, whose type is also inherited by the entire construct. A denotation of every comma operation evaluates both of its arguments in sequence, discarding the result of e_1 and delivering that of the second operand e_2. Formally:

$$\begin{aligned}&\mathcal{D}_{\mathrm{V}}[\![S,\, T,\, D,\, I \triangleright e_1\texttt{,}e_2]\!] = \mathrm{do}\\ &\quad (S_1,\, T_1,\, D_1,\, V_1,\, \tau_1) \leftarrow \mathcal{D}_{\mathrm{E}}(S,\, T,\, D,\, I \triangleright e_1)\\ &\quad (S_2,\, T_2,\, D_2,\, V_2,\, t_2,\, \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}(S_1,\, T_1,\, D_1,\, I \triangleright e_2)\\ &\quad \mathrm{return}\ (S_2,\, T_2,\, D_2,\, V_1 \cup V_2,\, t_2,\, [\![\mathsf{LET}\ () = \tau_1;\, \tau_2]\!])\end{aligned}$$

All of the remaining value operators "**`=`**", "**`*=`**", "**`/=`**", "**`%=`**", "**`+=`**", "**`-=`**", "**`<<=`**", "**`>>=`**", "**`&=`**", "**`^=`**", "**`|=`**", "**`++`**" and "**`--`**" introduce the set of fifteen *assignment operation* forms, intended to facilitate modification of values contained in memory-resident objects. The first operand to every assignment operation must always represent a modifiable l-value. The entire construct assumes that l-value's type, delivering the updated content of the underlying Etude object.

In particular, a *simple assignment operation* of the form "e_1 **`=`** e_2" replenishes the object designated by the l-value e_1 with an appropriately-converted value of e_2, provided that both expressions satisfy one of the following conditions:

① both operands have arithmetic types, or

② e_1 has a structure or union type, whose unqualified version is compatible with the type of e_2 and both operands represent object types, or

③ both e_1 and e_2 represent pointers to qualified or unqualified versions of compatible types and the referenced type of e_2 is qualified with a subset of the qualifications associated with the type referenced by e_1, or

④ one of the two expressions represents a pointer to an object or incomplete type, the other is a pointer to the "**`void`**" type and the referenced type of e_2 is qualified with a subset of the qualifications associated with the type referenced by e_1, or

⑤ e_1 has a pointer type and e_2 is a null pointer constant.

In all cases, the entire operation has the type of the first operand e_1. Its denotation evaluates, in an unspecified order, both the l-value e_1 and the value of e_2, with the later

converted into the type of the entire operation. Next, the converted value of e_2 is placed into the memory-resident object designated by e_1 as follows:

① if the e_1 has a bit field type, then the existing value of the entire bit field container at that memory location is retrieved and replenished with the value of e_2, using the unspecified $\mathcal{P}$ combinator described earlier in this section, with the result stored back into the same memory location.

② On the other hand, if e_2 has any other scalar type, then the object designated by its l-value is replenished with the value of e_2, converted into the first operand's type.

③ Finally, if e_1 represents a structure or union l-value, then the entire structure or union object depicted by the denotation of e_2 is copies into that l-value using an unspecified function "set", whose typical implementation is given in Appendix C.

In Haskell:

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e_1 \texttt{ = } e_2]\!] \\
&\texttt{= do } (S_1,\ T_1,\ D_1,\ V_1,\ t_1,\ \tau_1) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad (S_2,\ T_2,\ D_2,\ V_2,\ t_2,\ \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}\ \ (S_1,\ T_1,\ D_1,\ I \triangleright e_2) \\
&\qquad \mathrm{require}\ (\mathrm{BF}(t_1) \wedge \mathrm{AT}(t_2)) \\
&\qquad \mathrm{return}\ (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ t_1, \\
&\qquad\qquad [\![\texttt{LET T}_1 \texttt{ = } \tau_1,\ \texttt{T}_2 \texttt{ = } \tau_2; \\
&\qquad\qquad\ \texttt{LET T}_3 \texttt{ = GET [T}_1,\ [\![\bar{\mu}(t_1)]\!]\texttt{]}_{\Psi}; \\
&\qquad\qquad\ \texttt{LET T}_4 \texttt{ = RET } ([\![\phi(t_1)]\!]_{[\![\phi(t_2)]\!]}(\texttt{T}_2)); \\
&\qquad\qquad\ \texttt{LET () = SET [T}_1,\ [\![\bar{\mu}(t_1)]\!]\texttt{]}_{\Psi}\ \texttt{TO } [\![\mathcal{P}(t_1 \triangleright \texttt{T}_3,\ \texttt{T}_4)]\!]; \\
&\qquad\qquad\ \texttt{RET (T}_4\texttt{)}]\!]) \\
&\texttt{|| do } (S_1,\ T_1,\ D_1,\ V_1,\ t_1,\ \tau_1) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad (S_2,\ T_2,\ D_2,\ V_2,\ t_2,\ \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}\ \ (S_1,\ T_1,\ D_1,\ I \triangleright e_2) \\
&\qquad \mathrm{require}\ (\mathrm{AT}(t_1)\ \ \wedge \mathrm{AT}(t_2)) \vee \\
&\qquad\qquad (\mathrm{PTR}(t_1) \wedge \mathrm{PTR}(t_2) \wedge ((\mathrm{unq}(\mathcal{B}(t_1)) \approx \mathrm{unq}(\mathcal{B}(t_2))) \vee \\
&\qquad\qquad\qquad\qquad ((\mathrm{OBJ}(\mathcal{B}(t_1)) \vee \mathrm{INC}(\mathcal{B}(t_1))) \wedge \mathrm{VT}(\mathcal{B}(t_2))) \vee \\
&\qquad\qquad\qquad\qquad ((\mathrm{OBJ}(\mathcal{B}(t_2)) \vee \mathrm{INC}(\mathcal{B}(t_2))) \wedge \mathrm{VT}(\mathcal{B}(t_1)))) \wedge \\
&\qquad\qquad\quad (\mathrm{tq}(\mathcal{B}(t_2)) \subseteq \mathrm{tq}(\mathcal{B}(t_1)))) \vee \\
&\qquad\qquad (\mathrm{PTR}(t_1) \wedge \mathrm{NULL}(S,\ T,\ D,\ I \triangleright e_2)) \\
&\qquad \mathrm{return}\ (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ t_1, \\
&\qquad\qquad [\![\texttt{LET T}_1 \texttt{ = } \tau_1,\ \texttt{T}_2 \texttt{ = } \tau_2; \\
&\qquad\qquad\ \texttt{LET T}_3 \texttt{ = RET } ([\![\phi(t_1)]\!]_{[\![\phi(t_2)]\!]}(\texttt{T}_2)); \\
&\qquad\qquad\ \texttt{LET () = SET [T}_1,\ [\![\bar{\mu}(t_1)]\!]\texttt{]}_{[\![\phi(t_1)]\!]}\ \texttt{TO T}_3; \\
&\qquad\qquad\ \texttt{RET (T}_3\texttt{)}]\!]) \\
&\texttt{|| do } (S_1,\ T_1,\ D_1,\ V_1,\ t_1,\ \tau_1) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad (S_2,\ T_2,\ D_2,\ V_2,\ t_2,\ \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}\ \ (S_1,\ T_1,\ D_1,\ I \triangleright e_2) \\
&\qquad \mathrm{require}\ (\mathrm{SU}(t_1) \wedge \mathrm{OBJ}(t_1) \wedge \mathrm{unq}(t_1) \approx \mathrm{unq}(t_2)) \\
&\qquad \mathrm{return}\ (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ t_1, \\
&\qquad\qquad [\![\texttt{LET T}_1 \texttt{ = } \tau_1,\ \texttt{T}_2 \texttt{ = } \tau_2; \\
&\qquad\qquad\ \texttt{LET () = } [\![\text{set } [\![t_1 \triangleright \texttt{T}_1]\!] \text{ to } [\![t_2 \triangleright \texttt{T}_2]\!]]\!]; \\
&\qquad\qquad\ \texttt{RET (T}_1\texttt{)}]\!])
\end{aligned}
$$

Intuitively, the construction "set $(t_1 \triangleright \alpha_1)$ to $(t_2 \triangleright \alpha_2)$" which appears in the above definition of structure and union assignments, duplicates the structure and content of

an entire address space region described by the envelope of t_2 from the location given by the atom α_2 into a similarly-structured region beginning at the address α_1 and moulded into the envelope $\bar{\xi}(t_1)$. Since the precise semantics of this construction are left unspecified by the C Standard and, at any rate, cannot be captured exactly within the regime of the generic Etude fragment, its implementation is characterised in this chapter only by the following Haskell type signature:

$$\text{set } [\![\cdot]\!] \text{ to } [\![\cdot]\!] :: (\mathit{type} \triangleright \mathit{atom}_\nu) \to (\mathit{type} \triangleright \mathit{atom}_\nu) \to \mathit{term}_\nu$$

On every Etude implementation, the term "set $(t_1 \triangleright \alpha_1)$ to $(t_2 \triangleright \alpha_2)$" is always guaranteed to reduce into a trivial "RET ()", in the context of some appropriately-updated object environment $\Delta /\!\!/ (\sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2)$, as long as that environment is well-formed, α_1 and α_2 represent meaningful object atoms of appropriate formats, the two address space regions have identical sizes and either do not overlap at all, or else the overlap is exact, t_1 depicts a recursively-unqualified type and, finally, t_2 does not mention the "**volatile**" type qualifier:

$$\begin{aligned}
&\text{SET} :: \forall\, \Lambda, \Delta, t_1, \sigma_1, t_2, \sigma_2 \Rightarrow \\
&\quad \text{WF}(\Lambda) \to \text{WF}[\![\Delta /\!\!/ (\sigma_1, \bar{\xi}(t_1), \sigma_2, \bar{\xi}(t_2))]\!] \to \text{WF}[\![\#\sigma_{1\,O(\phi(t_1))}]\!] \to \text{WF}[\![\#\sigma_{2\,O(\phi(t_2))}]\!] \to \\
&\quad [\![\mathcal{S}(t_1) = \mathcal{S}(t_2) \wedge (\sigma_1 + \mathcal{S}(t_1) \leq \sigma_2 \vee \sigma_1 \geq \sigma_2 + \mathcal{S}(t_2) \vee \sigma_1 = \sigma_2)]\!] \to \\
&\quad [\![\text{rtq}(t_1) = \varnothing \wedge \text{rtq}(t_2) \subseteq \{\mathsf{C}\}]\!] \to \\
&\quad [\![\Lambda, \Delta \triangleright \text{set } [\![t_1 \triangleright \#\sigma_{1\,O(\phi(t_1))}]\!] \text{ to } [\![t_2 \triangleright \#\sigma_{2\,O(\phi(t_2))}]\!]]\!] \\
&\qquad \equiv [\![\Lambda, \Delta /\!\!/ (\sigma_1, \bar{\xi}(t_1), \sigma_2, \bar{\xi}(t_2)) \triangleright \text{RET } ()]\!]
\end{aligned}$$

in which the notation "$\Delta /\!\!/ (\sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2)$" depicts the environment derived by "set":

$$[\![\cdot]\!] /\!\!/ [\![\cdot]\!] :: \mathit{o\text{-}env} \to (\mathit{integer}, \mathit{envelope}, \mathit{integer}, \mathit{envelope}) \to \mathit{o\text{-}env}$$

Similarly to all other object environment operations from Section 4.5, the minimal algebraic properties of this function on every Etude implementation are captured by the following pair of Haskell theorems:

$$\begin{aligned}
&\text{COPY}_\alpha :: \forall\, \Delta, \sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2, \sigma, \phi \Rightarrow \\
&\quad \text{WF}(\Delta) \to \text{WF}[\![\Delta /\!\!/ (\sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2)]\!] \to \\
&\quad [\![(n, \phi, \varnothing) \in_{\text{A}} \bar{\xi}_1 \wedge (\sigma_1 + n, \phi, \varnothing) \in_{\text{A}} \bar{\xi}(\Delta)]\!] \to \\
&\quad [\![(n, \phi, \{\mathsf{C}\}) \in_{\text{A}} \bar{\xi}_2 \wedge (\sigma_2 + n, \phi, \{\mathsf{C}\}) \in_{\text{A}} \bar{\xi}(\Delta)]\!] \to \\
&\quad [\![\bar{\alpha}(\Delta /\!\!/ (\sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2) \triangleright \sigma_1 + n, \phi)]\!] \equiv [\![\bar{\alpha}(\Delta /\!\!/ (\sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2) \triangleright \sigma_2 + n, \phi)]\!] \\
&\text{COPY}_\alpha :: \forall\, \Delta, \sigma_1, \bar{\xi}_2, \sigma_2, \bar{\xi}_2 \Rightarrow \\
&\quad \text{WF}(\Delta) \to \text{WF}[\![\Delta /\!\!/ (\sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2)]\!] \to \\
&\quad [\![\bar{\xi}_i(\Delta /\!\!/ (\sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2)) = \bar{\xi}_i(\Delta) \backslash (\sigma_1, \sigma_1 + \mathcal{S}(\bar{\xi}_1))]\!]
\end{aligned}$$

Such well-formed object update operations will never affect the contents of any non-overlapping memory-resident objects currently allocated in the same address space, or for that matter, any other properties of the original object environment:

$$\begin{aligned}
&\text{COPY}_{\bar{\alpha}} :: \forall\, \Delta, \sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2, \sigma_k, \phi_k, \bar{\mu}_k \Rightarrow \\
&\quad \text{WF}(\Delta) \to \text{WF}[\![\Delta /\!\!/ (\sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2)]\!] \to \\
&\quad [\![(\sigma_k, \phi_k, \bar{\mu}_k) \in_{\text{A}} \bar{\xi}(\Delta)]\!] \to \\
&\quad [\![\sigma_k + \mathcal{S}(\phi_k) \leq \sigma_1 \vee \sigma_k \geq \sigma_1 + \mathcal{S}(\bar{\xi}_1)]\!] \to \\
&\quad [\![\bar{\alpha}(\Delta /\!\!/ (\sigma_1, \bar{\xi}_1, \sigma_2, \bar{\xi}_2) \triangleright \sigma_k, \phi_k)]\!] \equiv [\![\bar{\alpha}(\Delta \triangleright \sigma_k, \phi_k)]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{COPY}_\text{E} :: \forall\ \Delta,\ \sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2 \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta /\!\!/ (\sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2)]\!] \rightarrow \\
&\qquad [\![\bar{\xi}(\Delta /\!\!/ (\sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2)) = \bar{\xi}(\Delta)]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{COPY}_\text{S} :: \forall\ \Delta,\ \sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2 \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta /\!\!/ (\sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2)]\!] \rightarrow \\
&\qquad [\![\bar{\psi}(\Delta /\!\!/ (\sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2)) = \bar{\psi}(\Delta)]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{COPY}_\text{X} :: \forall\ \Delta,\ \sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2,\ \bar{\xi} \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta /\!\!/ (\sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2)]\!] \rightarrow \text{WF}[\![\Delta / \bar{\xi}]\!] \rightarrow \\
&\qquad \text{WF}[\![(\Delta /\!\!/ (\sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2)) / \bar{\xi}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{COPY}_\text{N} :: \forall\ \Delta,\ \sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2,\ \bar{\xi} \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta /\!\!/ (\sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2)]\!] \rightarrow \text{WF}[\![\Delta / \bar{\xi}]\!] \rightarrow \\
&\qquad [\![\sigma_c(\Delta /\!\!/ (\sigma_1,\ \bar{\xi}_1,\ \sigma_2,\ \bar{\xi}_2) \triangleright \bar{\xi}) = \sigma_c(\Delta \triangleright \bar{\xi})]\!]
\end{aligned}$$

On the other hand, all *compound assignments* of the form "e_1 *op*= e_2" are essentially equivalent to the simple assignment "e_1 = e_1 *op* e_2", effectively replenishing the targeted l-value with the result of applying the binary operator *op* to its existing content and the value of the second operand e_2, except that e_1 is evaluated only once by every instance of such an operation.

In all assignments of the form "e_1 **%=** e_2", "e_1 **<<=** e_2", "e_1 **>>=** e_2", "e_1 **&=** e_2", "e_1 **^=** e_2" and "e_1 **|=** e_2", both the l-value operand e_1 and the value e_2 must have an integral type. A formal denotation of such constructs is depicted by the notation "$\mathcal{D}_{\text{IA}}(S,\ T,\ D,\ I,\ op \triangleright e_1,\ e_2)$", in which *op* is the corresponding Etude operator "$\cdot|\cdot_\phi$", "$\ll_\phi$", "$\gg_\phi$", "$\triangle_\phi$", "$\bar{\triangle}_\phi$" or "$\bar{\nabla}_\phi$". In Haskell, this notation is implemented as follows:

$$\begin{aligned}
&\mathcal{D}_{\text{IA}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\qquad (S,\ S,\ D,\ I,\ \textit{binary-op} \triangleright \textit{expression},\ \textit{expression}) \rightarrow M(S,\ S,\ D,\ V,\ \textit{type},\ \textit{term}_V)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{IA}}[\![S,\ T,\ D,\ I,\ op \triangleright e_1,\ e_2]\!] \\
&\qquad = \text{do } (S_1,\ T_1,\ D_1,\ V_1,\ t_1,\ \tau_1) \leftarrow \mathcal{D}_{\text{MLV}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad\qquad (S_2,\ T_2,\ D_2,\ V_2,\ t_2,\ \tau_2) \leftarrow \mathcal{D}_\text{V}\ \ (S_1,\ T_1,\ D_1,\ I \triangleright e_2) \\
&\qquad\qquad \text{require } (\text{BF}(t_1) \wedge \text{AT}(t_2)) \\
&\qquad\qquad \text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ t_1, \\
&\qquad\qquad\qquad [\![\text{LET T}_1 = \tau_1,\ \text{T}_2 = \tau_2; \\
&\qquad\qquad\qquad\ \text{LET T}_3 = \text{GET } [\text{T}_1,\ [\![\bar{\mu}(t_1)]\!]]_\Psi; \\
&\qquad\qquad\qquad\ \text{LET T}_4 = \text{RET } ([\![\mathcal{U}(t_1 \triangleright \text{T}_3)]\!]\ op_{[\![\phi(t_1)]\!]}\ ([\![\phi(t_1)]\!]_{[\![\phi(t_2)]\!]}(\text{T}_2))); \\
&\qquad\qquad\qquad\ \text{LET } () = \text{SET } [\text{T}_1,\ [\![\bar{\mu}(t_1)]\!]]_\Psi \text{ TO } [\![\mathcal{P}(t_1 \triangleright \text{T}_3,\ \text{T}_4)]\!]; \\
&\qquad\qquad\qquad\ \text{RET } (\text{T}_4)]\!]) \\
&\qquad || \text{ do } (S_1,\ T_1,\ D_1,\ V_1,\ t_1,\ \tau_1) \leftarrow \mathcal{D}_{\text{MLV}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad\qquad (S_2,\ T_2,\ D_2,\ V_2,\ t_2,\ \tau_2) \leftarrow \mathcal{D}_\text{V}\ \ (S_1,\ T_1,\ D_1,\ I \triangleright e_2) \\
&\qquad\qquad \text{require } (\text{INT}(t_1) \wedge \text{INT}(t_2)) \\
&\qquad\qquad \text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ t_1, \\
&\qquad\qquad\qquad [\![\text{LET T}_1 = \tau_1,\ \text{T}_2 = \tau_2; \\
&\qquad\qquad\qquad\ \text{LET T}_3 = \text{GET } [\text{T}_1,\ [\![\bar{\mu}(t_1)]\!]]_{[\![\phi(t_1)]\!]}; \\
&\qquad\qquad\qquad\ \text{LET T}_4 = \text{RET } (\text{T}_3\ op_{[\![\phi(t_1)]\!]}\ ([\![\phi(t_1)]\!]_{[\![\phi(t_2)]\!]}(\text{T}_2))); \\
&\qquad\qquad\qquad\ \text{LET } () = \text{SET } [\text{T}_1,\ [\![\bar{\mu}(t_1)]\!]]_{[\![\phi(t_1)]\!]} \text{ TO T}_4; \\
&\qquad\qquad\qquad\ \text{RET } (\text{T}_4)]\!])
\end{aligned}$$

Specifically, each of this compound assignment forms is mapped onto its Etude equivalent by the following set of Haskell definitions:

$$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{\%=}\ e_2]\!] = \mathcal{D}_{\mathrm{IA}}(S,\ T,\ D,\ I,\ [\![\cdot\%\cdot]\!] \triangleright e_1,\ e_2)$$
$$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{<<=}\ e_2]\!] = \mathcal{D}_{\mathrm{IA}}(S,\ T,\ D,\ I,\ [\![\ll]\!] \triangleright e_1,\ e_2)$$
$$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{>>=}\ e_2]\!] = \mathcal{D}_{\mathrm{IA}}(S,\ T,\ D,\ I,\ [\![\gg]\!] \triangleright e_1,\ e_2)$$
$$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{\&=}\ e_2]\!] = \mathcal{D}_{\mathrm{IA}}(S,\ T,\ D,\ I,\ [\![\triangle]\!] \triangleright e_1,\ e_2)$$
$$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{\^{}=}\ e_2]\!] = \mathcal{D}_{\mathrm{IA}}(S,\ T,\ D,\ I,\ [\![\bigtriangledown]\!] \triangleright e_1,\ e_2)$$
$$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{|=}\ e_2]\!] = \mathcal{D}_{\mathrm{IA}}(S,\ T,\ D,\ I,\ [\![\bigtriangledown]\!] \triangleright e_1,\ e_2)$$

Similarly, in the compound assignments "e_1 ***=** e_2" and "e_1 **/=** e_2", the two operands e_1 and e_2 may assume arbitrary arithmetic types, with the entire denotation derived from the corresponding Etude operator "$\times_\phi$" or "$\div_\phi$" by the following function $\mathcal{D}_{\mathrm{AA}}$:

$$\mathcal{D}_{\mathrm{AA}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow$$
$$(S,\ S,\ D,\ I,\ \textit{binary-op} \triangleright \textit{expression},\ \textit{expression}) \to M(S,\ S,\ D,\ V,\ \textit{type},\ \textit{term}_{\mathrm{V}})$$

$$\mathcal{D}_{\mathrm{AA}}[\![S,\ T,\ D,\ I,\ op \triangleright e_1,\ e_2]\!]$$

$$= \text{do } (S_1,\ T_1,\ D_1,\ V_1,\ t_1,\ \tau_1) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e_1)$$
$$(S_2,\ T_2,\ D_2,\ V_2,\ t_2,\ \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}\ (S_1,\ T_1,\ D_1,\ I \triangleright e_2)$$
$$\text{require } (\mathrm{BF}(t_1) \wedge \mathrm{AT}(t_2))$$
$$\text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ t_1,$$
$$[\![\texttt{LET T}_1 = \tau_1,\ \texttt{T}_2 = \tau_2;$$
$$\texttt{LET T}_3 = \texttt{GET [T}_1,\ [\![\bar{\mu}(t_1)]\!]]_\Psi;$$
$$\texttt{LET T}_4 = \texttt{RET } ([\![\mathcal{U}(t_1 \triangleright \texttt{T}_3)]\!]\ op_{[\![\phi(t_1)]\!]}\ ([\![\phi(t_1)]\!]_{[\![\phi(t_2)]\!]}(\texttt{T}_2)));$$
$$\texttt{LET ()} = \texttt{SET [T}_1,\ [\![\bar{\mu}(t_1)]\!]]_\Psi\ \texttt{TO}\ [\![\mathcal{P}(t_1 \triangleright \texttt{T}_3,\ \texttt{T}_4)]\!];$$
$$\texttt{RET (T}_4)]\!])$$

$$\|\ \text{do } (S_1,\ T_1,\ D_1,\ V_1,\ t_1,\ \tau_1) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e_1)$$
$$(S_2,\ T_2,\ D_2,\ V_2,\ t_2,\ \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}\ (S_1,\ T_1,\ D_1,\ I \triangleright e_2)$$
$$\text{require } (\mathrm{AT}(t_1) \wedge \mathrm{AT}(t_2))$$
$$\text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ t_1,$$
$$[\![\texttt{LET T}_1 = \tau_1,\ \texttt{T}_2 = \tau_2;$$
$$\texttt{LET T}_3 = \texttt{GET [T}_1,\ [\![\bar{\mu}(t_1)]\!]]_{[\![\phi(t_1)]\!]};$$
$$\texttt{LET T}_4 = \texttt{RET (T}_3\ op_{[\![\phi(t_1)]\!]}\ ([\![\phi(t_1)]\!]_{[\![\phi(t_2)]\!]}(\texttt{T}_2)));$$
$$\texttt{LET ()} = \texttt{SET [T}_1,\ [\![\bar{\mu}(t_1)]\!]]_{[\![\phi(t_1)]\!]}\ \texttt{TO T}_4;$$
$$\texttt{RET (T}_4)]\!])$$

so that the actual meanings of these assignments can be captured in Haskell using the following translations:

$$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{*=}\ e_2]\!] = \mathcal{D}_{\mathrm{AA}}(S,\ T,\ D,\ I,\ [\![\times]\!] \triangleright e_1,\ e_2)$$
$$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{/=}\ e_2]\!] = \mathcal{D}_{\mathrm{AA}}(S,\ T,\ D,\ I,\ [\![\div]\!] \triangleright e_1,\ e_2)$$

The final two compound assignment forms "e_1 **+=** e_2" and "e_1 **-=** e_2" also accept a pair of arithmetic operands, applying their corresponding Etude operators "$+_\phi$" or "$-_\phi$" in the constructed denotation. In addition, their l-value operand e_1 may also represent a pointer to an object type, in which case the value operand e_2 must have an integral type. As in the earlier additive expression forms, the value of e_2 is multiplied by the size of the pointer's referenced type, before adding it to or subtracting it from the value of the pointer e_1. In effect, given a pointer variable p which, at the beginning

of the operation, contains the address of the n-th element in some C array object, the compound assignment "p **+=** i" sets p to the location of the element $n + i$ in the same array. Conversely, "p **−=** i", sets p to the element $n - i$, which, in Etude, is achieved by adding $i \times -\mathcal{S}(\mathcal{B}(t))$ to the original p's value, assuming that t represents the type of the construction's l-value operand. In this work, the meaning of all such compound assignment forms is formalised by the following semantic translations:

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \texttt{ += } e_2]\!] \\
&\quad = \mathcal{D}_{\mathrm{AA}}(S, T, D, I, [\![+]\!] \triangleright e_1, e_2) \\
&\quad \| \text{ do } (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e_1) \\
&\qquad\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}\ (S_1, T_1, D_1, I \triangleright e_2) \\
&\qquad\quad \text{require } (\mathrm{PTR}(t_1) \wedge \mathrm{OBJ}(\mathcal{B}(t_1)) \wedge \mathrm{INT}(t_2)) \\
&\qquad\quad \text{return } (S_2, T_2, D_2, V_1 \cup V_2, t_1, \\
&\qquad\qquad\qquad [\![\texttt{LET T}_1 \texttt{ = } \tau_1, \texttt{ T}_2 \texttt{ = } \tau_2; \\
&\qquad\qquad\qquad\ \texttt{LET T}_3 \texttt{ = GET [T}_1, [\![\bar{\mu}(t_1)]\!]]_{[\![\phi(t_1)]\!]}; \\
&\qquad\qquad\qquad\ \texttt{LET T}_4 \texttt{ = RET (T}_3 +_{[\![\phi(t_1)]\!]} ((\mathrm{Z}.\Phi_{[\![\phi(t_2)]\!]}(\mathrm{T}_2)) \times_{\mathrm{Z}.\Phi} \#[\![\mathcal{S}(\mathcal{B}(t_1))]\!]_{\mathrm{Z}.\Phi})); \\
&\qquad\qquad\qquad\ \texttt{LET () = SET [T}_1, [\![\bar{\mu}(t_1)]\!]]_{[\![\phi(t_1)]\!]} \texttt{ TO T}_4; \\
&\qquad\qquad\qquad\ \texttt{RET (T}_4\texttt{)}]\!])
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e_1 \texttt{ -= } e_2]\!] \\
&\quad = \mathcal{D}_{\mathrm{AA}}(S, T, D, I, [\![-]\!] \triangleright e_1, e_2) \\
&\quad \| \text{ do } (S_1, T_1, D_1, V_1, t_1, \tau_1) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e_1) \\
&\qquad\quad (S_2, T_2, D_2, V_2, t_2, \tau_2) \leftarrow \mathcal{D}_{\mathrm{V}}\ (S_1, T_1, D_1, I \triangleright e_2) \\
&\qquad\quad \text{require } (\mathrm{PTR}(t_1) \wedge \mathrm{OBJ}(\mathcal{B}(t_1)) \wedge \mathrm{INT}(t_2)) \\
&\qquad\quad \text{return } (S_2, T_2, D_2, V_1 \cup V_2, t_1, \\
&\qquad\qquad\qquad [\![\texttt{LET T}_1 \texttt{ = } \tau_1, \texttt{ T}_2 \texttt{ = } \tau_2; \\
&\qquad\qquad\qquad\ \texttt{LET T}_3 \texttt{ = GET [T}_1, [\![\bar{\mu}(t_1)]\!]]_{[\![\phi(t_1)]\!]}; \\
&\qquad\qquad\qquad\ \texttt{LET T}_4 \texttt{ = RET (T}_3 +_{[\![\phi(t_1)]\!]} ((\mathrm{Z}.\Phi_{[\![\phi(t_2)]\!]}(\mathrm{T}_2)) \times_{\mathrm{Z}.\Phi} \#[\![-\mathcal{S}(\mathcal{B}(t_1))]\!]_{\mathrm{Z}.\Phi})); \\
&\qquad\qquad\qquad\ \texttt{LET () = SET [T}_1, [\![\bar{\mu}(t_1)]\!]]_{[\![\phi(t_1)]\!]} \texttt{ TO T}_4; \\
&\qquad\qquad\qquad\ \texttt{RET (T}_4\texttt{)}]\!])
\end{aligned}
$$

In C, the four expression forms "**++**e", "e**++**", "**−−**e" and "e**−−**" represent specialised cases of compound assignments used to increase or decrease the value of a modifiable object e by the integer constant "**1**". In particular, the two prefix variants of these expressions with a concrete syntax of "**++**e" and "**−−**e" are equivalent precisely to the corresponding compound assignments "e **+= 1**" and "e **−= 1**":

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright \texttt{++}e]\!] = \mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e \texttt{ += 1}]\!] \\
&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright \texttt{--}e]\!] = \mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e \texttt{ -= 1}]\!]
\end{aligned}
$$

On the other hand, their postfix variants differ from "e **+= 1**" and "e **−= 1**" in that the original, rather than the updated value of e is returned by each of these operation. As a result, the formal denotations of these expressions differ from those of the earlier compound assignment forms only in the choice of Etude variable returned as the result of the entire operation. Like "e **+= 1**" and "e **−= 1**", "e**++**" and "e**−−**" accept arbitrary modifiable l-value operands of a bit-field, arithmetic or pointer type, so that their denotations must be translated into Etude as follows:

$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e\texttt{++}]\!]$

$$
\begin{aligned}
&= \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\ \tau_e) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e)\\
&\quad \text{require } (\mathrm{BF}(t_e))\\
&\quad \text{return } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\\
&\qquad [\![\texttt{LET T}_1 = \tau_e;\\
&\qquad\ \texttt{LET T}_2 = \texttt{GET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{\Psi};\\
&\qquad\ \texttt{LET T}_3 = \texttt{RET } ([\![\mathcal{U}(t_e \triangleright \texttt{T}_2)]\!]);\\
&\qquad\ \texttt{LET } () = \texttt{SET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{\Psi} \texttt{ TO } [\![\mathcal{P}(t_e \triangleright \texttt{T}_2,\ [\![\texttt{T}_3 +_{[\![\phi(t_e)]\!]} \#1_{[\![\phi(t_e)]\!]}]\!])]\!];\\
&\qquad\ \texttt{RET } (\texttt{T}_3)]\!])\\
&\quad \|\ \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\ \tau_e) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e)\\
&\quad\ \ \text{require } (\mathrm{AT}(t_e))\\
&\quad\ \ \text{return } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\\
&\qquad [\![\texttt{LET T}_1 = \tau_e;\\
&\qquad\ \texttt{LET T}_2 = \texttt{GET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{[\![\phi(t_e)]\!]};\\
&\qquad\ \texttt{LET } () = \texttt{SET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{[\![\phi(t_e)]\!]} \texttt{ TO } (\texttt{T}_2 +_{[\![\phi(t_e)]\!]} \#1_{[\![\phi(t_e)]\!]});\\
&\qquad\ \texttt{RET } (\texttt{T}_2)]\!])\\
&\quad \|\ \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\ \tau_e) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e)\\
&\quad\ \ \text{require } (\mathrm{PTR}(t_e) \wedge \mathrm{OBJ}(\mathcal{B}(t_e)))\\
&\quad\ \ \text{return } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\\
&\qquad [\![\texttt{LET T}_1 = \tau_e;\\
&\qquad\ \texttt{LET T}_2 = \texttt{GET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{[\![\phi(t_e)]\!]};\\
&\qquad\ \texttt{LET } () = \texttt{SET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{[\![\phi(t_e)]\!]} \texttt{ TO } (\texttt{T}_2 +_{[\![\phi(t_e)]\!]} \#[\![\mathcal{S}(\mathcal{B}(t_e))]\!]_{\mathrm{Z}.\Phi});\\
&\qquad\ \texttt{RET } (\texttt{T}_2)]\!])
\end{aligned}
$$

$\mathcal{D}_{\mathrm{V}}[\![S,\ T,\ D,\ I \triangleright e\texttt{--}]\!]$

$$
\begin{aligned}
&= \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\ \tau_e) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e)\\
&\quad \text{require } (\mathrm{BF}(t_e))\\
&\quad \text{return } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\\
&\qquad [\![\texttt{LET T}_1 = \tau_e;\\
&\qquad\ \texttt{LET T}_2 = \texttt{GET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{\Psi};\\
&\qquad\ \texttt{LET T}_3 = \texttt{RET } ([\![\mathcal{U}(t_e \triangleright \texttt{T}_2)]\!]);\\
&\qquad\ \texttt{LET } () = \texttt{SET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{\Psi} \texttt{ TO } [\![\mathcal{P}(t_e \triangleright \texttt{T}_2,\ [\![\texttt{T}_3 -_{[\![\phi(t_e)]\!]} \#1_{[\![\phi(t_e)]\!]}]\!])]\!];\\
&\qquad\ \texttt{RET } (\texttt{T}_3)]\!])\\
&\quad \|\ \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\ \tau_e) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e)\\
&\quad\ \ \text{require } (\mathrm{AT}(t_e))\\
&\quad\ \ \text{return } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\\
&\qquad [\![\texttt{LET T}_1 = \tau_e;\\
&\qquad\ \texttt{LET T}_2 = \texttt{GET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{[\![\phi(t_e)]\!]};\\
&\qquad\ \texttt{LET } () = \texttt{SET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{[\![\phi(t_e)]\!]} \texttt{ TO } (\texttt{T}_2 -_{[\![\phi(t_e)]\!]} \#1_{[\![\phi(t_e)]\!]});\\
&\qquad\ \texttt{RET } (\texttt{T}_2)]\!])\\
&\quad \|\ \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\ \tau_e) \leftarrow \mathcal{D}_{\mathrm{MLV}}(S,\ T,\ D,\ I \triangleright e)\\
&\quad\ \ \text{require } (\mathrm{PTR}(t_e) \wedge \mathrm{OBJ}(\mathcal{B}(t_e)))\\
&\quad\ \ \text{return } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\\
&\qquad [\![\texttt{LET T}_1 = \tau_e;\\
&\qquad\ \texttt{LET T}_2 = \texttt{GET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{[\![\phi(t_e)]\!]};\\
&\qquad\ \texttt{LET } () = \texttt{SET } [\texttt{T}_1,\ [\![\bar{\mu}(t_e)]\!]]_{[\![\phi(t_e)]\!]} \texttt{ TO } (\texttt{T}_2 +_{[\![\phi(t_e)]\!]} \#[\![-\mathcal{S}(\mathcal{B}(t_e))]\!]_{\mathrm{Z}.\Phi});\\
&\qquad\ \texttt{RET } (\texttt{T}_2)]\!])
\end{aligned}
$$

Last but not least, a value constructed from a function designator denotes the same term as that designator itself, while denotations of values obtained from an implicit conversion of an l-value with an object or an array type represent the actual content of the corresponding C object, as obtained by an application of the $\mathcal{V}$ combinator to the l-value's ordinary meaning. In all cases, the result has the pointer-promoted type of the original expression:

$$\begin{aligned}&\mathcal{D}_{\mathrm{V}}[\![S, T, D, I \triangleright e]\!] = \mathrm{do}\\&\quad (S_e, T_e, D_e, V_e, t_e, \tau_e) \leftarrow \mathcal{D}_{\mathrm{FD}}(S, T, D, I \triangleright e) \mid\mid \mathcal{D}_{\mathrm{LV}}(S, T, D, I \triangleright e)\\&\quad \mathrm{require}\ (\mathrm{FUN}(t_e) \vee \mathrm{OBJ}(t_e) \vee \mathrm{ARR}(t_e))\\&\quad \mathrm{return}\ (S_e, T_e, D_e, V_e, \mathrm{tq}(t_e) \cup \mathrm{pp}(t_e), \mathcal{V}(t_e \triangleright \tau_e))\end{aligned}$$

The retainment of type qualifiers by values obtained from such a conversion represents a small deviation from the strict letter of the C Standard, required for a proper formalisation of qualified structure and union types in the Etude framework. However, since these qualifiers are always ignored in all other semantic analysis of structure and union values, this abnormality does not impact the accuracy of our formalisation and, for all intents and purposes, may be safely ignored by more casual readers.

5.7.1.4 Void Expressions

In the C programming language, the expression syntax is also used to describe a rather different kind of entities that are generally known as *void expressions*. Intuitively, such constructs are only ever evaluated for their side effects, so that their type is always equal to a qualified or unqualified version of "**`void`**" and their denotations represent monadic Etude terms that deliver an empty list of results "`RET ()`". Formally, the meaning of every well-formed void expression is depicted by the following Haskell derivation:

$$\mathcal{D}_{\mathrm{VE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I \triangleright \mathit{expression}) \rightarrow M(S, S, D, V, \mathit{term}_{\mathrm{V}})$$

Most commonly, void expressions arise from calls to a C function with a "**`void`**" returned type. In particular, if "e **(**$\mathit{ael}_{\mathit{opt}}$**)**" represents a well-formed function call operation, in which the function designator e has one such type, then the operation is performed identically to its earlier value expression variant, except that no temporary variable is ever allocated for the function's returned value and that the derived Etude term does not deliver any such values to the surrounding program. In particular, the operand e is first evaluated together with any operands found in the argument expression list $\mathit{ael}_{\mathit{opt}}$, with values of all such arguments placed in appropriate temporary objects, whose complete list is then applied to the targeted Etude function. In Haskell, this translation can be expressed concisely as follows:

$$\begin{aligned}&\mathcal{D}_{\mathrm{VE}}[\![S, T, D, I \triangleright e\ \mathbf{(}\mathit{ael}_{\mathit{opt}}\mathbf{)}]\!] = \mathrm{do}\\&\quad (S_e, T_e, D_e, V_e, t_e, \tau_e) \leftarrow \mathcal{D}_{\mathrm{V}}\ \ (S,\ T,\ D,\ I \triangleright e)\\&\quad \mathrm{require}\ (\mathrm{PTR}(t_e) \wedge \mathrm{FUN}(\mathcal{B}(t_e)) \wedge \mathrm{VT}(\mathcal{B}(\mathcal{B}(t_e))))\\&\quad (S_a, T_a, D_a, V_a, W_a, \bar{\beta}) \leftarrow \mathcal{D}_{\mathrm{AEL}}(S_e, T_e, D_e, I, \mathrm{prot}(\mathcal{B}(t_e)) \triangleright \mathit{ael}_{\mathit{opt}})\\&\quad \mathrm{return}\ (S_a, T_a, D_a, V_e \cup V_a \cup W_a, [\![\mathtt{LET}\ [\![[\![\mathrm{T}_1 = \tau_e]\!] +\!\!+ \bar{\beta}]\!];\ \mathrm{T}_1([\![\mathrm{dom}(W_a)]\!])]\!])\end{aligned}$$

Further, an arbitrary C expression can be converted into the "**`void`**" type using an explicit cast operation, in which case the operand's meaning is always constructed by a separate function $\mathcal{D}_{\mathrm{E}}$ that is described later in the current section:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{VE}}[\![S,\ T,\ D,\ I \triangleright \texttt{(}\textit{type-name}\texttt{)}\,e]\!] = \text{do}\\
&\quad (S_t,\ T_t,\ D_t,\ t_t) \quad \leftarrow \mathcal{D}_{\mathrm{TN}}(S,\ T,\ D,\ I \triangleright \textit{type-name})\\
&\quad (S_e,\ T_e,\ D_e,\ V_e,\ \tau_e) \leftarrow \mathcal{D}_{\mathrm{E}}\ (S_t,\ T_t,\ D_t,\ I \triangleright e)\\
&\quad \text{require } (\mathrm{VT}(t_t))\\
&\quad \text{return } (S_e,\ T_e,\ D_e,\ V_e,\ \tau_e)
\end{aligned}$$

Void expressions may also assume the conditional or sequence syntax "e_1 `?` e_2 `:` e_3" and "e_1`,`e_2", provided that their e_2 and e_3 operands represent void expressions, as stipulated by the following pair of Haskell definitions:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{VE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3]\!] = \text{do}\\
&\quad (S_1,\ T_1,\ D_1,\ V_1,\ t_1,\ \tau_1) \leftarrow \mathcal{D}_{\mathrm{V}}\ (S,\ \ T,\ \ D,\ \ I \triangleright e_1)\\
&\quad (S_2,\ T_2,\ D_2,\ V_2,\ \tau_2) \quad \leftarrow \mathcal{D}_{\mathrm{VE}}(S_1,\ T_1,\ D_1,\ I \triangleright e_2)\\
&\quad (S_3,\ T_3,\ D_3,\ V_3,\ \tau_3) \quad \leftarrow \mathcal{D}_{\mathrm{VE}}(S_2,\ T_2,\ D_2,\ I \triangleright e_3)\\
&\quad \text{require } (\mathrm{SCR}(t_1))\\
&\quad \text{return } (S_3,\ T_3,\ D_3,\ V_1 \cup V_2 \cup V_3,\ [\![\mathsf{LET}\ \mathsf{T}_1 = \tau_1;\ \mathsf{IF}\ \mathsf{T}_1 \neq_{[\![\phi(t_1)]\!]} \#0_{[\![\phi(t_1)]\!]}\ \mathsf{THEN}\ \tau_2\ \mathsf{ELSE}\ \tau_3]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\mathrm{VE}}[\![S,\ T,\ D,\ I \triangleright e_1\texttt{,}e_2]\!] = \text{do}\\
&\quad (S_1,\ T_1,\ D_1,\ V_1,\ \tau_1) \quad \leftarrow \mathcal{D}_{\mathrm{E}}\ (S,\ \ T,\ \ D,\ \ I \triangleright e_1)\\
&\quad (S_2,\ T_2,\ D_2,\ V_2,\ \tau_2) \quad \leftarrow \mathcal{D}_{\mathrm{VE}}(S_1,\ T_1,\ D_1,\ I \triangleright e_2)\\
&\quad \text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ [\![\mathsf{LET}\ () = \tau_1;\ \tau_2]\!])
\end{aligned}$$

Last but not least, every parenthesised version of a void expression has the same denotation as its constituent, with no other semantic form ever permitted in such constructs:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{VE}}[\![S,\ T,\ D,\ I \triangleright \texttt{(}e\texttt{)}\,]\!] \ = \mathcal{D}_{\mathrm{VE}}(S,\ T,\ D,\ I \triangleright e)\\
&\mathcal{D}_{\mathrm{VE}}[\![S,\ T,\ D,\ I \triangleright \textit{other}]\!] = \text{reject}
\end{aligned}$$

In many contexts, a void expression can be obtained by an implicit conversion of an arbitrary well-formed function designator, l-value or value expression, in which case the expression's result is assigned to a temporary variable, whose value is otherwise ignored by the translated program. In this work, denotations of such implicitly-void expressions are formalised by the above-mentioned Haskell function $\mathcal{D}_{\mathrm{E}}$:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{E}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I \triangleright \textit{expression}_{opt}) \rightarrow M(S,\ S,\ D,\ V,\ \textit{term}_{\mathrm{V}})\\
&\mathcal{D}_{\mathrm{E}}[\![S,\ T,\ D,\ I \triangleright e_{opt}]\!]\\
&\quad = \text{do require } (e_{opt} = \epsilon)\\
&\qquad\quad \text{return } (S,\ T,\ D,\ \varnothing,\ [\![\mathsf{RET}\ ()]\!])\\
&\quad \mid\mid \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ \tau_e) \leftarrow \mathcal{D}_{\mathrm{VE}}(S,\ T,\ D,\ I \triangleright e_{opt})\\
&\qquad\quad \text{return } (S_e,\ T_e,\ D_e,\ V_e,\ \tau_e)\\
&\quad \mid\mid \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\ \tau_e)\\
&\qquad\quad \leftarrow \mathcal{D}_{\mathrm{FD}}(S,\ T,\ D,\ I \triangleright e_{opt}) \mid\mid \mathcal{D}_{\mathrm{LV}}(S,\ T,\ D,\ I \triangleright e_{opt}) \mid\mid \mathcal{D}_{\mathrm{V}}(S,\ T,\ D,\ I \triangleright e_{opt})\\
&\qquad\quad \text{return } (S_e,\ T_e,\ D_e,\ V_e,\ [\![\mathsf{LET}\ \mathsf{T}_1 = \tau_e;\ \mathsf{RET}\ ()]\!])
\end{aligned}$$

This construction is particularly useful for a concise formalisation of the "`,`" and void cast operations described earlier in this section, as well as *expression statements* discussed in Section 5.9. In the later context, the void expression operand may be omitted

entirely, resulting in the *null statement* construct "**;**" which, by the above translation, is taken to denote the trivial monadic term "RET ()".

5.7.2 Function Arguments

In every function call operation of the form "e **(**ael_{opt}**)**", the optional operand *ael* represents a comma-separated list of assignment expressions which depict the individual argument values supplied by the program to the function designated by the expression operand e. The concrete syntax of these lists is depicted in Haskell as follows:

argument-expression-list:
assignment-expression
argument-expression-list (*assignment-expression*)

It is convenient to view such constructs simply as sequences of zero or more expression entities, which can be easily derived from their concrete syntax by the following recursive process:

$\text{list}[\![\cdot]\!] :: \textit{argument-expression-list}_{opt} \to [\textit{expression}]$

$\text{list}[\![\textit{ael}\ \texttt{,}\ e]\!] = \text{list}(\textit{ael}) +\!\!+ [e]$
$\text{list}[\![e]\!] = [e]$
$\text{list}[\![]\!] = \varnothing$

At the beginning of every function call operation, the value of each argument is converted into the pointer-promoted version of the corresponding prototype entry, or else into the argument-promoted version of the expression's own type if no explicit type information is provided for the argument by the underlying function designator. In both cases, this converted value is stored into a temporary memory-resident object of an appropriate C type. Formally, the denotation of every such argument expression e is represented by the construct "$\mathcal{D}_{\text{AE}}(S, T, D, I, t_{opt} \triangleright e)$", in which t_{opt} represents either the pointer-promoted version of the corresponding prototype entry, or the omitted type "ϵ" if no such entry exists in the prototype:

$\mathcal{D}_{\text{AE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I, \textit{type}_{opt} \triangleright \textit{expression}) \to M(S, S, D, V, V, \textit{term}_{\text{V}})$

observing that all such denotations include, besides the usual S, T, D, V and τ components common to ordinary C expressions, an additional set of variable bindings W, which, intuitively, specifies the actual set of temporary Etude objects allocated for the function's argument values.

The constraints on such argument expressions are identical to those described earlier for simple assignment operations, except that no particular restrictions are imposed on the l-value's type qualification, so that the actual modification of the temporary argument object must be performed using Etude's initialisation operator "SET_{I}" instead of the earlier "SET" term form applied in Section 5.7.1.3. For the same reason, values of structure and union arguments are initialised using a separate implementation-defined construct "$\text{set}_{\text{I}}\ (t_1 \triangleright \alpha_1)$ to $(t_2 \triangleright \alpha_2)$", which is essentially equivalent to the earlier "set $(t_1 \triangleright \alpha_1)$ to $(t_2 \triangleright \alpha_2)$" combinator, except that the "set_{I}" form is also guaranteed to be defined for constant-qualified types t, provided that the corresponding

memory region also appears in the environment's initialiser envelope $\bar{\xi}_i(\Delta)$. Formally, the type signature of "set$_\text{I}$" is introduced as follows:

$$\text{set}_\text{I}\ [\![\cdot]\!]\ \text{to}\ [\![\cdot]\!] :: (\mathit{type} \triangleright \mathit{atom}_\nu) \rightarrow (\mathit{type} \triangleright \mathit{atom}_\nu) \rightarrow \mathit{term}_\nu$$

and its minimal semantics are described by the following theorem:

$$\begin{aligned}
&\text{SET}_\text{I} :: \forall\ \Lambda, \Delta, t_1, \sigma_1, t_2, \sigma_2 \Rightarrow \\
&\quad \text{WF}(\Lambda) \rightarrow \text{WF}[\![\Delta /\!/ (\sigma_1, \bar{\xi}(t_1), \sigma_2, \bar{\xi}(t_2))]\!] \rightarrow \text{WF}[\![\#\sigma_{1\,\mathcal{O}(\phi(t_1))}]\!] \rightarrow \text{WF}[\![\#\sigma_{2\,\mathcal{O}(\phi(t_2))}]\!] \rightarrow \\
&\quad [\![\mathcal{S}(t_1) = \mathcal{S}(t_2) \wedge (\sigma_1 + \mathcal{S}(t_1) \leq \sigma_2 \vee \sigma_1 \geq \sigma_2 + \mathcal{S}(t_2) \vee \sigma_1 = \sigma_2)]\!] \rightarrow \\
&\quad [\![\text{rtq}(t_1) \subseteq \{\text{C}\} \wedge \text{rtq}(t_2) \subseteq \{\text{C}\} \wedge \bar{\xi}(t_1) \subseteq \bar{\xi}_i(\Delta)]\!] \rightarrow \\
&\quad [\![\Lambda, \Delta \triangleright \text{set}_\text{I}\ [\![t_1 \triangleright \#\sigma_{1\,\mathcal{O}(\phi(t_1))}]\!]\ \text{to}\ [\![t_2 \triangleright \#\sigma_{2\,\mathcal{O}(\phi(t_2))}]\!]]\!] \\
&\qquad \equiv [\![\Lambda, \Delta /\!/ (\sigma_1, \bar{\xi}(t_1), \sigma_2, \bar{\xi}(t_2)) \triangleright \text{RET}\ ()]\!]
\end{aligned}$$

Using this definition, the meaning of a single argument expression with an explicitly-specified C type can be formalised relatively easily as follows:

$$\begin{aligned}
&\mathcal{D}_{\text{AE}}[\![S, T, D, I, t \triangleright e]\!] \\
&\quad = \text{do}\ (S_e, T_e, D_e, V_e, t_e, \tau_e) \leftarrow \mathcal{D}_\text{V}(S, T, D, I \triangleright e) \\
&\qquad \text{require}\ (\text{AT}(t) \wedge \text{AT}(t_e)) \vee \\
&\qquad\qquad (\text{PTR}(t) \wedge \text{PTR}(t_e) \wedge ((\text{unq}(\mathcal{B}(t)) \approx \text{unq}(\mathcal{B}(t_e))) \vee \\
&\qquad\qquad\qquad ((\text{OBJ}(\mathcal{B}(t)) \vee \text{INC}(\mathcal{B}(t))) \wedge \text{VT}(\mathcal{B}(t_e))) \vee \\
&\qquad\qquad\qquad ((\text{OBJ}(\mathcal{B}(t_e)) \vee \text{INC}(\mathcal{B}(t_e))) \wedge \text{VT}(\mathcal{B}(t))))) \vee \\
&\qquad\qquad (\text{PTR}(t) \wedge \text{NULL}(S, T, D, I \triangleright e)) \\
&\qquad \text{return}\ (S_e, T_e, D_e \cup \{\nu(D_e){:}\epsilon\}, V_e, \{\nu(D_e){:}\bar{\xi}(t)\}, \\
&\qquad\qquad [\![\text{LET}\ \text{T}_1 = \tau_e; \text{SET}_\text{I}\ [[\![\nu(D_e)]\!], \varnothing]_{[\![\phi(t)]\!]}\ \text{TO}\ ([\![\phi(t)]\!]_{[\![\phi(t_e)]\!]}(\text{T}_1))]\!]) \\
&\quad \|\ \text{do}\ (S_e, T_e, D_e, V_e, t_e, \tau_e) \leftarrow \mathcal{D}_\text{V}(S, T, D, I \triangleright e) \\
&\qquad \text{require}\ (\text{SU}(t) \wedge \text{OBJ}(t) \wedge \text{unq}(t) \approx \text{unq}(t_e)) \\
&\qquad \text{return}\ (S_e, T_e, D_e \cup \{\nu(D_e){:}\epsilon\}, V_e, \{\nu(D_e){:}\bar{\xi}(t)\}, \\
&\qquad\qquad [\![\text{LET}\ \text{T}_1 = \tau_e; [\![\text{set}_\text{I}\ (t \triangleright \nu(D_e))\ \text{to}\ [\![t_e \triangleright \text{T}_1]\!]]\!]]\!])
\end{aligned}$$

Otherwise, if the function prototype has provided us with no explicit type information about a given argument, then the resulting temporary l-value has the argument-promoted version of the expression's own type and, in all other respects, its denotation is derived identically to the meaning of a similar explicitly-typed argument assignment:

$$\begin{aligned}
&\mathcal{D}_{\text{AE}}[\![S, T, D, I, \epsilon \triangleright e]\!] \\
&\quad = \text{do}\ (S_e, T_e, D_e, V_e, t_e, \tau_e) \leftarrow \mathcal{D}_\text{V}(S, T, D, I \triangleright e) \\
&\qquad \text{require}\ (\text{SCR}(\text{ap}(t_e))) \\
&\qquad \text{return}\ (S_e, T_e, D_e \cup \{\nu(D_e){:}\epsilon\}, V_e, \{\nu(D_e){:}\bar{\xi}(\text{ap}(t_e))\}, \\
&\qquad\qquad [\![\text{LET}\ \text{T}_1 = \tau_e; \text{SET}_\text{I}\ [[\![\nu(D_e)]\!], \varnothing]_{[\![\phi(\text{ap}(t_e))]\!]}\ \text{TO}\ ([\![\phi(\text{ap}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\text{T}_1))]\!]) \\
&\quad \|\ \text{do}\ (S_e, T_e, D_e, V_e, t_e, \tau_e) \leftarrow \mathcal{D}_\text{V}(S, T, D, I \triangleright e) \\
&\qquad \text{require}\ (\text{SU}(\text{ap}(t_e)) \wedge \text{OBJ}(\text{ap}(t_e))) \\
&\qquad \text{return}\ (S_e, T_e, D_e \cup \{\nu(D_e){:}\epsilon\}, V_e, \{\nu(D_e){:}\bar{\xi}(\text{ap}(t_e))\}, \\
&\qquad\qquad [\![\text{LET}\ \text{T}_1 = \tau_e; [\![\text{set}_\text{I}\ (\text{ap}(t_e) \triangleright \nu(D_e))\ \text{to}\ [\![t_e \triangleright \text{T}_1]\!]]\!]]\!])
\end{aligned}$$

The above derivation of individual argument assignments may be extended trivially to an entire list of argument expressions, whereby the meaning τ_e of every individual

argument is converted into an Etude binding of the form "() = τ_e" as follows:

$$\begin{aligned}
&\mathcal{D}_{\text{AES}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\quad (S,\ S,\ D,\ I,\ \mathit{types}_{opt} \triangleright [\mathit{expression}]) \rightarrow M(S,\ S,\ D,\ V,\ V,\ \mathit{bindings}_{\mathrm{v}}) \\
&\mathcal{D}_{\text{AES}}[\![S,\ T,\ D,\ I,\ \epsilon \triangleright \bar{e}]\!] \\
&\quad = \text{do require } (\text{length}(\bar{e}) = 0) \\
&\qquad\quad \text{return } (S,\ T,\ D,\ \varnothing,\ \varnothing,\ \varnothing) \\
&\quad \mid\mid \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ W_e,\ \tau_e\ ) \leftarrow \mathcal{D}_{\text{AE}}\ (S,\ \ T,\ \ D,\ \ I,\ \epsilon \triangleright \text{head}(\bar{e})) \\
&\qquad\quad (S_a,\ T_a,\ D_a,\ V_a,\ W_a,\ \bar{\beta}_a) \leftarrow \mathcal{D}_{\text{AES}}(S_e,\ T_e,\ D_e,\ I,\ \epsilon \triangleright \text{tail}(\bar{e})) \\
&\qquad\quad \text{return } (S_a,\ T_a,\ D_a,\ V_e \cup V_a,\ W_e \cup W_a,\ [\![() = \tau_e]\!] +\!\!+ \bar{\beta}_a) \\
&\mathcal{D}_{\text{AES}}[\![S,\ T,\ D,\ I,\ \bar{t} \triangleright \bar{e}]\!] \\
&\quad = \text{do require } (\text{length}(\bar{t}) = \text{length}(\bar{e}) = 0) \\
&\qquad\quad \text{return } (S,\ T,\ D,\ \varnothing,\ \varnothing,\ \varnothing) \\
&\quad \mid\mid \text{do } (S_e,\ T_e,\ D_e,\ V_e,\ W_e,\ \tau_e\ ) \leftarrow \mathcal{D}_{\text{AE}}\ (S,\ \ T,\ \ D,\ \ I,\ \text{pp}(\text{head}(\bar{t})) \triangleright \text{head}(\bar{e})) \\
&\qquad\quad (S_a,\ T_a,\ D_a,\ V_a,\ W_a,\ \bar{\beta}_a) \leftarrow \mathcal{D}_{\text{AES}}(S_e,\ T_e,\ D_e,\ I,\ \text{tail}(\bar{t}) \triangleright \text{tail}(\bar{e})) \\
&\qquad\quad \text{return } (S_a,\ T_a,\ D_a,\ V_e \cup V_a,\ W_e \cup W_a,\ [\![() = \tau_e]\!] +\!\!+ \bar{\beta}_a)
\end{aligned}$$

In the actual function call operation, the above definition is applied directly only if the targeted function either includes a prototype without "`...`" suffix, or else if it omits the prototype altogether from its C type. As a special case, if the prototype consists entirely of a single "**`void`**" type entry, then the corresponding argument expression list must be empty and its denotation is formed from an empty argument variable set W and a null binding list $\bar{\beta}$ as follows:

$$\begin{aligned}
&\mathcal{D}_{\text{AEL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\quad (S,\ S,\ D,\ I,\ \mathit{prototype}_{opt} \triangleright \mathit{argument\text{-}expression\text{-}list}_{opt}) \rightarrow \\
&\quad M(S,\ S,\ D,\ V,\ V,\ \mathit{bindings}_{\mathrm{v}}) \\
&\mathcal{D}_{\text{AEL}}[\![S,\ T,\ D,\ I,\ \bar{t} \triangleright \mathit{ael}_{opt}]\!] = \text{do require } (\text{length}(\bar{t}) = 1 \wedge \text{VT}(\text{head}(\bar{t})) \wedge \mathit{ael}_{opt} = \epsilon) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{return } (S,\ T,\ D,\ \varnothing,\ \varnothing,\ \varnothing) \\
&\qquad\qquad\qquad\qquad\qquad\quad \mid\mid \text{do } \mathcal{D}_{\text{AES}}(S,\ T,\ D,\ I,\ \bar{t} \triangleright \text{list}(\mathit{ael}_{opt})) \\
&\mathcal{D}_{\text{AEL}}[\![S,\ T,\ D,\ I,\ \epsilon \triangleright \mathit{ael}_{opt}]\!] = \text{do } \mathcal{D}_{\text{AES}}(S,\ T,\ D,\ I,\ \epsilon \triangleright \text{list}(\mathit{ael}_{opt}))
\end{aligned}$$

In presence of the prototype suffix "`...`", the construction is rather more interesting. If ℓ represents the number of C types explicitly included in such a prototype, then the first ℓ arguments of the function are said to be *fixed*, while any remaining entries in the list are said to represent *variable arguments* of the function call operation, whose precise number may fluctuate between individual invocations of that C function. In lambda calculus, such constructs are inherently inexpressible, since every lambda abstraction always specifies a predetermined set of variable bindings utilised within its body. Accordingly, this aspect of the C language is most naturally modelled in Etude by combining all variable arguments into a single structure-like object of an unspecified layout, which is always appended to the ordinary portion of the argument list whenever the underlying function prototype ends in the "`...`" suffix. Conveniently, this approach also enables a very natural implementation of the "va_list" type in the Standard C Library that, ultimately, must accompany our completed C compiler, together with the associated macros "va_start", "va_arg" and "va_end" required by the C Standard.

In particular, any fixed arguments corresponding to the explicitly-typed prototype entries represent ordinary argument assignments to temporary l-values of these types, denoting a predetermined set of argument objects W_1 and bindings $\bar{\beta}_1$, except that the special "***void***" case is not applicable in such constructs. Further, a parameter-less denotation of any remaining variable arguments is obtained in an absence of explicit typing, producing a second set of temporary l-values W_2 and corresponding assignment bindings $\bar{\beta}_2$. In the entire operation, these variable arguments are then combined into a single Etude object ν_v, whose structure is obtained from the union of their individual envelopes. Each variable argument is placed in ν_v at a predetermined but unspecified byte offset, as computed by the unspecified function $\mathcal{L}$ of the following Haskell type:

$$\mathcal{L}[\![\cdot]\!] :: V \rightarrow (\nu \mapsto \mathit{integer})$$

Intuitively, given a set of individual argument objects W_2 derived from the function call's variable argument list, $\mathcal{L}(W_2)$ maps every variable $\nu_k \in \mathrm{dom}(W_2)$ to an offset n_k into the combined variable argument object ν_v, so that, in the denotation of the entire operation, all of these temporary variables are replaced with a single variable ν_v, whose envelope represents the union of the individual envelopes $W_2(\nu_k) \oplus n_k$. Further, in the set of assignment bindings $\bar{\beta}_2$, every occurrence of such a variable ν_k is replaced by an atom of the form "$\nu_v +_{\mathrm{O}.\Phi} \#n_{k\mathrm{Z}.\Phi}$", so that all of these variables are effectively eliminated from the program and, accordingly removed from the resulting item definition set D. Formally, this rather complicated derivation is implemented in Haskell as follows:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{AEL}}[\![S,\, T,\, D,\, I,\, \bar{t}\ldots \triangleright \mathit{ael}_{opt}]\!] = \mathrm{do}\\
&\quad (S_1,\, T_1,\, D_1,\, V_1,\, W_1,\, \bar{\beta}_1) \leftarrow \mathcal{D}_{\mathrm{AES}}(S,\ \ T,\ \ D,\ \ I,\, \bar{t} \triangleright \mathrm{take}\,(\mathrm{length}(\bar{t}),\, \mathrm{list}(\mathit{ael}_{opt})))\\
&\quad (S_2,\, T_2,\, D_2,\, V_2,\, W_2,\, \bar{\beta}_2) \leftarrow \mathcal{D}_{\mathrm{AES}}(S_1,\, T_1,\, D_1,\, I,\, \epsilon \triangleright \mathrm{drop}(\mathrm{length}(\bar{t}),\, \mathrm{list}(\mathit{ael}_{opt})))\\
&\quad (\nu_v) \leftarrow \nu(D_2 \backslash \mathrm{dom}(W_2))\\
&\quad \mathrm{return}\ (S_2,\, T_2,\, D_2 \backslash \mathrm{dom}(W_2) \cup \{\nu_v{:}\epsilon\},\, V_1 \cup V_2,\\
&\qquad\qquad W_1 \cup \{\nu_v{:}\textstyle\bigcup[W_2(\nu_k) \oplus n_k \mid \nu_k{:}n_k \leftarrow \mathcal{L}(W_2)]\},\\
&\qquad\qquad \bar{\beta}_1 +\!\!+ (\bar{\beta}_2 / \{\nu_k{:}[\![\nu_v +_{\mathrm{O}.\Phi} \#n_{k\mathrm{Z}.\Phi}]\!] \mid \nu_k{:}n_k \leftarrow \mathcal{L}(W_2)\}))
\end{aligned}$$

Although it may be argued that the above formalisation of the C function call semantics represents an over-specification of the language, extensive experience with implementation of C compilers has shown the above approach (or minor variations thereof) to be the only effective means of implementing variable argument lists on modern instruction set architectures. This empirical evidence strongly suggests that the semantic of variable argument lists are fundamentally connected to the so-called *C calling convention* modelled above.

Nevertheless, the reader should keep in mind that, since the program equivalence relation defined by the semantics of Etude in Chapter 4 is based only the sequence of the system calls generated by programs, the above formalisation of function call operations could, in principle, be proven equivalent to any other sensible definition, permitting optimising C compilers to tailor the C calling convention to the specific needs of a particular program's structure.

5.7.3 Constant Expressions

A conditional expression whose result is always known at the time of translation is known as a *constant expression*. In the C standard, this additional use of the expression syntax is captured by the following trivial Haskell type definition:

constant-expression :
conditional-expression

Such expressions have a number of important applications within the C language. Their precise meanings, as well as any constraints on the syntax of such constructs, depends on the context in which a constant expression appears within the program. The following sections contain a detailed discussion of five different classes of constant expressions, known as *integral constant expressions*, *null pointer constants*, *static initialiser expressions*, *arithmetic constant expressions* and *address constants*. Each of these syntactic entities is described individually in Sections 5.7.3.1, 5.7.3.2, 5.7.3.3 and 5.7.3.4, respectively.

Regardless of the precise nature of a particular constant expression e, every such construct denotes an atomic value rather than a monadic Etude term. In certain contexts, this atomic value must be reduced further into a simple numeric quantity, as depicted by the following Haskell combinator:

$$\mathcal{V}_{[\![\cdot]\!]}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \mathit{format} \rightarrow \mathit{atom}_{\nu} \rightarrow M(\mathit{rational})$$

For every well-formed arithmetic constant expression e of an integral C type t, the atomic denotation α of the constant e is always equivalent to the constant atomic form "$\#[\![\mathcal{V}(\alpha)]\!]_{[\![\phi(t)]\!]}$". Formally:

$$\begin{aligned}
&\text{INT}_{\alpha} :: \forall\, n :: \mathit{integer},\, S,\, S',\, T,\, T',\, D,\, D',\, I,\, e,\, t,\, \alpha \Rightarrow \\
&\quad [\![\text{INT}(t) \wedge \text{glb}(t) \leq n \leq \text{lub}(t)]\!] \rightarrow \\
&\quad [\![\mathcal{D}_{\text{ACE}}(S,\, T,\, D,\, I \triangleright e)]\!] = [\![S',\, T',\, D',\, t,\, \alpha]\!] \rightarrow \\
&\quad [\![\alpha]\!] \equiv [\![\#n_{[\![\phi(t)]\!]}]\!] \rightarrow \\
&\quad [\![\mathcal{V}_{\phi(t)}(\alpha) = n]\!]
\end{aligned}$$

Further, if t represents a floating type and α is equivalent to some rational constant atom of the form "$\#[\![(-1)^s \times m \times \text{r}(\phi(t))^{e\,-\,\text{p}(\phi(t))}]\!]_{[\![\phi(t)]\!]}$", then $\mathcal{V}_{\phi(t)}(\alpha)$ differs from α's true arithmetic value by less than $\text{r}(\phi(t))^{e\,-\,\text{p}(\phi(t))}$ in magnitude. The C Standard imposes no other requirements on the reduction of arithmetic constants, so that, in principle, it may produce results more precise than those achievable during actual execution of the program on any given floating point hardware. In Haskell, these relaxed constraints on the definition of $\mathcal{V}_{\phi}$ can be described as follows:

$$\begin{aligned}
&\text{FLT}_{\alpha} :: \forall\, s,\, m,\, e :: \mathit{integer},\, S,\, S',\, T,\, T',\, D,\, D',\, I,\, e,\, t,\, \alpha \Rightarrow \\
&\quad \text{WF}(\alpha) \rightarrow \\
&\quad [\![\text{FLT}(t) \wedge 0 \leq s \leq 1 \wedge 0 \leq m < \text{r}(\phi(t))^{\text{p}(\phi(t))} \wedge \text{E}_{\min}(\phi(t)) \leq e \leq \text{E}_{\max}(\phi(t))]\!] \rightarrow \\
&\quad [\![\mathcal{D}_{\text{ACE}}(S,\, T,\, D,\, I \triangleright e)]\!] = [\![S',\, T',\, D',\, t,\, \alpha]\!] \rightarrow \\
&\quad [\![\alpha]\!] \equiv [\![\#[\![(-1)^s \times m \times \text{r}(\phi(t))^{e\,-\,\text{p}(\phi(t))}]\!]_{[\![\phi(t)]\!]}]\!] \rightarrow \\
&\quad [\![|(-1)^s \times m \times \text{r}(\phi(t))^{e\,-\,\text{p}(\phi(t))} - \mathcal{V}_{\phi(t)}(\alpha)| < \text{r}(\phi(t))^{e\,-\,\text{p}(\phi(t))}]\!]
\end{aligned}$$

A typical implementation of this function is presented separately in Appendix C.

5.7.3.1 Integral Constant Expressions

An *integral constant expression* represents a value expression of an integral type, whose precise result is always known at the time of translation. Such constructs are used extensively in Sections 5.8 and 5.9 to describe lengths of array types and "**`case`**" label values. Accordingly, all such expressions always denote predetermined integer quantities, obtained from reduction of the corresponding Etude atoms by the $\mathcal{V}_\phi$ combinator described above. Formally, their denotations are constructed by the following Haskell translation:

$$\mathcal{D}_{\text{ICE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I \triangleright \textit{expression}) \rightarrow M(S,\ S,\ D,\ \textit{type},\ \textit{integer})$$

Intuitively, these expressions must be always constructed entirely from integral constants, floating constants applied to a cast operation, "**`sizeof`**" operations, casts of integral constant expressions to another integral type, or else from well-formed applications of the C operators "**`+`**", "**`-`**", "**`~`**", "**`!`**", "**`*`**", "**`/`**", "**`%`**", "**`<<`**", "**`>>`**", "**`<`**", "**`>`**", "**`<=`**", "**`>=`**", "**`==`**", "**`!=`**", "**`&`**", "**`^`**", "**`|`**", "**`&&`**", "**`||`**" and "**`?:`**", whose operands must also constitute integral constant expressions and whose true arithmetic results must fall within the range of values representable under the operation's C type.

In particular, every well-formed integer, character and enumeration constant represents an integral constant expression, whose type and numeric value is derived directly from the constant's lexical syntax by the process discussed earlier in Section 5.6 as follows:

$$\begin{aligned}
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \textit{integer-constant}]\!] = \text{do}\\
&\quad (t,\ n) \leftarrow \mathcal{D}_{\text{IC}}(\textit{integer-constant})\\
&\quad \text{return } (S,\ T,\ D,\ t,\ n)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \textit{character-constant}]\!] = \text{do}\\
&\quad (t,\ n) \leftarrow \mathcal{D}_{\text{CC}}(\textit{character-constant})\\
&\quad \text{return } (S,\ T,\ D,\ t,\ n)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \textit{enumeration-constant}]\!] = \text{do}\\
&\quad \text{require } (\textit{enumeration-constant} \in \text{dom}(S) \wedge \mathcal{L}(d) = [\![\textbf{\texttt{const}}\ [\![n(\mathcal{L}(d))]\!]]\!])\\
&\quad \text{return } (S,\ T,\ D,\ \mathcal{T}(d),\ n(\mathcal{L}(d)))\\
&\text{where } d = S(\textit{enumeration-constant})
\end{aligned}$$

Further, every well-formed application of the "**`sizeof`**" operator constitutes an integral constant expression of the "***`size_t`***" type, whose value is equal to the size of the C type derived from the expression's l-value, value or type name operand:

$$\begin{aligned}
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \textbf{\texttt{sizeof}}\ e]\!] = \text{do}\\
&\quad (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\ \tau_e) \leftarrow \mathcal{D}_{\text{LV}}(S,\ T,\ D,\ I \triangleright e)\ ||\ \mathcal{D}_{\text{V}}(S,\ T,\ D,\ I \triangleright e)\\
&\quad \text{require } (\text{OBJ}(t_e))\\
&\quad \text{return } (S_e,\ T_e,\ D_e \backslash \text{dom}(V_e),\ [\![\textbf{\texttt{size_t}}]\!],\ \mathcal{S}(t_e))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \textbf{\texttt{sizeof (}}\textit{type-name}\textbf{\texttt{)}}]\!] = \text{do}\\
&\quad (S_t,\ T_t,\ D_t,\ t_t) \leftarrow \mathcal{D}_{\text{TN}}(S,\ T,\ D,\ I \triangleright \textit{type-name})\\
&\quad \text{require } (\text{OBJ}(t_t))\\
&\quad \text{return } (S_t,\ T_t,\ D_t,\ [\![\textbf{\texttt{size_t}}]\!],\ \mathcal{S}(t_t))
\end{aligned}$$

More so, every cast operation of the form "**(***type-name***)***e*" belongs to this semantic family, provided that the specified type name represents an integral type and e constitutes an integral constant expression or a floating constant. The value of such an expression is obtained from the corresponding atom "$\phi'_{\phi}(\#n_{\phi})$", in which ϕ and ϕ' represent Etude formats derived from the respective C types of the entity's expression and type name operands, while n depicts the integral constant denotation of e:

$$\begin{aligned}
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \texttt{(}\textit{type-name}\texttt{)}e]\!] = \text{do}\\
&\quad (S_t,\ T_t,\ D_t,\ t_t) \leftarrow \mathcal{D}_{\text{TN}}(S,\ T,\ D,\ I \triangleright \textit{type-name})\\
&\quad (S_e,\ T_e,\ D_e,\ t_e,\ n_e) \leftarrow \mathcal{D}_{\text{ICE}}(S_t,\ T_t,\ D_t,\ I \triangleright e)\ ||\ \mathcal{D}_{\text{FCE}}(S_t,\ T_t,\ D_t,\ I \triangleright e)\\
&\quad \text{require } (\text{INT}(t_t))\\
&\quad (n) \leftarrow \mathcal{V}_{\phi(t_t)}[\![[\![\phi(t_t)]\!]_{[\![\phi(t_e)]\!]}\ \#n_{e[\![\phi(t_e)]\!]}]\!]\\
&\quad \text{return } (S_e,\ T_e,\ D_e,\ t_t,\ n)
\end{aligned}$$

in which the notation "$\mathcal{D}_{\text{FCE}}(S,\ T,\ D,\ I \triangleright e)$" obtains the meaning of an integral constant expression formed from a floating constant as follows:

$$\begin{aligned}
&\mathcal{D}_{\text{FCE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I \triangleright \textit{expression}) \rightarrow M(S,\ S,\ D,\ \textit{type},\ \textit{integer})\\
&\mathcal{D}_{\text{FCE}}[\![S,\ T,\ D,\ I \triangleright \textit{floating-constant}]\!] = \text{do } (t,\ x) \leftarrow \mathcal{D}_{\text{FC}}(\textit{floating-constant})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{return } (S,\ T,\ D,\ t,\ [x])\\
&\mathcal{D}_{\text{FCE}}[\![S,\ T,\ D,\ I \triangleright \texttt{(}e\texttt{)}]\!] \qquad\qquad = \text{do } \mathcal{D}_{\text{FCE}}(S,\ T,\ D,\ I \triangleright e)\\
&\mathcal{D}_{\text{FCE}}[\![S,\ T,\ D,\ I \triangleright \textit{other}]\!] \qquad\qquad = \text{do reject}
\end{aligned}$$

Similarly, the denotations of all parenthesised versions of an integral constant expression and all applications of the unary "**+**", "**−**" and "**~**" operator to such an operand constitute integral constant expressions with the following natural denotations:

$$\begin{aligned}
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \texttt{(}e\texttt{)}]\!] = \text{do } \mathcal{D}_{\text{ICE}}(S,\ T,\ D,\ I \triangleright e)\\
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \texttt{+}e]\!] = \text{do } (S_e,\ T_e,\ D_e,\ t_e,\ n_e) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ T,\ D,\ I \triangleright e)\\
&\qquad\qquad\qquad\qquad\quad (n) \leftarrow \mathcal{V}_{\phi(\text{ip}(t_e))}[\![[\![\phi(\text{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\#n_{e[\![\phi(t_e)]\!]})]\!]\\
&\qquad\qquad\qquad\qquad\quad \text{return } (S_e,\ T_e,\ D_e,\ \text{ip}(t_e),\ n)\\
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \texttt{-}e]\!] = \text{do } (S_e,\ T_e,\ D_e,\ t_e,\ n_e) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ T,\ D,\ I \triangleright e)\\
&\qquad\qquad\qquad\qquad\quad (n) \leftarrow \mathcal{V}_{\phi(\text{ip}(t_e))}[\![-_{[\![\phi(t_e)]\!]}([\![\phi(\text{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\#n_{e[\![\phi(t_e)]\!]}))]\!]\\
&\qquad\qquad\qquad\qquad\quad \text{return } (S_e,\ T_e,\ D_e,\ \text{ip}(t_e),\ n)\\
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \texttt{\~{}}e]\!] = \text{do } (S_e,\ T_e,\ D_e,\ t_e,\ n_e) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ T,\ D,\ I \triangleright e)\\
&\qquad\qquad\qquad\qquad\quad (n) \leftarrow \mathcal{V}_{\phi(\text{ip}(t_e))}[\![\sim_{[\![\phi(t_e)]\!]}([\![\phi(\text{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\#n_{e[\![\phi(t_e)]\!]}))]\!]\\
&\qquad\qquad\qquad\qquad\quad \text{return } (S_e,\ T_e,\ D_e,\ \text{ip}(t_e),\ n)
\end{aligned}$$

In the same vein, every well-formed application of the "*****", "**/**", "**%**", "**+**", "**−**", "**<<**", "**>>**", "**&**", "**^**" and "**|**" operator to a pair of integral constant operands belongs to this semantic family. For conciseness, the denotations of all such applications are derived from the corresponding binary Etude operator by the following common construction:

$$\begin{aligned}
&\mathcal{D}_{\text{BICE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow\\
&\quad (S,\ S,\ D,\ I,\ \textit{binary-op} \triangleright \textit{expression},\ \textit{expression}) \rightarrow M(S,\ S,\ D,\ \textit{type},\ \textit{integer})\\
&\mathcal{D}_{\text{BICE}}[\![S,\ T,\ D,\ I,\ op \triangleright e_1,\ e_2]\!] = \text{do}\\
&\quad (S_1,\ T_1,\ D_1,\ t_1,\ n_1) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1)\\
&\quad (S_2,\ T_2,\ D_2,\ t_2,\ n_2) \leftarrow \mathcal{D}_{\text{ICE}}(S_1,\ T_1,\ D_1,\ I \triangleright e_2)\\
&\quad (n) \leftarrow \mathcal{V}_{\phi(t_1 \boxtimes t_2)}[\![[\![\phi(t_1 \boxtimes t_2)]\!]_{[\![\phi(t_1)]\!]}(\#n_{1[\![\phi(t_1)]\!]})\ op_{[\![\phi(t_1 \boxtimes t_2)]\!]}\ [\![\phi(t_1 \boxtimes t_2)]\!]_{[\![\phi(t_2)]\!]}(\#n_{2[\![\phi(t_2)]\!]})]\!]\\
&\quad \text{return } (S_2,\ T_2,\ D_2,\ t_1 \boxtimes t_2,\ n)
\end{aligned}$$

so that:

$$\begin{aligned}
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{*}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![\times]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{/}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![\div]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{\%}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![\cdot|\cdot]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{+}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![+]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{-}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![-]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{<<}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![\ll]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{>>}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![\gg]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{\&}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![\triangle]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{\^{}}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![\veebar]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{|}\ e_2]\!] &= \mathcal{D}_{\text{BICE}}(S,\ T,\ D,\ I,\ [\![\vee]\!] \triangleright e_1,\ e_2)
\end{aligned}$$

Similarly, integral constant denotations of the relational and equality operators "**<**", "**>**", "**<=**", "**>=**", "**==**" and "**!=**" are also derived from their Etude equivalents using the $\mathcal{V}$ combinator, with the result always assuming the plain "***int***" type:

$$\begin{aligned}
&\mathcal{D}_{\text{RICE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow\\
&\quad (S,\ S,\ D,\ I,\ \textit{binary-op} \triangleright \textit{expression},\ \textit{expression}) \rightarrow M(S,\ S,\ D,\ \textit{type},\ \textit{integer})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{RICE}}[\![S,\ T,\ D,\ I,\ op \triangleright e_1,\ e_2]\!] = \text{do}\\
&\quad (S_1,\ T_1,\ D_1,\ t_1,\ n_1) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1)\\
&\quad (S_2,\ T_2,\ D_2,\ t_2,\ n_2) \leftarrow \mathcal{D}_{\text{ICE}}(S_1,\ T_1,\ D_1,\ I \triangleright e_2)\\
&\quad (n) \leftarrow \mathcal{V}_{\phi(\texttt{int})}[\![[\![\phi(t_1 \mathbin{⊛} t_2)]\!]_{[\![\phi(t_1)]\!]}(\#n_{1[\![\phi(t_1)]\!]})\ op_{[\![\phi(t_1 \mathbin{⊛} t_2)]\!]}\ [\![\phi(t_1 \mathbin{⊛} t_2)]\!]_{[\![\phi(t_2)]\!]}(\#n_{2[\![\phi(t_2)]\!]})]\!]\\
&\quad \text{return } (S_2,\ T_2,\ D_2,\ [\![\texttt{int}]\!],\ n)
\end{aligned}$$

In particular:

$$\begin{aligned}
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{<}\ e_2]\!] &= \mathcal{D}_{\text{RICE}}(S,\ T,\ D,\ I,\ [\![<]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{>}\ e_2]\!] &= \mathcal{D}_{\text{RICE}}(S,\ T,\ D,\ I,\ [\![>]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{<=}\ e_2]\!] &= \mathcal{D}_{\text{RICE}}(S,\ T,\ D,\ I,\ [\![\leq]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{>=}\ e_2]\!] &= \mathcal{D}_{\text{RICE}}(S,\ T,\ D,\ I,\ [\![\geq]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{==}\ e_2]\!] &= \mathcal{D}_{\text{RICE}}(S,\ T,\ D,\ I,\ [\![=]\!] \triangleright e_1,\ e_2)\\
\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{!=}\ e_2]\!] &= \mathcal{D}_{\text{RICE}}(S,\ T,\ D,\ I,\ [\![\neq]\!] \triangleright e_1,\ e_2)
\end{aligned}$$

Finally, the remaining four logical and conditional forms of integral constant expressions obtained from applications of the C operators "**!**", "**&&**", "**||**" and "**?:**" are formalised as follows:

$$\begin{aligned}
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \texttt{!}e]\!] = \text{do } (S_e,\ T_e,\ D_e,\ t_e,\ n_e) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ \ T,\ \ D,\ \ I \triangleright e)\\
&\qquad\qquad \text{return } (S_e,\ T_e,\ D_e,\ [\![\texttt{int}]\!],\ n_e = 0)\\
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{\&\&}\ e_2]\!] = \text{do } (S_1,\ T_1,\ D_1,\ t_1,\ n_1) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1)\\
&\qquad\qquad (S_2,\ T_2,\ D_2,\ t_2,\ n_2) \leftarrow \mathcal{D}_{\text{ICE}}(S_1,\ T_1,\ D_1,\ I \triangleright e_2)\\
&\qquad\qquad \text{return } (S_2,\ T_2,\ D_2,\ [\![\texttt{int}]\!],\ n_1 \neq 0 \wedge n_2 \neq 0)\\
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{||}\ e_2]\!] = \text{do } (S_1,\ T_1,\ D_1,\ t_1,\ n_1) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1)\\
&\qquad\qquad (S_2,\ T_2,\ D_2,\ t_2,\ n_2) \leftarrow \mathcal{D}_{\text{ICE}}(S_1,\ T_1,\ D_1,\ I \triangleright e_2)\\
&\qquad\qquad \text{return } (S_2,\ T_2,\ D_2,\ [\![\texttt{int}]\!],\ n_1 \neq 0 \vee n_2 \neq 0)\\
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3]\!] = \text{do } (S_1,\ T_1,\ D_1,\ t_1,\ n_1) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1)\\
&\qquad\qquad (S_2,\ T_2,\ D_2,\ t_2,\ n_2) \leftarrow \mathcal{D}_{\text{ICE}}(S_1,\ T_1,\ D_1,\ I \triangleright e_2)\\
&\qquad\qquad (S_3,\ T_3,\ D_3,\ t_3,\ n_3) \leftarrow \mathcal{D}_{\text{ICE}}(S_2,\ T_2,\ D_2,\ I \triangleright e_3)\\
&\qquad\qquad \text{return } (S_3,\ T_3,\ D_3,\ t_2 \mathbin{⊛} t_3,\ \text{if } n_1 \neq 0 \text{ then } n_2 \text{ else } n_3)\\
&\mathcal{D}_{\text{ICE}}[\![S,\ T,\ D,\ I \triangleright \textit{other}]\!] = \text{reject}
\end{aligned}$$

which, intuitively, computes the meaning of these operations from an appropriate comparison of their operands' integer values against the numeric constant 0. No other C operator may appear in the syntax of integral constant expression entities.

5.7.3.2 Null Pointer Constants

Every well-formed integral constant expression with a zero value, as well as all casts of such an expression into a pointer to the "**`void`**" type, are known as *null pointer constants*. Formally, the denotations of all such entities are constructed by the following simple Haskell derivation:

$$\mathcal{D}_{\text{NULL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I \triangleright \textit{expression}) \rightarrow M(S,\ S,\ D,\ \textit{type},\ \textit{atom}_v)$$

$$\mathcal{D}_{\text{NULL}}[\![S,\ T,\ D,\ I \triangleright \texttt{(}e\texttt{)}]\!] = \mathcal{D}_{\text{NULL}}(S,\ T,\ D,\ I \triangleright e)$$

$$\begin{aligned}
&\mathcal{D}_{\text{NULL}}[\![S,\ T,\ D,\ I \triangleright \texttt{(}\textit{type-name}\texttt{)}e]\!] = \text{do}\\
&\quad (S_t,\ T_t,\ D_t,\ t_t) \leftarrow \mathcal{D}_{\text{TN}}(S,\ T,\ D,\ I \triangleright \textit{type-name})\\
&\quad (S_e,\ T_e,\ D_e,\ t_e,\ n_e) \leftarrow \mathcal{D}_{\text{ICE}}(S_t,\ T_t,\ D_t,\ I \triangleright e)\ ||\ \mathcal{D}_{\text{FCE}}(S_t,\ T_t,\ D_t,\ I \triangleright e)\\
&\quad (n) \leftarrow \mathcal{V}_{\phi(t_t)}[\![[\![\phi(t_t)]\!]_{[\![\phi(t_e)]\!]}(\#n_{e[\![\phi(t_e)]\!]})]\!]\\
&\quad \text{require } (n = 0 \wedge (\text{INT}(t_t) \vee (\text{PTR}(t_t) \wedge \mathcal{B}(t_t) = [\![\texttt{void}]\!] \wedge \text{INT}(t_e))))\\
&\quad \text{return } (S_e,\ T_e,\ D_e,\ t_t,\ [\![\#0_{[\![\phi(t_e)]\!]}]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{NULL}}[\![S,\ T,\ D,\ I \triangleright e]\!] = \text{do}\\
&\quad (S_e,\ T_e,\ D_e,\ t_e,\ n_e) \leftarrow \mathcal{D}_{\text{ICE}}(S,\ T,\ D,\ I \triangleright e)\\
&\quad \text{require } (n_e = 0)\\
&\quad \text{return } (S_e,\ T_e,\ D_e,\ t_e,\ [\![\#0_{[\![\phi(t_e)]\!]}]\!])
\end{aligned}$$

In many contexts, it is also convenient to devise a boolean predicate function "NULL", which, given some expression e and its lexical context $(S,\ T,\ D,\ I)$, returns a true value if and only if e represents a null pointer constant, without the need to construct the full denotation of that expression. In Haskell, such predicate can be implemented trivially as follows:

$$\text{NULL}[\![\cdot]\!] :: (S,\ S,\ D,\ I \triangleright \textit{expression}) \rightarrow \textit{bool}$$

$$\text{NULL}[\![S,\ T,\ D,\ I \triangleright e]\!] = (\mathcal{D}_{\text{NULL}}(S,\ T,\ D,\ I \triangleright e) \neq \epsilon)$$

which derives the null pointer denotation of e in the standard "Maybe" monad and compares the resulting optional value against its monadic zero "Nothing".

5.7.3.3 Static Initialiser Expressions

A further application of the constant expression syntax is found in the context of *initialiser* entities, whose precise structure and semantics are defined later in Section 5.8.12. When used as an initialiser for an object with a static linkage, an expression must always assume one of the following four syntactic forms:

① an *arithmetic constant expression* of a structure defined in Section 5.7.3.4, or
② a null pointer constant, or
③ an *address constant*, described in Section 5.7.3.5, or
④ an application of the binary "**+**" or "**−**" operator to an address constant and an integral constant expression.

Formally, the all such static initialiser expressions denote Etude atoms, whose values are defined by the following Haskell construction:

$$\mathcal{D}_{\text{SIE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I \triangleright \textit{expression}) \rightarrow M(S,\ S,\ D,\ \textit{type},\ \textit{atom}_V)$$

$$\begin{aligned}
\mathcal{D}_{\text{SIE}}[\![S,\ T,\ D,\ I \triangleright \ \mathbf{(}e\mathbf{)}\ ]\!] \quad &= \text{do } \mathcal{D}_{\text{SIE}}\ \ (S,\ T,\ D,\ I \triangleright e) \\
\mathcal{D}_{\text{SIE}}[\![S,\ T,\ D,\ I \triangleright e_1 \ \mathbf{+}\ e_2]\!] &= \text{do } \mathcal{D}_{\text{ACE}}\ \ [\![S,\ T,\ D,\ I \triangleright e_1 \ \mathbf{+}\ e_2]\!] \\
&\quad \mid\mid \text{do } \mathcal{D}_{\text{NULL}}\ [\![S,\ T,\ D,\ I \triangleright e_1 \ \mathbf{+}\ e_2]\!] \\
&\quad \mid\mid \text{do } \mathcal{D}_{\text{ADDR}}[\![S,\ T,\ D,\ I \triangleright e_1 \ \mathbf{+}\ e_2]\!] \\
&\quad \mid\mid \text{do } (S_1,\ T_1,\ D_1,\ t_1,\ \alpha) \leftarrow \mathcal{D}_{\text{ADDR}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad\quad (S_2,\ T_2,\ D_2,\ t_2,\ n) \ \leftarrow \mathcal{D}_{\text{ICE}}\ \ (S_1,\ T_1,\ D_1,\ I \triangleright e_2) \\
&\qquad\quad \text{require } (\text{PTR}(t_1) \wedge \text{OBJ}(\mathcal{B}(t_1))) \\
&\qquad\quad \text{return } (S_2,\ T_2,\ D_2,\ t_1,\ [\![\alpha +_{[\![\phi(t_1)]\!]} \#[\![n \times \mathcal{S}(\mathcal{B}(t_1))]\!]_{[\![\phi(t_2)]\!]}]\!]) \\
&\quad \mid\mid \text{do } (S_1,\ T_1,\ D_1,\ t_1,\ n) \ \leftarrow \mathcal{D}_{\text{ICE}}\ \ (S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad\quad (S_2,\ T_2,\ D_2,\ t_2,\ \alpha) \leftarrow \mathcal{D}_{\text{ADDR}}(S_1,\ T_1,\ D_1,\ I \triangleright e_2) \\
&\qquad\quad \text{require } (\text{PTR}(t_2) \wedge \text{OBJ}(\mathcal{B}(t_2))) \\
&\qquad\quad \text{return } (S_2,\ T_2,\ D_2,\ t_2,\ [\![\alpha +_{[\![\phi(t_2)]\!]} \#[\![n \times \mathcal{S}(\mathcal{B}(t_2))]\!]_{[\![\phi(t_1)]\!]}]\!]) \\
\mathcal{D}_{\text{SIE}}[\![S,\ T,\ D,\ I \triangleright e_1 \ \mathbf{-}\ e_2]\!] &= \text{do } \mathcal{D}_{\text{ACE}}\ \ [\![S,\ T,\ D,\ I \triangleright e_1 \ \mathbf{-}\ e_2]\!] \\
&\quad \mid\mid \text{do } \mathcal{D}_{\text{NULL}}\ [\![S,\ T,\ D,\ I \triangleright e_1 \ \mathbf{-}\ e_2]\!] \\
&\quad \mid\mid \text{do } \mathcal{D}_{\text{ADDR}}[\![S,\ T,\ D,\ I \triangleright e_1 \ \mathbf{-}\ e_2]\!] \\
&\quad \mid\mid \text{do } (S_1,\ T_1,\ D_1,\ t_1,\ \alpha) \leftarrow \mathcal{D}_{\text{ADDR}}(S,\ \ T,\ \ D,\ \ I \triangleright e_1) \\
&\qquad\quad (S_2,\ T_2,\ D_2,\ t_2,\ n) \ \leftarrow \mathcal{D}_{\text{ICE}}\ \ (S_1,\ T_1,\ D_1,\ I \triangleright e_2) \\
&\qquad\quad \text{require } (\text{PTR}(t_1) \wedge \text{OBJ}(\mathcal{B}(t_1))) \\
&\qquad\quad \text{return } (S_2,\ T_2,\ D_2,\ t_1,\ [\![\alpha +_{[\![\phi(t_1)]\!]} \#[\![n \times -\mathcal{S}(\mathcal{B}(t_1))]\!]_{[\![\phi(t_2)]\!]}]\!]) \\
\mathcal{D}_{\text{SIE}}[\![S,\ T,\ D,\ I \triangleright e]\!] \quad &= \text{do } \mathcal{D}_{\text{ACE}}\ \ (S,\ T,\ D,\ I \triangleright e) \\
&\quad \mid\mid \text{do } \mathcal{D}_{\text{NULL}}\ (S,\ T,\ D,\ I \triangleright e) \\
&\quad \mid\mid \text{do } \mathcal{D}_{\text{ADDR}}(S,\ T,\ D,\ I \triangleright e)
\end{aligned}$$

An astute reader will observe that the denotation of such a constant expression may, on occasions, represent an ill-formed Etude atom, so that the ultimate validity of C initialiser expressions must be always determined by a further semantic scrutiny of the resulting Etude program against a set of implementation-defined criteria outlined earlier in Chapter 4.

5.7.3.4 Arithmetic Constant Expression

The family of *arithmetic constant expressions*, used to initialise static objects of an integral or floating C type, represents a slight relaxation of the integral constant expression constraints. In particular, arithmetic constant expressions support all of the C operators admitted into the earlier integral family, and, further, allow for their applications to arbitrary constant operands of any well-formed arithmetic C type. Accordingly, a formal definition of their denotations follows a structure similar to that of the previous definition described in Section 5.7.3.1 above, except that no specific restriction is imposed on the use of floating point operand values and that the resulting denotation is represented by a constant atomic value rather than a simple numeric quantity. Formally:

$$\mathcal{D}_{\text{ACE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I \triangleright \textit{expression}) \rightarrow M(S,\ S,\ D,\ \textit{type},\ \textit{atom}_V)$$

In particular, every well-formed integer, character and enumeration constants represents an arithmetic constant expression, whose type and atomic value is derived directly from

the constant's lexical syntax by the process described earlier in Section 5.6:

$$\begin{aligned}
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \texttt{sizeof}\ e]\!] = \text{do}\\
&\quad (S_e, T_e, D_e, V_e, t_e, \tau_e) \leftarrow \mathcal{D}_{\text{LV}}(S, T, D, I \triangleright e) \mid\mid \mathcal{D}_{\text{V}}(S, T, D, I \triangleright e)\\
&\quad \text{require } (\text{OBJ}(t_e))\\
&\quad \text{return } (S_e, T_e, D_e \backslash \text{dom}(V_e), [\![\texttt{size_t}]\!], [\![\#[\![\mathcal{S}(t_e)]\!]_{[\![\phi(\texttt{size_t})]\!]}]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \texttt{sizeof (}\textit{type-name}\texttt{)}]\!] = \text{do}\\
&\quad (S_t, T_t, D_t, t_t) \leftarrow \mathcal{D}_{\text{TN}}(S, T, D, I \triangleright \textit{type-name})\\
&\quad \text{require } (\text{OBJ}(t_t))\\
&\quad \text{return } (S_t, T_t, D_t, [\![\texttt{size_t}]\!], [\![\#[\![\mathcal{S}(t_t)]\!]_{[\![\phi(\texttt{size_t})]\!]}]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \texttt{(}\textit{type-name}\texttt{)}e]\!] = \text{do}\\
&\quad (S_t, T_t, D_t, t_t) \leftarrow \mathcal{D}_{\text{TN}}(S, T, D, I \triangleright \textit{type-name})\\
&\quad (S_e, T_e, D_e, t_e, \alpha) \leftarrow \mathcal{D}_{\text{ACE}}(S_t, T_t, D_t, I \triangleright e)\\
&\quad \text{require } (\text{AT}(t_t))\\
&\quad (x) \leftarrow \mathcal{V}_{\phi(t_t)}[\![[\![\phi(t_t)]\!]_{[\![\phi(t_e)]\!]}(\alpha)]\!]\\
&\quad \text{return } (S_e, T_e, D_e, t_t, [\![\#x_{[\![\phi(t_t)]\!]}]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \textit{constant}]\!] = \text{do } (t, x) \leftarrow \mathcal{D}_{\text{C}}(S \triangleright \textit{constant})\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{return } (S, T, D, t, [\![\#x_{[\![\phi(t)]\!]}]\!])\\
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \texttt{+}e]\!] \quad = \text{do } (S_e, T_e, D_e, t_e, \alpha) \leftarrow \mathcal{D}_{\text{ACE}}(S, T, D, I \triangleright e)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (x) \leftarrow \mathcal{V}_{\phi(\text{ip}(t_e))}[\![[\![\phi(\text{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\alpha)]\!]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{return } (S_e, T_e, D_e, \text{ip}(t_e), [\![\#x_{[\![\phi(\text{ip}(t_e))]\!]}]\!])\\
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \texttt{-}e]\!] \quad = \text{do } (S_e, T_e, D_e, t_e, \alpha) \leftarrow \mathcal{D}_{\text{ACE}}(S, T, D, I \triangleright e)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (x) \leftarrow \mathcal{V}_{\phi(\text{ip}(t_e))}[\![-_{[\![\phi(t_e)]\!]}([\![\phi(\text{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\alpha))]\!]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{return } (S_e, T_e, D_e, \text{ip}(t_e), [\![\#x_{[\![\phi(\text{ip}(t_e))]\!]}]\!])\\
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \texttt{\~{}}e]\!] \quad = \text{do } (S_e, T_e, D_e, t_e, \alpha) \leftarrow \mathcal{D}_{\text{ACE}}(S, T, D, I \triangleright e)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (x) \leftarrow \mathcal{V}_{\phi(\text{ip}(t_e))}[\![\sim_{[\![\phi(t_e)]\!]}([\![\phi(\text{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\alpha))]\!]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{return } (S_e, T_e, D_e, \text{ip}(t_e), [\![\#x_{[\![\phi(\text{ip}(t_e))]\!]}]\!])\\
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \texttt{!}e]\!] \quad = \text{do } (S_e, T_e, D_e, t_e, \alpha) \leftarrow \mathcal{D}_{\text{ACE}}(S, T, D, I \triangleright e)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (x) \leftarrow \mathcal{V}_{\phi[\![\texttt{int}]\!]}[\![\alpha =_{[\![\phi(t_e)]\!]} \#0_{[\![\phi(t_e)]\!]}]\!]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{return } (S_e, T_e, D_e, [\![\texttt{int}]\!], [\![\#x_{[\![\phi[\![\texttt{int}]\!]]\!]}]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1\ \texttt{\&\&}\ e_2]\!] = \text{do}\\
&\quad (S_1, T_1, D_1, t_1, \alpha_1) \leftarrow \mathcal{D}_{\text{ACE}}(S,\ T,\ D,\ I \triangleright e_1)\\
&\quad (S_2, T_2, D_2, t_2, \alpha_2) \leftarrow \mathcal{D}_{\text{ACE}}(S_1, T_1, D_1, I \triangleright e_2)\\
&\quad (x) \leftarrow \mathcal{V}_{\phi[\![\texttt{int}]\!]}[\![(\alpha_1 \neq_{[\![\phi(t_1)]\!]} \#0_{[\![\phi(t_1)]\!]}) \triangle_{[\![\phi[\![\texttt{int}]\!]]\!]} (\alpha_2 \neq_{[\![\phi(t_2)]\!]} \#0_{[\![\phi(t_2)]\!]})]\!]\\
&\quad \text{return } (S_2, T_2, D_2, [\![\texttt{int}]\!], [\![\#x_{[\![\phi[\![\texttt{int}]\!]]\!]}]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1\ \texttt{||}\ e_2]\!] = \text{do}\\
&\quad (S_1, T_1, D_1, t_1, \alpha_1) \leftarrow \mathcal{D}_{\text{ACE}}(S,\ T,\ D,\ I \triangleright e_1)\\
&\quad (S_2, T_2, D_2, t_2, \alpha_2) \leftarrow \mathcal{D}_{\text{ACE}}(S_1, T_1, D_1, I \triangleright e_2)\\
&\quad (x) \leftarrow \mathcal{V}_{\phi[\![\texttt{int}]\!]}[\![(\alpha_1 \neq_{[\![\phi(t_1)]\!]} \#0_{[\![\phi(t_1)]\!]}) \bigtriangledown_{[\![\phi[\![\texttt{int}]\!]]\!]} (\alpha_2 \neq_{[\![\phi(t_2)]\!]} \#0_{[\![\phi(t_2)]\!]})]\!]\\
&\quad \text{return } (S_2, T_2, D_2, [\![\texttt{int}]\!], [\![\#x_{[\![\phi[\![\texttt{int}]\!]]\!]}]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3]\!] = \text{do}\\
&\quad (S_1, T_1, D_1, t_1, \alpha_1) \leftarrow \mathcal{D}_{\text{ACE}}(S,\ T,\ D,\ I \triangleright e_1)\\
&\quad (S_2, T_2, D_2, t_2, \alpha_2) \leftarrow \mathcal{D}_{\text{ACE}}(S_1, T_1, D_1, I \triangleright e_2)\\
&\quad (S_3, T_3, D_3, t_3, \alpha_3) \leftarrow \mathcal{D}_{\text{ACE}}(S_2, T_2, D_2, I \triangleright e_3)\\
&\quad (x) \leftarrow \mathcal{V}_{\phi[\![\texttt{int}]\!]}(\alpha_1)\\
&\quad \text{return } (S_3, T_3, D_3, t_2 \boxtimes t_3, \text{if } x \neq 0 \text{ then } \alpha_2 \text{ else } \alpha_3)
\end{aligned}$$

Otherwise, all well-formed applications of the binary "`*`", "`/`", "`%`", "`+`", "`-`", "`<<`", "`>>`", "`&`", "`^`", "`|`", "`<`", "`>`", "`<=`", "`>=`", "`==`", "`!=`" operators to constant arithmetic expression operands and all parenthesised versions of such an expression also belong to this semantic family. Their denotations are derived from the following familiar mapping of these C operators to their Etude equivalents:

$$
\begin{aligned}
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ * } e_2]\!] &= \mathcal{D}_{\text{BACE}}(S, T, D, I, [\![\times]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ / } e_2]\!] &= \mathcal{D}_{\text{BACE}}(S, T, D, I, [\![\div]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ \% } e_2]\!] &= \mathcal{D}_{\text{IACE}}(S, T, D, I, [\![\cdot|\cdot]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ + } e_2]\!] &= \mathcal{D}_{\text{BACE}}(S, T, D, I, [\![+]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ - } e_2]\!] &= \mathcal{D}_{\text{BACE}}(S, T, D, I, [\![-]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ << } e_2]\!] &= \mathcal{D}_{\text{IACE}}(S, T, D, I, [\![\ll]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ >> } e_2]\!] &= \mathcal{D}_{\text{IACE}}(S, T, D, I, [\![\gg]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ \& } e_2]\!] &= \mathcal{D}_{\text{IACE}}(S, T, D, I, [\![\triangle]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ ^ } e_2]\!] &= \mathcal{D}_{\text{IACE}}(S, T, D, I, [\![\triangledown]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ | } e_2]\!] &= \mathcal{D}_{\text{IACE}}(S, T, D, I, [\![\triangledown]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ < } e_2]\!] &= \mathcal{D}_{\text{RACE}}(S, T, D, I, [\![<]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ > } e_2]\!] &= \mathcal{D}_{\text{RACE}}(S, T, D, I, [\![>]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ <= } e_2]\!] &= \mathcal{D}_{\text{RACE}}(S, T, D, I, [\![\leq]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ >= } e_2]\!] &= \mathcal{D}_{\text{RACE}}(S, T, D, I, [\![\geq]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ == } e_2]\!] &= \mathcal{D}_{\text{RACE}}(S, T, D, I, [\![=]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright e_1 \texttt{ != } e_2]\!] &= \mathcal{D}_{\text{RACE}}(S, T, D, I, [\![\neq]\!] \triangleright e_1, e_2)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \texttt{(}e\texttt{)}]\!] &= \mathcal{D}_{\text{ACE}}(S, T, D, I \triangleright e)\\
\mathcal{D}_{\text{ACE}}[\![S, T, D, I \triangleright \mathit{other}]\!] &= \text{reject}
\end{aligned}
$$

assuming that the individual meanings of all binary arithmetic and relational operations listed above are derived systematically from their Etude equivalents by the following Haskell constructions:

$$
\begin{aligned}
&\mathcal{D}_{\text{BACE}}[\![\cdot]\!],\ \mathcal{D}_{\text{IACE}}[\![\cdot]\!],\ \mathcal{D}_{\text{RACE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow\\
&\qquad (S, S, D, I, \mathit{binary\text{-}op} \triangleright \mathit{expression}, \mathit{expression}) \rightarrow M(S, S, D, \mathit{type}, \mathit{atom}_V)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}_{\text{BACE}}[\![S, T, D, I, \mathit{op} \triangleright e_1, e_2]\!] = \text{do}\\
&\qquad (S_1, T_1, D_1, t_1, \alpha_1) \leftarrow \mathcal{D}_{\text{ACE}}(S,\ T,\ D,\ I \triangleright e_1)\\
&\qquad (S_2, T_2, D_2, t_2, \alpha_2) \leftarrow \mathcal{D}_{\text{ACE}}(S_1, T_1, D_1, I \triangleright e_2)\\
&\qquad (x) \leftarrow \mathcal{V}_{\phi(t_1 \boxplus t_2)}[\![([\![\phi(t_1 \boxplus t_2)]\!]_{[\![\phi(t_1)]\!]}(\alpha_1))\ op_{[\![\phi(t_1 \boxplus t_2)]\!]}\ ([\![\phi(t_1 \boxplus t_2)]\!]_{[\![\phi(t_2)]\!]}(\alpha_2))]\!]\\
&\qquad \text{return } (S_2, T_2, D_2, t_1 \boxplus t_2, [\![\#x_{[\![\phi(t_1 \boxplus t_2)]\!]}]\!])
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}_{\text{IACE}}[\![S, T, D, I, \mathit{op} \triangleright e_1, e_2]\!] = \text{do}\\
&\qquad (S_1, T_1, D_1, t_1, \alpha_1) \leftarrow \mathcal{D}_{\text{ACE}}(S,\ T,\ D,\ I \triangleright e_1)\\
&\qquad (S_2, T_2, D_2, t_2, \alpha_2) \leftarrow \mathcal{D}_{\text{ACE}}(S_1, T_1, D_1, I \triangleright e_2)\\
&\qquad (x) \leftarrow \mathcal{V}_{\phi(t_1 \boxplus t_2)}[\![([\![\phi(t_1 \boxplus t_2)]\!]_{[\![\phi(t_1)]\!]}(\alpha_1))\ op_{[\![\phi(t_1 \boxplus t_2)]\!]}\ ([\![\phi(t_1 \boxplus t_2)]\!]_{[\![\phi(t_2)]\!]}(\alpha_2))]\!]\\
&\qquad \text{require } \text{INT}(t_1) \wedge \text{INT}(t_2)\\
&\qquad \text{return } (S_2, T_2, D_2, t_1 \boxplus t_2, [\![\#x_{[\![\phi(t_1 \boxplus t_2)]\!]}]\!])
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}_{\text{RACE}}[\![S, T, D, I, \mathit{op} \triangleright e_1, e_2]\!] = \text{do}\\
&\qquad (S_1, T_1, D_1, t_1, \alpha_1) \leftarrow \mathcal{D}_{\text{ACE}}(S,\ T,\ D,\ I \triangleright e_1)\\
&\qquad (S_2, T_2, D_2, t_2, \alpha_2) \leftarrow \mathcal{D}_{\text{ACE}}(S_1, T_1, D_1, I \triangleright e_2)\\
&\qquad (x) \leftarrow \mathcal{V}_{\phi(t_1 \boxplus t_2)}[\![([\![\phi(t_1 \boxplus t_2)]\!]_{[\![\phi(t_1)]\!]}(\alpha_1))\ op_{[\![\phi(t_1 \boxplus t_2)]\!]}\ ([\![\phi(t_1 \boxplus t_2)]\!]_{[\![\phi(t_2)]\!]}(\alpha_2))]\!]\\
&\qquad \text{return } (S_2, T_2, D_2, [\![\texttt{int}]\!], [\![\#x_{[\![\phi[\![\texttt{int}]\!]]\!]}]\!])
\end{aligned}
$$

A careful reader will observe that the meaning of every well-formed arithmetic constant

always represents a valid Etude atom of the form "$\#x_\phi$", whose reduction is performed internally by the compiler as part of the translation process, prior to an eventual execution of the translated program. This behaviour is explicitly mandated by the C standard and provides for a potential discrepancy between the evaluation rules of floating point operations in their ordinary and constant contexts, as discussed earlier in Section 5.6.

5.7.3.5 Address Constants

The final constant expression category, known as *address constants*, is used by the language for representation of the initial values assigned to statically-linked pointer variables. Intuitively, such expressions must represent function designators, l-values that designate a statically-linked object or else pointers to such l-values. They may be formed from almost arbitrary applications of the unary "`&`", "`*`" operators, as well as the "`.`", "`->`", array subscripts and pointer cast operations, provided that their operands also represent address constants. Formally, this semantic family is defined in Haskell as follows:

$$\mathcal{D}_{\text{ADDR}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \textit{expression}) \rightarrow M(S,\, S,\, D,\, \textit{type},\, \textit{atom}_v)$$

In particular, such constants must be always formed from applications of the unary operator "`&`" to a *constant l-value object*, or else by an implicit conversion of a constant l-value of a function or an array type. Cast operations are also permitted, provided that they convert a well-formed address or null pointer constant into another pointer type. Formally:

$$\mathcal{D}_{\text{ADDR}}[\![S,\, T,\, D,\, I \triangleright \texttt{(}e\texttt{)}]\!] = \mathcal{D}_{\text{ADDR}}(S,\, T,\, D,\, I \triangleright e)$$

$$\begin{aligned}
&\mathcal{D}_{\text{ADDR}}[\![S,\, T,\, D,\, I \triangleright \texttt{(}\textit{type-name}\texttt{)}e]\!] = \text{do}\\
&\quad (S_t,\, T_t,\, D_t,\, t_t) \leftarrow \mathcal{D}_{\text{TN}}(S,\, T,\, D,\, I \triangleright \textit{type-name})\\
&\quad (S_e,\, T_e,\, D_e,\, t_e,\, \alpha) \leftarrow \mathcal{D}_{\text{ADDR}}(S_t,\, T_t,\, D_t,\, I \triangleright e) \;||\; \mathcal{D}_{\text{NULL}}(S_t,\, T_t,\, D_t,\, I \triangleright e)\\
&\quad \text{require } (\text{PTR}(t_t))\\
&\quad \text{return } (S_e,\, T_e,\, D_e,\, t_t,\, [\![[\![\phi(t_t)]\!]_{[\![\phi(t_e)]\!]}(\alpha)]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ADDR}}[\![S,\, T,\, D,\, I \triangleright \texttt{\&}e]\!] = \text{do}\\
&\quad (S_e,\, T_e,\, D_e,\, t_e,\, \alpha) \leftarrow \mathcal{D}_{\text{CLV}}(S,\, T,\, D,\, I \triangleright e)\\
&\quad \text{require } (\neg\text{BF}(t_e))\\
&\quad \text{return } (S_e,\, T_e,\, D_e,\, [\![t_e\,\texttt{*}]\!],\, \alpha)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{ADDR}}[\![S,\, T,\, D,\, I \triangleright e]\!] = \text{do}\\
&\quad (S_e,\, T_e,\, D_e,\, t_e,\, \alpha) \leftarrow \mathcal{D}_{\text{CLV}}(S,\, T,\, D,\, I \triangleright e)\\
&\quad \text{require } (\text{FUN}(t_e) \vee \text{ARR}(t_e))\\
&\quad \text{return } (S_e,\, T_e,\, D_e,\, \text{pp}(t_e),\, \alpha)
\end{aligned}$$

Further, the set and denotations of *constant l-values*, which, intuitively, represent objects with an external, internal or static linkage, are formalised by the following Haskell construction:

$$\mathcal{D}_{\text{CLV}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \textit{expression}) \rightarrow M(S,\, S,\, D,\, \textit{type},\, \textit{atom}_v)$$

In particular, every string literal and every variable bound to a designator with an external, internal or static linkage form, as well as every parenthesised version of

another valid l-value constant always belongs to this semantic family, as captured by the following three translation rules:

$$\mathcal{D}_{\text{CLV}}[\![S,\, T,\, D,\, I \triangleright \textit{string-literal}]\!] = \text{do}$$
$$(t,\, \bar{\delta}) \leftarrow \mathcal{D}_{\text{SC}}(\textit{string-literal})$$
$$\text{return } (S,\, T,\, D \cup \{\nu(D){:}[\![\textsf{OBJ}\ (\bar{\delta})\ \textsf{OF}\ ([\![\bar{\xi}(t)]\!])]\!]\},\, t,\, \nu(D))$$

$$\mathcal{D}_{\text{CLV}}[\![S,\, T,\, D,\, I \triangleright x]\!] = \text{do}$$
$$\text{require } (x \in \text{dom}(S) \wedge \text{SL}(\mathcal{L}(S(x))))$$
$$\text{return } (S,\, T,\, D,\, \mathcal{T}(S(x)),\, \nu(\mathcal{L}(S(x))))$$

$$\mathcal{D}_{\text{CLV}}[\![S,\, T,\, D,\, I \triangleright \texttt{(}e\texttt{)}]\!] = \mathcal{D}_{\text{CLV}}(S,\, T,\, D,\, I \triangleright e)$$

Further, a constant l-value may represent a member of a constant structure, union or array object, obtained by a well-formed application of the direct member operator "**.**" to a constant l-value operand, an application of the indirect member operator "**->**" to an address constant, or else from an application of the array subscript operator "**[]**" to an address constant and an integral constant expression:

$$\mathcal{D}_{\text{CLV}}[\![S,\, T,\, D,\, I \triangleright e\texttt{.}x]\!] = \text{do}$$
$$(S_e,\, T_e,\, D_e,\, t_e,\, \alpha) \leftarrow \mathcal{D}_{\text{CLV}}(S,\, T,\, D,\, I \triangleright e)$$
$$(m) \leftarrow \bar{m}(t_e)(x)$$
$$\text{require } (\text{SU}(t_e) \wedge x \in \text{dom}(\bar{m}(t_e)))$$
$$\text{return } (S_e,\, T_e,\, D_e,\, \text{tq}(t_e) \cup \mathcal{T}(m),\, [\![[\![O(\phi(\mathcal{T}(m)))]\!]_{\text{O.}\Phi}(\alpha) +_{\text{O.}\Phi} \#[\![O(m)]\!]_{\text{Z.}\Phi}]\!])$$

$$\mathcal{D}_{\text{CLV}}[\![S,\, T,\, D,\, I \triangleright e\texttt{->}x]\!] = \text{do}$$
$$(S_e,\, T_e,\, D_e,\, t_e,\, \alpha) \leftarrow \mathcal{D}_{\text{ADDR}}(S,\, T,\, D,\, I \triangleright e)$$
$$(m) \leftarrow \bar{m}(\mathcal{B}(t_e))(x)$$
$$\text{require } (\text{SU}(\mathcal{B}(t_e)) \wedge x \in \text{dom}(\bar{m}(\mathcal{B}(t_e))))$$
$$\text{return } (S_e,\, T_e,\, D_e,\, \text{tq}(\mathcal{B}(t_e)) \cup \mathcal{T}(m),\, [\![[\![O(\phi(\mathcal{T}(m)))]\!]_{\text{O.}\Phi}(\alpha) +_{\text{O.}\Phi} \#[\![O(m)]\!]_{\text{Z.}\Phi}]\!])$$

$$\mathcal{D}_{\text{CLV}}[\![S,\, T,\, D,\, I \triangleright e_1\,\texttt{[}e_2\texttt{]}]\!]$$
$$= \text{do } (S_1,\, T_1,\, D_1,\, t_1,\, \alpha) \leftarrow \mathcal{D}_{\text{ADDR}}(S,\ T,\ D,\ I \triangleright e_1)$$
$$(S_2,\, T_2,\, D_2,\, t_2,\, n\,) \leftarrow \mathcal{D}_{\text{ICE}}\ (S_1,\, T_1,\, D_1,\, I \triangleright e_2)$$
$$\text{require } (\text{OBJ}(\mathcal{B}(t_1)))$$
$$\text{return } (S_2,\, T_2,\, D_2,\, \mathcal{B}(t_1),\, [\![\alpha +_{[\![\phi(t_1)]\!]} \#[\![n \times \mathcal{S}(\mathcal{B}(t_1))]\!]_{\text{Z.}\Phi}]\!])$$
$$\|\ \text{do } (S_1,\, T_1,\, D_1,\, t_1,\, n\,) \leftarrow \mathcal{D}_{\text{ICE}}\ (S,\ T,\ D,\ I \triangleright e_1)$$
$$(S_2,\, T_2,\, D_2,\, t_2,\, \alpha) \leftarrow \mathcal{D}_{\text{ADDR}}(S_1,\, T_1,\, D_1,\, I \triangleright e_2)$$
$$\text{require } (\text{OBJ}(\mathcal{B}(t_2)))$$
$$\text{return } (S_2,\, T_2,\, D_2,\, \mathcal{B}(t_2),\, [\![\alpha +_{[\![\phi(t_2)]\!]} \#[\![n \times \mathcal{S}(\mathcal{B}(t_2))]\!]_{\text{Z.}\Phi}]\!])$$

Last but not least, constant l-values may be constructed by applying the unary "*****" operator to the value of a well-formed address constant:

$$\mathcal{D}_{\text{CLV}}[\![S,\, T,\, D,\, I \triangleright \star e]\!] \quad = \text{do } (S_e,\, T_e,\, D_e,\, t_e,\, \alpha) \leftarrow \mathcal{D}_{\text{ADDR}}(S,\, T,\, D,\, I \triangleright e)$$
$$\text{return } (S_e,\, T_e,\, D_e,\, \mathcal{B}(t_e),\, \alpha)$$
$$\mathcal{D}_{\text{CLV}}[\![S,\, T,\, D,\, I \triangleright \textit{other}]\!] = \text{reject}$$

observing that, by definition, every such constant always represents a pointer to an l-value, whose content may be described either by a single Etude atom or else a sum of such an atom with an integer constant of the form "$\#n_{\text{Z.}\Phi}$", which makes it possible to implement all such constants under every relocatable program representation found in the industry.

5.8 Declarations

In C, the *declaration* syntax provides programs with a concrete means of introducing new entities into a translation union, be it variables, C types or type tags. Further, in Section 5.8.9 a subset of this syntax doubles as a source code representation of C types, in the form of entities known as *type names*. Due to their vastly diverse nature, the structure and semantics of C declarations follow a pattern more irregular than that of any other syntactic entity in the language, which makes for an interesting reading but does not help to simplify their formalisation. In the concrete syntax of the language, most declarations begin with a group of *declaration specifiers* that are usually followed by one or more *initialised declarators*. In particular, the canonical form of a C declaration generally assumes the following structure:

declaration:
    declaration-specifiers init-declarator-list$_{opt}$ **;**

declaration-list:
    declaration
    declaration-list declaration

Under the semantic framework adopted in this work, every such construct is always interpreted in the context of its current variable and tag scopes S and T, a set of item definitions D and the current scope index I. Similarly to C expressions, its denotation includes any required updates to S, T and D, together with a set of local variables V and an initialiser term τ, which, intuitively, prepares the initial content of the corresponding memory-resident object for declarations with automatic and register linkage forms and, in all other cases, defaults to the trivial Etude term "RET ()". Formally, these denotations are derived by the following Haskell function:

$$\mathcal{D}_{\mathrm{D}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I \triangleright \textit{declaration-list}_{opt}) \rightarrow M(S,\ S,\ D,\ V,\ \textit{term}_{\mathrm{V}})$$

In particular, in a single declaration with at least one declarator provided by its idl_{opt} component, the list of declaration specifiers is split into three components $\bar{sc}$, $\bar{tq}$ and $\bar{ts}$ as described in Section 5.8.1 below. The later is then promptly converted into a single C type t_b, known as the declaration's *base type*, using a simple algorithm defined later in Section 5.8.3, merged with the associated list of type qualifiers $\bar{tq}$ and applied to all of the supplied declarators in *idl*, eventually producing a collective meaning of the entire construct as follows:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{D}}[\![S,\ T,\ D,\ I \triangleright \textit{ds idl;}]\!] = \text{do}\\
&\quad (\bar{sc},\ \bar{tq},\ \bar{ts}) \leftarrow \mathcal{D}_{\mathrm{DS}}(\textit{ds})\\
&\quad (S_1,\ T_1,\ D_1,\ t_b) \leftarrow \mathcal{D}_{\mathrm{TS}}(S,\ T,\ D,\ I \triangleright \bar{ts})\\
&\quad (S_2,\ T_2,\ D_2,\ V,\ \tau) \leftarrow \mathcal{D}_{\mathrm{IR}}(S_1,\ T_1,\ D_1,\ I,\ \bar{sc},\ \bar{tq} +\!\!+ t_b \triangleright \textit{idl})\\
&\quad \text{return } (S_2,\ T_2,\ D_2,\ V,\ \tau)
\end{aligned}$$

If, however, no *init-declarator-list* is included in the syntax, then the construct represents a rather different kind of entity known as a *tag declaration*. Its declaration specifiers are scrutinised by a dedicated function $\mathcal{D}_{\mathrm{TAG}}$ that is defined separately in

Section 5.8.6. In Haskell:

$$\begin{aligned}&\mathcal{D}_{\mathrm{D}}[\![S,\ T,\ D,\ I \triangleright ds\texttt{;}\,]\!] = \text{do}\\&\quad (S',\ T',\ D',\ t) \leftarrow \mathcal{D}_{\mathrm{TAG}}(S,\ T,\ D,\ I \triangleright ds)\\&\quad \text{return } (S',\ T',\ D',\ \varnothing,\ [\![\texttt{RET ()}]\!])\end{aligned}$$

Last but not least, the denotations of two or more declarations can be naturally combined by obtaining a union of their local variable sets V_k and, further, composing their respective initialiser terms τ_1, τ_2 ... τ_n into a single Etude construct of the form "LET () = τ_1; LET () = τ_2; ... τ_n", so that initialisation of all objects declared earlier in the program is always completed before proceeding with the translation of any following declarations. Under this interpretation, an omitted list of declarations denotes an empty set of local variables $\varnothing$ and a singular initialiser term of the form "RET ()", as captured by the following pair of translation rules:

$$\begin{aligned}&\mathcal{D}_{\mathrm{D}}[\![S,\ T,\ D,\ I \triangleright dl\ d]\!] = \text{do } (S_1,\ T_1,\ D_1,\ V_1,\ \tau_1) \leftarrow \mathcal{D}_{\mathrm{D}}(S,\ \ T,\ \ D,\ \ I \triangleright dl)\\&\qquad\qquad\qquad\qquad\quad (S_2,\ T_2,\ D_2,\ V_2,\ \tau_2) \leftarrow \mathcal{D}_{\mathrm{D}}(S_1,\ T_1,\ D_1,\ I \triangleright d)\\&\qquad\qquad\qquad\qquad\quad \text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ [\![\texttt{LET () = }\tau_1\texttt{; }\tau_2]\!])\\&\mathcal{D}_{\mathrm{D}}[\![S,\ T,\ D,\ I \triangleright \epsilon]\!] \quad = \text{do return } (S,\ T,\ D,\ \varnothing,\ [\![\texttt{RET ()}]\!])\end{aligned}$$

5.8.1 Declaration Specifiers

In every C declaration of the form "*ds* idl_{opt};", the list *ds* represents a sequence of one or more *declaration specifier* entities, constructed from a mixture of three kinds of constructs, known as *storage class specifiers*, *type specifiers* and *type qualifiers*, respectively. The concrete syntax of such lists is captured naturally by the following Haskell type:

declaration-specifiers:
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

The three kinds of declaration specifiers are discussed individually in Sections 5.8.2, 5.8.3 and 5.8.7 later in this chapter. However, before their meaning can be analysed by the compiler, different kinds of specifies must be segregated from each other, as described by the following simple Haskell construction:

$$\begin{aligned}&\mathcal{D}_{\mathrm{DS}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow\\&\quad (\textit{declaration-specifiers}_{opt}) \rightarrow\\&\quad M(\{\textit{storage-class-specifier}\},\ \{\textit{type-qualifier}\},\ \{\textit{type-specifier}\})\end{aligned}$$

In particular, a well-formed C declaration may only ever include at most one of each particular storage class specifier, type qualifier and type specifier entity form. Further, the precise order in which these entities appear in the program's syntax is irrelevant to the declaration's semantics, so that $\mathcal{D}_{\mathrm{DS}}$ dissects an optional list of declaration specifiers into three independent sets $\bar{sc}$, $\bar{tq}$ and $\bar{ts}$ as follows:

$$\begin{aligned}&\mathcal{D}_{\mathrm{DS}}[\![\textit{storage-class-specifier } ds_{opt}]\!] = \text{do}\\&\quad (\bar{sc},\ \bar{tq},\ \bar{ts}) \leftarrow \mathcal{D}_{\mathrm{DS}}(ds_{opt})\\&\quad \text{require } (\textit{storage-class-specifier} \notin \bar{sc})\\&\quad \text{return } (\bar{sc} \cup \{\textit{storage-class-specifier}\},\ \bar{tq},\ \bar{ts})\end{aligned}$$

$\mathcal{D}_{\text{DS}}[\![\textit{type-specifier } ds_{opt}]\!]$ = do
    $(s\bar{c},\ t\bar{q},\ t\bar{s}) \leftarrow \mathcal{D}_{\text{DS}}(ds_{opt})$
    require $(\textit{type-specifier} \notin t\bar{s})$
    return $(s\bar{c},\ t\bar{q},\ t\bar{s} \cup \{\textit{type-specifier}\})$

$\mathcal{D}_{\text{DS}}[\![\textit{type-qualifier } ds_{opt}]\!]$ = do
    $(s\bar{c},\ t\bar{q},\ t\bar{s}) \leftarrow \mathcal{D}_{\text{DS}}(ds_{opt})$
    require $(\textit{type-qualifier} \notin t\bar{q})$
    return $(s\bar{c},\ t\bar{q} \cup \{\textit{type-qualifier}\},\ t\bar{s})$

$\mathcal{D}_{\text{DS}}[\![]\!]$ = return $(\varnothing, \varnothing, \varnothing)$

5.8.2 Storage Class Specifiers

In the concrete syntax of the language, the five keywords "**typedef**", "**extern**", "**static**", "**auto**" and "**register**" are known as *storage class specifiers*:

storage-class-specifier:
    typedef
    extern
    static
    auto
    register

Intuitively, such entities describe the linkage of a designator associated with some variable x of the C type t, which is introduced by the surrounding declaration as follows:

① All declarations with the "**typedef**" storage class specifier always have the "***type***" linkage form.

② Otherwise, if a declaration includes the "**extern**" storage class specifier, then x has the same linkage as any visible declaration of that identifier with the file scope, i.e., the linkage of an existing designator in S with a scope index of 0. If no such declaration can be found, then x is assigned the external linkage form "***extern*** x".

③ If a declaration in the file scope includes the "**static**" storage class specifier, then the identifier has the internal linkage form "***intern*** x".

④ However, if such a declaration appears in a nested scope with a non-zero scope index, then the declared identifier is assigned the private linkage "***private*** ν", in which the Etude variable ν is given a globally-unique name $\nu(D)$.

⑤ Likewise, if a declaration in a nested scope includes the "**auto**" or "**register**" storage class specifier, then x is assigned, respectively the automatic or register linkage form "***auto*** ν" or "***register*** ν", using a similar globally-unique Etude variable $\nu(D)$.

⑥ Further, such automatic linkage is also assigned to all declarations without any storage class specifiers, provided that the construct appears in a nested scope and that it does not assign a function type for the identifier.

⑦ On the other hand, a declaration without any storage class specifiers behaves exactly as if the construction included the "**extern**" storage class specifier whenever the entity appears within the file scope or if it designates a C function. In all other cases, x is assigned the external linkage form "***extern*** x".

No other combination of storage class specifiers may ever appear in a well-formed C declaration. In particular, as a direct consequence of this definition, any given C declaration can include at most one storage class specifier entity and no declaration that appears in the file scope may contain the "**`auto`**" or "**`extern`**" storage class specifier forms. In Haskell, the above rules can be formalised as follows:

$$\mathcal{D}_{\mathrm{SC}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I,\ \mathit{type},\ \mathit{identifier} \triangleright [\mathit{storage\text{-}class\text{-}specifier}]) \rightarrow M(\mathit{linkage})$$

$$\mathcal{D}_{\mathrm{SC}}[\![S,\ T,\ D,\ I,\ t,\ x \triangleright \bar{sc}]\!]$$

$$\begin{array}{lll}
\mid \bar{sc} = [\![\texttt{typedef}]\!] & & = \text{return } [\![\textbf{\textit{type}}]\!] \\
\mid \bar{sc} = [\![\texttt{extern}]\!] & \wedge\ (x \in \mathrm{dom}(S) \wedge I(S(x)) = 0) & = \text{return } \mathcal{L}(S(x)) \\
\mid \bar{sc} = [\![\texttt{extern}]\!] & \wedge\ (x \notin \mathrm{dom}(S) \vee I(S(x)) \neq 0) & = \text{return } [\![\textbf{\textit{extern}}\ x]\!] \\
\mid \bar{sc} = [\![\texttt{static}]\!] & \wedge\ (I = 0) & = \text{return } [\![\textbf{\textit{intern}}\ x]\!] \\
\mid \bar{sc} = [\![\texttt{static}]\!] & \wedge\ (I \neq 0) & = \text{return } [\![\textbf{\textit{private}}\ [\![\nu(D)]\!]]\!] \\
\mid \bar{sc} = [\![\texttt{auto}]\!] & \wedge\ (I \neq 0) & = \text{return } [\![\textbf{\textit{auto}}\ [\![\nu(D)]\!]]\!] \\
\mid \bar{sc} = [\![\texttt{register}]\!] & \wedge\ (I \neq 0) & = \text{return } [\![\textbf{\textit{register}}\ [\![\nu(D)]\!]]\!] \\
\mid \bar{sc} = [\![]\!] & \wedge\ \neg\mathrm{FUN}(t) \wedge (I \neq 0) & = \text{return } [\![\textbf{\textit{auto}}\ [\![\nu(D)]\!]]\!] \\
\mid \bar{sc} = [\![]\!] & \wedge\ \neg\mathrm{FUN}(t) \wedge (I = 0) & = \text{return } [\![\textbf{\textit{extern}}\ x]\!] \\
\mid \bar{sc} = [\![]\!] & \wedge\ \mathrm{FUN}(t) & = \mathcal{D}_{\mathrm{SC}}[\![S,\ T,\ D,\ I,\ t,\ x \triangleright \texttt{extern}]\!] \\
\mid \text{otherwise} & & = \text{reject}
\end{array}$$

5.8.3 Type Specifiers

In C, most simple types are represented by sets of entities known as *type specifiers*. The C Standard defines nine type specifier keywords and three composite type specifier structures, whose concrete syntax is captured collectively by the following grammar:

type-specifier:
- **`void`**
- **`char`**
- **`short`**
- **`int`**
- **`long`**
- **`float`**
- **`double`**
- **`signed`**
- **`unsigned`**
- *struct-or-union-specifier*
- *enum-specifier*
- *typedef-name*

The precise syntax and semantics of the "*enum-specifier*", "*struct-or-union-specifier*" and "*typedef-name*" type specifier forms is defined separately in Sections 5.8.4, 5.8.5 and 5.8.10 later in this chapter. Otherwise, every valid sorted list of such entities denotes a predetermined C type from Section 5.4, such that, intuitively, each meaningful list of type specifiers is mapped to an abstract C type with a lexically-identical name, except that all arithmetic types other than "***`char`***" and "***`int`***" are explicitly marked with the "***`signed`***" or "***`unsigned`***" keyword and that "***`int`***" is generally excluded

from the names of "***short***" and "***long***" type forms. Formally, this correspondence is captured by the following Haskell definition:

$\mathcal{D}_{\text{TS}}[\![\cdot]\!]$ :: (monad-fix M) $\Rightarrow$ (S, S, D, $I \triangleright$ [*type-specifier*]) $\rightarrow$ M(S, S, D, *type*)

$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{void}]\!]$ = return (S, T, D, [[***void***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{char}]\!]$ = return (S, T, D, [[***char***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{signed char}]\!]$ = return (S, T, D, [[***signed char***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{unsigned char}]\!]$ = return (S, T, D, [[***unsigned char***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{short}]\!]$ = return (S, T, D, [[***signed short***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{short int}]\!]$ = return (S, T, D, [[***signed short***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{signed short}]\!]$ = return (S, T, D, [[***signed short***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{signed short int}]\!]$ = return (S, T, D, [[***signed short***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{unsigned short}]\!]$ = return (S, T, D, [[***unsigned short***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{unsigned short int}]\!]$ = return (S, T, D, [[***unsigned short***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \epsilon]\!]$ = return (S, T, D, [[***int***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{int}]\!]$ = return (S, T, D, [[***int***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{signed}]\!]$ = return (S, T, D, [[***signed int***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{signed int}]\!]$ = return (S, T, D, [[***signed int***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{unsigned}]\!]$ = return (S, T, D, [[***unsigned int***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{unsigned int}]\!]$ = return (S, T, D, [[***unsigned int***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{long}]\!]$ = return (S, T, D, [[***signed long***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{long int}]\!]$ = return (S, T, D, [[***signed long***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{signed long}]\!]$ = return (S, T, D, [[***signed long***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{signed long int}]\!]$ = return (S, T, D, [[***signed long***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{unsigned long}]\!]$ = return (S, T, D, [[***unsigned long***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{unsigned long int}]\!]$ = return (S, T, D, [[***unsigned long***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{float}]\!]$ = return (S, T, D, [[***float***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{double}]\!]$ = return (S, T, D, [[***double***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright \texttt{long double}]\!]$ = return (S, T, D, [[***long double***]])
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright$ *struct-or-union-specifier*$]\!]$ = $\mathcal{D}_{\text{SUS}}$ (S, T, D, $I \triangleright$ *struct-or-union-specifier*)
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright$ *enum-specifier*$]\!]$ = $\mathcal{D}_{\text{ENS}}$ (S, T, D, $I \triangleright$ *enum-specifier*)
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright$ *typedef-name*$]\!]$ = $\mathcal{D}_{\text{TDN}}$(S, T, D, $I \triangleright$ *typedef-name*)
$\mathcal{D}_{\text{TS}}[\![S, T, D, I \triangleright$ *other*$]\!]$ = reject

5.8.4 Structure and Union Specifiers

A *structure or union specifier* consists of the keyword "**struct**" or "**union**", followed by an optional identifier and an optional list of *structure declarations* enclosed in braces. At least one of the *identifier* and *struct-declaration-list* components must be present, as described by the following set of Haskell data type definitions:

struct-or-union-specifier:
    struct-or-union identifier$_{opt}$ **{** *struct-declaration-list* **}**
    struct-or-union identifier

struct-or-union:
    struct
    union

Every such structure or union specifier denotes a C type derived from its syntax and the surrounding translation context using the following Haskell construction:

$\mathcal{D}_{\text{SUS}}[\![\cdot]\!]$ :: (monad-fix M) $\Rightarrow$ (S, S, D, $I \triangleright$ *struct-or-union-specifier*) $\rightarrow$ M(S, S, D, *type*)

In particular, if the entity includes no identifier, then a new globally-unique type tag "tag(D)" is introduced into the program, with the entire construct denoting a structure or union type "*struct-or-union* $[\![\text{tag}(D)]\!]$ $[\![\bar{m}]\!]$", in which the member list $\bar{m}$ is derived from the specifier's list of structure declarations *sdl*, after annotating every member $m_k \in \bar{m}$ with an appropriate offset value as follows:

$$\begin{aligned}&\mathcal{D}_{\text{SUS}}[\![S,\, T,\, D,\, I \triangleright \textit{struct-or-union}\ \{\textit{sdl}\}]\!] = \text{do}\\ &\quad (S',\, T',\, D',\, \bar{m}) \leftarrow \mathcal{D}_{\text{SD}}(S,\, T,\, D \cup \{\text{tag}(D){:}\epsilon\},\, I \triangleright \textit{sdl})\\ &\quad \text{require}\ \textstyle\bigwedge[\text{WF}(m_k)\ \mid\ m_k \leftarrow \mathcal{L}(\textit{struct-or-union} \triangleright \bar{m})]\\ &\quad \text{return}\ (S',\, T',\, D',\, [\![\textit{struct-or-union}\ [\![\text{tag}(D)]\!]\ [\![\mathcal{L}(\textit{struct-or-union} \triangleright \bar{m})]\!]]\!])\end{aligned}$$

In particular, the notation "$\mathcal{L}(\textit{struct-or-union} \triangleright \bar{m})$" represents an application of the following unspecified Haskell function $\mathcal{L}$, which, intuitively, annotates every member $m_k \in \bar{m}$ with an appropriate offset value within the newly-introduced structure or union type:

$$\mathcal{L}[\![\cdot]\!] :: (\textit{struct-or-union} \triangleright \textit{members}) \to \textit{members}$$

Although, in the process, $\mathcal{L}$ can insert new anonymous members into the list $\bar{m}$, the relative order of all named members and their C types will be always preserved by the construction, with the possible exception of any bit field offset values appearing in the original configuration of the list:

$$\begin{aligned}&\text{LAYOUT}_1 :: \forall\ \textit{su},\, \bar{m} \Rightarrow\\ &\quad [\![\mathcal{Z}(m_k)\ \mid\ m_k \leftarrow \mathcal{L}(\textit{su} \triangleright \bar{m}),\, \mathcal{N}(m_k) \neq \epsilon]\!] = [\![\mathcal{Z}(m_k)\ \mid\ m_k \leftarrow \bar{m},\, \mathcal{N}(m_k) \neq \epsilon]\!]\end{aligned}$$

where $\mathcal{Z}(m)$ discards all offset information from a given member m as follows:

$$\begin{aligned}&\mathcal{Z}[\![\cdot]\!] :: \textit{member} \to \textit{member}\\ &\mathcal{Z}[\![t\ x\ @\ n]\!]\ \mid\ \text{BF}(t)\quad\ = [\![[\![\text{tq}(t) +\!\!+ ([\![\mathcal{B}(t)]\!] : [\![\mathcal{W}(t)]\!]\ .\ 0)]\!]\ x\ @\ 0]\!]\\ &\qquad\quad\ \mid\ \text{otherwise} = [\![t\ x\ @\ 0]\!]\end{aligned}$$

Further, for every member found in the resulting list $\bar{m}$ and a well-formed Etude object atom α whose format describes the entire structure or union type in question, the sum of the member's offset with an appropriately-converted value of α is also guaranteed to represent a meaningful object atom:

$$\begin{aligned}&\text{LAYOUT}_2 :: \forall\ \textit{su},\, \nu,\, \bar{m},\, \phi \Rightarrow\\ &\quad [\![\phi = \phi[\![\textit{su}\ \nu\ [\![\mathcal{L}(\textit{su} \triangleright \bar{m})]\!]]\!]]\!] \to\\ &\quad (\forall\ n,\, m \Rightarrow \text{WF}[\![\#n_\phi]\!] \to [\![m \in \mathcal{L}(\textit{su} \triangleright \bar{m})]\!] \to\\ &\qquad \text{WF}[\![[\![O(\phi(\mathcal{T}(m)))]\!]_\phi(\#n_\phi) +_{[\![O(\phi(\mathcal{T}(m)))]\!]}\ \#[\![O(m)]\!]_{\text{Z.}\Phi}]\!])\end{aligned}$$

Finally, in the entire member list of every well-formed structure type, no two members will be ever allocated to overlapping regions in the structure's memory image. Specifically, let A, B and C represent three sets, such that A consists of all the distinct byte offsets allocated to some named member of $\bar{m}$, B specifies all distinct member offsets allocated to named bit field members and C includes every tuple $(n,\, i)$ in which n represents the offset of some bit field member from $\bar{m}$ and i represents one of the bit indices associated with that bit field. Further, let S and W represent the total size of all named

non-bit field members and the total width of all bit field members found in the structure, respectively. Then, in every member set $\bar{m}$ constructed by $\mathcal{L}[\![\texttt{struct} \triangleright \bar{m}]\!]$, the cardinality of A must be equal to $S + |B| \times \mathcal{S}(\Psi)$ and the cardinality of C must be equal to W. Formally:

$$\begin{aligned}
&\text{LAYOUT}_3 :: \forall\, \bar{m},\, A,\, B,\, S \Rightarrow \\
&\quad [\![A = \{O(m_k) + i \mid m_k \leftarrow \mathcal{L}[\![\texttt{struct} \triangleright \bar{m}]\!],\, i \leftarrow [0 \mathinner{..} \mathcal{S}(\mathcal{T}(m_k)) - 1],\, \mathcal{N}(m_k) \neq \epsilon\}]\!] \\
&\quad [\![B = \{O(m_k) \mid m_k \leftarrow \mathcal{L}[\![\texttt{struct} \triangleright \bar{m}]\!],\, \mathcal{N}(m_k) \neq \epsilon \wedge \text{BF}(\mathcal{T}(m_k))\}]\!] \rightarrow \\
&\quad [\![S = \textstyle\sum[\mathcal{S}(\mathcal{T}(m_k)) \mid m_k \leftarrow \bar{m},\, \mathcal{N}(m_k) \neq \epsilon \wedge \neg\text{BF}(\mathcal{T}(m_k))]]\!] \rightarrow \\
&\quad [\![|A| = S + |B| \times \mathcal{S}(\Psi)]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{LAYOUT}_4 :: \forall\, \bar{m},\, C,\, W \Rightarrow \\
&\quad [\![C = \{(O(m_k),\, O(\mathcal{T}(m_k)) + j) \\
&\qquad\qquad \mid m_k \leftarrow \mathcal{L}[\![\texttt{struct} \triangleright \bar{m}]\!],\, j \leftarrow [0 \mathinner{..} \mathcal{W}(\mathcal{T}(m_k)) - 1],\, \text{BF}(\mathcal{T}(m_k))\}]\!] \rightarrow \\
&\quad [\![W = \textstyle\sum[\mathcal{W}(\mathcal{T}(m_k)) \mid m_k \leftarrow \bar{m},\, \text{BF}(\mathcal{T}(m_k))]]\!] \rightarrow \\
&\quad [\![|C| = W]\!]
\end{aligned}$$

While the precise definition of $\mathcal{L}$ may, in principle, vary between individual compiler implementations, Appendix C includes an almost-universally accepted design of this function, in which individual members are always allocated to successive byte offsets within the structure and adjacent bit field containers are merged together whenever possible, in order to reduce the overall memory footprint of the entire construction.

In a well-formed C program, all elements of the resulting member list are subject to a further semantic scrutiny by the following Haskell construction:

$$\text{WF}[\![\cdot]\!] :: \mathit{member} \rightarrow \mathit{bool}$$

In particular, every well-formed member must be associated with a non-negative offset value and a well-formed object or bit field type. Further, each bit field member must be either unnamed, or else it must specify a type with a non-zero width, since all bit field entities of the form "$t\texttt{:}0.n$" are reserved for a rather specialised purpose discussed later in this section. Finally, in all members, the sum of the member's offset with the size of its type must be no greater than the largest value representable under the implementation-defined C type "***ptrdiff_t***". Formally:

$$\begin{aligned}
\text{WF}[\![m]\!] = &((\mathcal{N}(m) \neq \epsilon \wedge (\text{OBJ}(\mathcal{T}(m)) \vee (\text{BF}(\mathcal{T}(m)) \wedge \mathcal{W}(\mathcal{T}(m)) > 0))) \vee \\
&(\mathcal{N}(m) = \epsilon \wedge \text{BF}(\mathcal{T}(m)) \wedge \mathcal{W}(\mathcal{T}(m)) = 0)) \wedge \\
&0 \leq O(m) \leq \text{lub}(\texttt{ptrdiff_t}) - \mathcal{S}(\mathcal{T}(m))
\end{aligned}$$

Otherwise, the meaning of a structure specifier whose syntax includes both an identifier component x and a list of structure declarations *sdl* depends on the present binding of x in the current tag scope T. If no tag binding for this identifier is visible in T, or if the existing binding has been introduced in some outer scope of the program, then the specifier behaves similarly to the above anonymous version of the construct, except that, during processing of *sdl*, the tag name x is also bound to a designator with the type linkage and an incomplete structure or union type tagged with the newly-introduced unique tag variable "tag(D)". At the end of the entire construct, this type is then completed in the current scope with the list of members derived from the provided list

of structure declarations and returned as part of the entity's denotation. If, on the other hand, an existing binding of x is already present in the innermost current scope T and if that binding is associated with an incomplete structure or union type of the same kind as that specified by the entity's syntax, then the existing tag variable of x is completed with the list of structure or union members derived from *sdl*. In all cases, the resulting member list is laid out and validated using the $\mathcal{L}$ and WF functions described earlier:

$$
\begin{aligned}
&\mathcal{D}_{\text{SUS}}[\![S,\, T,\, D,\, I \triangleright \textit{struct-or-union}\ x\ \texttt{\{}\textit{sdl}\texttt{\}}]\!] \\
&\quad = \text{do require } (x \notin \text{dom}(T) \vee I(T(x)) \neq I) \\
&\qquad\quad (S',\, T',\, D',\, \bar{m}) \leftarrow \mathcal{D}_{\text{SD}}(S,\, T/\{x{:}[\![\texttt{type}\ \textit{struct-or-union}\ [\![\text{tag}(D)]\!]\ \texttt{@}\ I]\!]\}, \\
&\qquad\qquad\qquad\qquad D \cup \{\text{tag}(D){:}\epsilon\},\, I \triangleright \textit{sdl}) \\
&\qquad\quad (t) \leftarrow [\![\textit{struct-or-union}\ [\![\text{tag}(D)]\!]\ [\![\mathcal{C}_{\text{M}}(t \triangleright \mathcal{L}(\textit{struct-or-union} \triangleright \bar{m}))]\!]]\!] \\
&\qquad\quad \text{require } (\text{INC}(\mathcal{T}(T'(x)))) \wedge \bigwedge[\text{WF}(m_k) \mid m_k \leftarrow \mathcal{L}(\textit{struct-or-union} \triangleright \bar{m})] \\
&\qquad\quad \text{return } (\mathcal{C}[\![t \triangleright S']\!],\, \mathcal{C}[\![t \triangleright T']\!],\, D',\, t) \\
&\quad \| \text{ do require } (x \in \text{dom}(T) \wedge I(T(x)) = I) \wedge \\
&\qquad\qquad\qquad (\text{SU}(\mathcal{T}(T(x))) \wedge \text{INC}(\mathcal{T}(T(x)))) \wedge (\text{su}(\mathcal{T}(T(x))) = \textit{struct-or-union}) \\
&\qquad\quad (S',\, T',\, D',\, \bar{m}) \leftarrow \mathcal{D}_{\text{SD}}(S,\, T,\, D,\, I \triangleright \textit{sdl}) \\
&\qquad\quad (t) \leftarrow [\![\textit{struct-or-union}\ [\![\text{tag}(\mathcal{T}(T(x)))]\!]\ [\![\mathcal{C}_{\text{M}}(t \triangleright \mathcal{L}(\textit{struct-or-union} \triangleright \bar{m}))]\!]]\!] \\
&\qquad\quad \text{require } (\text{INC}(\mathcal{T}(T'(x)))) \wedge \bigwedge[\text{WF}(m_k) \mid m_k \leftarrow \mathcal{L}(\textit{struct-or-union} \triangleright \bar{m})] \\
&\qquad\quad \text{return } (\mathcal{C}[\![t \triangleright S']\!],\, \mathcal{C}[\![t \triangleright T']\!],\, D',\, t)
\end{aligned}
$$

Observe that, since the list of structure declarations is processed directly in the scope of the surrounding specifier, a programmer may be tempted to complete it before the end of the entire list. Since the C Standard permits only a single complete definition of a given type tag within a particular scope, any such attempts are foiled in the above formalisation of structure specifiers by ensuring that the supplied tag name x remains bound to an incomplete type at the end of its list of structure declarations.

Finally, a structure or union specifier without a declaration list refers simply to the existing meaning of the designated tag name. If no such binding is visible in the current scope, then the construct introduces a new incomplete structure or union type, tagged with the unique variable "tag(D)". Otherwise, it simply denotes the existing type of x in T, provided that this type represents a structure or union of the same kind as that requested in the entity's syntax. Formally:

$$
\begin{aligned}
&\mathcal{D}_{\text{SUS}}[\![S,\, T,\, D,\, I \triangleright \textit{struct-or-union}\ x]\!] \\
&\quad = \text{do require } (x \notin \text{dom}(T)) \\
&\qquad\quad \text{return } (S,\, T \cup \{x{:}[\![\texttt{type}\ \textit{struct-or-union}\ [\![\text{tag}(D)]\!]\ \texttt{@}\ I]\!]\}, \\
&\qquad\qquad\qquad\qquad D \cup \{\text{tag}(D){:}\epsilon\},\, [\![\textit{struct-or-union}\ [\![\text{tag}(D)]\!]]\!]) \\
&\quad \| \text{ do require } (x \in \text{dom}(T) \wedge \text{SU}(\mathcal{T}(T(x))) \wedge \text{su}(\mathcal{T}(T(x))) = \textit{struct-or-union}) \\
&\qquad\quad \text{return } (S,\, T,\, D,\, \mathcal{T}(T(x)))
\end{aligned}
$$

In the concrete syntax of the language, a *structure declaration* is always represented by a non-empty sequence of type specifiers and qualifiers, that are followed by a list of one or more *structure declarators* and concluded by the delimiter "`;`". Formally, such constructs denote a list of structure or union members derived from its individual declarators in the order of their appearance within the program, ensuring that no

two named members of a given structure or union are assigned the same identifier. In Haskell:

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

struct-declaration:
specifier-qualifier-list struct-declarator-list **;**

Similarly to the ordinary C declarations from Section 5.8, any type qualifiers and specifiers associated with a particular element of the sequence are split into two homogenous sets $t\bar{q}$ and $t\bar{s}$, with the later used to derive a C type t_b, using a mapping identical to that applied in Section 5.8.3 to ordinary C declaration forms. In combination with the qualifier set $t\bar{q}$, t_b serves as the base type in a derivation of the actual member entities from the following list of structure declarators:

$$\mathcal{D}_{\text{SD}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I \triangleright \textit{struct-declaration-list}) \rightarrow M(S, S, D, \textit{members})$$

$$\begin{aligned}
&\mathcal{D}_{\text{SD}}[\![S, T, D, I \triangleright \textit{sql sdl}\,\textbf{;}\,]\!] = \text{do} \\
&\quad (t\bar{q}, t\bar{s}) \leftarrow \mathcal{D}_{\text{SQL}}(\textit{sql}) \\
&\quad (S_1, T_1, D_1, t_b\ ) \leftarrow \mathcal{D}_{\text{TS}}(S,\ T,\ D,\ I \triangleright t\bar{s}) \\
&\quad (S_2, T_2, D_2, \bar{m}\ ) \leftarrow \mathcal{D}_{\text{SR}}(S_1, T_1, D_1, I, t\bar{q} +\!\!+ t_b \triangleright \textit{sdl}) \\
&\quad \text{return } (S_2, T_2, D_2, \bar{m})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{SD}}[\![S, T, D, I \triangleright \textit{sdl sd}]\!] = \text{do} \\
&\quad (S_1, T_1, D_1, \bar{m}_1) \leftarrow \mathcal{D}_{\text{SD}}(S,\ T,\ D,\ I \triangleright \textit{sdl}) \\
&\quad (S_2, T_2, D_2, \bar{m}_2) \leftarrow \mathcal{D}_{\text{SD}}(S_1, T_1, D_1, I \triangleright \textit{sd}) \\
&\quad \text{require } (\text{dom}(\bar{m}_1) \cap \text{dom}(\bar{m}_2) = \varnothing) \\
&\quad \text{return } (S_2, T_2, D_2, \bar{m}_1 +\!\!+ \bar{m}_2)
\end{aligned}$$

The actual syntax and semantics of a structure declaration's specifier-qualifier list are identical to those of ordinary declaration specifiers described earlier in Section 5.8.1, except that no storage class specifiers can ever appear in these constructs:

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

In particular, such lists are always split into type specifier and qualifier sets by the following definition analogous to the earlier formalisation of $\mathcal{D}_{\text{DS}}$:

$$\begin{aligned}
&\mathcal{D}_{\text{SQL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\quad (\textit{specifier-qualifier-list}_{opt}) \rightarrow M(\{\textit{type-qualifier}\}, \{\textit{type-specifier}\})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{SQL}}[\![\textit{type-specifier sql}_{opt}]\!] = \text{do} \\
&\quad (t\bar{q}, t\bar{s}) \leftarrow \mathcal{D}_{\text{SQL}}(\textit{sql}_{opt}) \\
&\quad \text{require } (\textit{type-specifier} \notin t\bar{s}) \\
&\quad \text{return } (t\bar{q}, t\bar{s} \cup \{\textit{type-specifier}\})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{SQL}}[\![\textit{type-qualifier sql}_{opt}]\!] = \text{do} \\
&\quad (t\bar{q}, t\bar{s}) \leftarrow \mathcal{D}_{\text{SQL}}(\textit{sql}_{opt}) \\
&\quad \text{require } (\textit{type-qualifier} \notin t\bar{q}) \\
&\quad \text{return } (t\bar{q} \cup \{\textit{type-qualifier}\}, t\bar{s})
\end{aligned}$$

$$\mathcal{D}_{\text{SQL}}[\![]\!] = \text{return } (\varnothing, \varnothing)$$

Every structure declarator, in turn, consists of an optional *declarator* with the syntax defined later in Sections 5.8.8 and an optional constant expression, at least one of which must always be present. Formally:

struct-declarator-list :
    struct-declarator
    struct-declarator-list **,** *struct-declarator*

struct-declarator :
    declarator
    $declarator_{opt}$ **:** *constant-expression*

In this work, the meaning of all such structure declarators is captured by the following Haskell definition:

$$\mathcal{D}_{\text{SR}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I, \mathit{type} \triangleright \mathit{struct\text{-}declarator\text{-}list) \rightarrow M(S, S, D, \mathit{members})}$$

In particular, a structure declarator without the "**:** *e*" suffix denotes a simple member entity, whose name and type is derived from the declarator's syntax by an algorithm $\mathcal{D}_{\text{R}}$ defined later in Section 5.8.8. In all cases, the result must represent an object type:

$$\begin{aligned}&\mathcal{D}_{\text{SR}}[\![S, T, D, I, t_b \triangleright r]\!] = \text{do}\\ &\quad (S', T', D', t, x) \leftarrow \mathcal{D}_{\text{R}}(S, T, D, I, t_b \triangleright r)\\ &\quad \text{require } (\text{OBJ}(t))\\ &\quad \text{return } (S', T', D', [\![t\ x\ \texttt{@}\ \bot]\!])\end{aligned}$$

On the other hand, a structure declarator of the form "*r* **:** *e*" denotes a bit field member, whose name and base type *t* is derived from the declarator *r* and whose width is specified by the integral constant expression *e*. In all well-formed C programs, this base type must always represent some qualified or unqualified version of the "***int***", "***signed int***" or "***unsigned int***" type, with any type qualifiers stripped from *t* and applied directly to the member's own C type. Further, *e* must always specify a positive integer no greater than the width of *t* in magnitude. Formally:

$$\begin{aligned}&\mathcal{D}_{\text{SR}}[\![S, T, D, I, t_b \triangleright r : e]\!] = \text{do}\\ &\quad (S_1, T_1, D_1, t, \ x) \leftarrow \mathcal{D}_{\text{R}}(S, T, D, I, t_b \triangleright r)\\ &\quad (S_2, T_2, D_2, t_e, n) \leftarrow \mathcal{D}_{\text{ICE}}(S_1, T_1, D_1, I \triangleright e)\\ &\quad \text{require } (\text{unq}(t) \in \{[\![\texttt{int}]\!], [\![\texttt{signed int}]\!], [\![\texttt{unsigned int}]\!]\} \wedge 1 \leq n \leq \mathcal{W}(t))\\ &\quad \text{return } (S_2, T_2, D_2, [\![[\![\text{tq}(t) +\!\!+ [\![[\![\text{unq}(t)]\!] \texttt{:} n \texttt{.} \bot]\!]]\!]\ x\ \texttt{@}\ \bot]\!])\end{aligned}$$

In addition, a degenerate bit field declarator of the form "**:** *e*" denotes an anonymous bit field member with a C type constructed similarly to the above derivation, except that the present variant of such entities also admits bit fields of a zero width:

$$\begin{aligned}&\mathcal{D}_{\text{SR}}[\![S, T, D, I, t \triangleright : e]\!] = \text{do}\\ &\quad (S', T', D', t, n) \leftarrow \mathcal{D}_{\text{ICE}}(S, T, D, I \triangleright e)\\ &\quad \text{require } (\text{unq}(t) \in \{[\![\texttt{int}]\!], [\![\texttt{signed int}]\!], [\![\texttt{unsigned int}]\!]\} \wedge 0 \leq n \leq \mathcal{W}(t))\\ &\quad \text{return } (S', T', D', [\![[\![\text{tq}(t) +\!\!+ [\![[\![\text{unq}(t)]\!] \texttt{:} n \texttt{.} \bot]\!]]\!]\ \texttt{@}\ \bot]\!])\end{aligned}$$

The reader should observe that, initially, the byte offset of every member and the bit offset of every bit field type appearing in a structure declaration list is left unspecified

by the above definition. These parameters are only assigned proper integer values by the $\mathcal{L}$ function described earlier in this section, once the complete list of all members for the entity has been extracted from the enclosing structure or union specifier.

Finally, a comma-separated list of two or more structure declarators denotes a list of members, derived in the order of their appearance within the program as follows:

$$\begin{aligned}&\mathcal{D}_{\text{SR}}[\![S,\ T,\ D,\ I,\ t \triangleright \mathit{srl}\texttt{,}\ \mathit{sr}]\!] = \text{do}\\ &\quad (S_1,\ T_1,\ D_1,\ \bar{m}_1) \leftarrow \mathcal{D}_{\text{SR}}(S,\ \ T,\ \ D,\ \ I,\ t \triangleright \mathit{srl})\\ &\quad (S_2,\ T_2,\ D_2,\ \bar{m}_2) \leftarrow \mathcal{D}_{\text{SR}}(S_1,\ T_1,\ D_1,\ I,\ t \triangleright \mathit{sr})\\ &\quad \text{require } (\text{dom}(\bar{m}_1) \cap \text{dom}(\bar{m}_2) = \varnothing)\\ &\quad \text{return } (S_2,\ T_2,\ D_2,\ \bar{m}_1 +\!\!+\ \bar{m}_2)\end{aligned}$$

5.8.5 Enumeration Specifiers

In a concrete syntax of the language, an *enumeration specifier* is represented by the keyword "**enum**", followed by an optional identifier and an optional list of *enumerators* enclosed in braces. At least one of the *identifier* and *enumerator-list* components must be present in every such construct, as described by the following BNF grammar:

enum-specifier:
enum *identifier*$_{opt}$ **{** *enumerator-list* **}**
enum *identifier*

Similarly to structure and union specifiers, all such enumeration specifiers denote a C type. Their denotations are derived by the following Haskell function:

$$\mathcal{D}_{\text{ENS}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I \triangleright \mathit{enum\text{-}specifier}) \rightarrow M(S,\ S,\ D,\ \mathit{type})$$

In particular, if the specifier's syntax includes a list of enumerators *enl*, then that entity is used to derive a list of integer values $\bar{n}$, using an algorithm $\mathcal{D}_{\text{EN}}$ described later in this section. The entire construct denotes an enumeration type of the form "***enum*** t ν", in which ν is given a unique value "tag(D')" and t represents some implementation-defined integral type derived from $\bar{n}$. For named specifiers of the form "**enum** x **{***enl***}**", the identifier x is also bound in the current tag scope T to a type designator with that enumeration type, provided that no previous declaration of the corresponding tag name x is visible in the innermost current scope of the entity:

$$\begin{aligned}&\mathcal{D}_{\text{ENS}}[\![S,\ T,\ D,\ I \triangleright \texttt{enum \{}\mathit{enl}\texttt{\}}]\!] = \text{do}\\ &\quad (S',\ T',\ D',\ \bar{n}) \leftarrow \mathcal{D}_{\text{EN}}(S,\ T,\ D,\ I \triangleright \mathit{enl})\\ &\quad \text{return } (S',\ T',\ D' \cup \{\text{tag}(D'){:}\epsilon\},\ [\![\texttt{enum}\ [\![\mathcal{T}_{\text{ENL}}(\bar{n})]\!]\ [\![\text{tag}(D')]\!]]\!])\end{aligned}$$

$$\begin{aligned}&\mathcal{D}_{\text{ENS}}[\![S,\ T,\ D,\ I \triangleright \texttt{enum}\ x\ \texttt{\{}\mathit{enl}\texttt{\}}]\!] = \text{do}\\ &\quad (S',\ T',\ D',\ \bar{n}) \leftarrow \mathcal{D}_{\text{EN}}(S,\ T,\ D,\ I \triangleright \mathit{enl})\\ &\quad \text{require } (x \notin \text{dom}(T) \vee I(T(x)) \neq I)\\ &\quad \text{return } (S',\ T'/\{x{:}[\![\texttt{type}\ [\![\texttt{enum}\ [\![\mathcal{T}_{\text{ENL}}(\bar{n})]\!]\ [\![\text{tag}(D')]\!]]\!]\ \texttt{@}\ I]\!]\},\ D' \cup \{\text{tag}(D'){:}\epsilon\},\\ &\qquad\qquad [\![\texttt{enum}\ [\![\mathcal{T}_{\text{ENL}}(\bar{n})]\!]\ [\![\text{tag}(D')]\!]]\!])\end{aligned}$$

The C Standard does not impose any particular restriction on the base type of the resulting enumeration. In the present work, this type is derived from the list of integer values

assigned to the construct's individual enumerators by the following implementation-defined Haskell function:

$\mathcal{T}_{\text{ENL}}[\![\cdot]\!] :: [\mathit{integer}] \to \mathit{type}$

A careful reading of the C Standard reveals that $\mathcal{T}_{\text{ENL}}(\bar{n})$ must always represent a basic integral type convertible into "***signed int***" under integral promotion:

$\text{ENUM} :: \forall\, \bar{n} \Rightarrow \text{WF}(\bar{n}) \to [\![\mathcal{T}_{\text{ENL}}(\bar{n}) \in \mathit{BIT} \wedge \text{ip}(\mathcal{T}_{\text{ENL}}(\bar{n})) = [\![\textbf{\textit{signed int}}]\!]]\!]$

On the other hand, enumeration specifiers without an explicit enumerator list must always refer to an existing tag name, that, in the current scope T, is bound to a designator with an enumeration type. Such entities denote the C type of the associated designator from T without affecting the surrounding translation context:

$\mathcal{D}_{\text{ENS}}[\![S, T, D, I \triangleright \textbf{enum}\ x]\!] = \text{do}$
  require $(x \in \text{dom}(T) \wedge \text{ET}(\mathcal{T}(T(x))))$
  return $(S, T, D, \mathcal{T}(T(x)))$

When present, the list of enumerators found in an enumeration specifier must consist of one or more comma-separated *enumerator* entities, every one of which is formed from an identifier and an optional constant expression known as the enumerator's *initialiser*, as represented by the following Haskell rendition of its concrete syntax:

enumerator-list:
  enumerator
  enumerator-list **,** *enumerator*

enumerator:
  identifier
  identifier **=** *constant-expression*

Every such enumerator introduces into the current scope S a new binding of the specified variable name to a designator with an appropriate constant linkage and the plain "***int***" type, provided that no other binding of that identifier exists in S, or that the previous binding has been introduced in some outer scope and that the integer value of the identifier's new linkage is representable under the "***int***" type. Further, the denotation of an enumerator list includes a sequence of all integer values associated with the linkages of any new constants introduced into the program by its translation, so that the meaning of all C enumerators can be modelled by the following Haskell function:

$\mathcal{D}_{\text{EN}}[\![\cdot]\!] :: (\text{monad-fix}\ M) \Rightarrow (S, S, D, I \triangleright \mathit{enumerator\text{-}list}) \to M(S, S, D, [\mathit{integer}])$

In particular, for enumerators of the form "x **=** e", e must represent an integral constant expression, whose numeric value is used to form a linkage of the resulting designator:

$\mathcal{D}_{\text{EN}}[\![S, T, D, I \triangleright x \textbf{ = } e]\!] = \text{do}$
  $(S', T', D', t, n) \leftarrow \mathcal{D}_{\text{ICE}}(S, T, D, I \triangleright e)$
  require $(x \notin \text{dom}(S') \vee I(S'(x)) \neq I) \wedge (\text{glb}[\![\textbf{\textit{int}}]\!] \leq n \leq \text{lub}[\![\textbf{\textit{int}}]\!])$
  return $(S'/\{x{:}[\![\textbf{\textit{const}}\ n\ \textbf{\textit{int @}}\ I]\!]\}, T, D, [n])$

$\mathcal{D}_{\text{EN}}[\![S,\, T,\, D,\, I \triangleright \mathit{enl}\texttt{,}\ x\ \texttt{=}\ e]\!] = \text{do}$
$\quad (S_1,\, T_1,\, D_1,\, \bar{n}) \leftarrow \mathcal{D}_{\text{EN}}(S,\, T,\, D,\, I \triangleright \mathit{enl})$
$\quad (S_2,\, T_2,\, D_2,\, t,\, n) \leftarrow \mathcal{D}_{\text{ICE}}(S_1,\, T_1,\, D_1,\, I \triangleright e)$
$\quad \text{require } (x \notin \text{dom}(S_2) \vee I(S_2(x)) \neq I) \wedge (\text{glb}[\![\texttt{int}]\!] \leq n \leq \text{lub}[\![\texttt{int}]\!])$
$\quad \text{return } (S_2/\{x{:}[\![\texttt{const}\ n\ \texttt{int @}\ I]\!]\},\, T_2,\, D_2,\, \bar{n} +\!\!+ [n])$

Otherwise, if the enumerator consists entirely of an identifier without an explicit expression initialiser, then its linkage is assigned a value one greater than that of the last enumerator preceding the current one in the specifier's syntax, or 0 if such an entity appears at the very beginning of the list. Formally:

$\mathcal{D}_{\text{EN}}[\![S,\, T,\, D,\, I \triangleright x]\!] = \text{do}$
$\quad \text{require } (x \notin \text{dom}(S) \vee I(S(x)) \neq I)$
$\quad \text{return } (S/\{x{:}[\![\texttt{const}\ 0\ \texttt{int @}\ I]\!]\},\, T,\, D,\, [0])$

$\mathcal{D}_{\text{EN}}[\![S,\, T,\, D,\, I \triangleright \mathit{enl}\texttt{,}\ x]\!] = \text{do}$
$\quad (S',\, T',\, D',\, \bar{n}) \leftarrow \mathcal{D}_{\text{EN}}(S,\, T,\, D,\, I \triangleright \mathit{enl})$
$\quad \text{require } (x \notin \text{dom}(S') \vee I(S'(x)) \neq I) \wedge (\text{glb}[\![\texttt{int}]\!] \leq \text{last}(\bar{n}) + 1 \leq \text{lub}[\![\texttt{int}]\!])$
$\quad \text{return } (S'/\{x{:}[\![\texttt{const}\ [\![\text{last}(\bar{n}) + 1]\!]\ \texttt{int @}\ I]\!]\},\, T',\, D',\, \bar{n} +\!\!+ [\text{last}(\bar{n}) + 1])$

5.8.6 Tag Declarations

As mentioned at the beginning of Section 5.8, a C declaration whose syntax does not include any declarator entities is known as a *tag declaration*. The list of specifiers in every such construct must consist entirely of a single structure, union or enumeration specifier entity. Formally, their meaning is derived by the following Haskell function:

$\mathcal{D}_{\text{TAG}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \mathit{declaration\text{-}specifiers}) \rightarrow M(S,\, S,\, D,\, \mathit{type})$

In particular, if the declaration assumes the form of "*struct-or-union x*", in which x is either undeclared in the current scope T, or else introduced only in some outer scope of the program, then the construct always extend T with a new binding of x to a unique incomplete structure or union type as follows:

$\mathcal{D}_{\text{TAG}}[\![S,\, T,\, D,\, I \triangleright \mathit{struct\text{-}or\text{-}union}\ x]\!] = \text{do}$
$\quad \text{require } (x \notin \text{dom}(T) \vee I(T(x)) \neq I)$
$\quad \text{return } (S,\, T/\{x{:}[\![\texttt{type}\ \mathit{struct\text{-}or\text{-}union}\ [\![\text{tag}(D)]\!]\ \texttt{@}\ I]\!]\},\, D \cup \{\text{tag}(D){:}\epsilon\},$
$\qquad\qquad [\![\mathit{struct\text{-}or\text{-}union}\ [\![\text{tag}(D)]\!]]\!])$

Otherwise, a tag declaration must consist entirely of a single named structure or union specifier whose syntax includes a list of structure declarations, or else an enumeration specifier with a non-empty enumerator list, behaving exactly like any ordinary occurrence of such an entity described earlier in Sections 5.8.4 and 5.8.5:

$\mathcal{D}_{\text{TAG}}[\![S,\, T,\, D,\, I \triangleright \mathit{struct\text{-}or\text{-}union}\ x\ \texttt{\{}\mathit{sdl}\texttt{\}}]\!] = \mathcal{D}_{\text{SUS}}[\![S,\, T,\, D,\, I \triangleright \mathit{struct\text{-}or\text{-}union}\ x\ \texttt{\{}\mathit{sdl}\texttt{\}}]\!]$
$\mathcal{D}_{\text{TAG}}[\![S,\, T,\, D,\, I \triangleright \texttt{enum}\ x_{\mathit{opt}}\ \texttt{\{}\mathit{enl}\texttt{\}}]\!] = \mathcal{D}_{\text{ENS}}[\![S,\, T,\, D,\, I \triangleright \texttt{enum}\ x_{\mathit{opt}}\ \texttt{\{}\mathit{enl}\texttt{\}}]\!]$
$\mathcal{D}_{\text{TAG}}[\![S,\, T,\, D,\, I \triangleright \mathit{other}]\!] = \text{reject}$

Intuitively, tag declarations allow C programs to hide an existing declaration of a structure or union type, in order to replace them with a new complete or incomplete binding of the same tag name. The reader should observe that the special behaviour of these declaration forms is only applicable to structure specifiers, since, in C, enumeration types are always completed upon their initial appearance within the program.

5.8.7 Type Qualifiers

In a concrete syntax of the language, the qualification of all C types described earlier in Section 5.4 is represented by a pair of *type qualifier* keywords featured in the following Haskell definition:

type-qualifier :
    `const`
    `volatile`

type-qualifier-list :
    type-qualifier
    type-qualifier-list type-qualifier

The "*type-qualifier-list*" syntax corresponds naturally to Haskell lists of type qualifiers, which, formally, can be derived from such entities using the following function:

$$\mathcal{D}_{\mathrm{TQ}}[\![\cdot]\!] :: \textit{type-qualifier-list}_{opt} \to [\textit{type-qualifier}]$$

$$\begin{aligned}
\mathcal{D}_{\mathrm{TQ}}[\![]\!] &= \varnothing \\
\mathcal{D}_{\mathrm{TQ}}[\![\textit{type-qualifier}]\!] &= [\textit{type-qualifier}] \\
\mathcal{D}_{\mathrm{TQ}}[\![\textit{type-qualifier-list type-qualifier}]\!] &= \mathcal{D}_{\mathrm{TQ}}(\textit{type-qualifier-list}) +\!\!+ [\textit{type-qualifier}]
\end{aligned}$$

The resulting list can be applied to an almost-arbitrary C type using one of the algorithms described earlier in Section 5.4.4. However, as discussed in Section 5.4.3, the reader should keep in mind that a single C type may be only qualified with at most one of the four syntactic forms "**`const`**", "**`volatile`**", "**`const volatile`**" or "**`volatile const`**". In particular, redundant qualifications such as "**`const const`**" are explicitly forbidden by the C Standard. However, an actual enforcement of these rules is encapsulated within the type classification predicates from Section 5.4 and generally disregarded during the actual semantic analysis of C declarations.

5.8.8 Declarators

In C, *declarators* are used to introduce new object identifiers into the current variable scope *S*. Formally, their syntax is depicted by the following set of three Haskell data type definitions:

declarator :
    pointer$_{opt}$ *direct-declarator*

direct-declarator :
    identifier
    `(` *declarator* **`)`**
    direct-declarator **`[`** *constant-expression*$_{opt}$ **`]`**
    direct-declarator **`(`** *parameter-type-list* **`)`**
    direct-declarator **`(`** *identifier-list*$_{opt}$ **`)`**

pointer :
    `*` *type-qualifier-list*$_{opt}$
    `*` *type-qualifier-list*$_{opt}$ *pointer*

In all cases, every occurrence of a given declarator *r* in a C program must be associated with a list of zero or more type specifiers by the surrounding declaration. As already

mentioned in Section 5.8, these specifiers are used to construct the *base type* for the declarator, which the entity refines systematically into its eventual denotation. Specifically, the denotation of every C declarator r consists of a C type and an identifier found at the heart of its concrete syntax. Formally, these denotations are obtained using the following translation function:

$$\mathcal{D}_{\mathrm{R}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I,\ \mathit{type} \triangleright \mathit{declarator}) \rightarrow M(S,\ S,\ D,\ \mathit{type},\ \mathit{identifier})$$

In particular, a simple declarator whose syntax is formed entirely from some identifier x always denotes that identifier, assigning to it the derivation's base type t_b itself. Further, a parenthesised declarator of the form "**`(`**r**`)`**" has the same denotation as its direct declarator component r, while a *pointer declarator* of the form "*pointer r*" denotes the same identifier and type as that depicted by r, in the context of a new base type derived from its *pointer* entity as follows:

$$\begin{aligned}
\mathcal{D}_{\mathrm{R}}[\![S,\ T,\ D,\ I,\ t_b \triangleright x]\!] &= \text{return } (S,\ T,\ D,\ t_b,\ x) \\
\mathcal{D}_{\mathrm{R}}[\![S,\ T,\ D,\ I,\ t_b \triangleright \texttt{(}r\texttt{)}]\!] &= \mathcal{D}_{\mathrm{R}}(S,\ T,\ D,\ I,\ t_b \triangleright r) \\
\mathcal{D}_{\mathrm{R}}[\![S,\ T,\ D,\ I,\ t_b \triangleright \mathit{pointer}\ r]\!] &= \mathcal{D}_{\mathrm{R}}(S,\ T,\ D,\ I,\ \mathcal{D}_{\mathrm{P}}(t_b \triangleright \mathit{pointer}) \triangleright r)
\end{aligned}$$

Specifically, the notation "$\mathcal{D}_{\mathrm{P}}(t_b \triangleright$ **`*`** $\mathit{tql}_{opt})$" represents a *tql*-qualified pointer to the base type t_b, while "$\mathcal{D}_{\mathrm{P}}(t_b \triangleright$ **`*`** $\mathit{tql}_{opt}\ p)$" further refines that pointer in accordance with the syntax of p, as captured by the following recursive algorithm:

$$\mathcal{D}_{\mathrm{P}}[\![\cdot]\!] :: (\mathit{type} \triangleright \mathit{pointer}) \rightarrow \mathit{type}$$

$$\begin{aligned}
\mathcal{D}_{\mathrm{P}}[\![t \triangleright \texttt{*}\ \mathit{tql}_{opt}\ \mathit{pointer}]\!] &= \mathcal{D}_{\mathrm{P}}(\mathcal{D}_{\mathrm{TQ}}(\mathit{tql}_{opt}) +\!\!+ [\![t\texttt{*}]\!] \triangleright \mathit{pointer}) \\
\mathcal{D}_{\mathrm{P}}[\![t \triangleright \texttt{*}\ \mathit{tql}_{opt}]\!] &= \mathcal{D}_{\mathrm{TQ}}(\mathit{tql}_{opt}) +\!\!+ [\![t\texttt{*}]\!]
\end{aligned}$$

Intuitively, these rules imply that a C declaration of the form "**`int *`**x**`;`**" denotes the plain "**`int`**" directly, while "ds **`*`**$\mathit{tql}\ x$**`;`**" denotes a *tql*-qualified pointer to the C type derived from the list of declaration specifiers ds. On the other hand, an *array declarator* of the form "r**`[`**e_{opt}**`]`**" denotes an array of its element type t_b, with a length described by the optional integral constant expression e_{opt}:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{R}}[\![S,\ T,\ D,\ I,\ t_b \triangleright r\texttt{[}e_{opt}\texttt{]}]\!] = \text{do} \\
&\qquad (S_r,\ T_r,\ D_r,\ t,\ x) \leftarrow \mathcal{D}_{\mathrm{R}}(S,\ T,\ D,\ I,\ [\![t_b\texttt{[}n_{opt}\texttt{]}]\!] \triangleright r) \\
&\qquad (S_e,\ T_e,\ D_e,\ n_{opt}) \leftarrow \mathcal{D}_{\mathrm{ALE}}(S_r,\ T_r,\ D_r,\ I \triangleright e_{opt}) \\
&\qquad \text{return } (S_e,\ T_e,\ D_e,\ t,\ x)
\end{aligned}$$

A a result, a declaration of the form "ds **`*`**$\mathit{tql}\ x$**`[];`**" represents an array of *tql*-qualified pointers to the type derived from ds, while "ds **`(*`**$\mathit{tql}\ x$**`)[];`**" represents a *tql*-qualified pointer to such an array. If no expression is included in the declarator's syntax, then the construct describes an incomplete array type of an unknown length. Otherwise, the type has a length equal to the expression's integer value, as captured by the following Haskell derivation:

$$\mathcal{D}_{\mathrm{ALE}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I \triangleright \mathit{constant\text{-}expression}_{opt}) \rightarrow M(S,\ S,\ D,\ \mathit{integer}_{opt})$$

$$\begin{aligned}
&\mathcal{D}_{\mathrm{ALE}}[\![S,\ T,\ D,\ I \triangleright e]\!] = \text{do} \\
&\qquad (S',\ T',\ D',\ t,\ n) \leftarrow \mathcal{D}_{\mathrm{ICE}}(S,\ T,\ D,\ I \triangleright e) \\
&\qquad \text{return } (S',\ T',\ D',\ n)
\end{aligned}$$

$$\mathcal{D}_{\mathrm{ALE}}[\![S,\ T,\ D,\ I \triangleright \epsilon]\!] = \text{return } (S,\ T,\ D,\ \epsilon)$$

Last but not least, all declarators of the form "r`(`*ptl*`)`" and "r`()`" describe function types that return the declarator's base type t_b. If the entity includes a parameter type list, then this list is used to construct the function's prototype. Otherwise, the result is a function without a prototype and the identifier list featured in the declarator's syntax must be empty, since function declarators with non-empty identifier lists are valid only in the context of a *function definition* considered later in Section 5.10.2:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{R}}[\![S,\, T,\, D,\, I,\, t_b \triangleright r\texttt{(}ptl\texttt{)}]\!] = \mathrm{do}\\
&\quad (S_r,\, T_r,\, D_r,\, t,\, x) \leftarrow \mathcal{D}_{\mathrm{R}}(S,\, T,\, D,\, I,\, [\![t_b\texttt{(}p_{opt}\texttt{)}]\!] \triangleright r)\\
&\quad (D_p,\, p_{opt}) \leftarrow \mathcal{D}_{\mathrm{PTL}}(S_r,\, T_r,\, D_r,\, I \triangleright ptl)\\
&\quad \mathrm{return}\ (S_r,\, T_r,\, D_p,\, t,\, x)\\
&\mathcal{D}_{\mathrm{R}}[\![S,\, T,\, D,\, I,\, t_b \triangleright r\texttt{(}xl_{opt}\texttt{)}]\!] = \mathrm{do}\\
&\quad (S_r,\, T_r,\, D_r,\, t,\, x) \leftarrow \mathcal{D}_{\mathrm{R}}(S,\, T,\, D,\, I,\, [\![t_b\texttt{()}]\!] \triangleright r)\\
&\quad \mathrm{require}\ (\mathrm{list}(xl_{opt}) = \varnothing)\\
&\quad \mathrm{return}\ (S_r,\, T_r,\, D_r,\, t,\, x)
\end{aligned}$$

In the concrete syntax of the language, actual function prototypes are described by entities known as *parameter type lists*. Intuitively, a parameter type list consists of a comma-separated sequence of *parameter declarations*, every one of which is formed from a list of declaration specifiers followed by a single declarator or an *abstract declarator* entity described later in Section 5.8.9. The entire construct can be also concluded with an optional "`...`" symbol. Alternatively, when no prototype information is to be provided for a given function, the names of its arguments may be described simply by a comma-separated list of identifiers, although, as already mentioned, non-empty identifier lists are only ever permitted in the syntax of *function definitions* scrutinised separately in Section 5.10.2. Formally, both kinds of function annotations are depicted by the following set of Haskell type definitions:

parameter-type-list:
    parameter-list
    parameter-list `, ...`

parameter-list:
    parameter-declaration
    parameter-list `,` *parameter-declaration*

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
    identifier
    identifier-list `,` *identifier*

where, for convenience, every *identifier-list* entity is generally treated as a Haskell list of identifiers, derived from the entity's syntax by the following recursive algorithm:

$$\begin{aligned}
&\mathrm{list}[\![\cdot]\!] :: \textit{identifier-list}_{opt} \rightarrow [\textit{identifier}]\\
&\mathrm{list}[\![xl\texttt{,}\ x]\!] = \mathrm{list}(xl) +\!\!+ [x]\\
&\mathrm{list}[\![x]\!] \quad\ = [x]\\
&\mathrm{list}[\![]\!] \quad\ = \varnothing
\end{aligned}$$

When present in a function declarator, a parameter type list always introduces a new variable and tag scope into the program. Intuitively, this scope spans all of its parameter declarations and is promptly discarded at the end of the list. Accordingly, the denotations of such constructs consist entirely of the intended prototype p and an updated set of item definitions D', excluding the usual scope components S' and T'. In particular, every parameter type list *pl* and its suffixed form "*pl*`, ...`" denotes a prototype whose type list is derived from the list of parameters *pl* and whose abstract syntax includes the "`...`" suffix if and only if such suffix is also present in the actual source code of the program. In Haskell, this derivation is represented by the following simple translation:

$$\mathcal{D}_{\text{PTL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I \triangleright \textit{parameter-type-list}_{opt}) \to M(D, \textit{prototype}_{opt})$$

$$\begin{aligned}
&\mathcal{D}_{\text{PTL}}[\![S, T, D, I \triangleright pl]\!] = \text{do} \\
&\quad (S', T', D', \bar{t}) \leftarrow \mathcal{D}_{\text{PL}}(S, T, D, I + 1 \triangleright pl) \\
&\quad \text{return } (D', [\![\bar{t}]\!])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{PTL}}[\![S, T, D, I \triangleright pl\texttt{, ...}]\!] = \text{do} \\
&\quad (S', T', D', \bar{t}) \leftarrow \mathcal{D}_{\text{PL}}(S, T, D, I + 1 \triangleright pl) \\
&\quad \text{return } (D', [\![\bar{t}\texttt{...}]\!])
\end{aligned}$$

$$\mathcal{D}_{\text{PTL}}[\![S, T, D, I \triangleright \epsilon]\!] = \text{return } (D, [\![]\!])$$

Every parameter declaration contributes the C type of a single parameter to the function prototype under construction. Further, the identifier of every non-abstract parameter declarator is bound in the current scope S to an appropriately-typed designator for a variable object, whose linkage is obtained by the algorithm described earlier in Section 5.8.2. In all cases, the parameter must have a well-formed object, function or incomplete type, must not include any storage class specifies other than "**`register`**" and its name must be either unbound in the current scope or else bound only in some outer scope of the program. Formally:

$$\mathcal{D}_{\text{PL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I \triangleright \textit{parameter-list}) \to M(S, S, D, \textit{types})$$

$$\begin{aligned}
&\mathcal{D}_{\text{PL}}[\![S, T, D, I \triangleright ds\ r]\!] = \text{do} \\
&\quad (s\bar{c}, t\bar{q}, t\bar{s}) \leftarrow \mathcal{D}_{\text{DS}}(ds) \\
&\quad (S_1, T_1, D_1, t_b) \leftarrow \mathcal{D}_{\text{TS}}(S, T, D, I \triangleright t\bar{s}) \\
&\quad (S_2, T_2, D_2, t, x) \leftarrow \mathcal{D}_{\text{R}}(S_1, T_1, D_1, I, t\bar{q} +\!\!+ t_b \triangleright r) \\
&\quad (\ell) \leftarrow \mathcal{D}_{\text{SC}}(S_2, T_2, D_2, I, \text{pp}(t), x \triangleright s\bar{c}) \\
&\quad \text{require } (s\bar{c} \subseteq \{\texttt{register}\} \wedge (\text{OBJ}(t) \vee \text{FUN}(t) \vee \text{INC}(t))) \wedge \\
&\qquad\quad (x \notin \text{dom}(S_2) \vee I(S_2(x)) \neq I) \\
&\quad \text{return } (S_2/\{x{:}[\![\ell\ t\ \texttt{@}\ I]\!]\}, T_2, D_2/\{\nu(\ell){:}\epsilon\}, [t])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{PL}}[\![S, T, D, I \triangleright ds\ ar_{opt}]\!] = \text{do} \\
&\quad (s\bar{c}, t\bar{q}, t\bar{s}) \leftarrow \mathcal{D}_{\text{DS}}(ds) \\
&\quad (S_1, T_1, D_1, t_b) \leftarrow \mathcal{D}_{\text{TS}}(S, T, D, I \triangleright t\bar{s}) \\
&\quad (S_2, T_2, D_2, t\ ) \leftarrow \mathcal{D}_{\text{AR}}(S_1, T_1, D_1, I, t\bar{q} +\!\!+ t_b \triangleright ar_{opt}) \\
&\quad \text{require } (s\bar{c} \subseteq \{\texttt{register}\} \wedge (\text{OBJ}(t) \vee \text{FUN}(t) \vee \text{INC}(t))) \\
&\quad \text{return } (S_2, T_2, D_2, [t])
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{PL}}[\![S, T, D, I \triangleright pl\texttt{,}\ pd]\!] = \text{do} \\
&\quad (S_1, T_1, D_1, \bar{t}_1) \leftarrow \mathcal{D}_{\text{PL}}(S,\ T,\ D,\ I \triangleright pl) \\
&\quad (S_2, T_2, D_2, \bar{t}_2) \leftarrow \mathcal{D}_{\text{PL}}(S_1, T_1, D_1, I \triangleright pd) \\
&\quad \text{return } (S_2, T_2, D_2, \bar{t}_1 +\!\!+ \bar{t}_2)
\end{aligned}$$

5.8.9 Type Names

A *type name* has a structure similar to a declaration in which the declarator's identifier has been omitted from the concrete syntax, except that only one such declarator, said to be *abstract*, may be present in a given type name entity and that no storage class specifiers may appear within its declaration specifier list. Further, in a well-formed C program, the abstract declarator is often omitted entirely from the syntax of such constructs. Formally, these structures are depicted in Haskell as follows:

type-name:
    specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
    pointer
    pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
    (*abstract-declarator* **)**
    direct-abstract-declarator$_{opt}$ **[** *constant-expression*$_{opt}$ **]**
    direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

Conceptually, the meaning of all type names is derived by an algorithm similar to that described in Section 5.8.8 for ordinary C declarators, except that their denotations consist entirely of a C type, without an associated identifier entity:

$$\mathcal{D}_{\text{TN}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I \triangleright \textit{type-name}) \to M(S, S, D, \textit{type})$$

$$\begin{aligned}
&\mathcal{D}_{\text{TN}}[\![S, T, D, I \triangleright \textit{sql } ar_{opt}]\!] = \text{do} \\
&\quad (\bar{tq}, \bar{ts}) \leftarrow \mathcal{D}_{\text{SQL}}(\textit{sql}) \\
&\quad (S_1, T_1, D_1, t_b) \leftarrow \mathcal{D}_{\text{TS}}(S,\ T,\ D,\ I \triangleright \bar{ts}) \\
&\quad (S_2, T_2, D_2, t\ ) \leftarrow \mathcal{D}_{\text{AR}}(S_1, T_1, D_1, I, \bar{tq} +\!\!+ t_b \triangleright ar_{opt}) \\
&\quad \text{require } (\text{OBJ}(t) \vee \text{FUN}(t) \vee \text{INC}(t)) \\
&\quad \text{return } (S_2, T_2, D_2, t)
\end{aligned}$$

In particular, a type name without an abstract declarator denotes directly the base type of its derivation, while an abstract declarator consisting entirely of a *pointer* entity describes an appropriate pointer to that base type. If such a *pointer* is also followed by some other abstract declarator *ar*, then the entire construct has the denotation of *ar*, obtained in the context of an adjusted base type derived from that *pointer*. Finally, a parenthesised abstract declarator of the form "**(***ar***)**" has the same denotation as *ar* itself:

$$\mathcal{D}_{\text{AR}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I, \textit{type} \triangleright \textit{abstract-declarator}_{opt}) \to M(S, S, D, \textit{type})$$

$$\begin{aligned}
\mathcal{D}_{\text{AR}}[\![S, T, D, I, t_b \triangleright \epsilon]\!] &= \text{return } (S, T, D, t_b) \\
\mathcal{D}_{\text{AR}}[\![S, T, D, I, t_b \triangleright \textit{pointer}]\!] &= \text{return } (S, T, D, \mathcal{D}_{\text{P}}(t_b \triangleright \textit{pointer})) \\
\mathcal{D}_{\text{AR}}[\![S, T, D, I, t_b \triangleright \textit{pointer ar}]\!] &= \mathcal{D}_{\text{AR}}(S, T, D, I, \mathcal{D}_{\text{P}}(t_b \triangleright \textit{pointer}) \triangleright ar) \\
\mathcal{D}_{\text{AR}}[\![S, T, D, I, t_b \triangleright \textbf{(}ar\textbf{)}]\!] &= \mathcal{D}_{\text{AR}}(S, T, D, I, t_b \triangleright ar)
\end{aligned}$$

Further, array and function types can be described by abstract declarators of the form "ar_{opt}**[**e_{opt}**]**" and "ar_{opt}**(**ptl_{opt}**)**", observing that no identifier list is ever admitted into

the later syntax. A derivation of their meanings proceeds in a manner analogous to that of earlier non-abstract declarators with similar structures:

$$\begin{aligned}&\mathcal{D}_{\text{AR}}[\![S,\, T,\, D,\, I,\, t_b \triangleright ar_{opt}\texttt{[}e_{opt}\texttt{]}\,]\!] = \text{do}\\ &\quad (S_1,\, T_1,\, D_1,\, t) \leftarrow \mathcal{D}_{\text{AR}}(S,\, T,\, D,\, I,\, [\![t\texttt{[}n_{opt}\texttt{]}]\!] \triangleright ar_{opt})\\ &\quad (S_2,\, T_2,\, D_2,\, n_{opt}) \leftarrow \mathcal{D}_{\text{ALE}}(S_1,\, T_1,\, D_1,\, I \triangleright e_{opt})\\ &\quad \text{return } (S_2,\, T_2,\, D_2,\, t)\end{aligned}$$

$$\begin{aligned}&\mathcal{D}_{\text{AR}}[\![S,\, T,\, D,\, I,\, t_b \triangleright ar_{opt}\texttt{(}ptl_{opt}\texttt{)}\,]\!] = \text{do}\\ &\quad (S_1,\, T_1,\, D_1,\, t) \leftarrow \mathcal{D}_{\text{AR}}(S,\, T,\, D,\, I,\, [\![t_b\texttt{(}p_{opt}\texttt{)}]\!] \triangleright ar_{opt})\\ &\quad (D_2,\, p_{opt}) \leftarrow \mathcal{D}_{\text{PTL}}(S_1,\, T_1,\, D_1,\, I \triangleright ptl_{opt})\\ &\quad \text{return } (S_1,\, T_1,\, D_2,\, t)\end{aligned}$$

5.8.10 Type Definitions

As already mentioned in Section 5.5, an identifier with a type linkage is known as a *typedef name*. In the concrete C grammar, such entities are depicted by the following trivial syntax:

typedef-name:
identifier

A given typedef name x is considered to be well-formed only in the context of a variable scope S which binds that identifier to a designator with the "**`type`**" linkage form. Such constructs always denote the type of that designator:

$$\mathcal{D}_{\text{TDN}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \textit{typedef-name}) \rightarrow M(S,\, S,\, D,\, \textit{type})$$

$$\begin{aligned}&\mathcal{D}_{\text{TDN}}[\![S,\, T,\, D,\, I \triangleright x]\!] = \text{do}\\ &\quad \text{require } (x \in \text{dom}(S) \wedge \mathcal{L}(S(x)) = [\![\texttt{type}]\!])\\ &\quad \text{return } (S,\, T,\, D,\, \mathcal{T}(S(x)))\end{aligned}$$

Since typedef names share a single name space with ordinary C variables, yet generally appear in vastly different context within the syntax of the language, they must be distinguished from variable names by propagating certain scope information throughout the lexical analysis of C programs. A complete formalisation of this well-known technique is beyond the scope of the present work and an interested reader is referred to the abundant literature on compiler construction for a further discussion of the topic.

5.8.11 Initialised Declarators

We are now ready to define the precise structure of *initialised declarator lists* that appear in the syntax of every C declaration described at the beginning of this section. Syntactically, every such list is represented by a comma-separated sequence of one or more declarators. Each declarator in the sequence may be also followed by an optional symbol "**=**" and an entity known as an *initialiser*. Formally:

init-declarator-list:
init-declarator
init-declarator-list **,** *init-declarator*

init-declarator:
declarator
declarator **=** *initialiser*

As described earlier in Section 5.8, the meaning of all such entities is modelled by an initialiser term τ, a set of local variables V and the usual updated context components S', T' and D'. In particular, every initialised declarator extends the current scope S with a binding of its identifier x to a designator with the declarator's type and a linkage derived from any storage class specifiers appearing in its syntax. Formally, a construction of this designator is modelled by the following algorithm $\mathcal{D}_{\mathrm{X}}$:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{X}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\qquad (S,\, S,\, D,\, I,\, [\textit{storage-class-specifier}],\, \textit{type} \triangleright \textit{identifier}) \rightarrow M(\textit{designator}) \\
&\mathcal{D}_{\mathrm{X}}[\![S,\, T,\, D,\, I,\, \bar{sc},\, t \triangleright x]\!] \\
&\qquad = \text{do } (\ell) \leftarrow \mathcal{D}_{\mathrm{SC}}(S,\, T,\, D,\, I,\, t,\, x \triangleright \bar{sc}) \\
&\qquad\qquad \text{require } (x \notin \mathrm{dom}(S) \vee I(S(x)) \neq I) \\
&\qquad\qquad \text{return } [\![\ell\ t\ \texttt{@}\ I]\!] \\
&\qquad \mid\mid \text{do } (\ell) \leftarrow \mathcal{D}_{\mathrm{SC}}(S,\, T,\, D,\, I,\, t,\, x \triangleright \bar{sc}) \\
&\qquad\qquad \text{require } (x \in \mathrm{dom}(S) \wedge I(S(x)) = I \wedge \mathcal{T}(S(x)) \approx t \wedge \mathcal{L}(S(x)) = \ell \wedge \mathrm{GL}(\ell)) \\
&\qquad\qquad \text{return } [\![\ell\ [\![\mathcal{T}(S(x)) \sqcup t]\!]\ \texttt{@}\ I]\!]
\end{aligned}$$

Intuitively, the above Haskell definition implies that, whenever a given variable name x is introduced into the current scope S, it is generally assigned its declarator's type t and a linkage derived from that type in combination with any storage class specifiers appearing in the declaration's syntax, using the algorithm $\mathcal{D}_{\mathrm{SC}}$ defined earlier in Section 5.8.2. However, if the current scope S already includes a binding of that variable to a designator with the innermost scope index I, then the new declaration of x must have a C type compatible with that of its existing designator and an identical external or internal linkage form. In this case, the new binding introduced by $\mathcal{D}_{\mathrm{X}}$ has a C type obtained by a composition of t with the type of the existing designator $S(x)$, as described earlier in Section 5.4.12. Accordingly, the denotation of all well-formed initialised C declarators can be modelled by the following translation:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{IR}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\qquad (S,\, S,\, D,\, I,\, [\textit{storage-class-specifier}],\, \textit{type} \triangleright \textit{init-declarator-list}) \rightarrow \\
&\qquad M(S,\, S,\, D,\, V,\, \textit{term}_{\nu})
\end{aligned}$$

The precise meaning of an initialised C declarator for some identifier x without an explicit initialiser is determined by the designator d derived from the C type and storage class of the surrounding declaration as follows:

① If d has a type linkage, then the declarator may specify an arbitrary well-formed object, function or incomplete type for its identifier. Such declarations never introduce new variable bindings into the program's set of item definitions D.

② Otherwise, if d has a global linkage form and a function type, or else if the declaration does not include an explicit "**`extern`**" storage class specifier and its designator describes a globally-linked entity of a well-formed object or incomplete type, then its associated Etude variable $\nu(\mathcal{L}(d))$ is bound in the resulting set of item definitions D to an imported Etude item of the form "IMP x", unless a previous binding of that variable already exists in the program.

③ Further, if the entity's syntax does not include an explicit "**`extern`**" storage class specifier and if the designator d has a global linkage with an object type, then the declaration constitutes a *tentative definition* for x, represented in the resulting set of item definitions by a binding of its Etude variable to the tentative item "ϵ" and, once again, leaving any existing bindings of that variable unchanged. Observe that such declarations cannot specify an incomplete type for the identifier.

④ More so, if the declaration prescribes a private linkage for the identifier, then, in the resulting set of item definitions, its Etude variable is bound to a *default object specification*, in which every member is given the initial value of "**`0`**". As for tentative C definitions, all such privately-linked variables must have object types.

⑤ Finally, if the variable's linkage has an automatic or register form "***`auto`*** ν" or "***`register`*** ν", then x must also assume a complete object type by the end of the entire declaration. Its Etude variable ν is mapped in D to the tentative linkage form "ϵ" and, in the resulting set of local variables V, to the envelope of its C type.

In all cases, the declaration's current scope S is extended with a binding of x to the constructed designator d and the associated initialiser term τ is given the trivial Etude form "RET ()". The set of local variables V is left empty if the declaration specifies a static or type linkage form for the identifier. Formally:

$$
\begin{aligned}
&\mathcal{D}_{\text{IR}}[\![S,\, T,\, D,\, I,\, s\bar{c},\, t_b \triangleright r]\!] \\
&\quad = \text{do } (S',\, T',\, D',\, t,\, x) \leftarrow \mathcal{D}_{\text{R}}(S,\, T,\, D,\, I,\, t_b \triangleright r) \\
&\qquad\quad (d) \leftarrow \mathcal{D}_{\text{X}}(S',\, T',\, D',\, I,\, s\bar{c},\, t \triangleright x) \\
&\qquad\quad \text{require } (\mathcal{L}(d) = [\![\textbf{\textit{type}}]\!] \wedge (\text{OBJ}(\mathcal{T}(d)) \vee \text{FUN}(\mathcal{T}(d)) \vee \text{INC}(\mathcal{T}(d)))) \\
&\qquad\quad \text{return } (S'/\{x{:}d\},\, T',\, D',\, \varnothing,\, [\![\text{RET }()]\!]) \\
&\quad \| \text{ do } (S',\, T',\, D',\, t,\, x) \leftarrow \mathcal{D}_{\text{R}}(S,\, T,\, D,\, I,\, t_b \triangleright r) \\
&\qquad\quad (d) \leftarrow \mathcal{D}_{\text{X}}(S',\, T',\, D',\, I,\, s\bar{c},\, t \triangleright x) \\
&\qquad\quad \text{require } (\text{GL}(\mathcal{L}(d)) \wedge \\
&\qquad\qquad\qquad (\text{FUN}(\mathcal{T}(d)) \vee (s\bar{c} = [\![\textbf{extern}]\!] \wedge (\text{OBJ}(\mathcal{T}(d)) \vee \text{INC}(\mathcal{T}(d)))))) \\
&\qquad\quad \text{return } (S'/\{x{:}d\},\, T',\, \{\nu(\mathcal{L}(d)){:}[\![\text{IMP } x]\!]\}/D',\, \varnothing,\, [\![\text{RET }()]\!]) \\
&\quad \| \text{ do } (S',\, T',\, D',\, t,\, x) \leftarrow \mathcal{D}_{\text{R}}(S,\, T,\, D,\, I,\, t_b \triangleright r) \\
&\qquad\quad (d) \leftarrow \mathcal{D}_{\text{X}}(S',\, T',\, D',\, I,\, s\bar{c},\, t \triangleright x) \\
&\qquad\quad \text{require } (\text{GL}(\mathcal{L}(d)) \wedge \text{OBJ}(\mathcal{T}(d)) \wedge s\bar{c} \neq [\![\textbf{extern}]\!]) \\
&\qquad\quad \text{return } (S'/\{x{:}d\},\, T',\, \{\nu(\mathcal{L}(d)){:}\epsilon\}/D',\, \varnothing,\, [\![\text{RET }()]\!]) \\
&\quad \| \text{ do } (S',\, T',\, D',\, t,\, x) \leftarrow \mathcal{D}_{\text{R}}(S,\, T,\, D,\, I,\, t_b \triangleright r) \\
&\qquad\quad (d) \leftarrow \mathcal{D}_{\text{X}}(S',\, T',\, D',\, I,\, s\bar{c},\, t \triangleright x) \\
&\qquad\quad \text{require } (\mathcal{L}(d) = [\![\textbf{\textit{private}}]\!] \wedge \text{OBJ}(\mathcal{T}(d))) \\
&\qquad\quad \text{return } (S'/\{x{:}d\},\, T', \\
&\qquad\qquad\qquad D'/\{\nu(\mathcal{L}(d)){:}[\![\text{OBJ } ([\![\bar{\delta}(\mathcal{T}(d))]\!]) \text{ OF } ([\![\bar{\xi}(\mathcal{T}(d))]\!])]\!]\},\, \varnothing,\, [\![\text{RET }()]\!]) \\
&\quad \| \text{ do } (S',\, T',\, D',\, t,\, x) \leftarrow \mathcal{D}_{\text{R}}(S,\, T,\, D,\, I,\, t_b \triangleright r) \\
&\qquad\quad (d) \leftarrow \mathcal{D}_{\text{X}}(S',\, T',\, D',\, I,\, s\bar{c},\, t \triangleright x) \\
&\qquad\quad \text{require } (\mathcal{L}(d) \in \{[\![\textbf{\textit{auto}}]\!], [\![\textbf{\textit{register}}]\!]\} \wedge \text{OBJ}(\mathcal{T}(d))) \\
&\qquad\quad \text{return } (S'/\{x{:}d\},\, T',\, D'/\{\nu(\mathcal{L}(d)){:}\epsilon\},\, \{\nu(\mathcal{L}(d)){:}\bar{\xi}(\mathcal{T}(d))\},\, [\![\text{RET }()]\!])
\end{aligned}
$$

in which $\bar{\delta}(t)$ represents the default data specification for an Etude object of the given C type t, constructed by converting every element $(n, \phi, \bar{\mu})$ from t's envelope into a datum of the form $(n, \phi, \bar{\mu}, \#0_\phi)$:

$$\bar{\delta}[\![\cdot]\!] :: \mathit{type} \to \mathit{data}_\nu$$

$$\bar{\delta}[\![t]\!] = \{[\![n_k,\ \phi_k,\ \bar{\mu}_k,\ \#0_{\phi_k}]\!] \mid [\![n_k,\ \phi_k,\ \bar{\mu}_k]\!] \leftarrow \bar{\xi}(t)\}$$

On the other hand, an initialised C declaration of the form "r **=** i" has a rather less versatile semantics, since, in all such constructs, the newly-introduced identifier x must be assigned a static, automatic or register linkage form and a C type that is complete by the end of the initialiser i. If the declaration specifies a static linkage for the identifier, then i must represent a *static initialiser* for the type of x, of the form scrutinised later in Section 5.8.12.1. Any existing binding of the corresponding Etude variable in D must have the tentative form "ϵ" or an imported form "IMP x". Further, if the declaration specifies a global linkage form for its identifier, then the entire construct must appear in the file scope with the index of 0. The denotation of such an initialiser represents an Etude object specification $\bar{\delta}$ which, in the resulting set of item definitions D, always replaces any previous binding of the Etude variable x.

Otherwise, if the declaration specifies an automatic or register linkage for the newly-introduced identifier, then i must assume the form of a *dynamic initialiser* described in Section 5.8.12.2, whose meaning is represented by an Etude term that, intuitively, populates the corresponding memory-resident object upon its original introduction into the program's address space. The associated Etude variable is always bound to a tentative item "ϵ" in the resulting set of item definitions D and, in the associated set of local variables V, it is described by an Etude envelope derived from the variable's C type. Formally:

$$\begin{aligned}
&\mathcal{D}_{\text{IR}}[\![S,\ T,\ D,\ I,\ s\bar{c},\ t_b \triangleright r \mathbf{=} i]\!] \\
&\quad = \text{do } (S_r,\ T_r,\ D_r,\ t_r,\ x) \leftarrow \mathcal{D}_{\text{R}}\ (S,\ \ T,\ \ D,\ \ I,\ t_b \triangleright r) \\
&\qquad\quad (S_i,\ T_i,\ D_i,\ t_i,\ \bar{\delta}) \leftarrow \mathcal{D}_{\text{SI}}(S_r,\ T_r,\ D_r,\ I,\ t_r \triangleright i) \\
&\qquad\quad (d) \leftarrow \mathcal{D}_{\text{X}}(S_i,\ T_i,\ D_i,\ I,\ s\bar{c},\ t_i \triangleright x) \\
&\qquad\quad \text{require } (\text{SL}(\mathcal{L}(d)) \wedge \text{OBJ}(\mathcal{T}(d)) \wedge (I = 0 \vee \neg\text{GL}(\mathcal{L}(d))) \wedge \\
&\qquad\qquad\qquad (\nu(\mathcal{L}(d)) \notin \text{dom}(D_i) \vee D_i(\nu(\mathcal{L}(d))) \in \{\epsilon,\ [\![\text{IMP } x]\!]\})) \\
&\qquad\quad \text{return } (S_i/\{x{:}d\},\ T_i,\ D_i/\{\nu(\mathcal{L}(d)){:}[\![\text{OBJ } (\bar{\delta}) \text{ OF } ([\![\bar{\xi}(\mathcal{T}(d))]\!])]\!]\},\ \varnothing,\ [\![\text{RET } ()]\!]) \\
&\quad \mid\mid \text{do } (S_r,\ T_r,\ D_r,\ t_r,\ x) \leftarrow \mathcal{D}_{\text{R}}\ (S,\ \ T,\ \ D,\ \ I,\ t_b \triangleright r) \\
&\qquad\quad (S_i,\ T_i,\ D_i,\ t_i,\ \tau) \leftarrow \mathcal{D}_{\text{DI}}(S_r,\ T_r,\ D_r,\ I,\ t_r,\ \nu(\mathcal{L}(d)) \triangleright i) \\
&\qquad\quad (d) \leftarrow \mathcal{D}_{\text{X}}(S_i,\ T_i,\ D_i,\ I,\ s\bar{c},\ t_i \triangleright x) \\
&\qquad\quad \text{require } (\mathcal{L}(d) \in \{[\![\mathbf{auto}]\!],\ [\![\mathbf{register}]\!]\} \wedge \text{OBJ}(\mathcal{T}(d))) \\
&\qquad\quad \text{return } (S_i/\{x{:}d\},\ T_i,\ D_i/\{\nu(\mathcal{L}(d)){:}\epsilon\},\ \{\nu(\mathcal{L}(d)){:}\bar{\xi}(\mathcal{T}(d))\},\ \tau)
\end{aligned}$$

Finally, when a single C declaration includes two or more initialised declarator entities, then the meanings of all such declarators are derived from a common base type described by the preceding list of declaration specifiers. In the resulting unified denotation, the local variable set V represents a union of all local variables introduced by

any automatic and register declarators in the list and the term τ evaluates these declarators' initialisers in the order of their appearance within the program, as captured by the following Haskell construction:

$$\begin{aligned}&\mathcal{D}_{\text{IR}}[\![S,\ T,\ D,\ I,\ \bar{sc},\ t_b \triangleright irl\texttt{,}\ ir]\!] = \text{do}\\ &\quad (S_1,\ T_1,\ D_1,\ V_1,\ \tau_1) \leftarrow \mathcal{D}_{\text{IR}}(S,\ \ T,\ \ D,\ \ I,\ \bar{sc},\ t_b \triangleright irl)\\ &\quad (S_2,\ T_2,\ D_2,\ V_2,\ \tau_2) \leftarrow \mathcal{D}_{\text{IR}}(S_1,\ T_1,\ D_1,\ I,\ \bar{sc},\ t_b \triangleright ir)\\ &\quad \text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ [\![\texttt{LET ()} = \tau_1;\ \tau_2]\!])\end{aligned}$$

5.8.12 Initialisation

Every *initialiser* appearing in a list of initialised declarators described above may represent either a single assignment expression, or else a comma-separated list of such expressions enclosed in braces. In the C Standard, a concrete syntax of these entities is captured by the following BNF grammar:

initialiser:
- *assignment-expression*
- **{** *initialiser-list* **}**
- **{** *initialiser-list* **,** **}**

initialiser-list:
- *initialiser*
- *initialiser-list* **,** *initialiser*

Intuitively, an initialiser describes the original value of the corresponding memory-resident Etude object upon its introduction into the program's address space by the associated C declarator. Before dwelling into the somewhat elaborate semantics of these entities, it is, however, useful to begin their formal treatment with a few simple auxiliary definitions.

First of all, an initialiser whose syntax consists entirely of a single expression, optionally enclosed in braces "{}", is known as an *expression initialiser*, while all initialisers with a concrete syntax of the form "**{***il***}**" or "**{***il***,}**" are said to be *aggregate*. Further, an expression initialiser constructed from a single string literal is also distinguished as a *string initialiser form*. Formally, these three kinds of C initialisers are identified by the following Haskell predicate functions:

$$\text{EXPR}\,[\![\cdot]\!],\ \text{STR}[\![\cdot]\!],\ \text{AGGR}[\![\cdot]\!] :: \textit{initialiser} \to \textit{bool}$$

$$\begin{aligned}\text{EXPR}\ &[\![\textit{assignment-expression}]\!] &&= \text{true}\\ \text{EXPR}\ &[\![\texttt{\{}\textit{assignment-expression}\texttt{\}}]\!] &&= \text{true}\\ \text{EXPR}\ &[\![\texttt{\{}\textit{assignment-expression}\texttt{,\}}]\!] &&= \text{true}\\ \text{EXPR}\ &[\![\textit{other}]\!] &&= \text{false}\end{aligned}$$

$$\begin{aligned}\text{STR}\ \ &[\![\textit{string-literal}]\!] &&= \text{true}\\ \text{STR}\ \ &[\![\texttt{\{}\textit{string-literal}\texttt{\}}]\!] &&= \text{true}\\ \text{STR}\ \ &[\![\texttt{\{}\textit{string-literal}\texttt{,\}}]\!] &&= \text{true}\\ \text{STR}\ \ &[\![\textit{other}]\!] &&= \text{false}\end{aligned}$$

$$\begin{aligned}\text{AGGR}&[\![\texttt{\{}\textit{initialiser-list}\texttt{\}}]\!] &&= \text{true}\\ \text{AGGR}&[\![\texttt{\{}\textit{initialiser-list}\texttt{,\}}]\!] &&= \text{true}\\ \text{AGGR}&[\![\textit{other}]\!] &&= \text{false}\end{aligned}$$

In the following discussion, expression initialisers are generally converted into plain C expressions with the help of the following Haskell coercion function:

$[\![\cdot]\!] :: \textit{initialiser} \to \textit{expression}$

$$\begin{array}{ll}
[\![\textit{assignment-expression}]\!] & = \textit{assignment-expression} \\
[\![\texttt{\{}\textit{assignment-expression}\texttt{\}}]\!] & = \textit{assignment-expression} \\
[\![\texttt{\{}\textit{assignment-expression}\texttt{,\ \}}]\!] & = \textit{assignment-expression}
\end{array}$$

Similarly, a string initialiser forms can be converted into a plain string literal entity using the following auxiliary definition:

$[\![\cdot]\!] :: \textit{initialiser} \to \textit{string-literal}$

$$\begin{array}{ll}
[\![\textit{string-literal}]\!] & = \textit{string-literal} \\
[\![\texttt{\{}\textit{string-literal}\texttt{\}}]\!] & = \textit{string-literal} \\
[\![\texttt{\{}\textit{string-literal}\texttt{,\ \}}]\!] & = \textit{string-literal}
\end{array}$$

Further, to simplify the following semantic analysis of aggregate initialisers, it is usually convenient to convert their initialiser sequences into plain Haskell lists, as facilitated by the following simple recursive construction:

$\text{list}[\![\cdot]\!] :: \textit{initialiser} \to [\textit{initialiser}]$

$$\begin{array}{ll}
\text{list}[\![\texttt{\{}\textit{il}\texttt{,\ \}}]\!] & = \text{list}[\![\texttt{\{}\textit{il}\texttt{\}}]\!] \\
\text{list}[\![\texttt{\{}\textit{il}\texttt{,\ }\ \textit{i}\texttt{\}}]\!] & = \text{list}[\![\texttt{\{}\textit{il}\texttt{\}}]\!] +\!\!+ [i] \\
\text{list}[\![\texttt{\{}\textit{i}\texttt{\}}]\!] & = [i]
\end{array}$$

If such a list is intended as a description of the initial value for a structure, union or array object, its successive elements are assigned to the corresponding *initialisable members* of that object. Formally, this member list is derived from the object's C type t by the construction $\bar{m}_1(t)$, as described by the following simple algorithm:

① If t represents a structure type, then $\bar{m}_1(t)$ consists of all named members found in the type's member list $\bar{m}(t)$, with every such member further requalified with any additional type qualifiers of t itself.

② Otherwise, if t represents a union type, then the list includes only the very first named member from $\bar{m}(t)$, observing that every complete union must, by construction, include at least one such member.

③ If, on the other hand, t represents an array type of some length n, then $\bar{m}_1(t)$ represents a sequence of n anonymous members, or else an infinite sequence if t is an incomplete type. All of these members are assigned the same element type of t and successive offset values equal to the respective multiples of that type's size, so that the kth element of such an array is always placed at the byte offset $k \times \mathcal{S}(\mathcal{B}(t))$ from the beginning of the entire object.

Formally:

$\bar{m}_1[\![\cdot]\!] :: \textit{type} \to \textit{members}$

$$\begin{array}{lll}
\bar{m}_1[\![t]\!] & \mid \text{SU}(t) \wedge \text{su}(t) = [\![\texttt{struct}]\!] & = [m_k \mid m_k \leftarrow \text{tq}(t) \cup \bar{m}(t),\ \mathcal{N}(m_k) \neq \epsilon] \\
& \mid \text{SU}(t) \wedge \text{su}(t) = [\![\texttt{union}]\!] & = \text{take}(1, [m_k \mid m_k \leftarrow \text{tq}(t) \cup \bar{m}(t),\ \mathcal{N}(m_k) \neq \epsilon]) \\
& \mid \text{ARR}(t) \wedge \text{length}(t) \neq \epsilon & = [[\![\mathcal{B}(t)]\!]\ \texttt{@}\ [\![k \times \mathcal{S}(\mathcal{B}(t))]\!] \mid k \leftarrow [0 \mathinner{..} \text{length}(t) - 1]] \\
& \mid \text{ARR}(t) \wedge \text{length}(t) = \epsilon & = [[\![\mathcal{B}(t)]\!]\ \texttt{@}\ [\![k \times \mathcal{S}(\mathcal{B}(t))]\!] \mid k \leftarrow [0 \mathinner{..}]]
\end{array}$$

Using these seven auxiliary definitions, the meanings of all C initialiser forms can be given a concise semantic definition presented in Sections 5.8.12.1 and 5.8.12.2 for the static and dynamic variants of these entities, respectively.

5.8.12.1 Static Initialisers

As suggested earlier in Section 5.8.11, the meaning of every static C initialiser is represented by an Etude data specification set, which determines the initial value of a corresponding statically-linked object at the beginning of the program's execution. Further, under certain conditions discussed shortly, such initialisers may adjust the C types of their associated declarators, which, accordingly, are also included in their complete denotations. In particular, the meaning of every static initialiser form is determined from its structure and the C type t of the associated designator d in the following manner:

① If t has a bit field type, then the initialiser must assume an expression form that is derived from a single static initialiser expression e of an arithmetic type, as described earlier in Section 5.7.3.3. The expression's atomic denotation is converted into the type t and packed using the familiar bit field constructor $\mathcal{P}$ into a single datum $(0, \Psi, \bar{\mu}, \#0_\Psi)$, in which Ψ is the standard bit field format and $\bar{\mu}$ represents the set of access attributes appropriate for the declarator's type.

② Similarly, if t has any other arithmetic or pointer type, then the initialiser must assume a static initialiser expression form, subject to the same typing constraints as those described earlier in Section 5.7.2 for function argument assignments. Its denotation consists of a single datum $(0, \phi, \bar{\mu}, \phi'_\phi(\alpha))$, in which ϕ' and $\bar{\mu}$ represent the format and access attributes of t, while α and ϕ are determined by the atomic denotation of the initialiser's expression and the Etude format of its unconverted type, respectively.

③ Otherwise, a string initialiser can be used to prepare the initial content of a complete or incomplete array type, provided that the array's length, if known, is greater or equal to one less then the length of the string literal's type, and that both t and the string literal itself represent arrays of character types, or else both depict arrays with elements that are compatible with a qualified or unqualified version of the implementation-defined type "**`wchar_t`**", so that all of the literal's characters with the possible exception of its implicit NUL terminator can be accommodated by the targeted object. In both cases, the resulting denotation consists of the data specification set derived from the string literal's syntax, except that, if the targeted array object is longer than its initialiser, then any remaining elements of the array are implicitly assigned the default data specification described earlier in Section 5.8.11. If the declarator has an incomplete type, then the initialiser also completes that array with the length of the specified string literal.

④ In all other cases, if t represents a non-scalar object type or an array of an unknown length, then its initialiser must always have an aggregate form, whose elements $\bar{i}$

are assigned to the successive initialisable members of t, forming a set of data specifications $\bar{\delta}$ using an algorithm $\mathcal{D}_{\text{SIL}}$ described later in this section. If t represents an incomplete array type prior to initialisation, then, in the resulting denotation, that type is completed with the length n equal to the number of t's members initialised by $\bar{i}$. Otherwise, $\bar{i}$ must not specify more elements than those required for a complete initialisation of t. If it specifies fewer elements than required, then, in $\bar{\delta}$, any remaining members are assigned their default data specifications.

In Haskell, the above algorithm can be implemented concisely as follows:

$$\mathcal{D}_{\text{SI}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D,\ I,\ \mathit{type} \triangleright \mathit{initialiser}) \rightarrow M(S,\ S,\ D,\ \mathit{type},\ \mathit{data}_V)$$

$$\begin{aligned}
&\mathcal{D}_{\text{SI}}[\![S,\ T,\ D,\ I,\ t \triangleright i]\!] \\
&\quad = \text{do require } (\text{BF}(t) \wedge \text{EXPR}(i)) \\
&\qquad\quad (S',\ T',\ D',\ t',\ \alpha) \leftarrow \mathcal{D}_{\text{SIE}}(S,\ T,\ D,\ I \triangleright i) \\
&\qquad\quad \text{require } (\text{AT}(t')) \\
&\qquad\quad \text{return } (S',\ T',\ D',\ t,\ [\![(0,\ \Psi,\ [\![\bar{\mu}(t)]\!],\ [\![\mathcal{P}[\![t \triangleright \#0_\Psi,\ [\![\phi(t)]\!]_{[\![\phi(t')]\!]}(\alpha)]\!]]\!])]\!]) \\
&\quad \| \text{ do require } (\text{AT}(t) \wedge \neg\text{BF}(t) \vee (\text{PTR}(t))) \wedge (\text{EXPR}(i)) \\
&\qquad\quad (S',\ T',\ D',\ t',\ \alpha) \leftarrow \mathcal{D}_{\text{SIE}}(S,\ T,\ D,\ I \triangleright i) \\
&\qquad\quad \text{require } (\text{AT}(t) \wedge \text{AT}(t')) \vee \\
&\qquad\qquad\qquad (\text{PTR}(t) \wedge \text{PTR}(t') \wedge ((\text{unq}(\mathcal{B}(t)) \approx \text{unq}(\mathcal{B}(t'))) \vee \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\text{VT}(\mathcal{B}(t')) \wedge (\text{OBJ}(\mathcal{B}(t)) \vee \text{INC}(\mathcal{B}(t)))) \vee \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\text{VT}(\mathcal{B}(t)) \wedge (\text{OBJ}(\mathcal{B}(t')) \vee \text{INC}(\mathcal{B}(t'))))) \wedge \\
&\qquad\qquad\qquad\qquad\qquad (\text{tq}(\mathcal{B}(t')) \subseteq \text{tq}(\mathcal{B}(t)))) \vee \\
&\qquad\qquad\qquad (\text{PTR}(t) \wedge \text{NULL}(S,\ T,\ D,\ I \triangleright i)) \\
&\qquad\quad \text{return } (S',\ T',\ D',\ t,\ [\![(0,\ \phi(t),\ \bar{\mu}(t),\ [\![\phi(t)]\!]_{[\![\phi(t')]\!]}(\alpha))]\!]) \\
&\quad \| \text{ do require } (\text{STR}(i) \wedge \text{ARR}(t)) \\
&\qquad\quad (t_i,\ \bar{\delta}) \leftarrow \mathcal{D}_{\text{SC}}(i) \\
&\qquad\quad \text{require } ((\text{CHR}(\mathcal{B}(t)) \wedge \text{CHR}(\mathcal{B}(t_i))) \vee (\text{unq}(\mathcal{B}(t)) \approx \mathcal{B}(t_i) \approx [\![\texttt{wchar_t}]\!])) \wedge \\
&\qquad\qquad\qquad (\text{length}(t) = \epsilon \vee \text{length}(t) \geq \text{length}(t_i) - 1) \\
&\qquad\quad \text{return } (S,\ T,\ D,\ \mathcal{C}_{\text{A}}(t,\ \text{length}(t_i)), \\
&\qquad\qquad\qquad \bar{\delta} \cup \bigcup[\bar{\delta}(\mathcal{T}(m_k)) \oplus \mathcal{O}(m_k) \mid m_k \leftarrow \text{drop}(|\bar{\delta}|,\ \bar{m}_{\text{I}}(\mathcal{C}_{\text{A}}(t,\ \text{length}(t_i))))]) \\
&\quad \| \text{ do require } ((\neg\text{SCR}(t) \wedge \text{OBJ}(t) \vee \text{ARR}(t)) \wedge \text{AGGR}(i)) \\
&\qquad\quad (S',\ T',\ D',\ \bar{\delta},\ n,\ \bar{i}') \leftarrow \mathcal{D}_{\text{SIL}}(S,\ T,\ D,\ I,\ \bar{m}_{\text{I}}(t) \triangleright \text{list}(i)) \\
&\qquad\quad \text{require } (\bar{i}' = \varnothing) \\
&\qquad\quad \text{return } (S',\ T',\ D',\ \mathcal{C}_{\text{A}}(t,\ n), \\
&\qquad\qquad\qquad \mathcal{P}(\bar{\delta} \cup \bigcup[\bar{\delta}(\mathcal{T}(m_k)) \oplus \mathcal{O}(m_k) \mid m_k \leftarrow \text{drop}(n,\ \bar{m}_{\text{I}}(\mathcal{C}_{\text{A}}(t,\ n)))])
\end{aligned}$$

The actual completion of an array's length is performed in the above definition using the following natural Haskell construction:

$$\mathcal{C}_{\text{A}}[\![\cdot]\!] :: (\mathit{type},\ \mathit{integer}) \rightarrow \mathit{type}$$

$$\begin{aligned}
\mathcal{C}_{\text{A}}[\![t,\ n]\!] \mid\ & (\text{ARR}(t) \wedge \text{length}(t) = \epsilon) &&= [\![[\![\mathcal{B}(t)]\!]\ \texttt{[}n\texttt{]}]\!] \\
\mid\ & \text{otherwise} &&= t
\end{aligned}$$

Further, in the above formalisation of static initialisers, all otherwise uninitialised members of a structure, union or array type are assigned their correct default values by computing a union of the set $\bar{\delta}$ derived from any explicitly specified initialisers for the object

with the default data specification sets $\bar{\delta}(\mathcal{T}(m_k)) \oplus \mathcal{O}(m_k)$ for every remaining uninitialised but initialisable member m_k. Intuitively, "$\bar{\delta} \oplus n$" simply relocates every datum $(n_k,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k) \in \bar{\delta}$ to a new offset $n_k + n$ within the surrounding structure, union or array, as depicted by the following list comprehension:

$$[\![\cdot]\!]\ \oplus [\![\cdot]\!] :: \mathit{data}_\nu \rightarrow \mathit{integer} \rightarrow \mathit{data}_\nu$$

$$[\![\bar{\delta}]\!] \oplus [\![n]\!] = \{[\![n_k + n,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k]\!] \mid [\![n_k,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k]\!] \leftarrow \bar{\delta}\}$$

Finally, the initialisation algorithm must *compact* all initialisers for every group of bit field members that, in $\bar{\delta}$, are placed by the structure layout algorithm $\mathcal{L}$ from Section 5.8.4 at the same offset into the entire object, into a single datum entity. In the above definition of $\mathcal{D}_{\mathrm{SI}}$, this process is represented by the notation "$\mathcal{P}(\bar{\delta})$". In particular, if $\bar{\delta}$ contains two or more elements with the same format ϕ and byte offset value n, then, in the compacted set of data specification $\mathcal{P}(\bar{\delta})$, all such elements are combined into a single tuple $(n,\ \phi,\ \bar{\mu}',\ \alpha')$, in which $\bar{\mu}'$ represents an intersection of the individual access attribute sets assigned to each of these overlapping elements and α' depicts a single Etude atom with a value derived by an implementation-defined construction $\mathcal{P}(\bar{\alpha})$. In this chapter, such atoms are introduced by the following Haskell function:

$$\mathcal{P}[\![\cdot]\!] :: \mathit{atoms}_\nu \rightarrow \mathit{atom}_\nu$$

Although the precise semantics of this construct are left unspecified in the generic fragment of the language, every implementation of a C compiler must ensure that, for every well-formed bit field type $t_k \in \bar{t}$ and atom $\alpha_k \equiv \mathcal{P}[\![t_k \triangleright \#0_\Psi,\ \alpha'_k]\!] \in \bar{\alpha}$, the atom's original value α'_k can be always retrieved from $\mathcal{P}(\bar{t} \triangleright \bar{\alpha})$ using the standard bit field extraction operation $\mathcal{U}(t_k \triangleright \mathcal{P}(\bar{t} \triangleright \bar{\alpha}))$, provided that all types in $\bar{t}$ specify non-overlapping bit ranges. Formally:

$$\begin{aligned}
&\mathrm{PACK_I} :: \forall\ \bar{\alpha},\ \bar{\alpha}'\ \bar{t} \Rightarrow \\
&\quad \mathrm{WF}(\bar{\alpha}) \rightarrow \mathrm{WF}(\bar{\alpha}') \rightarrow \\
&\quad [\![\mathrm{length}(\bar{\alpha}) = \mathrm{length}(\bar{\alpha}') = \mathrm{length}(\bar{t})]\!] \rightarrow \\
&\quad [\![|\bigcup[\{\mathcal{O}(t_k) \mathinner{..} \mathcal{O}(t_k) + \mathcal{W}(t_k) - 1\} \mid t_k \leftarrow \bar{t}]| = \textstyle\sum[\mathcal{W}(t_k) \mid t_k \leftarrow \bar{t}]]\!] \rightarrow \\
&\quad (\forall\ t_k \Rightarrow [\![t_k \in \bar{t}]\!] \rightarrow [\![\mathrm{BF}(t_k)]\!]) \rightarrow \\
&\quad (\forall\ \alpha_k,\ \alpha'_k,\ t_k \Rightarrow [\![(\alpha_k,\ \alpha'_k,\ t_k) \in \bar{\alpha}|\bar{\alpha}'|\bar{t}]\!] \rightarrow [\![\alpha_k]\!] \equiv [\![\mathcal{P}[\![t_k \triangleright \#0_\Psi,\ \alpha'_k]\!]]\!]) \rightarrow \\
&\quad (\forall\ \alpha_k,\ \alpha'_k,\ t_k \Rightarrow [\![(\alpha_k,\ \alpha'_k,\ t_k) \in \bar{\alpha}|\bar{\alpha}'|\bar{t}]\!] \rightarrow [\![\alpha'_k]\!] \equiv [\![\mathcal{U}[\![t_k \triangleright \alpha_k]\!]]\!])
\end{aligned}$$

Using this auxiliary definition, compaction of a well-formed Etude data specification set can be formalised concisely as the following recursive algorithm:

$$\mathcal{P}[\![\cdot]\!] :: \mathit{data}_\nu \rightarrow \mathit{data}_\nu$$

$$\begin{aligned}
\mathcal{P}[\![\bar{\delta}]\!] \mid (\bar{\delta} = \varnothing)\ &= \varnothing \\
\mid \mathrm{otherwise} &= \{[\![n,\ \phi,\ \bar{\mu}',\ \alpha']\!]\} \cup \mathcal{P}(\bar{\delta}')
\end{aligned}$$

$$\text{where } [\![n,\ \phi,\ \bar{\mu},\ \alpha]\!] = \min(\bar{\delta})$$

$$\begin{aligned}
\bar{\mu}' &= \textstyle\bigcap[\bar{\mu}_k \mid [\![n_k,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k]\!] \leftarrow \bar{\delta},\ n_k = n \wedge \phi_k = \phi] \\
\alpha' &= \mathcal{P}[\alpha_k \mid [\![n_k,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k]\!] \leftarrow \bar{\delta},\ n_k = n \wedge \phi_k = \phi] \\
\bar{\delta}' &= \{[\![n_k,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k]\!] \mid [\![n_k,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k]\!] \leftarrow \bar{\delta},\ n_k \neq n \vee \phi_k \neq \phi\}
\end{aligned}$$

Last but not least, the actual meaning of an initialiser list $\bar{i}$ for a given sequence of aggregate members $\bar{m}$ is represented by the following Haskell function:

$$\mathcal{D}_{\text{SIL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S, S, D, I, \mathit{members} \triangleright [\mathit{initialiser}]) \rightarrow M(S, S, D, \mathit{data}_{\text{V}}, \mathit{integer}, [\mathit{initialiser}])$$

$$\begin{aligned}
&\mathcal{D}_{\text{SIL}}[\![S, T, D, I, \bar{m} \triangleright \bar{i}]\!] \\
&\quad = \text{do require } (\bar{m} = \varnothing \vee \bar{i} = \varnothing) \\
&\qquad\quad \text{return } (S, T, D, \varnothing, 0, \bar{i}) \\
&\quad \| \text{ do require } (\text{OBJ}(\mathcal{T}(m_0)) \wedge \neg\text{SCR}(\mathcal{T}(m_0)) \wedge \neg\text{AGGR}(\text{head}(\bar{i}))) \\
&\qquad\quad (S_1, T_1, D_1, \bar{\delta}_1, n_1, \bar{i}_1) \leftarrow \mathcal{D}_{\text{SIL}}(S, T, D, I, \bar{m}_1(\mathcal{T}(m_0)) \triangleright \bar{i}) \\
&\qquad\quad (S_2, T_2, D_2, \bar{\delta}_2, n_2, \bar{i}_2) \leftarrow \mathcal{D}_{\text{SIL}}(S_1, T_1, D_1, I, \text{tail}(\bar{m}) \triangleright \bar{i}_1) \\
&\qquad\quad \text{return } (S_2, T_2, D_2, \mathcal{P}(\bar{\delta}_1 \oplus O(m_0)) \cup \bar{\delta}_2, 1 + n_2, \bar{i}_2) \\
&\quad \| \text{ do } (S_1, T_1, D_1, t, \bar{\delta}_1) \leftarrow \mathcal{D}_{\text{SI}}(S, T, D, I, \mathcal{T}(m_0) \triangleright \text{head}(\bar{i})) \\
&\qquad\quad (S_2, T_2, D_2, \bar{\delta}_2, n, \bar{i}') \leftarrow \mathcal{D}_{\text{SIL}}(S_1, T_1, D_1, I, \text{tail}(\bar{m}) \triangleright \text{tail}(\bar{i})) \\
&\qquad\quad \text{return } (S_2, T_2, D_2, \bar{\delta}_1 \oplus O(m_0) \cup \bar{\delta}_2, 1 + n, \bar{i}') \\
&\text{where } m_0 = \text{head}(\bar{m})
\end{aligned}$$

Intuitively, the above inductive algorithm processes all members $m_k \in \bar{m}$ in the order of their appearance in the list, progressively applying one or more elements from $\bar{i}$ as an initialiser for that member. In particular, if m_k represents a member of a non-scalar object type and the list $\bar{i}$ begins with a non-aggregate initialiser, then the construction obtains as many successive elements from $\bar{i}$ as required for all the initialisable members of m_k, using a recursive application of that initialisable member list to $\bar{i}$. Any initialisers remaining in $\bar{i}$ at the end of this process are then applied to the following members in $\bar{m}$. Otherwise, the very first element of $\bar{i}$ must represent a complete well-formed initialiser for the type of m_k and that initialiser alone is applied to the member's C type using the algorithm $\mathcal{D}_{\text{SI}}$ described earlier in this section. In all cases, the resulting denotation is formed from a compacted union of appropriately-relocated data specification sets for all explicitly-initialised members of $\bar{m}$, together with the actual number of these members and the list of any unused initialisers from $\bar{i}$. The algorithm terminates when either of the two lists $\bar{m}$ or $\bar{i}$ has been exhausted. If the end of $\bar{m}$ is reached first, then all members have been initialised completely and the remaining elements of $\bar{i}$ are made available for use with any following elements of the surrounding structure or array; otherwise, if $\bar{i}$ was exhausted before the end of $\bar{m}$ has been reached, then any remaining members of the structure will be eventually assigned their default data specifications as part of the earlier algorithm $\mathcal{D}_{\text{SI}}$.

5.8.12.2 Dynamic Initialisers

Last but not least, every well-formed *dynamic C initialiser* denotes a monadic Etude term whose evaluation is intended to populate the initial content of some memory-resident object located at a given address α within the program's memory image. If the initialiser has an expression form, then this term behaves as if that expression was

assigned to a function argument with a C type of the associated declarator, subject to the same constraints as those discussed earlier in Section 5.7.2 for such assignments. Otherwise, if an aggregate initialiser is used to initialise a non-scalar l-value, or if a string initialiser is utilised in preparation of an appropriately-typed array object, then the construct represents a *dynamic aggregate initialiser* and its semantics are defined later in this section. The precise meaning of every dynamic initialiser form is determined from the entity's syntax and the type t of the associated C declarator by the following algorithm:

① If the surrounding C declarator has a non-bit field scalar type, then the initialiser's value is evaluated and stored in the l-value described by the parameter α, using the familiar Etude constructor "SET_I". Observe that, in a well-formed C program, t will never assume a bit field form.

② Otherwise, if t represents a complete structure or union type and the initialiser has a non-aggregate form of an expression whose type is compatible with a qualified or unqualified variant of t, then the initialisation is performed using the construction "$\text{set}_\text{I}\ (t \triangleright \alpha)$ to $(t' \triangleright \alpha')$" discussed earlier in Section 5.7.2.

③ Finally, if t has a non-scalar type and the initialiser has an aggregate form, or else if t represents an array and the initialiser depicts a single string literal entity, then its *dynamic aggregate* meaning is derived using the algorithm $\mathcal{D}_{\text{DAI}}$ defined later in this section. Every such dynamic aggregate initialiser denotes a set of Etude data specifications $\bar{\delta}$ similar to that derived earlier from static initialisers of the same structure. In the resulting denotation, every datum $(n_k,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k) \in \bar{\delta}$ is converted into a binding of the form "$() = \mathsf{SET}_\mathsf{I}\ [[\![O(\phi_k)]\!]_{[\![\phi(t)]\!]}(\alpha) +_{[\![O(\phi_k)]\!]} \#n_{kZ.\Phi},\ \bar{\mu}_k]_{\phi_k}\ \mathsf{TO}\ \alpha_k$", which, intuitively, stores the value of α_k in a memory-resident object found at an offset n_k into the address space region located at the address α. All such bindings are then placed within a single Etude group, leaving the precise order of their evaluation unspecified.

In the first two cases, the initialiser may represent an arbitrary C expression, so that all temporary variables required for its evaluation must be introduced into the program's address space at the beginning of the construct and purged before proceeding with the translation of any following entities in the program. In Section 5.7, all such temporary objects were represented in an expression's denotation by its local variable set V, so that, following the principles from Section 4.6, the required list of object environment extensions must be modelled in the initialiser's denotation by a sequence of Etude terms "$\mathsf{NEW}\ (\bar{\xi}_k)$" for every envelope $\bar{\xi}_k$ bound to some variable in V. Formally, this term sequence is introduce by a *scope constructor* combinator $\mathcal{C}(V)$, which, intuitively, maps every binding $v_k{:}\bar{\xi}_k \in V$ to a partially-applied Etude term "$\mathsf{LET}\ v_k = \mathsf{NEW}\ (\bar{\xi}_k);$" and composes these terms into a single Etude construct using the list operator "$\odot$" defined in Appendix A. Further, the dual *scope destructor* combinator $\mathcal{D}(V)$ purges all such temporary objects from the program's address space once they are no longer required

for its execution, which, in Etude, is modelled by an analogous sequence of "DEL $(\bar{\xi}_k)$" term forms, applied in the reverse order of the corresponding objects' allocation, so that the minimal requirements of the address space contraction operations are always satisfied as described in Section 4.5. In Haskell, these two combinators are represented by the following pair of functions:

$$\mathcal{C}[\![\cdot]\!],\ \mathcal{D}[\![\cdot]\!] :: V \to (term_V \to term_V)$$

$$\mathcal{C}[\![V]\!] = \odot[[\![\texttt{LET}\ v_k\ \texttt{=}\ \texttt{NEW}\ (\bar{\xi}_k)\texttt{;}]\!] \mid v_k{:}\bar{\xi}_k \leftarrow V]$$
$$\mathcal{D}[\![V]\!] = \odot(\text{reverse}[[\![\texttt{LET}\ ()\ \texttt{=}\ \texttt{DEL}\ (\bar{\xi}_k)\texttt{;}]\!] \mid v_k{:}\bar{\xi}_k \leftarrow V])$$

Using this pair of auxiliary definitions, the meaning of every well-formed dynamic C initialiser can be formalised as follows:

$$\mathcal{D}_{\text{DI}}[\![\cdot]\!] :: (\text{monad-fix}\ M) \Rightarrow$$
$$(S,\ S,\ D,\ I,\ type,\ atom_V \triangleright initialiser) \to M(S,\ S,\ D,\ type,\ term_V)$$

$$\mathcal{D}_{\text{DI}}[\![S,\ T,\ D,\ I,\ t,\ \alpha \triangleright i]\!]$$

$$\begin{aligned}
&= \text{do require } (\text{SCR}(t) \wedge \neg\text{BF}(t) \wedge \text{EXPR}(i))\\
&\quad (S',\ T',\ D',\ V,\ t',\ \tau) \leftarrow \mathcal{D}_{\text{V}}(S,\ T,\ D,\ I \triangleright i)\\
&\quad \text{require } (\text{AT}(t) \wedge \text{AT}(t')) \vee\\
&\qquad (\text{PTR}(t) \wedge \text{PTR}(t') \wedge ((\text{unq}(\mathcal{B}(t)) \approx \text{unq}(\mathcal{B}(t'))) \vee\\
&\qquad\qquad ((\text{OBJ}(\mathcal{B}(t)) \vee \text{INC}(\mathcal{B}(t))) \wedge \text{VT}(\mathcal{B}(t'))) \vee\\
&\qquad\qquad ((\text{OBJ}(\mathcal{B}(t')) \vee \text{INC}(\mathcal{B}(t'))) \wedge \text{VT}(\mathcal{B}(t)))))) \vee\\
&\qquad (\text{PTR}(t) \wedge \text{NULL}(S,\ T,\ D,\ I \triangleright i))\\
&\quad \text{return } (S',\ T',\ D',\ t,\\
&\qquad [[\mathcal{C}(V)]];\\
&\qquad \texttt{LET}\ \text{T}_1\ \texttt{=}\ \tau\texttt{;}\\
&\qquad \texttt{LET}\ ()\ \texttt{=}\ \text{SET}_{\text{I}}\ [\alpha,\ [\![\bar{\mu}(t)]\!]]_{[\![\phi(t)]\!]}\ \texttt{TO}\ ([\![\phi(t)]\!]_{[\![\phi(t')]\!]}(\text{T}_1))\texttt{;}\\
&\qquad [\![\mathcal{D}(V)]\!];\\
&\qquad \texttt{RET}\ ()]])
\end{aligned}$$

$$\begin{aligned}
&\|\ \text{do require } (\text{SU}(t) \wedge \text{OBJ}(t) \wedge \neg\text{AGGR}(i))\\
&\quad (S',\ T',\ D',\ V,\ t',\ \tau) \leftarrow \mathcal{D}_{\text{V}}(S,\ T,\ D,\ I \triangleright i)\\
&\quad \text{require } (\text{unq}(t) \approx \text{unq}(t'))\\
&\quad \text{return } (S',\ T',\ D',\ t,\\
&\qquad [[\mathcal{C}(V)]];\\
&\qquad \texttt{LET}\ \text{T}_1\ \texttt{=}\ \tau\texttt{;}\\
&\qquad \texttt{LET}\ ()\ \texttt{=}\ [\![\text{set}_{\text{I}}\ (t \triangleright \alpha)\ \text{to}\ [\![t' \triangleright \text{T}_1]\!]]\!];\\
&\qquad [\![\mathcal{D}(V)]\!];\\
&\qquad \texttt{RET}\ ()]])
\end{aligned}$$

$$\begin{aligned}
&\|\ \text{do require } (\neg\text{SCR}(t) \wedge \text{AGGR}(i)) \vee (\text{ARR}(t) \wedge \text{STR}(i))\\
&\quad (S',\ T',\ D',\ t',\ \bar{\delta}) \leftarrow \mathcal{D}_{\text{DAI}}(S,\ T,\ D,\ I,\ t \triangleright i)\\
&\quad (\bar{\beta}) \leftarrow [[\![()\ \texttt{=}\ \text{SET}_{\text{I}}\ [[\![O(\phi_k)]\!]_{[\![\phi(t)]\!]}(\alpha) +_{[\![O(\phi_k)]\!]} \#n_{k\text{Z}.\Phi},\ \bar{\mu}_k]_{\phi_k}\ \texttt{TO}\ \alpha_k]\!]\\
&\qquad\qquad \mid [\![(n_k,\ \phi_k,\ \bar{\mu}_k,\ \alpha_k)]\!] \leftarrow \bar{\delta}]\\
&\quad \text{return } (S',\ T',\ D',\ t',\ [\![\texttt{LET}\ \bar{\beta};\ \texttt{RET}\ ()]\!])
\end{aligned}$$

In particular, if a dynamic aggregate initialiser is used to populate an object of a scalar C type, then the construction has a meaning identical to its earlier static initialiser interpretation from Section 5.8.12.1, as epitomised by the algorithm $\mathcal{D}_{\text{SI}}$. Further, in the

present context, a dynamic string initialiser also behaves similarly to its earlier static variant, except that the resulting object is never padded with the implicit "`0`" values, even when such a string is used to populate a dynamically-allocated array object of a length greater than itself, keeping in mind that the implicit NUL terminator is still retained at the end of every string literal's denotation. In every other form of a dynamic aggregate initialiser, the list of all initialiser entities found in the construct's syntax is applied recursively to the object's initialisable members using the algorithm $\mathcal{D}_{\text{DIL}}$ defined at the end of the present section. Formally:

$$\begin{aligned}
&\mathcal{D}_{\text{DAI}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\quad (S,\ S,\ D,\ I,\ \mathit{type} \triangleright \mathit{initialiser}) \rightarrow M(S,\ S,\ D,\ \mathit{type},\ \mathit{data}_{\text{v}}) \\
&\mathcal{D}_{\text{DAI}}[\![S,\ T,\ D,\ I,\ t \triangleright i]\!] \\
&\quad = \text{do require } (\text{SCR}(t)) \\
&\qquad\quad \mathcal{D}_{\text{SI}}(S,\ T,\ D,\ I,\ t \triangleright i) \\
&\quad \mid\mid \text{do require } (\text{STR}(i) \wedge \text{ARR}(t)) \\
&\qquad\quad (t_i,\ \bar{\delta}) \leftarrow \mathcal{D}_{\text{SC}}(i) \\
&\qquad\quad \text{require } ((\text{CHR}(\mathcal{B}(t)) \wedge \text{CHR}(\mathcal{B}(t_i))) \vee (\text{unq}(\mathcal{B}(t)) \approx \mathcal{B}(t_i) \approx [\![\texttt{wchar_t}]\!])) \wedge \\
&\qquad\qquad\quad (\text{length}(t) = \epsilon \vee \text{length}(t) \geq \text{length}(t_i) - 1) \\
&\qquad\quad \text{return } (S,\ T,\ D,\ \mathcal{C}_{\text{A}}(t,\ \text{length}(t_i)),\ \bar{\delta}) \\
&\quad \mid\mid \text{do require } ((\neg\text{SCR}(t) \wedge \text{OBJ}(t) \vee \text{ARR}(t)) \wedge \text{AGGR}(i)) \\
&\qquad\quad (S',\ T',\ D',\ \bar{\delta},\ n,\ \bar{i}') \leftarrow \mathcal{D}_{\text{DIL}}(S,\ T,\ D,\ I,\ \bar{m}_{\text{I}}(t) \triangleright \text{list}(i)) \\
&\qquad\quad \text{require } (\bar{i}' = \varnothing) \\
&\qquad\quad \text{return } (S',\ T',\ D',\ \mathcal{C}_{\text{A}}(t,\ n), \\
&\qquad\qquad\quad \mathcal{P}(\bar{\delta} \cup \bigcup[\bar{\delta}(\mathcal{T}(m_k)) \oplus O(m_k) \mid m_k \leftarrow \text{drop}(n,\ \bar{m}_{\text{I}}(\mathcal{C}_{\text{A}}(t,\ n)))]))
\end{aligned}$$

In particular, any aggregate initialiser forms that appear within the syntax of a dynamic aggregate initialiser are translated identically to their static equivalents, except that their individual members are processed recursively using the above function $\mathcal{D}_{\text{DAI}}$ instead of its earlier static variant $\mathcal{D}_{\text{SI}}$, in order to ensure that no default padding is ever inserted into any string-initialised arrays embedded within the targeted object. Accordingly, the actual translation of all such initialiser lists is formulated in Haskell as follows:

$$\begin{aligned}
&\mathcal{D}_{\text{DIL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\
&\quad (S,\ S,\ D,\ I,\ \mathit{members} \triangleright [\mathit{initialiser}]) \rightarrow M(S,\ S,\ D,\ \mathit{data}_{\text{v}},\ \mathit{integer},\ [\mathit{initialiser}]) \\
&\mathcal{D}_{\text{DIL}}[\![S,\ T,\ D,\ I,\ \bar{m} \triangleright \bar{i}]\!] \\
&\quad = \text{do require } (\bar{m} = \varnothing \vee \bar{i} = \varnothing) \\
&\qquad\quad \text{return } (S,\ T,\ D,\ \varnothing,\ 0,\ \bar{i}) \\
&\quad \mid\mid \text{do require } (\text{OBJ}(\mathcal{T}(m_0)) \wedge \neg\text{SCR}(\mathcal{T}(m_0)) \wedge \neg\text{AGGR}(\text{head}(\bar{i}))) \\
&\qquad\quad (S_1,\ T_1,\ D_1,\ \bar{\delta}_1,\ n_1,\ \bar{i}_1) \leftarrow \mathcal{D}_{\text{DIL}}(S,\ \ T,\ \ D,\ \ I,\ \bar{m}_{\text{I}}(\mathcal{T}(m_0)) \triangleright \bar{i}) \\
&\qquad\quad (S_2,\ T_2,\ D_2,\ \bar{\delta}_2,\ n_2,\ \bar{i}_2) \leftarrow \mathcal{D}_{\text{DIL}}(S_1,\ T_1,\ D_1,\ I,\ \text{tail}(\bar{m}) \triangleright \bar{i}_1) \\
&\qquad\quad \text{return } (S_2,\ T_2,\ D_2,\ \mathcal{P}(\bar{\delta}_1 \oplus O(m_0)) \cup \bar{\delta}_2,\ 1 + n_2,\ \bar{i}_2) \\
&\quad \mid\mid \text{do } (S_1,\ T_1,\ D_1,\ t,\ \bar{\delta}_1) \leftarrow \mathcal{D}_{\text{DAI}}(S,\ T,\ D,\ I,\ \mathcal{T}(m_0) \triangleright \text{head}(\bar{i})) \\
&\qquad\quad (S_2,\ T_2,\ D_2,\ \bar{\delta}_2,\ n,\ \bar{i}') \leftarrow \mathcal{D}_{\text{DIL}}(S_1,\ T_1,\ D_1,\ I,\ \text{tail}(\bar{m}) \triangleright \text{tail}(\bar{i})) \\
&\qquad\quad \text{return } (S_2,\ T_2,\ D_2,\ \bar{\delta}_1 \oplus O(m_0) \cup \bar{\delta}_2,\ 1 + n,\ \bar{i}') \\
&\quad \text{where } m_0 = \text{head}(\bar{m})
\end{aligned}$$

5.9 Statements

Most of the computational structure of C programs is captured in the syntax of entities known as *statements*. Statements represent both the most interesting and, semantically, the most complex aspect of the C programming language, since they expose a number of fundamental differences between the designs of typical imperative languages such as C and the declarative paradigms of lambda calculi. It is a precise formal specification of statement semantics to which we will now turn our attention.

In particular, every well-formed C statement may be classified into one of six general forms, giving rise to the following Haskell definitions of their concrete syntax:

statement:
    labelled-statement
    compound-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement

labelled-statement:
    identifier **:** *statement*
    case *constant-expression* **:** *statement*
    default **:** *statement*

compound-statement:
    { $\mathit{declaration\text{-}list}_{opt}$ $\mathit{statement\text{-}list}_{opt}$ **}**

expression-statement:
    $\mathit{expression}_{opt}$ **;**

selection-statement:
    if (*expression* **)** *statement*
    if (*expression* **)** *statement* **else** *statement*
    switch (*expression* **)** *statement*

iteration-statement:
    while (*expression* **)** *statement*
    do *statement* **while (** *expression* **) ;**
    for ($\mathit{expression}_{opt}$ **;** $\mathit{expression}_{opt}$ **;** $\mathit{expression}_{opt}$ **)** *statement*

jump-statement:
    goto *identifier* **;**
    continue **;**
    break **;**
    return $\mathit{expression}_{opt}$ **;**

Due to the peculiar sequential nature and scoping rules adopted by these entities, a representation of their meanings in Etude is rather less straight forward than the one featured in our earlier semantic treatment of C expressions from Section 5.7. Fundamentally, every group of consecutive C statements that begins with an optional *statement label* and ends in an explicit or implicit *jump* construct depicts a logical entity known as a *basic block* in the standard programming language nomenclature. Intuitively, such

blocks describe a group of machine instructions that are always executed in a predetermined purely-sequential manner by the program. For example, the following list of C statements defines three basic blocks: "L_1`:` s_1`;` **`goto`** L_2`;`", "L_2`:` s_2`;` s_3`;` **`goto`** L_1`;`" and "L_3`:` s_4`;` **`return`**`;`":

```
L1:     s1;
L2:     s2;
        s3;
        goto L1;
L3:     s4;
        return;
```

In Etude, every such basic block is represented by a separate lambda abstraction, whose parameter list includes all of the local variables visible to the corresponding set of statements in a C program, although, strictly speaking, such functions represent *extended basic blocks* under conventional meaning of the term, since, in general they may include multiple exit points or jumps. The set of all basic blocks derived from a particular C function is represented in the following translation by a finite map B, which associates the Etude variable of each label with its local variable set V. Further, during translation of all C statements, a finite map L maps all label identifiers visible in their *function scope* to the corresponding Etude variables. More so, a finite map C contains the mapping of all visible "**`case`**" labels to their associated integer values and, finally, a set J maintains the collection of all label names actually utilised by some "**`goto`**" statement within the function. Formally, these four structures assume the following Haskell types:

$B:$	$v \mapsto V$	*(basic block definitions)*
$L:$	$identifier \mapsto v$	*(label definitions)*
$C:$	$integer_{opt} \mapsto v$	*(case label definitions)*
$J:$	$\{identifier\}$	*(required label variables)*

This somewhat elaborate context of C statements reflects the fact that, in C, constructs such as "**`goto`**" may transcend the usual scoping structure of declarative programs. For example, in the following nonsensical code fragment:

```
L1:     int x = 1;
        s1;
        {
                int x = 2;
                s2;
        L2:     if (x) {
                        goto L1;
                }
                s3;
        }
        s4;
        goto L2;
```

the two declarations of x introduce a pair of semantically-distinct objects, each allocated

conceptually at the beginning of its respective scope. Only the first x is visible to the statements s_1 and s_4, while s_2 and s_3 see only the second. However, upon execution of the jump "**goto** L_1**;**" and at the end of the following statement s_3, the first x reappears in the current scope, so that its Etude counterpart must, somehow, be retained throughout the entire program fragment, despite the preceding intermittent invisibility of the corresponding C identifier. To complicate the matters further, the second x is actually allocated twice in the above list of statements, first by the declaration "**int** x **=** 2**;**" and then again by the jump "**goto** L_2**;**", which transcends the usual scoping structure and brings the second x into scope without executing its proper declaration.

Accordingly, every control transfer to a basic block, whether represented explicitly in the concrete syntax of a C program by one of the jump statements discussed later in Section 5.9.6, or implied by a selection or iteration statement from Sections 5.9.4 and 5.9.5, is modelled uniformly by a tail call to the corresponding Etude function, as constructed by the following Haskell derivation of its meaning:

$$\mathcal{J}[\![\cdot]\!] :: (B,\, V \triangleright \nu) \to term_\nu$$

$$\mathcal{J}[\![B,\, V \triangleright \nu]\!] = [\![[\![\mathcal{D}(V \backslash \mathrm{dom}(B(\nu)))]\!];\, [\![\mathcal{C}(B(\nu) \backslash \mathrm{dom}(V))]\!];\, \nu([\![\mathrm{dom}(B(\nu))]\!])]\!]$$

so that the notation "$\mathcal{J}(B,\, V \triangleright \nu)$" depicts a jump to a C label associated with the Etude variable ν, provided that V represents the set of all local C variables visible at the beginning of that label, with every such variable mapped to the Etude envelope of its respective C type, and the finite map B binds the Etude variables associated with all labels defined in the surrounding C function to their individual local variable sets.

Intuitively, the resulting Etude term begins by purging from the current address space any local variables in V that do not exist in the scope of the targeted label ν, using the scope destruction operation $\mathcal{D}(V \backslash \mathrm{dom}(B(\nu)))$ described earlier in Section 5.8.12.2. Next, all local definitions visible to ν but missing from the caller's scope are introduced using the scope constructor $\mathcal{C}(B(\nu) \backslash \mathrm{dom}(V))$, effectively recreating the C scope current at the point where the label denoted by ν was actually introduced into the program. Finally, the actual tail call or jump operation is modelled by a function application with a list of arguments that are lexically identical to the sorted list of all the local variables visible at the point of ν's definition, since, under the translation described in this chapter, the meaning of each locally-defined C object is always depicted by a predetermined Etude variable whose atomic value remains invariant throughout the entire execution of a given C function.

Accordingly, the meanings of all C statements are always derived in a semantic context that consists of thirteen individual components S, T, D, I, B, L, V, t_r, t_s, ν_r, ν_c, ν_b and τ_κ, where S, T, D and I represent the usual current variable and tag scopes, set of item definitions and scope index. In addition, the two finite maps B and L describe the complete set of all basic blocks and label identifiers of the surrounding C function, retaining the same invariant values throughout the translation of a particular function's body. Finally, the remaining seven components V, t_r, t_s, ν_r, ν_c, ν_b and τ_κ, further refine

the statement's semantic context as follows:

① V represents the set of all local Etude variables visible to the statement under translation, with each such variable mapped to the envelope of its declared C type.

② t_r depicts the *returned type* of the surrounding C function. If this type is not "**`void`**", then the first variable from the associated V set is always bound to a memory-resident object destined to hold the function's returned value.

③ t_s specifies the optional *selection type* of the controlling expression e in the innermost surrounding statement of the form "**`switch (`***e*`)`** *s*". It is used in determination of the precise numeric values assigned to any "**`case`**" labels encountered during translation. This component is omitted from the statement's semantic context during analysis of entities that appear outside of any "**`switch`**" statements, whereby its value is represented by the symbol "ϵ" described in Appendix A.

④ v_r represents the *return label* variable, bound to an implicit basic block that is targeted by all "**`return`**" statements within the current C function.

⑤ v_c specifies the optional *continuation label*, bound to the basic block targeted by any "**`continue`**" statement forms within the current innermost iteration statement. If no such statement exists, v_c is given the value of "ϵ".

⑥ Similarly, v_b specifies the optional *break label*, bound to the basic block targeted by any "**`break`**" statement forms within the current innermost "**`switch`**" or iteration statement, once again defaulting to "ϵ" if no such statement exists in the program.

⑦ Finally, τ_κ represents the *continuation term*, intuitively equal to the combined denotation of all C statements that, in the program's lexical syntax, follow the one currently under translation. Its presence reflects the fact that the structure of a C statement alone is generally insufficient to determine the basic block to which the entity belongs. Consider, for example, the following program fragment:

```
L1 :    s1 ;
        {
                    s2 ;
        L2 :        s3 ;
        }
        s4 ;
```

which, conceptually, defines two basic blocks "L_1`:` s_1`;` s_2`;`" and "L_2`:` s_3`;` s_4`;`". Unfortunately, there is no simple way for s_4 to known that it belongs to the label L_2, or for the compound statement "s_2`;` L_2`:` s_3`;`" to discover that it spans two basic blocks, neither of which is actually complete within that statement. In order to resolve all such complications in a uniform manner, the denotation of every C statement is systematically composed with its respective continuation term and shoved into the set of item definitions D as soon as the boundary of a complete basic block to which the statement belongs has been discovered by the compiler. Specifically, in the above example, the continuation terms of s_1, s_2 and s_3 represent the denotations of s_2, "**`goto`** s_3`;`" and s_4, respectively.

The actual denotations derived from all C statements consist of the usual updated current scopes S and T, the set of item definitions D, as well as five statement-specific components B', L', C', J' and τ, which, intuitively, fulfil the following individual rôles:

① The set of *new basic block definitions* B' represents a strict subset of the basic block definitions B from the statement's context, consisting precisely of those basic block definitions that have been introduced into the program by the statement under semantic scrutiny. Intuitively, a union of all such sets B'_s for all statements s appearing within the lexical syntax of a given C function represents the complete value of the context component B supplied to every one of these statements, in recognition of the fact that, in C, a jump to a given basic block often precedes that block's actual appearance within the program's concrete syntax.

② Similarly, the set of *new named label definitions* L' represents the subset of L contributed to the surrounding C function by the statement being scrutinised. Each label is mapped in L' to the Etude variable of the corresponding basic block.

③ Further, the set of *case label definitions* C' maps all case labels introduced by the current statement to the Etude variables of their respective basic blocks. Every such label is represented in the domain of C' either by its concrete integer value, or else by the symbol "ϵ" for labels introduced by the "**`default`**" statement form described later in Section 5.9.1.

④ To facilitate correct diagnostics of "**`goto`**" statements, the set of all label identifiers that are actually accessed by some explicit jump within the currently-examined program fragment is returned in the statement's denotation as the component J'. The reader should observe that the value of J' is only ever examined later in Section 5.10.2, once the entire C function has been processed, since, until that time, the complete set of all defined label cannot be known to the compiler.

⑤ Finally, the Etude term τ represents the actual computation performed by the current statement, which, as already mentioned earlier, implicitly incorporates its continuation term τ_κ. Further, as shown in Section 5.9.1, the meanings of all labelled statement forms such as "x**`:`**s_1", "**`case`** e**`:`**s_2" and "**`default:`**s_3" are defined as implicit jumps to their respective basic blocks. The actual Etude terms derived from s_1, s_2 and s_3 are always incorporated into the bodies of the corresponding Etude functions and are never returned as part of the actual term denotations of their surrounding labelled statement entities.

Formally, these denotations are derived by the following Haskell construction:

$$\begin{aligned}&\mathcal{D}_\mathrm{S}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \\ &\quad (S,\ S,\ D,\ B,\ L,\ V,\ I,\ \mathit{type},\ \mathit{type}_{opt},\ \nu,\ \nu_{opt},\ \nu_{opt},\ \mathit{term}_\nu \triangleright \mathit{statement\text{-}list}_{opt}) \rightarrow \\ &\quad M(S,\ S,\ D,\ B,\ L,\ C,\ J,\ \mathit{term}_\nu)\end{aligned}$$

observing that this definition is naturally extended to encompass meanings of entire statement lists. For the sake of presentation, the actual definition of $\mathcal{D}_\mathrm{S}$ is presented separately for each statement form in the following seven sections.

5.9.1 Labelled Statements

In C, all constructs of the form "x`:` s", "**`case`** e`:` s" and "**`default`**`:` s" are known collectively as *labelled statements*. Every such entity consists of an arbitrary *nested statement* s, further associated with an annotation known as a *label*, which may be fashioned from an identifier x, a construct of the form "**`case`** e", or the "**`default`**" keyword. Conceptually, every labelled statement introduces a new basic block into the program, which, in the eventual Etude module, is represented by a function of the form "$\lambda\bar{\nu}.\tau$", whose parameter list $\bar{\nu}$ is formed from the domain of the statement's set of local variables V and whose body is equal to the denotation of the nested statement s. As already mentioned earlier, the denotation of s already incorporates its continuation term τ_κ and, accordingly, constitutes a complete body of that basic block.

The new function item is always contributed to the set of item definitions D and returned as part of the statement's denotation. Further, the corresponding unique Etude variable $\nu(D)$ is added to the set of new basic block definitions B', whereby it is bound to the statement's set of local variables V. The actual Etude term derived from the statement represents a simple jump to the new basic block, as depicted by the Haskell construction "$\mathcal{J}(B,\ V \triangleright \nu(D))$" defined earlier.

Further, if the statement's label consists entirely of a single identifier x, then x must not appear among the label names L_s derived from its nested statement s. In the resulting denotation, the identifier x is added to L_s, with a binding to the newly-introduced basic block variable $\nu(D)$. In Haskell:

$$
\begin{aligned}
&\mathcal{D}_\mathrm{S}[\![S,\ T,\ D,\ B,\ L,\ V,\ I,\ t_r,\ t_s,\ \nu_r,\ \nu_c,\ \nu_b,\ \tau_\kappa \triangleright x\texttt{:}\ s]\!] = \mathrm{do}\\
&\quad (D') \leftarrow D \cup \{\nu(D){:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_s]\!]\}\\
&\quad (S_s,\ T_s,\ D_s,\ B_s,\ L_s,\ C_s,\ J_s,\ \tau_s) \leftarrow \mathcal{D}_\mathrm{S}(S,\ T,\ D',\ B,\ L,\ V,\ I,\ t_r,\ t_s,\ \nu_r,\ \nu_c,\ \nu_b,\ \tau_\kappa \triangleright s)\\
&\quad \mathrm{require}\ (x \notin \mathrm{dom}(L_s))\\
&\quad \mathrm{return}\ (S_s,\ T_s,\ D_s,\ B_s \cup \{\nu(D){:}V\},\ L_s \cup \{x{:}\nu(D)\},\ C_s,\ J_s,\ \mathcal{J}(B,\ V \triangleright \nu(D)))
\end{aligned}
$$

On the other hand, in a labelled statement of the form "**`case`** e`:` s", the C expression e must always represent an integral constant, whose numeric value n is converted into the type of the surrounding switch statement t_s and, in the resulting set of case definitions C', mapped to the newly-introduced basic block variable $\nu(D)$. Every such statement must be always embedded within some "**`switch`**" construct, which, in the following Haskell definition, is asserted by requiring the selection type t_s to have some concrete value other than "ϵ". As for identifier labels, the converted value of n must not appear in the set C_s that has been derived from the construct's nested statement s:

$$
\begin{aligned}
&\mathcal{D}_\mathrm{S}[\![S,\ T,\ D,\ B,\ L,\ V,\ I,\ t_r,\ t_s,\ \nu_r,\ \nu_c,\ \nu_b,\ \tau_\kappa \triangleright \texttt{case}\ e\texttt{:}\ s]\!] = \mathrm{do}\\
&\quad \mathrm{require}\ (t_s \neq \epsilon)\\
&\quad (D') \leftarrow D \cup \{\nu(D){:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_s]\!]\}\\
&\quad (S_e,\ T_e,\ D_e,\ t,\ n) \leftarrow \mathcal{D}_\mathrm{ICE}(S,\ T,\ D',\ I \triangleright e)\\
&\quad (S_s,\ T_s,\ D_s,\ B_s,\ L_s,\ C_s,\ J_s,\ \tau_s) \leftarrow \mathcal{D}_\mathrm{S}(S_e,\ T_e,\ D_e,\ B,\ L,\ V,\ I,\ t_r,\ t_s,\ \nu_r,\ \nu_c,\ \nu_b,\ \tau_\kappa \triangleright s)\\
&\quad (n') \leftarrow \mathcal{V}_{\phi(t_s)}[\![[\![\phi(t_s)]\!]_{[\![\phi(t)]\!]}(\#n_{[\![\phi(t)]\!]})]\!]\\
&\quad \mathrm{require}\ (n' \notin \mathrm{dom}(C_s))\\
&\quad \mathrm{return}\ (S_s,\ T_s,\ D_s,\ B_s \cup \{\nu(D){:}V\},\ L_s,\ C_s \cup \{n'{:}\nu(D)\},\ J_s,\ \mathcal{J}(B,\ V \triangleright \nu(D)))
\end{aligned}
$$

Finally, every construction of the form "**`default:`** s" must also appear within the context of an enclosing "**`switch`**" statement. In the resulting set of case labels C', the new basic block is represented by the distinguished case label "ϵ", which, once again, cannot appear among the labels of its nested statement s. Formally:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{S}}[\![S,\, T,\, D,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau_\kappa \triangleright \texttt{default:}\ s]\!] = \mathrm{do} \\
&\quad \text{require } (t_s \neq \epsilon) \\
&\quad (D') \leftarrow D \cup \{\nu(D){:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_s]\!]\} \\
&\quad (S_s,\, T_s,\, D_s,\, B_s,\, L_s,\, C_s,\, J_s,\, \tau_s) \leftarrow \mathcal{D}_{\mathrm{S}}(S,\, T,\, D',\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau_\kappa \triangleright s) \\
&\quad \text{require } (\epsilon \notin \mathrm{dom}(C_s)) \\
&\quad \text{return } (S_s,\, T_s,\, D_s,\, B_s \cup \{\nu(D){:}V\},\, L_s,\, C_s \cup \{\epsilon{:}\nu(D)\},\, J_s,\, \mathcal{J}(B,\, V \triangleright \nu(D)))
\end{aligned}$$

5.9.2 Compound Statements or Blocks

Multiple C statements are often combined into a single *compound statement*, which, in the concrete syntax of the language, is represented by an optional list of declarations followed by a list of zero or more statements, with the entire construct enclosed in braces. Every such compound statement "**{** dl_{opt} sl_{opt} **}**" introduces a new current scope into the program, whose index I is one greater than that of its surrounding semantic context. Further, the context of the statement list sl_{opt} is extended with the set of local variable definitions V_d introduced by any non-static declarations from dl_{opt}. The resulting Etude term begins by introducing into the program's address space all of these newly-defined local variables, as depicted by the the scope constructor $\mathcal{C}(V_d)$. Next, the underlying local objects are populated with their desired initial contents by evaluating the initialiser term τ_d derived from the declaration list dl_{opt}. Finally, the actual computational meaning τ_s of the supplied statement list sl_{opt} is evaluated. An appropriate scope destructor $\mathcal{D}(V_d)$ is also prefixed to the continuation term of sl_{opt}, in order to ensure that all of the Etude objects local to the new scope are purged from the program's address space upon completion of the statement's execution. Formally, this denotation is represented in Haskell as follows:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{S}}[\![S,\, T,\, D,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau_\kappa \triangleright \{\, dl_{opt}\ sl_{opt}\, \}]\!] = \mathrm{do} \\
&\quad (S_d,\, T_d,\, D_d,\, V_d,\, \tau_d) \leftarrow \mathcal{D}_{\mathrm{D}}(S,\, T,\, D,\, I+1 \triangleright dl_{opt}) \\
&\quad (S_s,\, T_s,\, D_s,\, B_s,\, L_s,\, C_s,\, J_s,\, \tau_s) \\
&\qquad \leftarrow \mathcal{D}_{\mathrm{S}}(S_d,\, T_d,\, D_d,\, B,\, L,\, V \cup V_d,\, I+1,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, [\![[\![\mathcal{D}(V_d)]\!];\, \tau_\kappa]\!] \triangleright sl_{opt}) \\
&\quad \text{return } (S,\, T,\, D_s,\, B_s,\, L_s,\, C_s,\, J_s,\, [\![[\![\mathcal{C}(V_d)]\!];\, \mathsf{LET}\ () = \tau_d;\, \tau_s]\!])
\end{aligned}$$

The reader should observe that the new variable and tag scopes do not persist past the end of the construct, so that the resulting denotation includes the original values of its prevailing current scopes S and T, instead of those derived from the statement list sl_{opt}.

5.9.3 Expression and Null Statements

Any C expression can be also turned into an *expression statement* by appending a semicolon to its concrete syntax. Further, the actual expression component can be omitted entirely from such constructs, in which case the entity is said to represent the *null statement* form "**`;`**". Regardless of its precise syntax, every expression statement denotes a simple Etude term of the form "$\mathcal{C}(V_e)$; $\mathsf{LET}\ () = \tau_e$; $\mathcal{D}(V_e)$; τ_κ", in which τ_e represents

the void denotation of its optional expression component. Intuitively, this denotation evaluates that void denotation, after introducing any required temporary variables into the program's address space, which are promptly discarded before proceeding with execution of the continuation term τ_κ. Formally:

$$\begin{aligned}&\mathcal{D}_\mathrm{S}[\![S,\, T,\, D,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau_\kappa \triangleright e_{opt}]\!] = \mathrm{do}\\ &\quad (S_e,\, T_e,\, D_e,\, V_e,\, \tau_e) \leftarrow \mathcal{D}_\mathrm{E}(S,\, T,\, D,\, I \triangleright e_{opt})\\ &\quad \mathrm{return}\ (S_e,\, T_e,\, D_e,\, \varnothing,\, \varnothing,\, \varnothing,\, \varnothing,\, [\![[\![\mathcal{C}(V_e)]\!];\ \mathsf{LET}\ ()\ =\ \tau_e;\ [\![\mathcal{D}(V_e)]\!];\ \tau_\kappa]\!])\end{aligned}$$

In other words, expression statements never introduce any new labels or local variables into the program, so that the B', L', C and J components of their denotations always represent the empty set $\varnothing$. The fact that the continuation term usually pertains to an expression which is only encountered much later in the translation process is immaterial, since Haskell's lazy evaluation mechanism is perfectly capable of alluding to events that are yet to come. In particular, the continuation term is always incorporated, but never actually constructed until the C statement from which it is to be derived has been encountered in the program, deemed well-formed and processed by the compiler.

This simple definition underpins the entire expressive power of the continuation term approach to the specification of C statements pursued in this work. In particular, given that C expressions are the sole means provided by the language for representation of all non-trivial computations in a program, the above inclusion of the continuation term τ_κ in their denotations ensures that every other form of a C statement always denotes an Etude term which implicitly incorporates the remainder of a basic block to which the statement belongs. As we have already seen in Section 5.9.1, this basic block is contributed to the constructed Etude program by the translation of its introducing label and, thereafter, it is always referenced solely through tail calls (or jumps) to the Etude variable associated with that label.

5.9.4 Selection Statements

The language recognises three kinds of *selection statements*, often referred to as the "**`if`**", "**`if-else`**" and "**`switch`**" statement forms. Conceptually, all three variants of a selection statement represent a fork in the program's control flow graph, whereby only some subset of the statement's nested components are evaluated as determined dynamically during the construct's execution.

Upon completion of its evaluation, every selection statement always proceeds with the reduction of its continuation term τ_κ. In order to avoid replication of τ_κ on every possible execution path through the program, all of the following denotations introduce into the statement's set of item definitions D' a new basic block $\nu(D')$ with the body of "$\lambda\bar{v}.\tau_\kappa$", in which $\bar{v}$ includes all of the local variables visible at the beginning of the construct. During translation the entity's nested statements, their continuation terms are systematically replaced with a simple jump to this new basic block, as depicted by the standard notation "$\mathcal{J}(B,\, V \triangleright \nu(D'))$" described earlier in this section.

The parenthesised C expression which appears in the syntax of every selection statement and most of the iteration statement forms described later in Section 5.9.5 is known as the entity's *controlling expression*. Formally, such expressions must always represent values of a scalar C type. Their denotations are evaluated in a scope delimited by the usual constructor and destructor terms $\mathcal{C}(V_e)$ and $\mathcal{D}(V_e)$, where V_e is the set of temporary variables introduced by the expression. They always deliver the result obtained from comparison of the expression's value for inequality with the constant "**0**" of an appropriate C type, as depicted by the following translation of these entities:

$$\mathcal{D}_{\text{CTRL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D,\, I \triangleright \mathit{expression}) \rightarrow M(S,\, S,\, D,\, \mathit{term}_{\mathrm{v}})$$

$$\begin{aligned}
&\mathcal{D}_{\text{CTRL}}[\![S,\, T,\, D,\, I \triangleright e]\!] = \text{do}\\
&\quad (S',\, T',\, D',\, V,\, t,\, \tau) \leftarrow \mathcal{D}_{\mathrm{v}}(S,\, T,\, D,\, I \triangleright e)\\
&\quad \text{require } (\mathrm{SCR}(t))\\
&\quad \text{return } (S',\, T',\, D',\, [\![[\![\mathcal{C}(V)]\!];\, \mathsf{LET}\ \mathsf{T}_1 = \tau;\, [\![\mathcal{D}(V)]\!];\, \mathsf{RET}\ (\mathsf{T}_1 \neq_{[\![\phi(t)]\!]} \#0_{[\![\phi(t)]\!]})]\!])
\end{aligned}$$

In particular, every selection statement of the form "**if (***e***)** *s*" represents the Etude term "$\mathsf{LET}\ \mathsf{T}_1 = \tau_e;\ \mathsf{IF}\ \mathsf{T}_1\ \mathsf{THEN}\ \tau_s\ \mathsf{ELSE}\ \tau'_\kappa$", in which τ_e and τ_s depict the respective denotations of the controlling expression e and the nested statement s, while $\nu(D_s)$ denotes a unique basic block variable bound to the construct's continuation term τ_κ. In other words, such statements evaluate s if and only if e has a non-zero value. Formally:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{S}}[\![S,\, T,\, D,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau_\kappa \triangleright \textbf{if (}e\textbf{)}\ s]\!] = \text{do}\\
&\quad (\tau'_\kappa) \leftarrow \mathcal{J}(B,\, V \triangleright \nu(D_s))\\
&\quad (S_e,\, T_e,\, D_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{CTRL}}(S,\, T,\, D,\, I \triangleright e)\\
&\quad (S_s,\, T_s,\, D_s,\, B_s,\, L_s,\, C_s,\, J_s,\, \tau_s) \leftarrow \mathcal{D}_{\mathrm{S}}(S_e,\, T_e,\, D_e,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau'_\kappa \triangleright s)\\
&\quad \text{return } (S_s,\, T_s,\, D_s \cup \{\nu(D_s){:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_\kappa]\!]\},\, B_s \cup \{\nu(D_s){:}V\},\, L_s,\, C_s,\, J_s,\\
&\qquad\qquad [\![\mathsf{LET}\ \mathsf{T}_1 = \tau_e;\ \mathsf{IF}\ \mathsf{T}_1\ \mathsf{THEN}\ \tau_s\ \mathsf{ELSE}\ \tau'_\kappa]\!])
\end{aligned}$$

Similarly, a selection statement of the form "**if (***e***)** s_1 **else** s_2" evaluates either of the two nested statements s_1 or s_2, whenever its controlling expression e has a non-zero or zero value, respectively. Formally, the resulting denotation represents an Etude term of the form "$\mathsf{LET}\ \mathsf{T}_1 = \tau_e;\ \mathsf{IF}\ \mathsf{T}_1\ \mathsf{THEN}\ \tau_1\ \mathsf{ELSE}\ \tau_2$", in which τ_e, τ_1 and τ_2 depict the respective denotations of e, s_1 and s_2. Similarly to the earlier "**if**" statement form, τ_κ is saved in the body of a new basic block $\nu(D_2)$ and the continuation terms of both nested statements are replaced with a jump to that block. In Haskell:

$$\begin{aligned}
&\mathcal{D}_{\mathrm{S}}[\![S,\, T,\, D,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau_\kappa \triangleright \textbf{if (}e\textbf{)}\ s_1\ \textbf{else}\ s_2]\!] = \text{do}\\
&\quad (\tau'_\kappa) \leftarrow \mathcal{J}(B,\, V \triangleright \nu(D_2))\\
&\quad (S_e,\, T_e,\, D_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{CTRL}}(S,\, T,\, D,\, I \triangleright e)\\
&\quad (S_1,\, T_1,\, D_1,\, B_1,\, L_1,\, C_1,\, J_1,\, \tau_1) \leftarrow \mathcal{D}_{\mathrm{S}}(S_e,\, T_e,\, D_e,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau'_\kappa \triangleright s_1)\\
&\quad (S_2,\, T_2,\, D_2,\, B_2,\, L_2,\, C_2,\, J_2,\, \tau_2) \leftarrow \mathcal{D}_{\mathrm{S}}(S_1,\, T_1,\, D_1,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau'_\kappa \triangleright s_2)\\
&\quad \text{require } (\mathrm{dom}(L_1) \cap \mathrm{dom}(L_2) = \varnothing \wedge \mathrm{dom}(C_1) \cap \mathrm{dom}(C_2) = \varnothing)\\
&\quad \text{return } (S_2,\, T_2,\, D_2 \cup \{\nu(D_2){:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_\kappa]\!]\},\\
&\qquad\qquad B_1 \cup B_2 \cup \{\nu(D_2){:}V\},\, L_1 \cup L_2,\, C_1 \cup C_2,\, J_1 \cup J_2,\\
&\qquad\qquad [\![\mathsf{LET}\ \mathsf{T}_1 = \tau_e;\ \mathsf{IF}\ \mathsf{T}_1\ \mathsf{THEN}\ \tau_1\ \mathsf{ELSE}\ \tau_2]\!])
\end{aligned}$$

The reader should observe that the two nested statements appearing in such constructs must derive non-overlapping sets L' and C', in order to ensure that all statement and case labels always remain unique within their respective scopes.

On the other hand, selection statements of the form "**`switch (`***e*`)`** *s*" have a rather different semantic interpretation. In every such construct, the controlling expression *e* must assume an integral type that, after integral promotion, will serve as the selection type t_s throughout translation of the nested statement *s*. The entire construct represents a *multi-way jump*, which, in Etude, can be modelled by a term of the following form:

$$\begin{array}{l}\quad\ \mathsf{IF}\ \mathrm{T}_1 =_\phi n_1\ \mathsf{THEN}\ \mathcal{J}[\![B,\ V \triangleright \nu_1]\!] \\ \mathsf{ELSE\ IF}\ \mathrm{T}_1 =_\phi n_2\ \mathsf{THEN}\ \mathcal{J}[\![B,\ V \triangleright \nu_2]\!] \\ \qquad\qquad\vdots \\ \mathsf{ELSE\ IF}\ \mathrm{T}_1 =_\phi n_k\ \mathsf{THEN}\ \mathcal{J}[\![B,\ V \triangleright \nu_k]\!] \\ \mathsf{ELSE}\ \mathcal{J}[\![B,\ V \triangleright \nu_d]\!]\end{array}$$

in which $n_1 \ldots n_k$ are the integer values of all "**`case`**" labels introduced by the nested statement *s*, $\nu_1 \ldots \nu_k$ represent the Etude variables bound to these labels' respective basic blocks and ν_d depicts a designated basic block bound to the special "ϵ" label introduced by the statement's "**`default`**" label, or else the basic block of its continuation statement τ_κ if no such label is present in the syntax of *s*. The Etude variable ν_d also doubles as the break label within *s*, so that all "**`break;`**" statements included within the syntax of *s* represent a jump to that basic block under their later interpretation in Section 5.9.6. The return and continuation labels ν_r and ν_c are not affected by these constructs. Formally, the denotation of every "**`switch`**" statement is represented by the following Haskell translation:

$$\begin{array}{l}\mathcal{D}_\mathrm{S}[\![S,\ T,\ D,\ B,\ L,\ V,\ I,\ t_r,\ t_s,\ \nu_r,\ \nu_c,\ \nu_b,\ \tau_\kappa \triangleright \texttt{switch (}e\texttt{)}\ s]\!] = \mathrm{do} \\ \quad (\tau'_\kappa) \leftarrow \mathcal{J}(B,\ V \triangleright \nu(D_s)) \\ \quad (\mathring{\tau}\ ) \leftarrow [\![\odot[[\![\mathsf{IF}\ \mathrm{T}_2 =_{[\![\phi(\mathrm{ip}(t_e))]\!]}\ \#n_k{}_{[\![\phi(\mathrm{ip}(t_e))]\!]}\ \mathsf{THEN}\ [\![\mathcal{J}(B,\ V \triangleright \nu_k)]\!]\ \mathsf{ELSE}]\!]\ |\ n_k{:}\nu_k \leftarrow C_s]]\!] \\ \quad (S_e,\ T_e,\ D_e,\ V_e,\ t_e,\ \tau_e) \leftarrow \mathcal{D}_\mathrm{V}(S,\ T,\ D,\ I \triangleright e) \\ \quad (S_s,\ T_s,\ D_s,\ B_s,\ L_s,\ C_s,\ J_s,\ \tau_s) \\ \qquad \leftarrow \mathcal{D}_\mathrm{S}(S_e,\ T_e,\ D_e,\ B,\ L,\ V,\ I,\ t_r,\ \mathrm{ip}(t_e),\ \nu_r,\ \nu_c,\ \nu(D_s),\ \tau'_\kappa \triangleright s) \\ \quad \mathrm{require}\ (\mathrm{INT}(t_e)) \\ \quad \mathrm{return}\ (S_s,\ T_s,\ D_s \cup \{\nu(D_s){:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_\kappa]\!]\},\ B_s \cup \{\nu(D_s){:}V\},\ L_s,\ \varnothing,\ J_s, \\ \qquad\qquad [\![[\![\mathcal{C}(V_e)]\!]; \\ \qquad\qquad\ \mathsf{LET}\ \mathrm{T}_1 = \tau_e; \\ \qquad\qquad\ \mathsf{LET}\ \mathrm{T}_2 = [\![\phi(\mathrm{ip}(t_e))]\!]_{[\![\phi(t_e)]\!]}(\mathrm{T}_1); \\ \qquad\qquad\ [\![\mathcal{D}(V_e)]\!]; \\ \qquad\qquad\ \mathring{\tau}; \\ \qquad\qquad\ \mathcal{J}(B,\ V \triangleright (\{\epsilon{:}\nu(D_s)\}/C_s)(\epsilon))]\!])\end{array}$$

Observe that the case labels of *s* remain confined within their innermost surrounding "**`switch`**" statement, so that the resulting set C' is always emptied by the above definition, regardless of the precise form assumed by a given "**`switch`**" statement.

5.9.5 Iteration Statements

The language also defines three forms of *iteration statements* or *loops*, commonly known as the "**`while`**", "**`do-while`**" and "**`for`**" statement forms, respectively. Every such construct evaluates its nested statement repeatedly for as long as its controlling expression retains a non-zero value. Each iteration statement introduces at least two

new basic blocks into the program, associated with the implicit continuation and break labels ν_c and ν_b that serve as a target for all occurrences of the jump statement forms "**`continue;`**" and "**`break;`**" within its syntax, as described later in Section 5.9.6.

In particular, if L_1 and L_2 are two otherwise-unused label identifiers and if "`...`" represents the remainder of the current basic block, then an iteration statement of the form "**`while (`***e*`)`** *s*" evaluates its nested statement *s* zero or more times, corresponding loosely to the following control flow structure, except that no new scope is introduced for its nested statement *s*:

```
L1:     if (e) {
                s;
                goto L1;
        }
        else {
                goto L2;
        }
L2:     ...
```

Any occurrences of the "**`continue;`**" and "**`break;`**" statement forms within the above code fragment represent jumps to the respective labels L_1 and L_2, which, in the following Haskell translation, is modelled by setting the continuation and break labels ν_c and ν_b to the globally-unique Etude variables $\nu(D)$ and $\nu(D_s)$ associated with these labels during analysis of the loop body *s*. The selection type t_s is not affected by these constructs. Formally:

$$
\begin{aligned}
&\mathcal{D}_S[\![S,\, T,\, D,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau_\kappa \triangleright \textbf{`while (`}e\textbf{`)`}\ s]\!] = \text{do}\\
&\quad (\nu_c') \leftarrow \nu(D)\\
&\quad (\nu_b') \leftarrow \nu(D_s)\\
&\quad (\tau_\ell\,) \leftarrow [\![\mathsf{LET\ T_1 = } \tau_e\mathsf{;\ IF\ T_1\ THEN\ } \tau_s \mathsf{\ ELSE\ } [\![\mathcal{J}(B,\, V \triangleright \nu_b')]\!]]\!]\\
&\quad (\tau_\ell') \leftarrow \mathcal{J}(B,\, V \triangleright \nu_c')\\
&\quad (S_e,\, T_e,\, D_e,\, \tau_e) \leftarrow \mathcal{D}_{\text{CTRL}}(S,\, T,\, D \cup \{\nu_c' {:} [\![\lambda [\![\text{dom}(V)]\!].\tau_\ell]\!]\},\, I \triangleright e)\\
&\quad (S_s,\, T_s,\, D_s,\, B_s,\, L_s,\, C_s,\, J_s,\, \tau_s) \leftarrow \mathcal{D}_S(S_e,\, T_e,\, D_e,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c',\, \nu_b',\, \tau_\ell' \triangleright s)\\
&\quad \text{return } (S_s,\, T_s,\, D_s \cup \{\nu_b' {:} [\![\lambda [\![\text{dom}(V)]\!].\tau_\kappa]\!]\},\, B_s \cup \{\nu_c' {:} V,\, \nu_b' {:} V\},\, L_s,\, C_s,\, J_s,\, \tau_\ell')
\end{aligned}
$$

Similarly, the iteration statement "**`do`** *s* **`while (`***e*`);`**" evaluates *s* one or more times until the controlling expression *e* attains the value of "**`0`**". In other words, the control structure of these statements can be described roughly by the following code fragment:

```
L1:     s;
L2:     if (e) {
                goto L1;
        }
        else {
                goto L3;
        }
L3:     ...
```

where, once again, the three identifiers L_1, L_2 and L_3 represent otherwise-unused basic block variables, with L_1 and L_3 also doubling as the continuation and break la-

bels during translation of the loop body s, as required for a proper translation of any "**`continue;`**" and "**`break;`**" statements appearing within it:

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{S}}[\![S,\, T,\, D,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau_\kappa \triangleright \texttt{do}\ s\ \texttt{while (}e\texttt{);}]\!] = \mathrm{do}\\
&\quad (\nu_i) \leftarrow \nu(D)\\
&\quad (\nu_c') \leftarrow \nu(D_s)\\
&\quad (\nu_b') \leftarrow \nu(D_e)\\
&\quad (\tau_\ell) \leftarrow [\![\mathsf{LET\ T_1 = } \tau_e\mathsf{;\ IF\ T_1\ THEN\ } [\![\mathcal{J}(B,\, V \triangleright \nu_i)]\!]\ \mathsf{ELSE\ } [\![\mathcal{J}(B,\, V \triangleright \nu_b')]\!]]\!]\\
&\quad (\tau_\ell') \leftarrow \mathcal{J}(B,\, V \triangleright \nu_c')\\
&\quad (S_s,\, T_s,\, D_s,\, B_s,\, L_s,\, C_s,\, J_s,\, \tau_s)\\
&\qquad \leftarrow \mathcal{D}_{\mathrm{S}}(S,\, T,\, D \cup \{\nu_i{:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_s]\!]\},\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c',\, \nu_b',\, \tau_\ell' \triangleright s)\\
&\quad (S_e,\, T_e,\, D_e,\, \tau_e) \leftarrow \mathcal{D}_{\mathrm{CTRL}}(S_s,\, T_s,\, D_s \cup \{\nu_c'{:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_\ell]\!]\},\, I \triangleright e)\\
&\quad \mathrm{return}\ (S_e,\, T_e,\, D_e \cup \{\nu_b'{:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_\kappa]\!]\},\\
&\qquad\qquad B_s \cup \{\nu_i{:}V,\, \nu_c'{:}V,\, \nu_b'{:}V\},\, L_s,\, C_s,\, J_s,\, \mathcal{J}(B,\, V \triangleright \nu_i))
\end{aligned}
$$

Finally, every iteration statement of the form "**`for (`**e_{opt1}**`;`** e_2**`;`** e_{opt3}**`)`** s" always begins with an evaluation of the optional expression e_{opt1}, which, similarly to the earlier "**`while`**" statement forms, is followed by zero or more iterations over the loop body "s**`;`** e_{opt3}**`;`**", subject to the controlling expression e_2 retaining a non-zero value. Like "**`do`**-**`while`**" statements, every "**`for`**" construct introduces three new basic blocks into the program and arranges them into the following control flow structure:

```
            e_opt1;
L_1:        if (e_2) {
                    s;
                    goto L_2;
            }
            else {
                    goto L_3;
            }
L_2:        e_opt3;
            goto L_1;
L_3:        ...
```

in which the continuation and break labels are represented by the later unique identifiers L_2 and L_3. Formally, the meanings of all "**`for`**" statements are described in Haskell by the following translation semantics:

$$
\begin{aligned}
&\mathcal{D}_{\mathrm{S}}[\![S,\, T,\, D,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c,\, \nu_b,\, \tau_\kappa \triangleright \texttt{for (}e_{opt1}\texttt{;}\ e_2\texttt{;}\ e_{opt3}\texttt{)}\ s]\!] = \mathrm{do}\\
&\quad (\nu_i) \leftarrow \nu(D_1)\\
&\quad (\nu_c') \leftarrow \nu(D_2)\\
&\quad (\nu_b') \leftarrow \nu(D_s)\\
&\quad (\tau_\ell) \leftarrow [\![\mathsf{LET\ T_1 = } \tau_2\mathsf{;\ IF\ T_1\ THEN\ } \tau_s\ \mathsf{ELSE\ } [\![\mathcal{J}(B,\, V \triangleright \nu_b')]\!]]\!]\\
&\quad (\tau_\ell') \leftarrow \mathcal{J}(B,\, V \triangleright \nu_c')\\
&\quad (\tau_c) \leftarrow [\![[\![\mathcal{C}(V_3)]\!]\mathsf{;\ LET\ () = } \tau_3\mathsf{;\ } [\![\mathcal{D}(V_3)]\!]\mathsf{;\ } \mathcal{J}(B,\, V \triangleright \nu_i)]\!]\\
&\quad (S_1,\, T_1,\, D_1,\, V_1,\, \tau_1) \leftarrow \mathcal{D}_{\mathrm{E}}(S,\, T,\, D,\, I \triangleright e_{opt1})\\
&\quad (S_2,\, T_2,\, D_2,\, \tau_2) \leftarrow \mathcal{D}_{\mathrm{CTRL}}(S_1,\, T_1,\, D_1 \cup \{\nu_i{:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_\ell]\!]\},\, I \triangleright e_2)\\
&\quad (S_3,\, T_3,\, D_3,\, V_3,\, \tau_3) \leftarrow \mathcal{D}_{\mathrm{E}}(S_2,\, T_2,\, D_2/\{\nu_c'{:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_c]\!]\},\, I \triangleright e_{opt3})\\
&\quad (S_s,\, T_s,\, D_s,\, B_s,\, L_s,\, C_s,\, J_s,\, \tau_s) \leftarrow \mathcal{D}_{\mathrm{S}}(S_3,\, T_3,\, D_3,\, B,\, L,\, V,\, I,\, t_r,\, t_s,\, \nu_r,\, \nu_c',\, \nu_b',\, \tau_\ell' \triangleright s)\\
&\quad \mathrm{return}\ (S_s,\, T_s,\, D_s \cup \{\nu_b'{:}[\![\lambda[\![\mathrm{dom}(V)]\!].\tau_\kappa]\!]\},\, B_s \cup \{\nu_i{:}V,\, \nu_c'{:}V,\, \nu_b'{:}V\},\, L_s,\, C_s,\, J_s,\\
&\qquad\qquad [\![[\![\mathcal{C}(V_1)]\!]\mathsf{;\ LET\ () = } \tau_1\mathsf{;\ } [\![\mathcal{D}(V_1)]\!]\mathsf{;\ } \mathcal{J}(B,\, V \triangleright \nu_i)]\!])
\end{aligned}
$$

Finally, if the controlling expression e_2 is omitted from the syntax of a given "**for**" statement, then the entity behaves as if an implicit controlling expression with a constant non-zero value was inserted into the program by compiler:

$$\mathcal{D}_\mathrm{S}[\![S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright \texttt{for (}e_{opt1}\texttt{;; }e_{opt3}\texttt{) }s]\!] \\ = \mathcal{D}_\mathrm{S}[\![S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright \texttt{for (}e_{opt1}\texttt{; 1; }e_{opt3}\texttt{) }s]\!]$$

A close inspection of the control-flow graph constructed by the above translation of iteration statements reveals that the nested statement s in the "**do**-**while**" constructs is always evaluated at least once for every appearance of such an entity on the program's execution path. In all other iteration statements, however, evaluation of s is subject to the controlling expression e having a non-zero value at the beginning of the construct. As we shall soon discover in Section 5.9.6, in all three cases, the ordinary sequence of events may be also interrupted by explicit jumps to the continuation and break labels introduced by these entities, as represented in the concrete syntax of the language by the two jump statement forms "**continue;**" and "**break;**" that are specially reserved for that purpose.

5.9.6 Jump Statements

Finally, the class of *jump statements* includes four simple syntactic forms known respectively as the "**goto**", "**continue**", "**break**" and "**return**" statements. The first three of these have trivial translations into Etude, with every statement of the form "**goto** x**;**" representing a jump to a named label bound in the surrounding function scope L to the specified identifier x. For the purposes of later error diagnostics, x is also included in the returned set of referenced labels J'. On the other hand, every C statement of the form "**continue;**", "**break;**" and "**return;**" represents an explicit jump to the corresponding continuation, break or return label ν_c, ν_b or ν_r. Such jumps are well-formed only if that label is actually included in the statement's context, i.e., if it does not have the omitted value "ϵ". Formally:

$$\mathcal{D}_\mathrm{S}[\![S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright \texttt{goto }x\texttt{;}]\!] = \text{do} \\ \quad \text{return } (S, T, D, \varnothing, \varnothing, \varnothing, \{x\}, \mathcal{J}(B, V \triangleright L(x)))$$

$$\mathcal{D}_\mathrm{S}[\![S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright \texttt{continue;}]\!] = \text{do} \\ \quad \text{require } (\nu_c \neq \epsilon) \\ \quad \text{return } (S, T, D, \varnothing, \varnothing, \varnothing, \varnothing, \mathcal{J}(B, V \triangleright \nu_c))$$

$$\mathcal{D}_\mathrm{S}[\![S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright \texttt{break;}]\!] = \text{do} \\ \quad \text{require } (\nu_b \neq \epsilon) \\ \quad \text{return } (S, T, D, \varnothing, \varnothing, \varnothing, \varnothing, \mathcal{J}(B, V \triangleright \nu_b))$$

$$\mathcal{D}_\mathrm{S}[\![S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright \texttt{return;}]\!] = \text{do} \\ \quad \text{return } (S, T, D, \varnothing, \varnothing, \varnothing, \varnothing, \mathcal{J}(B, V \triangleright \nu_r))$$

On the other hand, a "**return** e**;**" statement whose syntax includes some C expression e is only admissible within C functions whose returned type t_r represents an object other than an array. Such statements precede the jump to ν_r with an assignment of e's value to a temporary l-value bound to the least entry in the function's local variable

set V. In particular, the supplied C expression e represents a dynamic initialiser for that temporary l-value, whose denotation $\mathcal{D}_{\mathrm{DI}}[\![S, T, D, I, t_r, \min(\mathrm{dom}(V)) \triangleright \texttt{(}e\texttt{)}]\!]$ was described earlier in Section 5.8.12.2. In Haskell:

$\mathcal{D}_{\mathrm{S}}[\![S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright \texttt{return}\ e\texttt{;}]\!]$ = do
  $(S_e, T_e, D_e, t_e, \tau_e) \leftarrow \mathcal{D}_{\mathrm{DI}}[\![S, T, D, I, t_r, \min(\mathrm{dom}(V)) \triangleright \texttt{(}e\texttt{)}]\!]$
  require $(\mathrm{OBJ}(t_r) \wedge \neg\mathrm{ARR}(t_r))$
  return $(S_e, T_e, D_e, \varnothing, \varnothing, \varnothing, \varnothing, [\![\mathsf{LET}\ () = \tau_e; [\![\mathcal{J}(B, V \triangleright \nu_r)]\!]]\!])$

A careful reader will observe that none of the above five labelled statement forms ever mentions the continuation term τ_κ within their denotations. In fact, any computations that follow an unconditional jump in a C function can never be evaluated during ordinary execution of a well-formed C program, so that, in effect, the remainder of any basic block which includes such a jump statement is deemed *unreachable* and, as such, it is discarded by the above translations of these constructs.

5.9.7 Lists of Statements

To complete our formalisation of C statements, we must also specify the translation semantics of entire statement lists. Formally, the concrete syntax of such lists is depicted by the following BNF grammar:

statement-list:
  statement
  statement-list statement

Although, in this work, the denotations of all C statements are described by a single function $\mathcal{D}_{\mathrm{S}}$ that always operates on an entire statement list, in reality, sequences of multiple statemnets can only ever appear within the compound statement forms described earlier in Section 5.9.2. In all other contexts, such lists are formed by an implicit coercion of a singular statement form, whose presence, however, is always hidden in the presentation for the sake of conciseness.

Intuitively, a list of the form "$s_1\ s_2$" always evaluates both of its statements in the order of their appearance within the program. Formally, its denotation is equal to the meaning τ_1 of the first statement s_1, taken in the context of a continuation term τ_2 derived from the following statement s_2, which, in turn, is supplied the continuation term τ_κ of the entire construction, provided that the sets of new label entities introduced by each statement in the list remain disjoint from each other. In Haskell:

$\mathcal{D}_{\mathrm{S}}[\![S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright sl\ s]\!]$ = do
  $(S_1, T_1, D_1, B_1, L_1, C_1, J_1, \tau_1) \leftarrow \mathcal{D}_{\mathrm{S}}(S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_2 \triangleright sl)$
  $(S_2, T_2, D_2, B_2, L_2, C_2, J_2, \tau_2) \leftarrow \mathcal{D}_{\mathrm{S}}(S_1, T_1, D_1, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright s)$
  require $(\mathrm{dom}(L_1) \cap \mathrm{dom}(L_2) = \varnothing \wedge \mathrm{dom}(C_1) \cap \mathrm{dom}(C_2) = \varnothing)$
  return $(S_2, T_2, D_2, B_1 \cup B_2, L_1 \cup L_2, C_1 \cup C_2, J_1 \cup J_2, \tau_1)$

$\mathcal{D}_{\mathrm{S}}[\![S, T, D, B, L, V, I, t_r, t_s, \nu_r, \nu_c, \nu_b, \tau_\kappa \triangleright \epsilon]\!]$ = return $(S, T, D, \varnothing, \varnothing, \varnothing, \varnothing, \tau_\kappa)$

The reader should observe that, under the above translation, an empty statement list denotes simply its own continuation term τ_κ.

5.10 External Definitions

Traditionally, large C programs are generally developed as a collection of separate *source files*, every one of which corresponds, intuitively, to a single Etude module of the form described earlier in Section 4.7. Formally, such independent units of compilation are represented in the C grammar by syntactic entities known as *translation units*, which, in concrete terms, consist of one or more *external declarations*, as captured by the following Haskell definition of their syntax:

translation-unit :
external-declaration
translation-unit external-declaration

The precise syntax and semantics of the individual external declarations in a translation union are discussed separately in Section 5.10.1, whereby a list of such entities, taken in the context of an initially empty current scopes and item definition set $\varnothing$, is used to derive the complete scopes S and T, the set of item definitions D and the initialiser term τ for the entire translation unit.

Once all external declarations in a given source file have been processed, any tentative definitions "ϵ" remaining in D for Etude variables that, in the ultimate scope S current at the end of the translation unit, are associated with a designator of a static linkage form, are converted into specification of an implicitly-initialised C object, i.e., into an Etude item definition of the form "OBJ $(\bar{\delta}_k)$ OF $(\bar{\xi}_k)$", in which $\bar{\delta}_k$ and $\bar{\xi}_k$ represent the default data specification and envelope derived from the variable's C type, as described earlier in the respective Sections 5.8.11 and 5.4.6.

Once this adjustment has been performed for all tentative definitions in the translation unit, the collective denotation of its external declarations can be used to derive a complete Etude module of the form "MODULE τ EXPORT $\bar{\xi}$ WHERE $\bar{\iota}$", in which the module initialiser τ represents the composition of the individual declaration initialisers derived from all the external declarations in the unit.

Intuitively, the module's export set $\bar{\xi}$ includes the bindings of all variables introduced into the translation unit with an external linkage form, other than those which are simply imported from some other source file of the entire program. In particular, for every identifier-like Etude variable x that, in the ultimate scope S current at the end of translation, is assigned a designator with an external linkage and, in the associated set of item definitions D, is associated with any Etude item form other than "IMP x", the resulting set $\bar{\xi}$ will contain a binding of x to its own name.

Further, the item set $\bar{\iota}$ of the resulting module consists of all the static definitions introduced by the translation unit, including those bound to the imported item forms "IMP x", but excluding any tentative definitions for objects with a non-static linkage. Keeping in mind that, throughout this chapter, all such tentative definitions were distinguished in D by bindings to the special symbol "ϵ", $\bar{\iota}$ consists simply of all the bindings $(\nu_k{:}\iota_k) \in D$ in which $\iota_k \neq \epsilon$.

In Haskell, the entire algorithm can be depicted quite naturally as follows:

$$\mathcal{D}_{\text{TU}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow \textit{translation-unit} \rightarrow M(\textit{module}_\nu)$$

$$\begin{aligned}
&\mathcal{D}_{\text{TU}}[\![tu]\!] = \text{do} \\
&\quad (S,\, T,\, D,\, V,\, \tau) \leftarrow \mathcal{D}_{\text{EDL}}(\varnothing,\, \varnothing,\, \varnothing \triangleright tu) \\
&\quad (D') \leftarrow D/\{[\![\nu(\mathcal{L}(g_k))]\!] = \textsf{OBJ}\ ([\![\bar{\delta}(\mathcal{T}(g_k))]\!])\ \textsf{OF}\ ([\![\bar{\xi}(\mathcal{T}(g_k))]\!]) \\
&\qquad\qquad\qquad\qquad \mid x_k{:}g_k \leftarrow S,\, \text{SL}(\mathcal{L}(g_k)) \wedge D(\nu(\mathcal{L}(g_k))) = \epsilon\} \\
&\quad (\,\bar{\xi}\,) \leftarrow \{[\![x_k = x_k]\!] \mid x_k{:}g_k \leftarrow S,\ \mathcal{L}(g_k) = [\![\textbf{\textit{extern}}]\!] \wedge D'(x_k) \neq [\![\textsf{IMP}\ x_k]\!]\} \\
&\quad (\,\bar{\iota}\,) \leftarrow \{[\![\nu_k = \iota_k]\!] \mid \nu_k{:}\iota_k \leftarrow D,\, \iota_k \neq \epsilon\} \\
&\quad \text{require } (V = \varnothing \wedge \textstyle\bigwedge [D'(\nu(\mathcal{L}(g_k))) \neq [\![\textsf{IMP}\ x_k]\!] \mid (x_k{:}g_k) \leftarrow S,\, \mathcal{L}(g_k) = [\![\textbf{\textit{intern}}]\!]]) \\
&\quad \text{return } [\![\textsf{MODULE}\ \tau\ \textsf{EXPORT}\ \bar{\xi}\ \textsf{WHERE}\ \bar{\iota}]\!]
\end{aligned}$$

The reader should observe that the set of local variable definitions V derived from the external C declarations must always remain empty and that, by the end of every source file, all internally-linked entities must include a proper object definition somewhere within their surrounding translation unit, so that no C variable x with an internal linkage in S should be bound in D to the imported item form "IMP x".

5.10.1 External Declarations

In a C program, every *external declaration* constitutes either an ordinary declaration construct, whose syntax and semantics were the subject of Section 5.8, or else a new kind of entity known as a *function definition*, whose meaning will be described shortly in Section 5.10.2. In Haskell, this dual syntax of external C declarations can be captured by the following BNF grammar:

external-declaration:
function-definition
declaration

Formally, the translation of a C source file into Etude is performed by threading the supplied current scope and item definition set through all of its external declarations, in the order of their appearance within the program, also composing the individual initialiser terms τ_k derived from each declaration into a monadic sequence of the form "LET () = τ_1; LET () = τ_2; ... LET () = τ_n;". In Haskell:

$$\mathcal{D}_{\text{EDL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D \triangleright \textit{translation-unit}) \rightarrow M(S,\, S,\, D,\, V,\, \textit{term}_\nu)$$

$$\begin{aligned}
&\mathcal{D}_{\text{EDL}}[\![S,\, T,\, D \triangleright \textit{edl ed}]\!] = \text{do} \\
&\quad (S_1,\, T_1,\, D_1,\, V_1,\, \tau_1) \leftarrow \mathcal{D}_{\text{EDL}}(S,\ \ T,\ \ D\ \ \triangleright \textit{edl}) \\
&\quad (S_2,\, T_2,\, D_2,\, V_2,\, \tau_2) \leftarrow \mathcal{D}_{\text{EDL}}(S_1,\, T_1,\, D_1 \triangleright \textit{ed}) \\
&\quad \text{return } (S_2,\, T_2,\, D_2,\, V_1 \cup V_2,\, [\![\textsf{LET}\ () = \tau_1;\, \tau_2]\!])
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}_{\text{EDL}}[\![S,\, T,\, D \triangleright \textit{function-definition}]\!] &= \mathcal{D}_{\text{FD}}(S,\, T,\, D \triangleright \textit{function-definition}) \\
\mathcal{D}_{\text{EDL}}[\![S,\, T,\, D \triangleright \textit{declaration}]\!] &= \mathcal{D}_{\text{D}}(S,\, T,\, D,\, 0 \triangleright \textit{declaration})
\end{aligned}$$

recalling that all three of the finite maps S, T and D supplied in the context of an external declaration will be empty at the beginning of every translation unit. The reader should also observe that, in every standard and well-formed C program, all external declarations will, by construction, always produce the trivial initialiser term "RET ()",

so that the resulting initialiser for an entire Etude module produced the program's translation will always remain semantically indistinguishable from a single "RET ()" term. During a subsequent linking stage of compilation, this term will be replaced with a call to the program's designated "**main**" function, although all of the relevant details of that replacement remain beyond the scope of the present work. Accordingly, the declaration initialisers are retained under the above translation only in the interest of its forward compatibility with any future extensions to the C Standard.

5.10.2 Function Definitions

A new function is always introduced into a C program by an entity known as a *function definition*, whose concrete syntax is defined by the following Haskell data type:

function-definition:
declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

In other words, every function definition consists of a C declaration with a single declarator and no initialisers, together with a compound statement known as the *function body*. The list of declaration specifiers may be omitted from the syntax if no particular specifiers are applicable to the function being defined. The compound statement may be also preceded by an optional list of declarations, intended to refine the meanings of any function arguments in an absence of an explicit prototype in the associated declarator.

The derivation of a function definitions' formal meaning bootstraps the entire statement translation process presented earlier in Section 5.9. Accordingly, the construction $\mathcal{D}_{\mathrm{FD}}[\![S, T, D \triangleright ds_{opt}\ r\ dl_{opt}\ s]\!]$ has a somewhat esoteric formulation, which is best described informally by the following algorithm:

① First of all, the construct's list of declaration specifiers is used to derive the storage class $s\bar{c}$, qualification $t\bar{q}$ and base type t_b for the function's designator, using the standard algorithms $\mathcal{D}_{\mathrm{DS}}$ and $\mathcal{D}_{\mathrm{TS}}$ defined earlier in Sections 5.8.1 and 5.8.3.

② Next, a meaning of the following declarator r is established using the algorithm $\mathcal{D}_{\mathrm{FR}}$ described later in this section. Besides the usual C type t and identifier x found in the denotations of ordinary C declarators, this algorithm also derives the set of argument names $\bar{x}_2$ from the construct's identifier list whenever such a list appears in the syntax, as well as an additional pair of scopes S' and T' current at the end of any prevailing parameter type list in r.

③ Further, the new function is introduced into its declarator's scope S_2 with a binding of the identifier x to a designator d, derived from the storage class $s\bar{c}$ and the C type of the specified declarator r using the algorithm $\mathcal{D}_{\mathrm{X}}$ defined earlier in Section 5.8.11. In the associated set of item definitions D_2, the variable x is also bound to an Etude function, whose parameter list and term represent, respectively, the set of all local variables V'_3 visible to the compound statement s and the actual denotation τ of that statement, as derived is steps 5 and 6 of the present algorithm.

④ Now, the list of parameter declarations dl_{opt} is processed using the algorithm $\mathcal{D}_{\text{PDL}}$, in the context of the parameter scopes S' and T' derived from r. The resulting denotation includes a new set of identifiers $\bar{x}_3$, formed precisely from those parameters to the new function whose C types are declared by dl_{opt}, together with the set of all local variables V_3 that have been associated with these parameters.

⑤ Once dl_{opt} has been analysed, all identifiers from $\bar{x}_2$ that are still left without a local declaration are introduced into the program as automatically-linked entities of the plain "**`int`**" type. Further, the initial set of all local variables V_3' visible upon entry into the newly-created C function is defined as a union of all local variables V_2 declared in the parameter type list of the declarator r, together with the set V_3 of those that are declared by dl_{opt} and, finally, any variables introduced by the implicit "**`int`**" bindings of the otherwise undeclared parameters from $\bar{x}_2$.

⑥ Having completed these preparatory tasks, we are now ready to scrutinise the function's body statement s, which is processed directly using the algorithm $\mathcal{D}_{\text{S}}$ defined earlier in Section 5.9. The statement's function scope L is determined by the set of all label identifiers derived from s itself. Similarly, the collection of all basic block definitions B available throughout the construct's translation is set to contain all of the basic blocks B' introduced by the function's body, together with an additional *exit block* $\nu(D_4)$ that is appended to both B and D_4 with a trivial definition of "$\lambda[\![\text{dom}(V_3')]\!]$.`RET ()`". Intuitively, this block represents the exit point from the entire function and, during translation of s, it is made available to the statement as a binding of its return label ν_r. A jump to that label also serves as the statement's continuation term τ_κ. Finally, the function's returned type t_r is set to the base type of the designator r, while the continuation and break labels ν_c and ν_b and the selection type t_s are left with their default values of "ϵ".

⑦ In every well-formed function definition, the resulting designator d must have a static linkage, with a C type of a function returning either an object type other than an array, or else a qualified or unqualified version of the "**`void`**" type. Further, at most one of the parameter type lists and parameter declaration components can be ever supplied in a single function definition, so that, if the declarator r specifies a non-empty function prototype for the new entity, then the associated declaration list dl_{opt} must always remain empty. The set of all label names J that are actually referenced by s must also represent a subset of its function scope L, in order to ascertain that all "**`goto`**" statements in the program refer to valid statement labels. Last but not least, the set of case labels C derived from the function's body must be empty by the end of the entire construct, so that no "**`case`**" label may ever appear outside of a well-formed "**`switch`**" statement.

⑧ Finally, in the denotation of the entire function definition, the local scope of s is discarded and the returned Etude term is given the trivial form "`RET ()`", in order to facilitate its incorporation into a larger list of external declarations as described earlier in Section 5.10.1.

In Haskell, these translation semantics of C functions can be formalised as follows:

$$\mathcal{D}_{\text{FD}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D \triangleright \textit{function-definition}) \rightarrow M(S,\, S,\, D,\, V,\, \textit{term}_{\nu})$$

$$\begin{aligned}
&\mathcal{D}_{\text{FD}}[\![S,\, T,\, D \triangleright \textit{ds}_{opt}\ r\ \textit{dl}_{opt}\ s]\!] = \text{do}\\
&\quad (s\bar{c},\, t\bar{q},\, t\bar{s}) \leftarrow \mathcal{D}_{\text{DS}}(\textit{ds}_{opt})\\
&\quad (S_1,\, T_1,\, D_1,\, t_b) \leftarrow \mathcal{D}_{\text{TS}}(S,\, T,\, D,\, 0 \triangleright t\bar{s})\\
&\quad (S_2,\, S',\, T_2,\, T',\, D_2,\, V_2,\, t,\, x,\, \bar{x}_2) \leftarrow \mathcal{D}_{\text{FR}}(S_1,\, T_1,\, D_1,\, t\bar{q} +\!\!+ t_b \triangleright r)\\
&\quad (\ d\ ) \leftarrow \mathcal{D}_{\text{X}}(S_2,\, T_2,\, D_2,\, 0,\, s\bar{c},\, t \triangleright x)\\
&\quad (S_2') \leftarrow S_2/\{x{:}d\}\\
&\quad (D_2') \leftarrow D_2/\{x{:}[\![\lambda[\![\text{dom}(V_3')]\!].\tau]\!]\}\\
&\quad (S_3,\, T_3,\, D_3,\, V_3,\, \bar{x}_3) \leftarrow \mathcal{D}_{\text{PDL}}(S_2'/S',\, T_2/T',\, D_2' \triangleright \textit{dl}_{opt})\\
&\quad (S_3') \leftarrow S_3/\{x_k{:}[\![\textbf{\texttt{auto}}\ \nu_k\ \textbf{\texttt{int}}\ \texttt{@}\ 1]\!] \mid (x_k{:}\nu_k) \leftarrow (\bar{x}_2 \backslash \bar{x}_3) | [\nu(D_3)...]\}\\
&\quad (D_3') \leftarrow D_3 \cup \{\nu(\mathcal{L}(S_3'(x_k))){:}\epsilon \mid x_k \leftarrow \bar{x}_2 \backslash \bar{x}_3\}\\
&\quad (V_3') \leftarrow V_2 \cup V_3 \cup \{\nu(\mathcal{L}(S_3'(x_k))){:}\bar{\xi}(\mathcal{T}(S_3'(x_k))) \mid x_k \leftarrow \bar{x}_2 \backslash \bar{x}_3\}\\
&\quad (S_4,\, T_4,\, D_4,\, B,\, L,\, C,\, J,\, \tau)\\
&\qquad \leftarrow \mathcal{D}_{\text{S}}(S_3',\, T_3,\, D_3',\, B',\, L,\, V_3',\, 1,\, \mathcal{B}(\mathcal{T}(d)),\, \epsilon,\, \nu(D_4),\, \epsilon,\, \epsilon,\, \mathcal{J}(B',\, V_3' \triangleright \nu(D_4)) \triangleright s)\\
&\quad (D_4') \leftarrow D_4/\{\nu(D_4){:}[\![\lambda[\![\text{dom}(V_3')]\!].\text{RET }()]\!]\}\\
&\quad (\ B'\ ) \leftarrow B \cup \{\nu(D_4){:}V_3'\}\\
&\quad \text{require } (\text{SL}(\mathcal{L}(d))) \wedge\\
&\qquad (\text{FUN}(\mathcal{T}(d)) \wedge ((\text{OBJ}(\mathcal{B}(\mathcal{T}(d))) \wedge \neg\text{ARR}(\mathcal{B}(\mathcal{T}(d)))) \vee \text{VT}(\mathcal{B}(\mathcal{T}(d))))) \wedge\\
&\qquad (\bar{x}_3 \subseteq \bar{x}_2) \wedge\\
&\qquad (J \subseteq \text{dom}(L)) \wedge (C = \varnothing)\\
&\quad \text{return } (S_2',\, T_2,\, D_4',\, \varnothing,\, [\![\text{RET }()]\!])
\end{aligned}$$

In every such construct, the function's type must be specified directly in r, rather than through a type name in the associated list of declaration specifiers. In order to capture this requirement formally, it is convenient to introduce a notion of *simple declarators*, whose concrete syntax must be formed either from a single identifier, or else from a parenthesised version of such a declarator. In Haskell, these constructs are characterised by the following derivation, and their denotations consist solely of the specified identifier itself:

$$\mathcal{D}_{\text{SR}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (\textit{declarator}) \rightarrow M(\textit{identifier})$$

$$\begin{aligned}
\mathcal{D}_{\text{SR}}[\![\ \texttt{(}r\texttt{)}\ ]\!] &= \mathcal{D}_{\text{SR}}(r)\\
\mathcal{D}_{\text{SR}}[\![x]\!] &= \text{return } (x)\\
\mathcal{D}_{\text{SR}}[\![\textit{other}]\!] &= \text{reject}
\end{aligned}$$

With the help of this auxiliary definition, the meaning of all declarators appearing within a function definition can be described by the following algorithm $\mathcal{D}_{\text{FR}}$, akin to its earlier variant $\mathcal{D}_{\text{R}}$ that, in Section 5.8.8 was used to scrutinise the semantics of ordinary C declarators:

$$\begin{aligned}
&\mathcal{D}_{\text{FR}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow\\
&\quad (S,\, S,\, D,\, \textit{type} \triangleright \textit{declarator}) \rightarrow M(S,\, S,\, S,\, S,\, D,\, V,\, \textit{type},\, \textit{identifier},\, \{\textit{identifier}\})
\end{aligned}$$

In particular, like $\mathcal{D}_{\text{R}}$, $\mathcal{D}_{\text{FR}}$ proclaims every nested declarator "$\texttt{(}r\texttt{)}$" to represent a simple semantic equivalent of r itself and all declarators of the form "*pointer r*" to assume the meaning of r under the new pointer base type $\mathcal{D}_{\text{P}}(t_b \triangleright \textit{pointer})$, in which t_b depicts the prevailing base type of the entire construction. On the other hand, declarators consisting

entirely of a single identifier are never permitted in the present context:

$$\begin{aligned}
\mathcal{D}_{\text{FR}}[\![S,\, T,\, D,\, t_b \triangleright \texttt{(}r\texttt{)}]\!] &= \mathcal{D}_{\text{FR}}(S,\, T,\, D,\, t_b \triangleright r)\\
\mathcal{D}_{\text{FR}}[\![S,\, T,\, D,\, t_b \triangleright \mathit{pointer}\ r]\!] &= \mathcal{D}_{\text{FR}}(S,\, T,\, D,\, \mathcal{D}_{\text{P}}(t_b \triangleright \mathit{pointer}) \triangleright r)\\
\mathcal{D}_{\text{FR}}[\![S,\, T,\, D,\, t_b \triangleright x]\!] &= \text{reject}
\end{aligned}$$

Further, array declarators of the form "$r\texttt{[}e_{opt}\texttt{]}$" denote the array type "$t_b\texttt{[}n_{opt}\texttt{]}$", in which the element type t_b is equal to the construct's base type, identically to the earlier ordinary applications of this syntax:

$$\begin{aligned}
&\mathcal{D}_{\text{FR}}[\![S,\, T,\, D,\, t_b \triangleright r\texttt{[}e_{opt}\texttt{]}]\!] = \text{do}\\
&\quad (S_r,\, S',\, T_r,\, T',\, D_r,\, V,\, t,\, x,\, \bar{x}) \leftarrow \mathcal{D}_{\text{FR}}(S,\, T,\, D,\, [\![t_b\texttt{[}n_{opt}\texttt{]}]\!] \triangleright r)\\
&\quad (S_e,\, T_e,\, D_e,\, n_{opt}) \leftarrow \mathcal{D}_{\text{ALE}}(S_r,\, T_r,\, D_r,\, 0 \triangleright e_{opt})\\
&\quad \text{return } (S_e,\, S',\, T_e,\, T',\, D_e,\, V,\, t,\, x,\, \bar{x})
\end{aligned}$$

The peculiar nature of function declarators only becomes apparent in constructs of the form "$r\texttt{(}\mathit{ptl}\texttt{)}$" and "$r\texttt{(}xl_{opt}\texttt{)}$".

In particular, if the supplied declarator r that accompanies a non-empty parameter type list *ptl* consists entirely of an optionally-parenthesised identifier x (in other words, if it depicts a simple declarator), then the construction itself determines the actual type of the C function being defined. Accordingly, its denotation consists of the declarator's identifier x, the function type "$t_b\texttt{(}p\texttt{)}$" whose prototype p is derived from the parameter type list *ptl* and a local variable set formed from all of the Etude variables introduced by *ptl*, with an addition of a designated Etude object intended for storage of the function's return value. In V, this object is bound to the unique local variable $\nu(D)$ with an envelope derived from the function's returned type t_b, using the construction $\bar{\xi}_{\text{R}}(\nu(D),\, t_b)$ described later in the present section. The identifier set $\bar{x}$ returned by such entities is left empty, while the prototype scopes S' and T' consist precisely of those variables and type tags that have been introduced into the program locally by the parameter type list *ptl*. In all other cases, declarators of this form are processed identically to their ordinary usage from Section 5.8.8, deriving the meanings of their direct declarators r under the adjusted base type "$t_b\texttt{(}p_{opt}\texttt{)}$". Formally:

$$\begin{aligned}
&\mathcal{D}_{\text{FR}}[\![S,\, T,\, D,\, t_b \triangleright r\texttt{(}\mathit{ptl}\texttt{)}]\!]\\
&\quad = \text{do } (x) \leftarrow \mathcal{D}_{\text{SR}}(r)\\
&\qquad\quad (S',\, T',\, D',\, V,\, p) \leftarrow \mathcal{D}_{\text{FPTL}}(S,\, T,\, D \cup \{\nu(D){:}\epsilon\} \triangleright \mathit{ptl})\\
&\qquad\quad \text{return } (S,\, \{x_k{:}S'(x_k) \mid x_k \leftarrow \text{dom}(S'),\, I(S'(x_k)) = 1\},\\
&\qquad\qquad\qquad T,\, \{x_k{:}T'(x_k) \mid x_k \leftarrow \text{dom}(T'),\, I(T'(x_k)) = 1\},\\
&\qquad\qquad\qquad D',\, V \cup \bar{\xi}_{\text{R}}(\nu(D),\, t_b),\, [\![t_b\texttt{(}p\texttt{)}]\!],\, x,\, \varnothing)\\
&\quad \mathbin{||}\ \text{do } (S_r,\, S',\, T_r,\, T',\, D_r,\, V,\, t,\, x,\, \bar{x}) \leftarrow \mathcal{D}_{\text{FR}}(S,\, T,\, D,\, [\![t_b\texttt{(}p_{opt}\texttt{)}]\!] \triangleright r)\\
&\qquad\quad (D_p,\, p_{opt}) \leftarrow \mathcal{D}_{\text{PTL}}(S_r,\, T_r,\, D_r,\, 0 \triangleright \mathit{ptl})\\
&\qquad\quad \text{return } (S_r,\, S',\, T_r,\, T',\, D_p,\, V,\, t,\, x,\, \bar{x})
\end{aligned}$$

Similarly, a function declarator of the form "$r\texttt{(}xl_{opt}\texttt{)}$", in which xl_{opt} represents a list of zero or more identifiers and r constitutes a simple declarator, denotes the list of identifiers $\bar{x}$ derived from the syntax of xl_{opt}, the declarator's name x, the type of a function "$t_b\texttt{()}$" without any prototype information and a singular set of local variables

formed entirely from the result object $\bar{\xi}_{\mathrm{R}}(\nu(D), t_b)$, that is bound to a globally-unique Etude variable $\nu(D)$. The prototype scopes S' and T' are always left empty by such declarators. In all such constructs, the identifier list xl_{opt} must represent a proper set, so that any particular identifier may appear at most once in a given list of parameter names and, further, none of these identifiers may redeclare an existing typedef name that, in the current scope S, is bound to a designator with a type linkage. Otherwise, such entities are treated in a fashion analogous to that prescribed for their ordinary usage in Section 5.8.8, so that the identifier list xl_{opt} must be left empty whenever r does not represent a simple declarator. In Haskell:

$$\mathcal{D}_{\mathrm{FR}}[\![S,\, T,\, D,\, t_b \triangleright r\texttt{(}xl_{opt}\texttt{)}]\!]$$

$$= \mathrm{do}\ (x) \leftarrow \mathcal{D}_{\mathrm{SR}}(r)$$
$$\quad \mathrm{require}\ (\mathrm{length}(\mathrm{list}(xl_{opt})) = |\mathrm{list}(xl_{opt})| \wedge$$
$$\quad\quad \bigwedge[x_k \notin \mathrm{dom}(S) \vee (I(S(x_k)) \neq 1 \wedge \mathcal{L}(S(x_k)) \neq [\![\texttt{type}]\!]) \mid x_k \leftarrow \mathrm{list}(xl_{opt})]$$
$$\quad \mathrm{return}\ (S,\, \varnothing,\, T,\, \varnothing,\, D \cup \{\nu(D){:}\epsilon\},\, \bar{\xi}_{\mathrm{R}}(\nu(D), t_b),\, [\![t_b\texttt{()}]\!],\, x,\, \mathrm{list}(xl_{opt}))$$

$$\|\ \mathrm{do}\ (S_r,\, S',\, T_r,\, T',\, D_r,\, V,\, t,\, x,\, \bar{x}) \leftarrow \mathcal{D}_{\mathrm{FR}}(S,\, T,\, D,\, [\![t_b\texttt{()}]\!] \triangleright r)$$
$$\quad \mathrm{require}\ (\mathrm{list}(xl_{opt}) = \varnothing)$$
$$\quad \mathrm{return}\ (S_r,\, S',\, T_r,\, T',\, D_r,\, V,\, t,\, x,\, \bar{x})$$

The restriction on redeclaration of typedef names within identifier lists has been imposed in the C Standard in order to resolve subtle syntactic ambiguity in the published BNF grammar of the language. In the present work, this constraint serves no other useful purpose and it is included in the above translation only for the sake of completeness.

In both cases, the collection of additional objects that are introduced implicitly into the resulting set of local variables for storage of the function's return value is determined by its returned type t, as represented by the notation "$\bar{\xi}_{\mathrm{R}}(\nu,\, t)$". In most cases, $\bar{\xi}_{\mathrm{R}}$ produces a singleton set, consisting of a single variable binding that associates ν with the Etude envelope derived from the C type t, provided that the function returns an object type other than an array. However, if t represents a qualified or unqualified version of the "**`void`**" type, then no returned variable is introduced into the local scope, so that $\bar{\xi}_{\mathrm{R}}[\![\nu,\, \texttt{void}]\!]$ always produces the empty set $\varnothing$:

$$\bar{\xi}_{\mathrm{R}}[\![\cdot]\!] :: (\nu,\, \mathit{type}) \rightarrow V$$

$$\begin{array}{lll} \bar{\xi}_{\mathrm{R}}[\![\nu,\, t]\!] & \mid \mathrm{OBJ}(t) \wedge \neg\mathrm{ARR}(t) & = \{\nu{:}\bar{\xi}(t)\} \\ & \mid \mathrm{VT}(t) & = \varnothing \end{array}$$

Every list of parameter types appearing within a function declarator is processed similarly to the ordinary usage of such constructs described earlier in Section 5.8.8, except that, in the present context, these entities also derive the set of all Etude variables associated with the individual parameters of the new function, in addition to the actual prototype information. Formally, this new interpretation of such lists is characterised by the following Haskell function:

$$\mathcal{D}_{\mathrm{FPTL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\, S,\, D \triangleright \mathit{parameter\text{-}type\text{-}list}) \rightarrow M(S,\, S,\, D,\, V,\, \mathit{prototype})$$

If such a list consists entirely of a single storage class specifier "**`void`**", then the resulting set of local variables V is left empty. Otherwise, the meaning of such entities is determined from their parameter lists pl, using the algorithm $\mathcal{D}_{\text{FPL}}$ described shortly:

$$\mathcal{D}_{\text{FPTL}}[\![S,\ T,\ D \triangleright \texttt{void}]\!] = \text{return } (S,\ T,\ D,\ \varnothing,\ [\![\texttt{void}]\!])$$

$$\begin{aligned}&\mathcal{D}_{\text{FPTL}}[\![S,\ T,\ D \triangleright pl]\!] = \text{do}\\ &\quad (S',\ T',\ D',\ V,\ \bar{t}) \leftarrow \mathcal{D}_{\text{FPL}}(S,\ T,\ D \triangleright pl)\\ &\quad \text{return } (S',\ T',\ D',\ V,\ [\![\bar{t}]\!])\end{aligned}$$

$$\begin{aligned}&\mathcal{D}_{\text{FPTL}}[\![S,\ T,\ D \triangleright pl\texttt{, ...}]\!] = \text{do}\\ &\quad (S',\ T',\ D',\ V,\ \bar{t}) \leftarrow \mathcal{D}_{\text{FPL}}(S,\ T,\ D \triangleright pl)\\ &\quad \text{return } (S',\ T',\ D',\ V,\ [\![\bar{t}\texttt{...}]\!])\end{aligned}$$

In particular, every parameter declaration found in a function definition must specify a distinct identifier x that is unique within its scope, so that abstract declarators are not permitted in such constructs, except for the special "**`void`**" case mentioned earlier. Further, the declaration's syntax must not include any storage class specifiers other than "**`register`**" and the semantics of its declarator must assign to x a C type t that is complete after pointer promotion. The construct extends its current scope S with a binding of x to a designator with the unadjusted type t and a linkage derived from the supplied storage class $s\bar{c}$ using the usual algorithm $\mathcal{D}_{\text{SC}}$ defined in Section 5.8.2. The formal denotation of such parameters always includes a set of local variables V that binds individual parameter variables to Etude envelopes of their pointer-promoted types. In Haskell:

$$\mathcal{D}_{\text{FPL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D \triangleright \textit{parameter-list}) \rightarrow M(S,\ S,\ D,\ V,\ \textit{types})$$

$$\begin{aligned}&\mathcal{D}_{\text{FPL}}[\![S,\ T,\ D \triangleright ds\ r]\!] = \text{do}\\ &\quad (s\bar{c},\ t\bar{q},\ t\bar{s}) \leftarrow \mathcal{D}_{\text{DS}}(ds)\\ &\quad (S_1,\ T_1,\ D_1,\ t_b) \leftarrow \mathcal{D}_{\text{TS}}(S,\ T,\ D,\ 1 \triangleright t\bar{s})\\ &\quad (S_2,\ T_2,\ D_2,\ t,\ x) \leftarrow \mathcal{D}_{\text{R}}(S_1,\ T_1,\ D_1,\ 1,\ t\bar{q} +\!\!+ t_b \triangleright r)\\ &\quad (\ell) \leftarrow \mathcal{D}_{\text{SC}}(S_2,\ T_2,\ D_2,\ 1,\ \text{pp}(t),\ x \triangleright s\bar{c})\\ &\quad \text{require } (s\bar{c} \subseteq \{\texttt{register}\} \wedge \text{OBJ}(\text{pp}(t)) \wedge (x \notin \text{dom}(S_2) \vee I(S_2(x)) \neq 1))\\ &\quad \text{return } (S_2/\{x{:}[\![\ell\ t\ \texttt{@}\ 1]\!]\},\ T_2,\ D_2/\{\nu(\ell){:}\epsilon\},\ \{\nu(\ell){:}\bar{\xi}(\text{pp}(t))\},\ [t])\end{aligned}$$

$$\begin{aligned}&\mathcal{D}_{\text{FPL}}[\![S,\ T,\ D \triangleright pl\texttt{, }pd]\!] = \text{do}\\ &\quad (S_1,\ T_1,\ D_1,\ V_1,\ \bar{t}_1) \leftarrow \mathcal{D}_{\text{FPL}}(S,\ \ T,\ \ D\ \ \triangleright pl)\\ &\quad (S_2,\ T_2,\ D_2,\ V_2,\ \bar{t}_2) \leftarrow \mathcal{D}_{\text{FPL}}(S_1,\ T_1,\ D_1 \triangleright pd)\\ &\quad \text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ \bar{t}_1 +\!\!+ \bar{t}_2)\end{aligned}$$

$$\mathcal{D}_{\text{FPL}}[\![S,\ T,\ D \triangleright ds\ ar_{opt}]\!] = \text{reject}$$

On the other hand, if the function definition's declarator includes an identifier list instead of parameter types, then the names of all function arguments are determined by those identifiers and their C types are described by a list of zero or more ordinary C declarations that follow the declarator in the concrete syntax of the entire definition. Conceptually, all of these declarations are injected into the program at the beginning of the scope associated with the function's compound statement, in a manner similar to the earlier ordinary applications of the declaration syntax scrutinised in Section 5.8, except

that, in the present context, no declaration may include any storage class specifiers other than "**register**" and that every element of the list must specify at least one initialised declarator, so that tag declarations are explicitly disallowed within such lists:

$$\mathcal{D}_{\text{PDL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow (S,\ S,\ D \triangleright \textit{declaration-list}_{opt}) \to M(S,\ S,\ D,\ V,\ \{\textit{identifier}\})$$

$$\begin{aligned}
&\mathcal{D}_{\text{PDL}}[\![S,\ T,\ D \triangleright \textit{ds irl;}]\!] = \text{do}\\
&\quad (\bar{sc},\ \bar{tq},\ \bar{ts}) \leftarrow \mathcal{D}_{\text{DS}}(\textit{ds})\\
&\quad \text{require } (\bar{sc} \subseteq \{\textbf{register}\})\\
&\quad (S_1,\ T_1,\ D_1,\ t_b) \leftarrow \mathcal{D}_{\text{TS}}(S,\ T,\ D,\ 1 \triangleright \bar{ts})\\
&\quad (S_2,\ T_2,\ D_2,\ V,\ \bar{x}) \leftarrow \mathcal{D}_{\text{PRL}}(S_1,\ T_1,\ D_1,\ \bar{sc},\ \bar{tq} \nleftarrow t_b \triangleright \textit{irl})\\
&\quad \text{return } (S_2,\ T_2,\ D_2,\ V,\ \bar{x})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{PDL}}[\![S,\ T,\ D \triangleright \textit{dl d}]\!] = \text{do}\\
&\quad (S_1,\ T_1,\ D_1,\ V_1,\ \bar{x}_1) \leftarrow \mathcal{D}_{\text{PDL}}(S,\ T,\ D \triangleright \textit{dl})\\
&\quad (S_2,\ T_2,\ D_2,\ V_2,\ \bar{x}_2) \leftarrow \mathcal{D}_{\text{PDL}}(S_1,\ T_1,\ D_1 \triangleright \textit{d})\\
&\quad \text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ \bar{x}_1 \cup \bar{x}_2)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{PDL}}[\![S,\ T,\ D \triangleright \epsilon]\!] \quad = \text{return } (S,\ T,\ D,\ \varnothing,\ \varnothing)\\
&\mathcal{D}_{\text{PDL}}[\![S,\ T,\ D \triangleright \textit{ds;}]\!] = \text{reject}
\end{aligned}$$

Each initialised declarator of such a construct has a meaning similar to that of an analogous parameter declaration with the same identifier and C type, except that this type is not contributed to the prototype of the newly-defined function. Explicit initialisers are always disallowed in the syntax of function definitions, so that the semantics of this final construct in the C grammar can be described by the following simple translation:

$$\begin{aligned}
&\mathcal{D}_{\text{PRL}}[\![\cdot]\!] :: (\text{monad-fix } M) \Rightarrow\\
&\quad (S,\ S,\ D,\ [\textit{storage-class-specifier}],\ \textit{type} \triangleright \textit{init-declarator-list}) \to\\
&\quad M(S,\ S,\ D,\ V,\ \{\textit{identifier}\})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{PRL}}[\![S,\ T,\ D,\ \bar{sc},\ t_b \triangleright r]\!] = \text{do}\\
&\quad (S_r,\ T_r,\ D_r,\ t,\ x) \leftarrow \mathcal{D}_{\text{R}}(S,\ T,\ D,\ 1,\ t_b \triangleright r)\\
&\quad (\ell) \leftarrow \mathcal{D}_{\text{SC}}(S_r,\ T_r,\ D_r,\ 1,\ \text{pp}(t),\ x \triangleright \bar{sc})\\
&\quad \text{require } (\text{OBJ}(\text{pp}(t)) \wedge (x \notin \text{dom}(S_r) \vee I(S_r(x)) \neq 1))\\
&\quad \text{return } (S_r/\{x{:}[\![\ell\ t\ \texttt{@}\ 1]\!]\},\ T_r,\ D_r/\{\nu(\ell){:}\epsilon\},\ \{\nu(\ell){:}\bar{\xi}(\text{pp}(t))\},\ \{x\})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_{\text{PRL}}[\![S,\ T,\ D,\ \bar{sc},\ t_b \triangleright \textit{irl, ir}]\!] = \text{do}\\
&\quad (S_1,\ T_1,\ D_1,\ V_1,\ \bar{x}_1) \leftarrow \mathcal{D}_{\text{PRL}}(S,\ T,\ D,\ \bar{sc},\ t_b \triangleright \textit{irl})\\
&\quad (S_2,\ T_2,\ D_2,\ V_2,\ \bar{x}_2) \leftarrow \mathcal{D}_{\text{PRL}}(S_1,\ T_1,\ D_1,\ \bar{sc},\ t_b \triangleright \textit{ir})\\
&\quad \text{return } (S_2,\ T_2,\ D_2,\ V_1 \cup V_2,\ \bar{x}_1 \cup \bar{x}_2)
\end{aligned}$$

$$\mathcal{D}_{\text{PRL}}[\![S,\ T,\ D,\ \bar{sc},\ t_b \triangleright r \texttt{ = } i]\!] = \text{reject}$$

With this final definition, we have now attained a complete formal description of the C programming language. Although arguably obtuse at times, our translation of C programs into Etude is no more difficult to absorb than any other semantic notation, while carrying the additional benefit of a fully-usable compiler implementation, whose correctness is guaranteed by its automatic derivation from the same source as that used to describe the behaviour of all C programs to the eventual users of the system.

5.11 Extended Example

Much can be said about the translation of C programs described on the previous 124 pages. The reader should, by now, be aware that the $\mathcal{D}_{\text{TU}}$ algorithm defined in Section 5.10 specifies both the semantic meaning of an entire translation unit in a C program and a compiler for that unit that targets some unspecified Etude implementation, such as the one described later in Chapter 6 and Appendix C. When applied to some translation unit *tu* in the context of an exception monad akin to the one represented by the standard Haskell type "Maybe", $\mathcal{D}_{\text{TU}}(tu)$ always derives an Etude module *m* that captures the unit's precise meaning, or else returns the monad's exceptional value "fail" if *tu* fails to satisfy some of the constraints imposed on its syntax by the C Standard. If desired, the precise reason for rejection of *tu* could be easily incorporated into that "fail" value. In fact, such diagnostic messages are already present in the actual Haskell source code from which this presentation has been prepared and they are only hidden in its typesetting for the sake of conciseness.

By its very nature, the semantic translation captured in Section 5.10 by the function $\mathcal{D}_{\text{TU}}$ can only handle translation units that, in Haskell, are represented as well-defined finite data structures. For example, the translation process would always diverge if the undefined Haskell value "$\perp$" or some infinitely-recursive list of syntactic entities was to be embedded somewhere within *tu*. Due to the inherent unsolvability of the halting problem, there is very little that can be done to alleviate this limitation of the intentional compiler design technique pursued in this work. Fortunately, in real life, all translation units are produced by a parser from their finite textual representations and, accordingly, in practice, they will never contain such rogue entities.

Nevertheless, it seems advisable to formalise the precise termination guarantees for our compiler. Under the extended Haskell type system from Chapter 3, this criteria can be can described concisely by the following trivial theorem, which, intuitively, states that, for every well-defined C translation unit *tu*, the Etude module produced by its translation $\mathcal{D}_{\text{TU}}(tu)$ is likewise guaranteed to be well-formed:

$$\text{TU} :: \forall\, tu \Rightarrow \text{WF}(tu) \rightarrow \text{WF}(\mathcal{D}_{\text{TU}}(tu) :: module_{opt})$$

assuming that the default definition of the property "WF:: $translation\text{-}unit_{opt} \rightarrow \star$" has been derived implicitly from the structure of the Haskell type "$translation\text{-}unit_{opt}$", in accordance with the algorithm described in Section 3.6, which ensures that the value of *tu* does not contain any infinite or otherwise undefined components embedded within its syntax and, further, observing that the exceptional value "Nothing" of the Haskell monad "Maybe" is always well-formed under every such implicitly-derived definition of that property.

A detailed proof of this theorem is left as an exercise for the reader. Strictly speaking, it is not required for a simple establishment of the linear correctness property for our compiler, since, under its definition in Chapter 2, that property is only concerned with those compiler inputs that result in a meaningful Etude program after transla-

tion. A detailed formulation of this proof is complicated by the translation's reliance on partial Haskell functions, circular definitions and outright-infinite data structures. Nevertheless, it is my belief that a substantial progress in the area could be made with the aid of capable termination checkers such as AProVE [Giesl 04], which is already geared towards analysis of Haskell programs.

In the meantime, let us conclude our treatment of C with a detailed translation of a realistic C program into its Etude representation. In particular, let us consider the following textbook implementation of the quick sort algorithm in C:

```
void swap(int *a, int *b) {
        int t = *a;
        *a = *b;
        *b = t;
}

void sort(int a[], int i, int j) {
        if (j > i + 1) {
                int p = a[i], l = i + 1, r = j;
                while (l < r) {
                        if (a[l] <= p) {
                                ++l;
                        }
                        else {
                                swap(&a[l], &a[--r]);
                        }
                }
                swap(&a[--l], &a[i]);
                sort(a, i, l);
                sort(a, r, i);
        }
}
```

From this input, the algorithm $\mathcal{D}_{\mathrm{TU}}$ derives the following Etude module:

```
MODULE
    LET () = ();
    RET ()
EXPORT
    "iswap" = swap
    "sort"  = sort
WHERE
    swap  = λV2, V3. ...
    V5    = λV2, V3. ...
    sort  = λV7, V8, V9. ...
    V13   = λV7, V8, V9, V10, V11, V12. ...
    V16   = λV7, V8, V9, V10, V11, V12. ...
    V17   = λV7, V8, V9, V10, V11, V12. ...
    V26   = λV7, V8, V9. ...
    V27   = λV7, V8, V9. ...
```

which exports precisely the two externally-linked C functions `swap` and `sort`, bound

internally to Etude variables of the same names. The constructed module initialiser represents a sequence of two trivial Etude terms "RET ()" contributed to the program by the pair of external declarations in the above translation unit. Given a simple transformation of the program derived from the algebraic semantics of Etude terms described in Section 4.6, this initialiser could be easily reduced into a single "RET ()" construct, but, for the sake of conciseness, all such optimisations are avoided by our compiler at the present time.

The set of item definitions in the constructed module contains the bindings of *swap* and *sort* to their corresponding Etude functions, together with a specification of six additional functions V_5, V_{13}, V_{16}, V_{17}, V_{26} and V_{27} that serve as denotations of the individual basic blocks in the program. Their parameter lists reflect the variable scopes current at the beginning of the corresponding labelled statements and their actual bodies are discussed systematically throughout the remainder of this section.

In particular, the definition of *swap* introduces two basic blocks *swap* and V_5 into the module, corresponding to the function's entry and exit points, respectively. The actual Etude functions used to represent these basic blocks accept two parameter variables V_2 and V_3, which correspond to the respective parameter declarations "**int** ⋆*a*" and "**int** ⋆*b*". Observe that, since the C function *swap* has the return type of "***void***", the parameter list of its Etude translation does not include a result object, so that the variable V_1 reserved for that purpose by $\mathcal{D}_{\text{TU}}$ does not appear in the following Etude code:

swap = λV_2, V_3.

First of all, *swap* begins with a translation of the C declaration "**int** *t*". Under the MMIX architecture defined in Chapter 6 and the corresponding C implementation from Appendix C, all objects of this C type are associated with the Etude format "Z.32", so that this declaration is translated into the term "NEW (0, Z.32)" and bound to a new Etude variable V_4, which will henceforth represent its C counterpart *t* within the program, observing that the object's envelope does not include any access attributes, since *t* has an unqualified type:

```
LET V4 = NEW (0, Z.32);
```

In the resulting monadic sequence, this declaration is followed immediately by an Etude depiction of its initialiser "*t* **=** ⋆*a*":

```
LET () = LET T1 = LET T1 = LET T1 = V2;
                                GET [T1]O.62;
                           GET [T1]Z.32;
         LET () = SETI [V4]Z.32 TO (Z.32Z.32(T1));
         RET ();
```

The innermost "LET ... GET" term form of this construct obtains the value of the argument *a*, as depicted by the Etude variable V_2, which, under the MMIX architecture, is assigned the standard object format "O.62" common to all integer pointers. Next, the surrounding "LET ... GET" implements the indirection operation "⋆*a*" and, finally, the

following "LET ... SET_I" populates the memory-resident object *t* with an actual result of that operation. Throughout the entire construction, every intermediate result is stored in the same temporary Etude variable T_1.

Next comes the Etude rendition of the assignment operation "*****a **=** *****b", whose translation follows a similar structure to that of the initialiser described above, observing, however, that, in the present construct, the two operands of "**=**" are placed in the same Etude "LET" group, in order to capture the unspecified aspect of their evaluation order. Specifically, the following term binds the l-value "*****a" to its Etude counterpart T_1, the value of "*****b" to T_2 and, finally, saves the result of converting "*****a" into the type of the targeted object as the temporary Etude variable T_3:

```
LET () = LET T1 = LET T1 = LET T1 = V2;
                               GET [T1]o.62,
                           T2 = LET T1 = LET T1 =V3;
                                         GET [T1]o.62;
                               GET [T1]z.32;
                    LET T3 = z.32z.32(T2);
                    LET () = SET [T1]z.32 TO T3;
                    RET (T3);
         RET ();
```

In the same manner, the next expression statement "*****b **=** *t*" is depicted by a "LET" structure of the following form:

```
LET () = LET T1 = LET T1 = LET T1 = V3;
                               GET [T1]o.62,
                           T2 = LET T1 = V4;
                               GET [T1]z.32;
                    LET T3 = z.32z.32(T2);
                    LET () = SET [T1]z.32 TO T3;
                    RET (T3);
         RET ();
```

Before returning to the caller, the function purges its local variable *t* from the program's address space using a term of the form "DEL ($\bar{\xi}$)", in which $\bar{\xi}$ is the same envelope as that applied by the corresponding "NEW ($\bar{\xi}$)" term earlier within the function. Observe that Etude is perfectly capable of figuring out that the operation pertains to the most-recently introduced object, which, in this case, represents the function's sole contribution to its address space, i.e., the required variable *t*:

```
LET () = DEL (0, z.32);
```

Once the original address space current upon the original entry into *swap* has been restored, we can proceed with a jump to the function's exit block V_5, as depicted by the following tail call, whose argument list is constructed from all of the remaining variables in the current scope, i.e, the two function arguments *a* and *b*:

```
V5(V2, V3)
```

Like every such exit block in the program, an actual body of the Etude function V_5 consists entirely of the trivial Etude term "RET ()", that, in effect, leaves the continuation passing structure of the function's control-flow graph and returns immediately to its original caller:

$V_5 = \lambda V_2,\ V_3.$
RET ()

A translation of the *sort* function is rather more complicated. In its Etude representation, the three parameter variables *a*, *i* and *j* are depicted by V_7, V_8 and V_9. Once again, since this C function has a "**void**" type, the variable V_6 reserved for its returned object is left unused in the program:

sort $= \lambda V_7,\ V_8,\ V_9.$

First of all, *sort* begins with a trivial Etude term "RET ()" derived from the empty list of declarations hidden at the beginning of its body statement:

LET () = ();

It then proceeds with an evaluation of the controlling expression "*j* **>** *i* **+** 1" found in its outermost "**if**" statement, using a nested "LET" structure akin to the earlier depiction of the assignment operations in *swap*:

LET T_1 = LET T_1 = LET T_1 = LET T_1 = V_9;
GET $[T_1]_{Z.32}$,
T_2 = LET T_1 = LET T_1 = V_8;
GET $[T_1]_{Z.32}$,
T_2 = $\#1_{Z.32}$;
RET $((Z.32_{Z.32}(T_1)) +_{Z.32} (Z.32_{Z.32}(T_2)))$;
RET $((Z.32_{Z.32}(T_1)) >_{Z.32} (Z.32_{Z.32}(T_2)))$;
RET $(T_1 \neq_{Z.32} \#0_{Z.32})$;

Once T_1 has been set to the desired condition, its value is applied in the actual "**if**" statement as follows. In the "true" branch, we begin with a declaration of the three integer variables *p*, *l* and *r* found at the top of the corresponding compound statement, which will henceforth be represented by V_{10}, V_{11} and V_{12}, respectively:

IF T_1 THEN
LET V_{10} = NEW (0, Z.32);
LET V_{11} = NEW (0, Z.32);
LET V_{12} = NEW (0, Z.32);

Next, the new memory-resident object *p* that is specified by the first of these variables V_{10} must be populated with its initialiser expression "*a* **[** *i* **]**":

LET () = LET () = LET () = LET T_1 = LET T_1 = LET T_1 = V_7,
T_2 = LET T_1 = V_8;
GET $[T_1]_{Z.32}$;
RET $(T_1 +_{O.62}$
$((Z.64_{Z.32}(T_2)) \times_{Z.64} \#4_{Z.64}))$;
GET $[T_1]_{Z.32}$;
LET () = SET_I $[V_{10}]_{Z.32}$ TO $(Z.32_{Z.32}(T_1))$;
RET ();

In the above term, the reader will doubtless notice the meticulous depiction of the pointer arithmetic performed by the initialiser, recalling that all array subscript operations such as "*a*`[`*i*`]`" are essentially rendered equivalent to "`*((`*a*`)+`*i*`)`" by their translation in Section 5.7. Under that interpretation, the l-value examined by the above initialiser is depicted by the Etude expression "$\mathrm{T}_1 +_{\mathrm{O}.62} ((\mathsf{Z}.64_{\mathsf{Z}.32}(\mathrm{T}_2)) \times_{\mathsf{Z}.64} \#4_{\mathsf{Z}.64})$" which, intuitively, increments the value of the pointer *a* that is stored in T_1 by the product of the integer *i* (or T_2), converted into the standard integer format "Z.64" and multiplied by the size of the pointer's referenced type, equal to 4 bytes on the MMIX architecture.

Similarly, the C variable *l*, which masquerades as V_{11} in the constructed Etude module, is initialised to "*i* `+` 1":

```
LET T1 = LET T1 = LET T1 = V8;
                          GET [T1]z.32,
                 T2 = #1z.32;
          RET ((Z.32z.32(T1)) +z.32 (Z.32z.32(T2)));
LET () = SETi [V11]z.32 TO (Z.32z.32(T1));
RET ();
```

Finally, the variable *r* (or V_{12}) is assigned the value of *j* as requested by the source program:

```
LET T1 = LET T1 = V9;
         GET [T1]z.32;
LET () = SETi [V12]z.32 TO (Z.32z.32(T1));
RET ();
```

The unusually-deep nesting structure of these initialisers is a direct result of the means by which the denotations of individual initialiser expressions are composed into a single Etude term during their translation in Section 5.8.11. In particular, given two initialisers with the meaning of τ_1 and τ_2, the collective denotation of the surrounding declarator list would have the form of "`LET () =` τ_1; τ_2", although the reader should not be overly alarmed by this complexity, given that even the simplest of all optimising compilers would break no sweat when asked to reduce the resulting structure back into its pleasantly-flat representation.

Once all three of our local variables have been assigned their correct initial values, we are ready to proceed with a call to the body of the translated "**`while`**" loop, whose basic block is bound below to the Etude variable V_{13}, supplying it with all six local variables visible in the current scope, i.e., *a*, *i*, *j*, *p*, *l* and *r*:

```
V13(V7, V8, V9, V10, V11, V12)
```

Otherwise, in the implied "**`else`**" branch of the above "**`if`**" statement, control is simply transferred to the conditional's exit block V_{26}, whose local scope consists entirely of the three function arguments *a*, *i* and *j*:

```
ELSE
    V26(V7, V8, V9)
```

The actual body of the loop found at the heart of *swap* is represented in Etude by the following function:

$$V_{13} = \lambda V_7,\ V_8,\ V_9,\ V_{10},\ V_{11},\ V_{12}.$$

which, predictably, begins with an evaluation of the controlling expression "*l* **<** *r*" from the corresponding "**while**" statement:

$$\begin{array}{l}\mathsf{LET}\ T_1 = \mathsf{LET}\ T_1 = \mathsf{LET}\ T_1 = \mathsf{LET}\ T_1 = V_{11};\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{GET}\ [T_1]_{z.32},\\ \qquad\qquad\qquad\qquad\quad T_2 = \mathsf{LET}\ T_1 = V_{12};\\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{GET}\ [T_1]_{z.32};\\ \qquad\qquad\qquad\qquad\ \mathsf{RET}\ ((z.32_{z.32}(T_1)) <_{z.32} (z.32_{z.32}(T_2)));\\ \qquad\qquad\quad \mathsf{RET}\ (T_1 \neq_{z.32} \#0_{z.32});\end{array}$$

If that expression has a true value, we can proceed with an evaluation of the loop's nested statement. Once again, that statement begins with an empty set of declaration initialisers that are translated into the trivial Etude term "RET ()":

$$\begin{array}{l}\mathsf{IF}\ T_1\ \mathsf{THEN}\\ \quad \mathsf{LET}\ ()\ = ();\end{array}$$

Here, the first task on the program's agenda is an evaluation of the controlling expression "*a* **[** *l* **]** **<=** *p*" from the "**if**" statement embedded within the loop's body:

$$\begin{array}{l}\mathsf{LET}\ T_1 = \mathsf{LET}\ \ T_1 = \mathsf{LET}\ T_1 = \mathsf{LET}\ T_1 = \mathsf{LET}\ T_1 = V_7,\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad T_2 = \mathsf{LET}\ T_1 = V_{11};\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{GET}\ [T_1]_{z.32};\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{RET}\ (T_1 +_{o.62}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ((z.64_{z.32}(T_2)) \times_{z.64} \#4_{z.64}));\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{GET}\ [T_1]_{z.32},\\ \qquad\qquad\qquad\qquad\qquad\quad T_2 = \mathsf{LET}\ T_1 = V_{10};\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{GET}\ [T_1]_{z.32};\\ \qquad\qquad\qquad\qquad\qquad\ \mathsf{RET}\ ((z.32_{z.32}(T_1)) \leq_{z.32} (z.32_{z.32}(T_2)));\\ \qquad\qquad\qquad\quad \mathsf{RET}\ \ (T_1 \neq_{z.32} \#0_{z.32});\end{array}$$

If true, this expression prompts a reduction of the unary increment operation "**++***l*", as represented by the following predictable "LET" structure:

$$\begin{array}{l}\mathsf{IF}\ T_1\ \mathsf{THEN}\\ \quad \mathsf{LET}\ ()\ \ = \mathsf{LET}\ T_1 = \mathsf{LET}\ T_1 = V_{11},\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad T_2 = \#1_{z.32};\\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{LET}\ T_3 = \mathsf{GET}\ [T_1]_{z.32};\\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{LET}\ T_4 = T_3 +_{z.32} (z.32_{z.32}(T_2));\\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{LET}\ ()\ = \mathsf{SET}\ [T_1]_{z.32}\ \mathsf{TO}\ T_4;\\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{RET}\ (T_4);\\ \qquad\qquad\qquad\quad \mathsf{RET}\ ();\end{array}$$

Once the value of *l* has been updated, we simply proceed with a jump to the exit block V_{16} of the current selection statement, retaining all six of the C variables *a*, *i*, *j*, *p*, *l* and *r* from its surrounding scope:

$$V_{16}(V_7,\ V_8,\ V_9,\ V_{10},\ V_{11},\ V_{12})$$

Otherwise, the program allocates a pair of internal pointer variables V_{14} and V_{15}, whose Etude format "O.64" corresponds directly to the C type "**`int *`**", as required for storage of the argument values during the upcoming call to the *`swap`* function:

```
ELSE
    LET V14 = NEW (0, O.62);
    LET V15 = NEW (0, O.62);
```

The actual address of that function is then placed into the temporary variable T_1:

```
LET () = LET T1 = swap,
```

and the temporary argument object V_{14}, which corresponds to the function's initial operand *`a`*, is populated with its required value of "**`&`***`a`***`[`***`1`***`]`**":

```
() = LET T1 = LET T1 = V7,
                  T2 = LET T1 = V11;
                       GET [T1]z.32;
             RET (T1 +o.62 ((z.64z.32(T2)) ×z.64 #4z.64));
    SETi [V14]o.62 TO (o.62o.62(T1)),
```

Similarly, the second argument V_{15} is set to "**`&`***`a`***`[--`***`r`***`]`**":

```
() = LET T1 = LET T1 = V7,
                  T2 = LET T1 = V12,
                           T2 = #1z.32;
                       LET T3 = GET [T1]z.32;
                       LET T4 = T3 −z.32 (z.32z.32(T2));
                       LET () = SET [T1]z.32 TO T4;
                       RET (T4);
             RET (T1 +o.62 ((z.64z.32(T2)) ×z.64 #4z.64));
    SETi [V15]o.62 TO (o.62o.62(T1));
```

at which point we are ready to perform the actual function application:

```
T1(V14, V15);
```

Next, we wrap up the representation of the current expression statement by purging the temporary variables V_{15} and V_{14} from the program's address space, in the reverse order of their earlier introduction by the corresponding "NEW" terms:

```
LET () = DEL (0, O.62);
LET () = DEL (0, O.62);
```

Finally, we can conclude the entire "**`if`**" statement with a jump to its exit block V_{16}, once again retaining the entire scope of the surrounding statement:

```
V16(V7, V8, V9, V10, V11, V12)
```

Otherwise, the loop's controlling expression "*`l`* **`<`** *`r`*" must have a false value and, accordingly, we can leave the "**`while`**" statement immediately with a jump to its exit block V_{17} that is rendered in Etude as follows:

```
ELSE
    V17(V7, V8, V9, V10, V11, V12)
```

As already mentioned, the Etude variable V_{16} is bound to the exit block of the inner "`if`" statement in the body of *swap*, which simply transfers control back to the beginning of the loop, as represented by the following tail call to the corresponding Etude function V_{13}:

$$V_{16} = \lambda V_7,\ V_8,\ V_9,\ V_{10},\ V_{11},\ V_{12}.\\ \quad V_{13}(V_7,\ V_8,\ V_9,\ V_{10},\ V_{11},\ V_{12})$$

On the other hand, the loop's own exit block V_{17} continues as follows:

$$V_{17} = \lambda V_7,\ V_8,\ V_9,\ V_{10},\ V_{11},\ V_{12}.$$

First, we need to allocate a pair of new Etude variables V_{18} and V_{19} for the impending call to the C function *swap*:

$$\begin{array}{l}\mathsf{LET}\ V_{18} = \mathsf{NEW}\ (0,\ \mathsf{O}.62);\\ \mathsf{LET}\ V_{19} = \mathsf{NEW}\ (0,\ \mathsf{O}.62);\end{array}$$

As per the original C program, their values are initialised to "`&`*a*`[--`*l*`]`" and "`&`*a*`[`*i*`]`". The actual call is then performed by the application term "$T_1(V_{18},\ V_{19})$":

$$\begin{array}{l}\mathsf{LET}\ () = \mathsf{LET}\ T_1 = \mathit{swap},\\ \quad () = \mathsf{LET}\ T_1 = \mathsf{LET}\ T_1 = V_7,\\ \quad\quad T_2 = \mathsf{LET}\ T_1 = V_{11},\\ \quad\quad\quad T_2 = \#1_{\mathsf{Z}.32};\\ \quad\quad\quad \mathsf{LET}\ T_3 = \mathsf{GET}\ [T_1]_{\mathsf{Z}.32};\\ \quad\quad\quad \mathsf{LET}\ T_4 = T_3 -_{\mathsf{Z}.32} (\mathsf{Z}.32_{\mathsf{Z}.32}(T_2));\\ \quad\quad\quad \mathsf{LET}\ () = \mathsf{SET}\ [T_1]_{\mathsf{Z}.32}\ \mathsf{TO}\ T_4;\\ \quad\quad\quad \mathsf{RET}\ (T_4);\\ \quad\quad \mathsf{RET}\ (T_1 +_{\mathsf{O}.62} ((\mathsf{Z}.64_{\mathsf{Z}.32}(T_2)) \times_{\mathsf{Z}.64} \#4_{\mathsf{Z}.64}));\\ \quad \mathsf{SET}_{\mathsf{I}}\ [V_{18}]_{\mathsf{O}.62}\ \mathsf{TO}\ (\mathsf{O}.62_{\mathsf{O}.62}(T_1)),\\ \quad () = \mathsf{LET}\ T_1 = \mathsf{LET}\ T_1 = V_7,\\ \quad\quad T_2 = \mathsf{LET}\ T_1 = V_8;\\ \quad\quad\quad \mathsf{GET}\ [T_1]_{\mathsf{Z}.32};\\ \quad\quad \mathsf{RET}\ (T_1 +_{\mathsf{O}.62} ((\mathsf{Z}.64_{\mathsf{Z}.32}(T_2)) \times_{\mathsf{Z}.64} \#4_{\mathsf{Z}.64}));\\ \quad \mathsf{SET}_{\mathsf{I}}\ [V_{19}]_{\mathsf{O}.62}\ \mathsf{TO}\ (\mathsf{O}.62_{\mathsf{O}.62}(T_1));\\ T_1(V_{18},\ V_{19});\end{array}$$

At the end of the entire expression statement, these temporary variables are no longer required and, accordingly, they are purged from the program's address space, ignoring the fact that they could be easily reused in a subsequent call to *sort*:

$$\begin{array}{l}\mathsf{LET}\ () = \mathsf{DEL}\ (0,\ \mathsf{O}.62);\\ \mathsf{LET}\ () = \mathsf{DEL}\ (0,\ \mathsf{O}.62);\end{array}$$

Next comes a translation of the expression statement "*sort*`(`*a*`, `*i*`, `*l*`);`", for which we reserve the three Etude variables V_{20}, V_{21} and V_{22}, observing that the first argument of *sort* represents a pointer, but the other two have the integer format "Z.32":

$$\begin{array}{l}\mathsf{LET}\ V_{20} = \mathsf{NEW}\ (0,\ \mathsf{O}.62);\\ \mathsf{LET}\ V_{21} = \mathsf{NEW}\ (0,\ \mathsf{Z}.32);\\ \mathsf{LET}\ V_{22} = \mathsf{NEW}\ (0,\ \mathsf{Z}.32);\end{array}$$

The actual function call operation "*sort* **(***a, i, l***)**" is then performed by the following sequence of Etude assignments and application terms:

```
LET ()  = LET T1 = sort,
              () = LET T1 = V7;
                   SETI [V20]O.62 TO (O.62O.62(T1)),
              () = LET T1 = LET T1 = V8;
                            GET [T1]Z.32;
                   SETI [V21]Z.32 TO (Z.32Z.32(T1)),
              () = LET T1 = LET T1 = V11;
                            GET [T1]Z.32;
                   SETI [V22]Z.32 TO (Z.32Z.32(T1));
          T1(V20, V21, V22);
```

Once again, the three argument objects V_{22}, V_{21} and V_{20} are quietly discarded at the end of that expression statement, in the proper reverse order of their earlier allocation:

```
LET ()  = DEL (0, Z.32);
LET ()  = DEL (0, Z.32);
LET ()  = DEL (0, O.62);
```

Similarly, the second call to *swap* with the argument list of "**(***a, r, i***)**" is translated into the following monadic sequence:

```
LET V23 = NEW (0, O.62);
LET V24 = NEW (0, Z.32);
LET V25 = NEW (0, Z.32);

LET ()  = LET T1 = sort,
              () = LET T1 = V7;
                   SETI [V23]O.62 TO (O.62O.62(T1)),
              () = LET T1 = LET T1 = V12;
                            GET [T1]Z.32;
                   SETI [V24]Z.32 TO (Z.32Z.32(T1)),
              () = LET T1 = LET T1 = V8;
                            GET [T1]Z.32;
                   SETI [V25]Z.32 TO (Z.32Z.32(T1));
          T1(V23, V24, V25);

LET ()  = DEL (0, Z .32);
LET ()  = DEL (0, Z .32);
LET ()  = DEL (0, O .62);
```

At this point, all that remains in the program is the purging from its address space of the three C variables *p*, *l* and *r*, which are about to leave the scope of the inner compound statement in the body of *sort*:

```
LET ()  = DEL (0, Z.32);
LET ()  = DEL (0, Z.32);
LET ()  = DEL (0, Z.32);
```

Finally, we are ready to conclude the outer "**if**" statement in *sort* with a jump to its exit block V_{26}:

```
V26(V7, V8, V9)
```

where the actual implementation of that exit block is depicted in Etude by a trivial jump to the exit block v_{27} of the entire `sort` function:

$$\begin{aligned}&\mathrm{v}_{26} = \lambda \mathrm{v}_7,\ \mathrm{v}_8,\ \mathrm{v}_9.\\&\qquad \mathrm{v}_{27}(\mathrm{v}_7,\ \mathrm{v}_8,\ \mathrm{v}_9)\end{aligned}$$

Last but not least, the exit block of `swap` itself simply returns to its ultimate caller:

$$\begin{aligned}&\mathrm{v}_{27} = \lambda \mathrm{v}_7,\ \mathrm{v}_8,\ \mathrm{v}_9.\\&\qquad \mathsf{RET}\ ()\end{aligned}$$

Perhaps the most striking feature of the above translation is its extraordinary verbosity, whereby a mere 15 lines of C has been converted into a whooping 182 lines of Etude term definitions. Unfortunately, such code explosion is all but unavoidable under translation of a high-level language akin to C into a low-level program representation such as Etude, in which all of the computational complexities imposed by the underlying instruction set architecture must be exposed with an excruciating amount of detail.

In this work, the problem of loquacity is further aggravated by our omission of even the simples optimising program transformations, which could be easily applied to the above Etude modules. For example, if we chose to propagate trivial atomic bindings such as "$\mathsf{LET}\ \bar{\nu} = \bar{\alpha};\ \tau$" into the body term τ, eliminate redundant operations such as automorphic conversions of the form "$\phi_\phi(\alpha)$" that specify identical source and target formats and flatten the program's deep nesting of "LET" statements, then the above module would be reduced to less then a half of its present length. It would also be easy to demonstrate that all such optimisations preserve the algebraic model of Etude defined earlier in Chapter 4 and, accordingly, to detach them from the actual translation semantics of the C programming language.

However, when it comes to a reasoning about the actual properties of individual C programs, the formal model presented in this chapter should, in principle, be able to facilitate arbitrary verification tasks for all C programs that are well-formed under the portable subset of the language. When combined with the additional information derived from its MMIX specialisation in Chapter 6 and Appendix C, even programs that rely on various implementation-defined behaviours can be analysed successfully. While it is certainly true that, due to the great amount of detail introduced by the translation, most non-trivial verification tasks would require a sensible engineering regime and, more likely than not, some degree of automation from appropriate tools, the relative directness with which the individual C constructs have been mapped onto their Etude counterparts ensures that such tools and practices can, in principle, be devised entirely within the formal framework presented in this work.

6

GENERATING CODE

Errors using inadequate data are much less
than those using no data at all.
— Charles Babbage (1792–1871)

Ultimately, the elementary purpose of every compiler is a representation of its input programs in the designated target language, a rôle that has been traditionally assigned to a portion of its implementation known as a *back end*. It is this final stage of the entire translation process to which I will now turn the reader's attention.

A satisfactory formal verification of a compiler's back end poses some non-trivial challenges beyond those already encountered in the previous two chapters. Usually, the rôle of the target language is assumed by an *instruction set architecture* of some computational hardware such as ARM, MIPS, SPARC or x86. In these languages, programs are represented universally by sequences of primitive entities known as *instructions*, every one of which is encoded into a bit pattern known variously as a *byte* or a *word*, depending on its precise size and semantics. Accordingly, the structure of typical target programs differs significantly from their rendition under languages such as C or Etude, which offer a vastly disparate abstraction of variables, objects and control flow structures. These fundamental differences already pose a substantial challenge for the design of a robust translation system, but even if we could somehow overcome all such difficulties, the task of establishing correctness for a compiler's back end remains complicated by a number of further issues.

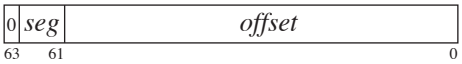
First, of all, in the final stage of translation, we cannot readily assume the approach from Chapter 5, whereby the source language was defined through a mapping of its syntactic constructs onto the compiler's intermediate program representation. It would be futile to attempt a definition of Etude in terms of any given instruction set architecture, since most users would shiver at the thought of their programs having incomparable meanings on different computers. It would be equally risky to attack the problem using conventional extensional techniques, given that a formal establishment of an equivalence between the disparate semantic models of these two computational paradigms would require a precise definition of the targeted system, which few architectures are prepared to provide in the real world. In the interest of relevance, it would be vastly preferable to retain the view of the compiler's target as a formal language rather than an algebra, refraining from a direct operational description of the behaviours exhibited by its machine instructions and, instead, formulating their semantic interpretation through a translation into some other well-understood calculus of computable functions, similar in spirit to the one described in Chapter 5 for our source language C.

As demonstrated earlier in Chapter 2, if the translation semantics of both the source and the target language can be rendered as a pair of total functions ρ and ψ that share a common codomain in the form of the compiler's intermediate program representation, then a correctness of the entire translation system rests on a single equivalence judgement $\rho \circ \hat{\psi} \circ \psi = \rho$, known as *the principle of linear correctness*, in which $\hat{\psi}$ represents the actual code generation function of the translation system. For the present compiler, this judgement is established formally in Section 6.4, together with a formal definition of the translation function $\hat{\psi}$. The actual semantics ψ of our target architecture are described separately in Section 6.2, while a suitable rendition of ρ has been already provided in Chapter 5, in the form of the function $\mathcal{D}_{\text{TU}}$, which maps all valid C programs to their Etude counterparts.

In order to sidestep any religious debates on the relative merits of various popular instruction set architectures such as SPARC or x86, in this chapter I choose to focus on a translation of Etude programs into the MMIX architecture, which was designed for the specific purpose of presenting low-level assembly programs [Knuth 05]. The instruction set of MMIX provides a very realistic sample of the typical computational facilities supported by its industrial counterparts and, in principle, the architecture could be easily rendered into an efficient hardware implementation. Nevertheless, the unparalleled uniformity of all MMIX instruction forms greatly simplify the following discussion, which permits us to focus on the actual academic issues at hand instead of the peculiarities of some given computational system.

6.1 The MMIX Architecture

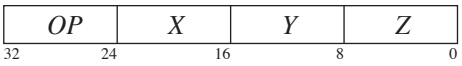
The MMIX instruction set architecture describes a simple yet capable von Neumann machine, in which both programs and data are stored within the same 64-bit virtual address space. The upper half of that address space, i.e., the set of all memory locations with negative address values, is reserved for use by the operating system and remains inaccessible to all ordinary programs of the kind scrutinised in this chapter. The remaining 63 bits of every virtual address are subdivided as follows:



The 2-bit field *seg* selects one of four 2^{61}-byte *segments* numbered 0, 1, 2 and 3, which, by convention, are dedicated to the program's code, data, heap and hardware-maintained data stack, respectively. The 61-bit *offset* field is used to specify the precise location of every memory-resident object stored within one of these segments. In particular, an n-byte object stored in bytes o, $o+1$ ... $o+n-1$ of a given segment s can be found under the address of $2^{61}s + o$. The MMIX architecture utilises the big-endian addressing convention, so that the eight most significant bits of such an object are always placed at the address $2^{61}s + o$.

Besides this *random-access memory*, MMIX also provides 256 individual *general purpose registers* with names of the form **$**0, **$**1, **$**3 and so on. Every one of these registers is capable of storing a single *word* of data formed from a sequence of 64 bits. Depending on its context, such word may be interpreted variously as a natural number n in the range $0 \leq n < 2^{64}$, a signed integer z in the range $-2^{63} \leq z < 2^{63}$, or else as a double precision floating point quantity of the form defined in the IEEE Standard for Binary Floating-Point Arithmetic [IEEE 754]. Further, the architecture defines 32 *special purpose registers*, although only one of these, namely the *remainder register* **rR**, is relevant to a translation of ordinary C programs. Most of the other 31 special purpose registers are reserved for various uses by the operating system and are not discussed further in this work. The architecture follows in the RISC tradition of MIPS and SPARC, in that most of its arithmetic instructions are designed to operate only on values stored in general purpose registers, with all other data storage facilities made accessible only through appropriate *load* and *store* instructions, which facilitate movement of information between general purpose registers and their special purpose counterparts or the program's address space.

Every MMIX instruction is represented uniformly by a 32-bit unsigned integer value, whose binary encoding is organised into four 8-bit fields as follows:



In particular, the *OP* field stored in bits 24 to 31 is known as the instruction's *opcode*. Intuitively, it selects one of the 256 possible MMIX instruction forms such as addition and multiplication, which happen to be assigned the respective opcode numbers 32 and 24. The remaining fields X, Y and Z contain up to three 8-bit operands for the instruction. In most cases, each of these operands selects one of the 256 general purpose registers **$**0 ... **$**255 described earlier, which, in the following discussion, is represented symbolically by the notations "**$***X*", "**$***Y*" and "**$***Z*" for the X, Y and Z operands, respectively. Typically, the register **$***X* is used to store the operation's actual result, while **$***Y* and **$***Z* represent two input values for the instruction. For some opcodes, however, one or more of these three fields may be interpreted directly as an *immediate operand*, in which case its value is always taken to depict an unsigned numeric constant n in the range $0 \leq n \leq 255$.

Although, in real-life, every MMIX program is ultimately represented as a binary object that describes its initial address space configuration and memory image, for the sake of exposition, the following discussion is restricted to an analysis of a simpler symbolic representation of such programs, whereby an abstract syntax of the individual object files, from which that memory image will be eventually assembled by the system's linker, is rendered as a symbolic *MMIX module*. In particular, every such module is represented by a structure of the form "**LET** S **TEXT** T **DATA** D **IN** σ_0", assembled from the *entry symbol* σ_0 and three finite maps S, T and D, known as the module's

symbol table, *text* and *data segments*, respectively. For simplicity, in this work all data segments are represented directly by a set of Etude item bindings, so that, in the following discussion, we will not concern ourselves with the trivial but tedious details of their translation. On the other hand, the text segment of an MMIX module contains the actual instructions of the program, grouped together into sets known as *sections*, that will be eventually inserted into the address space of a complete program at various locations chosen in an unspecified manner by the system's linking algorithm. Until then, the precise addresses of all segments are represented by abstract variables known as *symbols*. Formally, every symbol is depicted either by some integer label "**#***n*", or else by the distinguished value "**LOC**", reserved for a somewhat specialised purpose described later in Section 6.4. The module's symbol table S maps any of its symbols that are exported or imported by the module to appropriate Haskell string values that serve as these symbols' external names during the later linking process. In Haskell, the structure of all symbolic MMIX modules can be represented by the following BNF grammar:

MMIX-module:
    LET *MMIX-symbols* **TEXT** *MMIX-text* **DATA** *MMIX-data* **IN** *MMIX-symbol*

MMIX-symbols:
    MMIX-symbol $\mapsto$ *string*

MMIX-text:
    MMIX-symbol $\mapsto$ *MMIX-section*

MMIX-section:
    $[$*MMIX-instr*$]$

MMIX-data:
    MMIX-symbol $\mapsto$ $item_{MMIX\text{-}var}$

MMIX-symbol:
    # *integer*
    LOC

The actual content of every text section is depicted simply by a list of symbolic MMIX instructions. Every such instruction consists of its opcode, followed by the three operand fields X, Y and Z and an optional symbol. The entire construct is generally written as "*OP* `X, Y, Z @` σ_{opt}", except that the "`@`" keyword is usually omitted for conciseness whenever the instruction's syntax does not include a symbol component. The value of each operand field is depicted simply by a numeric representation of its binary encoding, so that the abstract syntax of all symbolic instructions can be captured in Haskell using the following data type definition:

MMIX-instr:
    MMIX-opcode integer `,` *integer* `,` *integer* `@` $MMIX\text{-}symbol_{opt}$

In a binary representation of MMIX modules, symbolic annotations are traditionally stored separately from the actual instruction stream as *relocation records*. However, since their direct incorporation into the "*MMIX-instr*" type provides for a much more eloquent depiction of the architecture's semantics in Section 6.2, in this work I omit

all details of such binary representations of MMIX instructions, with an understanding that their typical implementation could be easily formalised if deemed necessary at some future time.

Last but not least, values of the individual MMIX opcodes are always depicted in the symbolic representation by their standard *mnemonics* such as "**ADDU**" and "**MULU**". Formally, the complete list of all such MMIX mnemonics figures in the following definition of the Haskell type "*MMIX-opcode*":

MMIX-opcode: one of

TRAP	**FCMP**	FUN	FEQL	**FADD**	**FIX**	**FSUB**	**FIXU**
FLOT	FLOT_I	**FLOTU**	FLOTU_I	SFLOT	SFLOT_I	SFLOTU	SFLOTU_I
FMUL	FCMPE	FUNE	FEQLE	**FDIV**	FSQRT	FREM	FINT
MUL	MUL_I	**MULU**	MULU_I	**DIV**	DIV_I	**DIVU**	DIVU_I
ADD	ADD_I	**ADDU**	ADDU_I	SUB	SUB_I	**SUBU**	SUBU_I
2ADDU	2ADDU_I	4ADDU	4ADDU_I	8ADDU	8ADDU_I	16ADDU	16ADDU_I
CMP	CMP_I	**CMPU**	CMPU_I	NEG	NEG_I	NEGU	NEGU_I
SL	SL_I	**SLU**	SLU_I	**SR**	SR_I	**SRU**	SRU_I
BN	BN_B	**BZ**	BZ_B	BP	BP_B	BOD	BOD_B
BNN	BNN_B	BNZ	BNZ_B	BNP	BNP_B	BEV	BEV_B
PBN	PBN_B	PBZ	PBZ_B	PBP	PBP_B	PBOD	PBOD_B
PBNN	PBNN_B	PBNZ	PBNZ_B	PBNP	PBNP_B	PBEV	PBEV_B
CSN	CSN_I	CSZ	CSZ_I	CSP	CSP_I	CSOD	CSOD_I
CSNN	CSNN_I	CSNZ	CSNZ_I	CSNP	CSNP_I	CSEV	CSEV_I
ZSN	$\mathbf{ZSN_I}$	ZSZ	$\mathbf{ZSZ_I}$	ZSP	$\mathbf{ZSP_I}$	ZSOD	ZSOD_I
ZSNN	$\mathbf{ZSNN_I}$	ZSNZ	$\mathbf{ZSNZ_I}$	ZSNP	$\mathbf{ZSNP_I}$	ZSEV	ZSEV_I
LDB	$\mathbf{LDB_I}$	LDBU	$\mathbf{LDBU_I}$	LDW	$\mathbf{LDW_I}$	LDWU	$\mathbf{LDWU_I}$
LDT	$\mathbf{LDT_I}$	LDTU	$\mathbf{LDTU_I}$	LDO	$\mathbf{LDO_I}$	LDOU	$\mathbf{LDOU_I}$
LDSF	LDSF_I	LDHT	LDHT_I	CSWAP	CSWAP_I	LDUNC	LDUNC_I
LDVTS	LDVTS_I	PRELD	PRELD_I	PREGO	PREGO_I	GO	$\mathbf{GO_I}$
STB	STB_I	STBU	$\mathbf{STBU_I}$	STW	STW_I	STWU	$\mathbf{STWU_I}$
STT	STT_I	STTU	$\mathbf{STTU_I}$	STO	STO_I	STOU	$\mathbf{STOU_I}$
STSF	STSF_I	STHT	STHT_I	STCO	STCO_I	STUNC	STUNC_I
SYNCD	SYNCD_I	PREST	PREST_I	SYNCID	SYNCID_I	PUSHGO	PUSHGO_I
OR	OR_I	ORN	ORN_I	NOR	NOR_I	**XOR**	XOR_I
AND	AND_I	ANDN	ANDN_I	NAND	NAND_I	NXOR	NXOR_I
BDIF	BDIF_I	WDIF	WDIF_I	TDIF	TDIF_I	ODIF	ODIF_I
MUX	MUX_I	SADD	SADD_I	MOR	MOR_I	MXOR	MXOR_I
SETH	SETMH	SETML	**SETL**	**INCH**	**INCMH**	**INCML**	INCL
ORH	ORMH	ORML	ORL	ANDNH	ANDNMH	ANDNML	ANDNL
JMP	JMP_B	PUSHJ	PUSHJ_B	**GETA**	GETA_B	**PUT**	PUT_I
POP	RESUME	SAVE	UNSAVE	SYNC	SWYM	**GET**	TRIP

In this definition, all opcodes are listed in the order of their actual numeric values, so that "**TRAP**" represents opcode 0, "**FCMP**" stands for the numeric value of 1 and so on, up to "**TRIP**", which depicts the final opcode value 255. In order to focus the present discussion on its primary topic of compiler design, rather than the often esoteric

semantics of the MMIX architecture, I restrict the present study to a small subset of all valid MMIX instructions. Specifically, only the 50 opcodes whose mnemonics are emphasised in the above definition of "*MMIX-instr*" are ever mentioned in the remainder of this chapter or assigned a precise semantic interpretation. Further, even though the architecture defines a very capable set of exception handling facilities, for simplicity I assume that all exceptions have been invariably disabled for the entire duration of a program's execution, so that any invalid arithmetic operations can be always assumed to deliver their predictable default values. More so, I take for granted that the processor has been configured permanently with the default IEEE rounding mode for all floating point operations, also computing any denormalised results to their maximum available precisions. The ensuing semantics of the selected 50 MMIX instruction forms are presented with meticulous detail in the following section.

6.2 Semantics of MMIX

In this work, the meanings of all MMIX programs are described through their translation into Etude, as defined by a Haskell function $\mathcal{D}_{\mathtt{M}}$. In particular, the semantic significance of a symbolic module "**LET** S **TEXT** T **DATA** D **IN** σ_0" is represented by the Etude construct "MODULE τ EXPORT $\bar{\xi}$ WHERE $\bar{\iota}$", in which the initialiser term τ represents an application of the Etude function derived from a jump to the start symbol σ_0, the export list $\bar{\xi}$ is extracted from the MMIX symbol table S and the set of item definitions $\bar{\iota}$ is derived from the module's data, symbol and text segments as follows:

$$\mathcal{D}_{\mathtt{M}}[\![\cdot]\!] :: \textit{MMIX-module} \rightarrow \textit{module}_{\textit{MMIX-var}}$$

$$\begin{aligned}
&\mathcal{D}_{\mathtt{M}}[\![\mathbf{LET}\ S\ \mathbf{TEXT}\ T\ \mathbf{DATA}\ D\ \mathbf{IN}\ \sigma_0]\!] \\
&\qquad = [\![\mathsf{MODULE}\ \tau\ \mathsf{EXPORT}\ \bar{\xi}\ \mathsf{WHERE}\ \bar{\iota}]\!] \\
&\quad \text{where } \tau = [\![\sigma_0.0(\pi)]\!] \\
&\qquad \bar{\xi} = \{(S(\sigma_k){:}\sigma_k.0) \mid \sigma_k \leftarrow \mathrm{dom}(S),\ \sigma_k \in (\mathrm{dom}(T) \cup \mathrm{dom}(D))\} \\
&\qquad \bar{\iota} = \{(\sigma_k.0{:}\mathsf{IMP}\ [\![S(\sigma_k)]\!]) \mid \sigma_k \leftarrow \mathrm{dom}(S),\ \sigma_k \notin (\mathrm{dom}(T) \cup \mathrm{dom}(D))\} \\
&\qquad\quad \cup \{(\sigma_k.0{:}D(\sigma_k)) \mid \sigma_k \leftarrow \mathrm{dom}(D)\} \\
&\qquad\quad \cup \{(\sigma_k.0{:}\mathcal{D}_{\mathtt{I}}(\sigma_k.n_k \triangleright \iota_k)) \mid \sigma_k \leftarrow \mathrm{dom}(T),\ (\iota_k{:}n_k) \leftarrow T(\sigma_k)|[0,\ 4\ ...]\}
\end{aligned}$$

In particular, the resulting export list $\bar{\xi}$ includes a subset of the module's symbol table S, corresponding to those symbols from the domain of S whose bindings are provided locally by the associated data or text segment. On the other hand, every undeclared symbol $\sigma_k \in \mathrm{dom}(S)$ is introduced into the constructed module's set of item definitions $\bar{\iota}$ with a binding to an imported Etude item "IMP x", in which x represents the symbol's external name as specified in S. Further, $\bar{\iota}$ also includes the bindings of all object items defined in the supplied data segment D. Finally, it is also extended with a set of function items derived from the text segment T. In particular, each instruction found at some byte offset n into one the text sections σ_k in the entire source module materialises in $\bar{\iota}$ as a binding of the MMIX variable "$\sigma_k.4n$" to an Etude function derived from the instruction's syntax by an algorithm $\mathcal{D}_{\mathtt{I}}$ defined throughout the remainder of the present section.

In the resulting program representation, all Etude variables are depicted universally by values of the following Haskell type:

MMIX-var :
  $ *integer*
  rR
  MMIX-symbol **.** *integer*

Intuitively, variables of the form "**$***N*" are used to denote the 256 general purpose registers provided by every MMIX processor, while the "σ**.***n*" constructs depict symbolic memory locations, or names assigned to various forms of Etude item definitions. Every such variable consists of an MMIX symbol σ that must be associated with some relocatable segment of the surrounding module, together with an integer n that represents a byte offset into that segment's memory-resident image. Finally, a designated symbol "**rR**" represents the architecture's *remainder register* mentioned earlier in Section 6.1.

As implied by the above definition of $\mathcal{D}_{\mathbf{M}}$, every MMIX instruction denotes a separate Etude function that is bound to a symbolic variable "σ**.**4*n*", derived from the instruction's location within the program's eventual address space. The body of such a function represents a monadic term that captures the instruction's operational behaviour. Strictly speaking, only the "**TRAP**" instruction, which is mapped to an Etude system call operation, as well as all of the memory access operations such as "$\mathbf{LDOU}_{\mathbf{I}}$" and "$\mathbf{STOU}_{\mathbf{I}}$", must be represented by monadic term forms, since all other instructions always denote terms that are reducible to pure atomic expressions lifted into the term monad using a trivial "RET" construct. Nevertheless, for the sake of exposition, we will treat all MMIX instructions equally in the following translation.

Further, in the interest of consistency, the denotation of every MMIX instruction accepts the same list of parameters, which includes every conceivable general purpose register **$***N*, as well as the remainder register **rR**. For conciseness, in the following presentation this parameter list will be always represented collectively by the notation "π", which can be presented with a formal mandate by the following Haskell construction:

π :: $parameters_{MMIX\text{-}var}$

π = [**$**0 ... **$**255] ++ [**rR**]

Given the above definition, a jump to a text segment address denoted by some Etude variable ν can be depicted simply as an Etude tail call of the form "$\nu(\pi)$". In a purely-sequential MMIX program, such jumps are performed implicitly at the end of every machine instruction. Specifically, if a given instruction is located at some memory address "$\sigma.n$", then the following instruction is generally represented by the Etude variable "$\sigma.[\![n+4]\!]$", so that the denotations of most MMIX instructions end in an Etude application term of the form "$[\![\sigma.n \oplus 4]\!](\pi)$", in which the notation "$\nu \oplus n$" adjusts the existing offset value of the symbolic variable ν by n bytes as follows:

$[\![\cdot]\!] \oplus [\![\cdot]\!]$ :: *MMIX-var* $\rightarrow$ *integer* $\rightarrow$ *MMIX-var*

$[\![\sigma.n]\!] \oplus [\![m]\!] = [\![\sigma.[\![n+m]\!]]\!]$

In particular, the precise semantics of a given MMIX instruction ι that is located at some variable address ν within the program's address space is represented by the notation "$\mathcal{D}_{\mathtt{I}}(\nu \triangleright \iota)$", which maps every such instruction to an appropriate Etude function with an equivalent operational behaviour. Formally, the algorithm $\mathcal{D}_{\mathtt{I}}$ is defined by a Haskell construction of the following type:

$$\mathcal{D}_{\mathtt{I}}[\![\cdot]\!] :: (\mathit{MMIX\text{-}var} \triangleright \mathit{MMIX\text{-}instr}) \to \mathit{function}_{\mathit{MMIX\text{-}var}}$$

Most simple MMIX operations such as "**ADDU** \$$X$, \$$Y$, \$$Z$" denote Etude functions of the form "$\lambda\pi.\mathtt{LET}\ \$X = \$Y\ op_\phi\ \$Z;\ [\![\nu \oplus 4]\!](\pi)$", in which "$op_\phi$" represents an appropriate binary Etude operator and format. In all cases, ϕ will be set to one of the three standard arithmetic formats "N.64", "Z.64" or "R.64", as appropriate for the instruction's interpretation of its operand and result values. In particular, this simple semantic translation is adopted directly by the following thirteen MMIX instruction forms:

$$\begin{aligned}
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{FADD}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ +_{\mathrm{R}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{FSUB}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ -_{\mathrm{R}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{FMUL}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \times_{\mathrm{R}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{FDIV}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \div_{\mathrm{R}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{ADDU}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ +_{\mathrm{N}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{SUBU}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ -_{\mathrm{N}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{MULU}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \times_{\mathrm{N}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{AND}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \triangle_{\mathrm{N}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{XOR}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \bar{\vee}_{\mathrm{N}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{OR}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \vee_{\mathrm{N}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{SLU}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \ll_{\mathrm{N}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{SRU}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \gg_{\mathrm{N}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{SR}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \gg_{\mathrm{Z}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!]
\end{aligned}$$

The reader should observe that, by convention, opcodes whose mnemonics begin with "**F**" always represent rational or floating point operations, while all remaining instruction forms interpret their arguments as integer values. A few MMIX mnemonics are also supplied in pairs of the form "***XXX***" and "***XXXU***", in which case the earlier variant performs the required operation under the integer format "Z.64", while the later assumes the default unsigned integer model of a modulo-2^{64} arithmetic.

On MMIX, the integer division instructions "**DIVU**" and "**DIV**" compute simultaneously both the quotient and the remainder from the corresponding arithmetic operation. The later is always stored in the remainder register **rR**, so that the formal meanings of these two instruction forms can be captured concisely by the following translations:

$$\begin{aligned}
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{DIVU}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \div_{\mathrm{N}.64}\ \$Z,\ \mathbf{rR} = \$Y\ \cdot|\cdot_{\mathrm{N}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathbf{DIV}\ X, Y, Z]\!] &= [\![\lambda\pi.\mathtt{LET}\ \$X = \$Y\ \div_{\mathrm{Z}.64}\ \$Z,\ \mathbf{rR} = \$Y\ \cdot|\cdot_{\mathrm{Z}.64}\ \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!]
\end{aligned}$$

Further, the "**FCMP**", "**CMPU**" and "**CMP**" instructions perform comparison of two numeric quantities, returning -1, 0 or 1 whenever their first operand is less than, equal to or greater than the second. Mathematically, this can be expressed as a subtraction operation of the form "$(\$Z < \$Y) - (\$Y < \$Z)$", assuming the standard encoding of

relational operations as the numeric constants 1 and 0 for true and false boolean values, respectively. In Etude, the resulting term structures are represented as follows:

$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{FCMP}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = (\$Z <_{\mathsf{R}.64}\ \$Y) -_{\mathsf{Z}.64} (\$Y <_{\mathsf{R}.64}\ \$Z);\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{CMPU}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = (\$Z <_{\mathsf{N}.64}\ \$Y) -_{\mathsf{Z}.64} (\$Y <_{\mathsf{N}.64}\ \$Z);\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{CMP}\ \ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = (\$Z <_{\mathsf{Z}.64}\ \$Y) -_{\mathsf{Z}.64} (\$Y <_{\mathsf{Z}.64}\ \$Z);\ [\![\nu \oplus 4]\!](\pi)]\!]$$

In order to facilitate more general comparison operations, the architecture also provides a set of six instructions with the mnemonic forms "$\mathtt{ZSN}_{\mathtt{I}}$", "$\mathtt{ZSZ}_{\mathtt{I}}$", "$\mathtt{ZSP}_{\mathtt{I}}$", "$\mathtt{ZSNN}_{\mathtt{I}}$", "$\mathtt{ZSNZ}_{\mathtt{I}}$" and "$\mathtt{ZSNP}_{\mathtt{I}}$". Intuitively, each of these instructions sets the register $\$X$ to the integer constant Z if $\$Y$ has, respectively, a negative, zero, positive, non-negative, non-zero or non-positive value and, in all other cases, continues evaluation with $\$X$ set to zero. Mathematically, this behaviour can be modelled by expressions of the form $(\$Y\ op\ 0) \times Z$, so that the meanings of these six instruction forms are captured in this work as follows:

$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{ZSN}_{\mathtt{I}}\ \ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = (\$Y <_{\mathsf{Z}.64} \#0_{\mathsf{N}.64}) \times_{\mathsf{N}.64} \#Z_{\mathsf{N}.64};\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{ZSZ}_{\mathtt{I}}\ \ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = (\$Y =_{\mathsf{Z}.64} \#0_{\mathsf{N}.64}) \times_{\mathsf{N}.64} \#Z_{\mathsf{N}.64};\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{ZSP}_{\mathtt{I}}\ \ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = (\$Y >_{\mathsf{Z}.64} \#0_{\mathsf{N}.64}) \times_{\mathsf{N}.64} \#Z_{\mathsf{N}.64};\ [\![\nu \oplus 4]\!](\pi)]\!]$$

$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{ZSNN}_{\mathtt{I}}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = (\$Y \geq_{\mathsf{Z}.64} \#0_{\mathsf{N}.64}) \times_{\mathsf{N}.64} \#Z_{\mathsf{N}.64};\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{ZSNZ}_{\mathtt{I}}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = (\$Y \neq_{\mathsf{Z}.64} \#0_{\mathsf{N}.64}) \times_{\mathsf{N}.64} \#Z_{\mathsf{N}.64};\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{ZSNP}_{\mathtt{I}}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = (\$Y \leq_{\mathsf{Z}.64} \#0_{\mathsf{N}.64}) \times_{\mathsf{N}.64} \#Z_{\mathsf{N}.64};\ [\![\nu \oplus 4]\!](\pi)]\!]$$

On MMIX, conversions between integers and floating point numbers are facilitated by a set of four opcode mnemonics "**FIX**", "**FIXU**", "**FLOT**" and "**FLOTU**". In all four of the associated instruction forms, the immediate operand Y is used to specify a rounding mode for the operation, but, in this chapter, only those variants of these instructions whose rounding modes correspond precisely to the behaviour expected of the corresponding C operation are formalised, so that their complete translation into Etude is specified by the following six definitions:

$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{FIXU}\ \ X,\ 1,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = \mathsf{N}.64_{\mathsf{R}.64}(\$Z);\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{FIX}\ \ \ X,\ 1,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = \mathsf{Z}.64_{\mathsf{R}.64}(\$Z);\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{FLOTU}\ X,\ 0,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = \mathsf{R}.64_{\mathsf{N}.64}(\$Z);\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{FLOT}\ \ X,\ 0,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = \mathsf{R}.64_{\mathsf{Z}.64}(\$Z);\ [\![\nu \oplus 4]\!](\pi)]\!]$$

Much of the opcode space in the MMIX instruction set is occupied by operations dedicated to manipulation of memory-resident data. In particular, instructions with opcodes of the form "$\mathtt{STXU}_{\mathtt{I}}$" update the memory location $\$Y +_{\mathsf{N}.64} \#Z_{\mathsf{N}.64}$ with the least-significant 8, 16, 32 or 64 bits of the specified register $\$X$:

$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{STBU}_{\mathtt{I}}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ () = \mathsf{SET}\ [\$Y +_{\mathsf{N}.64} \#Z_{\mathsf{N}.64}]_{\mathsf{N}.8}\ \ \mathsf{TO}\ \$X;\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{STWU}_{\mathtt{I}}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ () = \mathsf{SET}\ [\$Y +_{\mathsf{N}.64} \#Z_{\mathsf{N}.64}]_{\mathsf{N}.16}\ \mathsf{TO}\ \$X;\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{STTU}_{\mathtt{I}}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ () = \mathsf{SET}\ [\$Y +_{\mathsf{N}.64} \#Z_{\mathsf{N}.64}]_{\mathsf{N}.32}\ \mathsf{TO}\ \$X;\ [\![\nu \oplus 4]\!](\pi)]\!]$$
$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{STOU}_{\mathtt{I}}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ () = \mathsf{SET}\ [\$Y +_{\mathsf{N}.64} \#Z_{\mathsf{N}.64}]_{\mathsf{N}.64}\ \mathsf{TO}\ \$X;\ [\![\nu \oplus 4]\!](\pi)]\!]$$

Conversely, the eight "$\mathtt{LDXU}_{\mathtt{I}}$" and "$\mathtt{LDX}_{\mathtt{I}}$" operations retrieve a 1, 2, 4 or 8-byte object located at the address specified by the sum of the general purpose register $\$Y$ and the

immediate operand $\#Z_{\text{N.64}}$, interpreting the object's bit pattern as a signed or unsigned integer value that is written directly into the specified register $\$X$. Formally:

$$\begin{aligned}
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{LDBU_I}\ X,\ Y,\ Z]\!] &= [\![\lambda\pi.\textsf{LET}\ \$X = \textsf{GET}\ [\$Y +_{\text{N.64}} \#Z_{\text{N.64}}]_{\text{N.8}};\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{LDWU_I}\ X,\ Y,\ Z]\!] &= [\![\lambda\pi.\textsf{LET}\ \$X = \textsf{GET}\ [\$Y +_{\text{N.64}} \#Z_{\text{N.64}}]_{\text{N.16}};\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{LDTU_I}\ X,\ Y,\ Z]\!] &= [\![\lambda\pi.\textsf{LET}\ \$X = \textsf{GET}\ [\$Y +_{\text{N.64}} \#Z_{\text{N.64}}]_{\text{N.32}};\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{LDOU_I}\ X,\ Y,\ Z]\!] &= [\![\lambda\pi.\textsf{LET}\ \$X = \textsf{GET}\ [\$Y +_{\text{N.64}} \#Z_{\text{N.64}}]_{\text{N.64}};\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{LDB_I}\ X,\ Y,\ Z]\!] &= [\![\lambda\pi.\textsf{LET}\ \$X = \textsf{GET}\ [\$Y +_{\text{N.64}} \#Z_{\text{N.64}}]_{\text{Z.8}};\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{LDW_I}\ X,\ Y,\ Z]\!] &= [\![\lambda\pi.\textsf{LET}\ \$X = \textsf{GET}\ [\$Y +_{\text{N.64}} \#Z_{\text{N.64}}]_{\text{Z.16}};\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{LDT_I}\ X,\ Y,\ Z]\!] &= [\![\lambda\pi.\textsf{LET}\ \$X = \textsf{GET}\ [\$Y +_{\text{N.64}} \#Z_{\text{N.64}}]_{\text{Z.32}};\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{LDO_I}\ X,\ Y,\ Z]\!] &= [\![\lambda\pi.\textsf{LET}\ \$X = \textsf{GET}\ [\$Y +_{\text{N.64}} \#Z_{\text{N.64}}]_{\text{Z.64}};\ [\![\nu \oplus 4]\!](\pi)]\!]
\end{aligned}$$

The "**GET**" and "**PUT**" instructions provide access to the 32 special purpose registers of MMIX. In this chapter, however, we are only ever interested in the sixth such register **rR**, so that only the following two translation rules are required for a complete implementation of our verified C compiler:

$$\begin{aligned}
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{GET}\ X,\ 0,\ 6]\!] &= [\![\lambda\pi.\textsf{LET}\ \$X = \mathtt{rR};\ [\![\nu \oplus 4]\!](\pi)]\!] \\
\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{PUT}\ 6,\ 0,\ Z]\!] &= [\![\lambda\pi.\textsf{LET}\ \mathtt{rR} = \$Z;\ [\![\nu \oplus 4]\!](\pi)]\!]
\end{aligned}$$

The "$\mathtt{GO_I}$" instruction performs an unconditional jump to a memory location determined dynamically by the sum of its two operands $\$Y$ and $\#Z_{\text{N.64}}$. In all cases, the address of the following instruction $\nu \oplus 4$ is also stored in the specified general purpose register $\$X$, for use as the continuation term once the targeted MMIX procedure has completed its job:

$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{GO_I}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\textsf{LET}\ \$X = [\![\nu \oplus 4]\!];\ (\$Y +_{\text{N.64}} \#Z_{\text{N.64}})(\pi)]\!]$$

On the other hand, a related *branch instruction* with the mnemonic "**BZ**" selects one of two possible addresses for its continuation term, based on a value specified in the operand $\$X$. In particular, if that register contains the value of zero, then the instruction performs a jump to an address determined from the sum of its own location ν and the 16-bit offset $4(2^8Y + Z)$ as specified by its immediate operands Y and Z. Otherwise, execution will continue normally at the following address $\nu \oplus 4$. In Etude, this construction can be represented as follows:

$$\begin{aligned}
&\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{BZ}\ X,\ Y,\ Z]\!] \\
&\quad = [\![\lambda\pi.\textsf{IF}\ \$X =_{\text{N.64}} \#0_{\text{N.64}}\ \textsf{THEN}\ [\![\nu \oplus 4(2^8Y + Z)]\!](\pi)\ \textsf{ELSE}\ [\![\nu \oplus 4]\!](\pi)]\!]
\end{aligned}$$

The actual location of the following instruction can be also obtained using the MMIX operation "**GETA**", which stores the numeric value of its address directly in the specified general purpose register $\$X$:

$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \mathtt{GETA}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\textsf{LET}\ \$X = [\![\nu \oplus 4(2^8Y + Z)]\!];\ [\![\nu \oplus 4]\!](\pi)]\!]$$

The reader will observe that all of the MMIX instruction forms described so far have operated solely on variable operands and small non-negative constants representable as a single unsigned byte value. To facilitate an efficient synthesis of larger numeric quantities, MMIX provides a block of four instructions with the mnemonics of "**SETL**", "**INCML**", "**INCML**", "**INCMH**" and "**INCH**". All of these instructions accept a pair of

immediate operands Y and Z, interpreted as a single unsigned integer YZ with a value of the form $2^k(2^8Y + Z)$, where k is equal to 0, 16, 32 or 48 for the "**`L`**", "**`ML`**", "**`MH`**" and "**`H`**" variants of these instructions, respectively. In particular, the "**`SETL`**" operation sets **`$`**X to the value of this constant directly, while the three "**`INCXX`**" instruction forms update **`$`**X with the sum of its existing value and the constant $2^k(2^8Y + Z)$:

$$\begin{aligned}
&\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \texttt{SETL}\ \ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = \#[\![2^8Y + Z]\!]_{\mathsf{N.64}};\ [\![\nu \oplus 4]\!](\pi)]\!]\\
&\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \texttt{INCML}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = \$X +_{\mathsf{N.64}} \#[\![2^{16}(2^8Y + Z)]\!]_{\mathsf{N.64}};\ [\![\nu \oplus 4]\!](\pi)]\!]\\
&\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \texttt{INCMH}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = \$X +_{\mathsf{N.64}} \#[\![2^{32}(2^8Y + Z)]\!]_{\mathsf{N.64}};\ [\![\nu \oplus 4]\!](\pi)]\!]\\
&\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \texttt{INCH}\ \ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \$X = \$X +_{\mathsf{N.64}} \#[\![2^{48}(2^8Y + Z)]\!]_{\mathsf{N.64}};\ [\![\nu \oplus 4]\!](\pi)]\!]
\end{aligned}$$

The above four instruction forms also accept an optional fourth symbolic operand, permitting MMIX programs to perform explicit arithmetic operations on the concrete values of any symbolic variables introduced into its modules by a later linking stage of compilation. In particular, if one of these instructions is annotated with a given symbol σ, then the register **`$`**X is updated with an appropriate 16-bit subset of a value assigned by the linker to the MMIX variable σ**`.`**n, in which the offset n is equal to the constant $2^k(2^8Y + Z)$, determined by the instruction's immediate operands Y and Z in a manner appropriate for its opcode. Formally:

$$\begin{aligned}
&\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \texttt{SETL}\ \ X,\ Y,\ Z\ \texttt{@}\ \sigma]\!]\\
&\qquad = [\![\lambda\pi.\mathsf{LET}\ \$X = \sigma\texttt{.}[\![2^8Y + Z]\!] \mathbin{\triangle_{\mathsf{N.64}}} \#[\![2^{16} - 2^0]\!]_{\mathsf{N.64}};\ [\![\nu \oplus 4]\!](\pi)]\!]\\
&\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \texttt{INCML}\ X,\ Y,\ Z\ \texttt{@}\ \sigma]\!]\\
&\qquad = [\![\lambda\pi.\mathsf{LET}\ \$X = \$X +_{\mathsf{N.64}} (\sigma\texttt{.}[\![2^{16}(2^8Y + Z)]\!] \mathbin{\triangle_{\mathsf{N.64}}} \#[\![2^{32} - 2^{16}]\!]_{\mathsf{N.64}});\ [\![\nu \oplus 4]\!](\pi)]\!]\\
&\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \texttt{INCMH}\ X,\ Y,\ Z\ \texttt{@}\ \sigma]\!]\\
&\qquad = [\![\lambda\pi.\mathsf{LET}\ \$X = \$X +_{\mathsf{N.64}} (\sigma\texttt{.}[\![2^{32}(2^8Y + Z)]\!] \mathbin{\triangle_{\mathsf{N.64}}} \#[\![2^{48} - 2^{32}]\!]_{\mathsf{N.64}});\ [\![\nu \oplus 4]\!](\pi)]\!]\\
&\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \texttt{INCH}\ \ X,\ Y,\ Z\ \texttt{@}\ \sigma]\!]\\
&\qquad = [\![\lambda\pi.\mathsf{LET}\ \$X = \$X +_{\mathsf{N.64}} (\sigma\texttt{.}[\![2^{32}(2^8Y + Z)]\!] \mathbin{\triangle_{\mathsf{N.64}}} \#[\![2^{64} - 2^{48}]\!]_{\mathsf{N.64}});\ [\![\nu \oplus 4]\!](\pi)]\!]
\end{aligned}$$

Last but not least, an MMIX operation of the form "**`TRAP`** X**`,`** Y**`,`** Z" denotes an invocation of the operating system facility identified by a 24-bit integer encoded in the instruction's immediate operands X, Y and Z as follows:

$$\mathcal{D}_{\mathtt{I}}[\![\nu \triangleright \texttt{TRAP}\ X,\ Y,\ Z]\!] = [\![\lambda\pi.\mathsf{LET}\ \pi = [\![2^{16}X + 2^8Y + Z]\!](\pi);\ [\![\nu \oplus 4]\!](\pi)]\!]$$

The reader should observe that, on MMIX, all such system calls are always assumed to return a new value for every available MMIX register. In the remainder of this chapter, "**`TRAP`**" instructions will receive only a superficial treatment, since their satisfactory formalisation would inherently involve a detailed specification of the entire operating system leveraged by the program under semantic scrutiny.

None of the remaining 206 MMIX opcodes are required for the simple translation of Etude programs into their MMIX counterparts presented later in Section 6.4, although many of these omitted instruction forms could be easily utilised by a smarter version of our compiler in order to improve, sometimes dramatically, an operational efficiency of the resulting programs. For example, most of the above instruction forms are also accompanied by analogous operations on immediate operand values, alleviating the need for a continuous synthesis of many simple numeric constants using the

"**SETL**" instruction family. Nevertheless, in this work, I avoid a formal specification and utilisation of all such instructions, in order to focus the present discussion on the actual challenge of compiler verification.

6.3 Etude on MMIX

In order to utilise the above semantic translation as a realistic operational interpretation of MMIX programs, we must also refine the generic Etude language described in Chapter 4 for the specific requirements of the MMIX instruction set architecture. In particular, all implementation-specified language parameters, such as the precise ranges of numeric quantities representable under individual Etude formats, the behaviour of mathematically-undefined operations, or the precise implementation and structure of the program's address space, must be assigned appropriate concrete values, in order to align the earlier generic algebraic interpretation of Etude programs with the operational reality of the underlying computational hardware.

6.3.1 Atoms and Their Formats

First of all, every Etude implementation must describe its atomic representation of all numeric quantities that are supported by the language. As described in Section 4.4, this representation rests crucially on a language parameter known as the byte width ω, which, on every 64-bit architectures such as MMIX, is always assigned the value of 8:

$$\omega :: \mathit{integer}$$
$$\omega = 8$$

Further, in this chapter, the encoding component of every Etude format is represented simply by an integer value, as described by the following type equations:

$$\mathit{encoding}:$$
$$\quad \mathit{integer}$$

Intuitively, such encoding value specifies the number of bits required for an accurate depiction of all numeric quantities representable under a given well-formed Etude format. Accordingly, the encoding of every natural and integer format must always represent a sensible bit width in the range 1 ... 64 for the object. On the other hand, formats from all other genres must be assigned the standard encoding value of 64, except that three additional object formats "O.63", "O.62" and "O.61" may be used to describe memory locations which are guaranteed to be aligned on 2, 4 or 8 byte boundaries, respectively. Formally, the set of all valid formats on the MMIX architecture is characterised by the following well-formedness property:

$$\text{data } \mathrm{WF}\ [\![\cdot]\!] :: \mathit{format} \to \star$$
$$\begin{aligned}\text{where } \mathrm{WF}_{\mathsf{N}} &:: \forall\, \varepsilon \Rightarrow\ [\![1 \leq \varepsilon \leq 64]\!] \to \mathrm{WF}[\![\mathsf{N}.\varepsilon]\!]\\ \mathrm{WF}_{\mathsf{Z}} &:: \forall\, \varepsilon \Rightarrow\ [\![1 \leq \varepsilon \leq 64]\!] \to \mathrm{WF}[\![\mathsf{Z}.\varepsilon]\!]\\ \mathrm{WF}_{\mathsf{O}} &:: \forall\, \varepsilon \Rightarrow [\![61 \leq \varepsilon \leq 64]\!] \to \mathrm{WF}[\![\mathsf{O}.\varepsilon]\!]\\ \mathrm{WF}_{\mathsf{R}} &:: \mathrm{WF}[\![\mathsf{R}.64]\!]\\ \mathrm{WF}_{\mathsf{F}} &:: \mathrm{WF}[\![\mathsf{F}.64]\!]\end{aligned}$$

In particular, the standard encoding Φ is always assigned the value of 64:

$\Phi :: \textit{encoding}$
$\Phi = 64$

THEOREM 6-1: *(Validity of standard formats)* Every standard Etude format is well-formed under the above definition of WF and Φ:

$\text{WF}_\Phi :: \forall\ \gamma \Rightarrow \text{WF}(\gamma) \rightarrow \text{WF}[\![\gamma.\Phi]\!]$

PROOF: Trivial, by case inspection of the "WF:: *format* $\rightarrow \star$" predicate definition:

$\text{WF}_\Phi\ [\![P :: \text{WF}[\![\mathsf{N}]\!]]\!] = \text{WF}_\text{N}\ (\text{DEFN} :: \quad [\![1 \leq \Phi \leq 64]\!])$
$\text{WF}_\Phi\ [\![P :: \text{WF}[\![\mathsf{Z}]\!]]\!] = \text{WF}_\text{Z}\ (\text{DEFN} :: \quad [\![1 \leq \Phi \leq 64]\!])$
$\text{WF}_\Phi\ [\![P :: \text{WF}[\![\mathsf{O}]\!]]\!] = \text{WF}_\text{O}\ (\text{DEFN} :: [\![61 \leq \Phi \leq 64]\!])$
$\text{WF}_\Phi\ [\![P :: \text{WF}[\![\mathsf{R}]\!]]\!] = \text{WF}_\text{R}$
$\text{WF}_\Phi\ [\![P :: \text{WF}[\![\mathsf{F}]\!]]\!] = \text{WF}_\text{F}$ □

THEOREM 6-2: *(Validity of integral formats)* For every valid integral format "$\mathsf{Z}.\varepsilon$", its natural counterpart "$\mathsf{N}.\varepsilon$" is likewise well-formed:

$\text{WF}_\text{I} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\mathsf{Z}.\varepsilon]\!] \rightarrow \text{WF}[\![\mathsf{N}.\varepsilon]\!]$

PROOF: Once again, this theorem follows naturally from the earlier definition of the format well-formedness axioms WF_Z and WF_N:

$\text{WF}_\text{I}\ [\![\text{WF}_\text{Z}\ Q]\!] = \text{WF}_\text{N}\ Q$ □

For conciseness, in the remainder of this section most of such trivially justified theorems are presented as simple corollaries, leaving a precise formulation of the required proof terms as an exercise to a keen reader.

As described earlier in Section 4.6, every well-formed Etude format ϕ is associated with some predetermined object format $O(\phi)$ that, intuitively, can be used to describe locations of ϕ-formatted objects in the program's address space. On MMIX, these formats are always given the encoding of $64 - \lfloor \log_2(\mathcal{S}(\phi)) \rfloor$, in which $\mathcal{S}(\phi)$ depicts the byte size of the referenced objects, as described later in this section:

$O[\![\cdot]\!] :: \textit{format} \rightarrow \textit{format}$
$O[\![\phi]\!] = [\![\mathsf{O}.[\![64 - \lfloor \log_2(\mathcal{S}(\phi)) \rfloor]\!]]\!]$

Further, the *width* of every well-formed Etude format ϕ, depicted in Chapter 4 by the notation "$\mathcal{W}(\phi)$", is always equal to the format's encoding value ε:

$\mathcal{W}[\![\cdot]\!] :: \textit{format} \rightarrow \textit{integer}$
$\mathcal{W}[\![\phi]\!] = \varepsilon(\phi)$

On the other hand, the size of a valid format is equal to the least integer $k \in \{1, 2, 4, 8\}$, such that $8k \geq \mathcal{W}(\phi)$. Mathematically, this integer can be obtained by the following concise formula:

$\mathcal{S}[\![\cdot]\!] :: \textit{format} \rightarrow \textit{integer}$
$\mathcal{S}[\![\phi]\!] = 2^{\lfloor \log_2 \lfloor (\mathcal{W}(\phi) + 7)/8 \rfloor \rfloor}$

COROLLARY 6-3: *(Geometry of well-formed formats)* Every well-formed Etude format has a positive size and width, with the later no greater than the product of its size and the byte width parameter ω. Further, the size of every integer format "Z.ε" is equal to that of the corresponding natural format "N.ε". Formally:

$$\begin{aligned}
&\text{SIZE}_\phi &&:: \forall\ \phi \Rightarrow \text{WF}(\phi) &&\rightarrow [\![\mathcal{S}(\phi) > 0]\!]\\
&\text{WIDTH}_\phi &&:: \forall\ \phi \Rightarrow \text{WF}(\phi) &&\rightarrow [\![0 < \mathcal{W}(\phi) \leq \omega \times \mathcal{S}(\phi)]\!]\\
&\text{SIZE}_\text{Z} &&:: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{Z}.\varepsilon]\!] &&\rightarrow [\![\mathcal{S}[\![\text{Z}.\varepsilon]\!] = \mathcal{S}[\![\text{N}.\varepsilon]\!]]\!]
\end{aligned}$$

The MMIX architecture supports a full range of double precision floating point arithmetic defined in IEEE 754 Standard for Binary Floating Point Arithmetic [IEEE 754]. Accordingly, the four floating point parameters r, p, E_{min} and E_{max} assume the following values for the 64-bit rational format "R.64":

$$\text{r}[\![\cdot]\!],\ \text{p}[\![\cdot]\!],\ \text{E}_\text{min}[\![\cdot]\!],\ \text{E}_\text{max}[\![\cdot]\!] :: \textit{format} \rightarrow \textit{integer}$$

$$\begin{aligned}
\text{r}\quad &[\![\text{R}.64]\!] = \quad 2\\
\text{p}\quad &[\![\text{R}.64]\!] = \quad 53\\
\text{E}_\text{min}\ &[\![\text{R}.64]\!] = -1021\\
\text{E}_\text{max}\ &[\![\text{R}.64]\!] = \quad 1024
\end{aligned}$$

COROLLARY 6-4: *(Minimal floating point requirements)*

$$\begin{aligned}
&\text{RADIX} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{r}\,[\![\text{R}.\varepsilon]\!] > 1]\!]\\
&\text{PREC}\ \ :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{p}[\![\text{R}.\varepsilon]\!] > 0]\!]\\
&\text{EMIN}\ \ :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{E}_\text{min}\,[\![\text{R}.\varepsilon]\!] < 0]\!]\\
&\text{EMAX}\ :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\text{R}.\varepsilon]\!] \rightarrow [\![\text{E}_\text{max}[\![\text{R}.\varepsilon]\!] > 0]\!]
\end{aligned}$$

As described in Section 4.4, the greatest finite value representable under a given well-formed Etude format ϕ is known as that format's *least upper bound*, depicted by the notation "lub(ϕ)". For natural formats, this bound is always set to $2^{\mathcal{W}(\phi)} - 1$. On the other hand, the $2^{\mathcal{W}(\phi)}$ possible bit patterns found in integer and pointer values are always interpreted under the *two's complement encoding*, so that the greatest representable number has the value of $2^{\mathcal{W}(\phi)\,-\,1} - 1$. Finally, in accordance with the IEEE Standard 754, the least upper bound of every rational format is equal to $(1 - \text{r}(\phi)^{-\text{p}(\phi)}) \times \text{r}(\phi)^{\text{E}_\text{max}(\phi)}$. Formally, these three rules are captured by the following Haskell definition:

$$\text{lub}[\![\cdot]\!] :: \textit{format} \rightarrow \textit{rational}$$

$$\begin{aligned}
\text{lub}[\![\text{N}.\varepsilon]\!] &= 2^{\mathcal{W}[\![\text{N}.\varepsilon]\!]} - 1\\
\text{lub}[\![\text{Z}.\varepsilon]\!] &= 2^{\mathcal{W}[\![\text{Z}.\varepsilon]\!]\,-\,1} - 1\\
\text{lub}[\![\text{R}.\varepsilon]\!] &= (1 - \text{r}[\![\text{R}.\varepsilon]\!]^{-\text{p}[\![\text{R}.\varepsilon]\!]}) \times \text{r}[\![\text{R}.\varepsilon]\!]^{\text{E}_\text{max}[\![\text{R}.\varepsilon]\!]}\\
\text{lub}[\![\text{F}.\varepsilon]\!] &= 2^{63} - 1\\
\text{lub}[\![\text{O}.\varepsilon]\!] &= 2^{63} - 1
\end{aligned}$$

Further, the format's *greatest lower bound*, or the least finite quantity representable under it, is always set to 0 for natural formats. For signed integer quantities and pointer address values, it is always set to the least value $-2^{\mathcal{W}(\phi)\,-\,1}$ representable under the corresponding two's complement encoding within $\mathcal{W}(\phi)$ bits of information,

while, for all well-formed rational formats, it is equal precisely to the negation of the corresponding least upper bound. In Haskell:

$$\text{glb}[\![\cdot]\!] :: \textit{format} \rightarrow \textit{rational}$$

$$\begin{aligned}
\text{glb}[\![\mathsf{N}.\varepsilon]\!] &= 0 \\
\text{glb}[\![\mathsf{Z}.\varepsilon]\!] &= -2^{\mathcal{W}[\![\mathsf{Z}.\varepsilon]\!]\,-\,1} \\
\text{glb}[\![\mathsf{R}.\varepsilon]\!] &= -\text{lub}[\![\mathsf{R}.\varepsilon]\!] \\
\text{glb}[\![\mathsf{F}.\varepsilon]\!] &= -2^{63} \\
\text{glb}[\![\mathsf{O}.\varepsilon]\!] &= -2^{63}
\end{aligned}$$

COROLLARY 6-5: *(Minimal format bounds requirements)* The bounds of all well-formed arithmetic formats satisfy the following minimal requirements:

$$\begin{aligned}
&\text{GLB}_\phi :: \forall\ \phi \Rightarrow \text{WF}(\phi) \rightarrow [\![\text{glb}(\phi) \leq 0]\!] \\
&\text{GLB}_\text{N} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\mathsf{N}.\varepsilon]\!] \rightarrow [\![\text{glb}[\![\mathsf{N}.\varepsilon]\!] = 0]\!] \\
&\text{GLB}_\text{Z} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\mathsf{Z}.\varepsilon]\!] \rightarrow [\![\text{glb}[\![\mathsf{Z}.\varepsilon]\!] \leq -\text{lub}[\![\mathsf{Z}.\varepsilon]\!]]\!] \\
&\text{GLB}_\text{R} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\mathsf{R}.\varepsilon]\!] \rightarrow [\![\text{glb}[\![\mathsf{R}.\varepsilon]\!] = -\text{r}[\![\mathsf{R}.\varepsilon]\!]^{\text{p}[\![\mathsf{R}.\varepsilon]\!]} - 1 \times \text{r}[\![\mathsf{R}.\varepsilon]\!]^{\text{E}_{\max}[\![\mathsf{R}.\varepsilon]\!]\,-\,\text{p}[\![\mathsf{R}.\varepsilon]\!]}]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{LUB}_\phi :: \forall\ \phi \Rightarrow \text{WF}(\phi) \rightarrow [\![\text{lub}(\phi) > 0]\!] \\
&\text{LUB}_\text{N} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\mathsf{N}.\varepsilon]\!] \rightarrow [\![\text{lub}[\![\mathsf{N}.\varepsilon]\!] = 2^{\mathcal{W}[\![\mathsf{N}.\varepsilon]\!]} - 1]\!] \\
&\text{LUB}_\text{Z} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\mathsf{Z}.\varepsilon]\!] \rightarrow [\![\text{lub}[\![\mathsf{Z}.\varepsilon]\!] < \text{lub}[\![\mathsf{N}.\varepsilon]\!]]\!] \\
&\text{LUB}_\text{R} :: \forall\ \varepsilon \Rightarrow \text{WF}[\![\mathsf{R}.\varepsilon]\!] \rightarrow [\![\text{lub}[\![\mathsf{R}.\varepsilon]\!] = -\text{glb}[\![\mathsf{R}.\varepsilon]\!]]\!]
\end{aligned}$$

Using the above definitions, we can now capture the precise well-formedness property applicable to all Etude atoms under the MMIX architecture. Perhaps surprisingly, this can be accomplished by only a pair of simple axioms WF_NC and $\text{WF}_\equiv$, The relevant Haskell property definition assumes the following form:

$$\begin{aligned}
&\text{data WF}\ [\![\cdot]\!] :: (\text{ord}\ \nu) \Rightarrow \textit{atom}_\nu \rightarrow \star \\
&\quad \text{where}\ \text{WF}_\text{NC}\ [\![\cdot]\!] \quad :: \forall\ x :: \textit{integer} \Rightarrow [\![0 \leq x < 2^{64}]\!] \rightarrow \text{WF}[\![\#x_{\text{N}.64}]\!] \\
&\qquad\quad \text{WF}_\equiv\ [\![\cdot]\!]\ [\![\cdot]\!] :: \forall\ \alpha_1, \alpha_2 \Rightarrow \text{WF}(\alpha_1) \rightarrow (\alpha_1 \equiv \alpha_2) \rightarrow \text{WF}(\alpha_2)
\end{aligned}$$

In other words, a constant natural atom of the form "$\#x_\phi$" is always guaranteed to be well-formed if x depicts a non-negative integer representable in 64 bits of pure binary representation, i.e., one with a value less than 2^{64}. Otherwise, $\text{WF}_\equiv$ guarantees the validity of all elements found in a given equivalence class of Etude atoms, whenever at least one of these elements can be shown to be well-formed. Formally, the equivalence relation "$\alpha_1 \equiv \alpha_2$" is defined as a definitional equality of the two atoms' respective normal forms, obtained from their reduction by the evaluation function $\mathcal{E}$. In particular:

$$\begin{aligned}
&\text{data}\ [\![\cdot]\!] \equiv [\![\cdot]\!] :: (\text{ord}\ \nu) \Rightarrow \textit{atom}_\nu \rightarrow \textit{atom}_\nu \rightarrow \star \\
&\quad \text{where}\ \text{EQV}_\alpha\ [\![\cdot]\!] :: \forall\ \alpha_1, \alpha_2 \Rightarrow [\![\mathcal{E}(\alpha_1) = \mathcal{E}(\alpha_2)]\!] \rightarrow (\alpha_1 \equiv \alpha_2)
\end{aligned}$$

THEOREM 6-6: *(Atom equivalence)* The "$\equiv$" property constitutes a reflexive, symmetric and transitive relation over Etude atoms:

$$\begin{aligned}
&\text{REFL}_\alpha :: \forall\ \alpha \rightarrow (\alpha \equiv \alpha) \\
&\text{SYMM}_\alpha :: \forall\ \alpha_1, \alpha_2 \Rightarrow (\alpha_1 \equiv \alpha_2) \rightarrow (\alpha_2 \equiv \alpha_1) \\
&\text{TRANS}_\alpha :: \forall\ \alpha_1, \alpha_2, \alpha_3 \Rightarrow (\alpha_1 \equiv \alpha_2) \rightarrow (\alpha_2 \equiv \alpha_3) \rightarrow (\alpha_1 \equiv \alpha_3)
\end{aligned}$$

PROOF: All three of the above theorems are justified directly by the corresponding properties of Haskell's definitional equality "=":

$$\begin{aligned}
&\text{REFL}_\alpha \quad [\![\alpha]\!] &&= \text{EQV}_\alpha\ (\text{REFL}\ [\![\mathcal{E}(\alpha)]\!]) \\
&\text{SYMM}_\alpha\ [\![\text{EQV}_\alpha\ P]\!] &&= \text{EQV}_\alpha\ (\text{SYMM}\ P) \\
&\text{TRANS}_\alpha\ [\![\text{EQV}_\alpha\ P]\!]\ [\![\text{EQV}_\alpha\ Q]\!] &&= \text{EQV}_\alpha\ (\text{TRANS}\ P\ Q) \quad \square
\end{aligned}$$

THEOREM 6-7: *(Equivalence of identical atoms)* All well formed pairs of structurally identical atoms are equivalent:

$$\text{EQV}_{\mathcal{E}} :: \forall\ \alpha_1,\ \alpha_2 \Rightarrow \text{WF}(\alpha_1) \rightarrow \text{WF}(\alpha_2) \rightarrow [\![\mathcal{E}(\alpha_1) = \mathcal{E}(\alpha_2)]\!] \rightarrow (\alpha_1 \equiv \alpha_2)$$

PROOF: By definition of the automatically derived structural identity relation "=", we have $[\![\alpha_1' = \alpha_2']\!] \rightarrow [\![\alpha_1' = \alpha_2']\!]$ for all atomic expressions α_1' and α_2'. Accordingly, we can establish the required equivalence by a simple application of that standard result to the assumed equality proof of $\mathcal{E}(\alpha_1) = \mathcal{E}(\alpha_2)$ in the following manner:

$$\begin{aligned}
&\text{EQV}_{\mathcal{E}}\ [\![P :: \text{WF}(\alpha_1)]\!]\ [\![Q :: \text{WF}(\alpha_2)]\!]\ [\![R :: [\![\mathcal{E}(\alpha_1) = \mathcal{E}(\alpha_2)]\!]]\!] = \text{EQV}_\alpha\ (\text{L}\ R) \\
&\quad \text{where}\ \text{L}_1 :: \forall\ \alpha_1',\ \alpha_2' \Rightarrow [\![\alpha_1' = \alpha_2']\!] \rightarrow [\![\alpha_1' = \alpha_2']\!] = \text{TRIV} \quad \square
\end{aligned}$$

In this work, $\mathcal{E}$ represents the operational semantics of all closed and well-formed Etude atoms under the MMIX architecture. By association, therefore, it also defines the precise operational behaviours of any C and MMIX programs whose meanings have been described through a translation into the present Etude variant.

A reader should observe that, according to the following definition of $\mathcal{E}$, an operational semantics of Etude atoms represents a partial Haskell function, undefined for open and otherwise meaningless syntactic entities. Accordingly, by theorem EQV_α, all such illegal constructs are grouped into a single equivalence class, since, in Haskell, the definitional equality always holds between a pair of undefined values "$\perp$".

When defined, however, the value produced by $\mathcal{E}$ will always represent some well-formed constant atom "$\#x_{\text{N.64}}$", i.e., one in which the rational number x depicts a non-negative integer quantity less than 2^{64} in magnitude. Intuitively, this implies that all MMIX programs are geared solely towards manipulation of 64-bit unsigned integer values, whose binary representation may, however be interpreted variously as a signed integer z in the range $-2^{63} \leq z < 2^{63}$, or even a rational number, with the later interpretation relying on the somewhat esoteric encoding of the IEEE 754 double precision values. Accordingly, on MMIX, every well-formed Etude atom "$\#x_\phi$" is always equivalent to some other 64-bit natural number with a binary representation identical to that of x under the rules of the format ϕ.

Formally, the binary encoding of a given numeric quantity x under a well-formed Etude format ϕ is depicted by the notation "$\text{enc}_\phi(x)$". In particular, if ϕ represents a valid natural or integer format, then the encoding of x is always equal to its integral portion $\lfloor x \rfloor$ after its reduction modulo $2^{\mathcal{W}(\phi)}$, except that, if ϕ belongs to the integer genre "Z" and $\lfloor x \rfloor \bmod 2^{\mathcal{W}(\phi)} \geq 2^{\mathcal{W}(\phi)-1}$, then $2^{64} - 2^{\mathcal{W}(\phi)}$ is added to its value, effectively copying the integer's sign bit into all otherwise unoccupied bit positions in the resulting

64-bit natural number. On the other hand, the encoding of x under a format from the object or function genre represents the value of $\lceil x \rceil \bmod 2^{64} - \lceil x \rceil \bmod 2^k$, where k is equal to the number of *alignment bits* specified by the format, defined as $64 - \varepsilon(\phi)$ and 2 for object and functional formats, respectively:

$$\mathrm{enc}_{[\![\cdot]\!]}[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \mathit{format} \to \mathit{rational} \to \mathit{atom}_\nu$$

$$\begin{array}{lll}
\mathrm{enc}_{[\![\mathsf{N}.\varepsilon]\!]}\ [\![x]\!] & \mid\ 1 \le \varepsilon \le 64 & = [\![\#[\![\lceil x \rceil \bmod 2^{\varepsilon}]\!]_{\mathsf{N}.64}]\!] \\
\mathrm{enc}_{[\![\mathsf{Z}.\varepsilon]\!]}\ [\![x]\!] & \mid\ 1 \le \varepsilon \le 64 & = [\![\#[\![\lceil x \rceil \bmod 2^{\varepsilon} + (2^{64} - 2^{\varepsilon}) \times [x][\varepsilon - 1]]\!]_{\mathsf{N}.64}]\!] \\
\mathrm{enc}_{[\![\mathsf{O}.\varepsilon]\!]}\ [\![x]\!] & \mid\ 61 \le \varepsilon \le 64 & = [\![\#[\![\lceil x \rceil \bmod 2^{64} - \lceil x \rceil \bmod 2^{64-\varepsilon}]\!]_{\mathsf{N}.64}]\!] \\
\mathrm{enc}_{[\![\mathsf{F}.64]\!]}[\![x]\!] & & = [\![\#[\![\lceil x \rceil \bmod 2^{64} - \lceil x \rceil \bmod 2^{2}]\!]_{\mathsf{N}.64}]\!]
\end{array}$$

The corresponding binary encoding of rational constants "$\#x_{\mathsf{R}.64}$" is somewhat more elaborate. In general, a precise representation of such constants relies on two additional parameters p and b, whose values are one less than those of the corresponding floating point parameters $\mathrm{p}[\![\mathsf{R}.64]\!]$ and $\mathrm{E}_{\max}[\![\mathsf{R}.64]\!]$, respectively:

$$p, b :: \mathit{integer}$$

$$\begin{array}{l}
p = \mathrm{p}[\![\mathsf{R}.64]\!] - 1 \\
b = \mathrm{E}_{\max}[\![\mathsf{R}.64]\!] - 1
\end{array}$$

The b parameter is often referred to as the *bias* of a floating point format. Using these parameters, every well-formed rational constant can be encoded as a triple of three integers s, e and m known as the *sign*, *exponent* and *mantissa*, respectively, such that $0 \le s \le 1$, $0 \le e \le 2b + 1$ and $0 \le m < 2^p$. If $x = 0$, then all three components are set to 0, so that "$\#0_{\mathsf{R}.64}$" is always encoded directly as "$\#0_{\mathsf{N}.64}$". Otherwise, the 64-bit encoding n of the rational number x is equal to the sum $2^{63}s + 2^p e + m$, so that the sign s is stored in the most-significant bit of n, the mantissa m is placed in the least-significant p bits of the resulting number and the exponent is slipped into the remaining 11 bits between s and m. In all cases, the sign bit s is assigned the value of 1 if and only if $x < 0$ and remains clear for all non-negative rational numbers. Further, the exponent e is generally equal to $\lfloor \log_2 |x| \rfloor + b$, clipped into the required range $0 \le e \le 2b + 1$. If $-b < \lfloor \log_2 |x| \rfloor \le b$, then x is said to represent a *normalised floating point number* and the mantissa m is set to $\mathrm{round}(2^{b+p-e}|x|) - 2^p$. Otherwise, if $\lfloor \log_2 |x| \rfloor \le -b$, then x is said to be *denormalised*, resulting in the mantissa of $\mathrm{round}(2^{b+p-1}|x|)$. Finally, all values of x for which $\lfloor \log_2 |x| \rfloor > b$ are semantically equivalent to $\pm\infty$ and are encoded with $m = 0$. Formally:

$$\begin{array}{lll}
\mathrm{enc}_{[\![\mathsf{R}.64]\!]}[\![x]\!] & \mid\ x = 0 & = [\![\#[\![0]\!]_{\mathsf{N}.64}]\!] \\
& \mid\ e \le 0 & = [\![\#[\![2^{63}s + \mathrm{round}(2^{b+p-1}|x|)]\!]_{\mathsf{N}.64}]\!] \\
& \mid\ e \le 2b & = [\![\#[\![2^{63}s + 2^p e + \mathrm{round}(2^{b+p-e}|x|) - 2^p]\!]_{\mathsf{N}.64}]\!] \\
& \mid\ e > 2b & = [\![\#[\![2^{63}s + 2^p(2b+1)]\!]_{\mathsf{N}.64}]\!]
\end{array}$$

$$\begin{array}{l}
\text{where } s = x < 0 \\
\qquad\ e = \lfloor \log_2 |x| \rfloor + b
\end{array}$$

In the following definition of the operational semantics $\mathcal{E}$, it is often necessary to reverse the above process, in order to obtain a concrete numeric value of an atom with

a given binary encoding. In particular, the notation "$\mathrm{dec}_\phi [\![\#n_{\mathrm{N}.64}]\!]$" depicts an integer or rational value reconstructed from its binary encoding n in accordance with the specified format ϕ. In Haskell, it is implemented as follows:

$$\mathrm{dec}_{[\![\cdot]\!]}[\![\cdot]\!] :: (\mathrm{ord}\ v) \Rightarrow \mathit{format} \to \mathit{atom}_v \to \mathit{rational}$$

$$\begin{array}{lll}
\mathrm{dec}_{[\![\mathrm{N}.\varepsilon]\!]}\ [\![\#n_{\mathrm{N}.64}]\!] & \mid\ 1 \leq \varepsilon \leq 64 & = n \\
\mathrm{dec}_{[\![\mathrm{Z}.\varepsilon]\!]}\ [\![\#n_{\mathrm{N}.64}]\!] & \mid\ 1 \leq \varepsilon \leq 64 & = n - 2^{64}(n \geq 2^{63}) \\
\mathrm{dec}_{[\![\mathrm{O}.\varepsilon]\!]}\ [\![\#n_{\mathrm{N}.64}]\!] & \mid\ 61 \leq \varepsilon \leq 64 & = n - 2^{64}(n \geq 2^{63}) \\
\mathrm{dec}_{[\![\mathrm{F}.64]\!]}\ [\![\#n_{\mathrm{N}.64}]\!] & & = n - 2^{64}(n \geq 2^{63}) \\
\mathrm{dec}_{[\![\mathrm{R}.64]\!]}\ [\![\#n_{\mathrm{N}.64}]\!] & \mid\ e = 0 & = (-1)^s \times m/2^{b\,+\,p\,-\,1} \\
& \mid\ 1 \leq e \leq 2b & = (-1)^s \times (m + 2^p)/2^{b\,+\,p\,-\,e}
\end{array}$$

$$\begin{array}{lll}
\text{where } s & = n[63] \\
e & = n[p \mathinner{..} 62] \\
m & = n[0 \mathinner{..} p-1]
\end{array}$$

in which the notations "$n[i \mathinner{..} j]$" represent the value of bits $i \mathinner{..} j$ in the binary representation of the integer n, as obtained by the corresponding Haskell construction defined in Appendix A. The reader should observe that the result of $\mathrm{dec}_\phi [\![\#n_{\mathrm{N}.64}]\!]$ is meaningless if the integer n falls outside the required range $0 \leq n < 2^{64}$, or if it doesn't represent a properly sign-extended quantity as appropriate for the underlying format ϕ. However, since, by construction, such applications of "dec" can never occur in the following specification of the operational semantics $\mathcal{E}$, is seems wasteful to enforce these restrictions directly in the above Haskell definition.

In addition to the finite floating point numbers whose values are interpreted by the above function, the IEEE 754 arithmetic model allows for two distinct representations of zero, written as $+0$ and -0 respectively, as well as two infinities $\pm\infty$ and, further, two distinguished non-numeric quantities depicted by the notations "$\pm$NaN". Whenever one of these values is applied to an Etude operator under the format "R.64", the result is determined by the rules of the IEEE Standard rather than the conventional mathematical wisdom. In hexadecimal notation, the binary encodings of these six specialised quantities are specified as follows:

$$+0,\ -0,\ +\infty,\ -\infty,\ +\mathrm{NaN},\ -\mathrm{NaN} :: (\mathrm{ord}\ v) \Rightarrow \mathit{atom}_v$$

$$\begin{array}{ll}
+0 & = [\![\#[\![\,0000000000000000_{16}]\!]_{\mathrm{N}.64}]\!] \\
-0 & = [\![\#[\![\,8000000000000000_{16}]\!]_{\mathrm{N}.64}]\!] \\
+\infty & = [\![\#[\![7\mathrm{FF}0000000000000_{16}]\!]_{\mathrm{N}.64}]\!] \\
-\infty & = [\![\#[\![\mathrm{FFF}0000000000000_{16}]\!]_{\mathrm{N}.64}]\!] \\
+\mathrm{NaN} & = [\![\#[\![7\mathrm{FF}8000000000000_{16}]\!]_{\mathrm{N}.64}]\!] \\
-\mathrm{NaN} & = [\![\#[\![\mathrm{FFF}8000000000000_{16}]\!]_{\mathrm{N}.64}]\!]
\end{array}$$

For convenience, we also define a pair of additional notations "$\{\pm 0\}$" and "$\{\pm\infty\}$" to stand for the following trivial subsets of these distinguished floating point values:

$$\{\pm 0\},\ \{\pm\infty\} :: (\mathrm{ord}\ v) \Rightarrow \{\mathit{atom}_v\}$$

$$\begin{array}{ll}
\{\pm 0\} & = \{+0,\ -0\} \\
\{\pm\infty\} & = \{+\infty,\ -\infty\}
\end{array}$$

In general, a given floating point encoding n represents a finite rational number only if its biased exponent $n[p \mathinner{..} 62]$ has a value less than or equal to $2b$. Otherwise, n encodes either an infinity of an appropriate sign, or, if the mantissa $n[0 \mathinner{..} p-1]$ has a non-zero value, a special non-numeric quantity known as a *NaN*. The two floating point values depicted by the earlier notations "$\pm$NaN" constitute merely a representative example of a NaN that is synthesised internally by the processor as a result of certain undefined mathematical operation. Formally, the complete sets of all finite and NaN floating point quantities can be identified by the following pair of Haskell predicates "FIN" and "NaN":

$$\mathrm{FIN}[\![\cdot]\!],\ \mathrm{NaN}[\![\cdot]\!] :: (\mathrm{ord}\ v) \Rightarrow \mathit{atom}_v \rightarrow \mathit{bool}$$

$$\begin{aligned}\mathrm{FIN}\ [\![\#n_{\mathrm{N.64}}]\!] &= (0 \leq n[p \mathinner{..} 62] \leq 2b)\\ \mathrm{NaN}[\![\#n_{\mathrm{N.64}}]\!] &= (n[p \mathinner{..} 62] = 2b+1 \wedge n[0 \mathinner{..} p-1] \neq 0)\end{aligned}$$

A reader interested in the actual motivation behind the above definitions is referred to the *IEEE 754 Standard for Binary Floating Point Arithmetic* [IEEE 754], which covers all of their underlying intricacies with much more details than permitted by the restricted scope of the present work. It should, however, be observed that all of the above definitions presuppose a particular configuration of the MMIX processor, in which all floating point arithmetic is always rounded correctly to the nearest even value as per the standard Haskell "round" function. For simplicity, I assume that the system has been configured in this way prior to commencement of every program's execution and that no well-formed MMIX instruction sequence will ever attempt to deviate from that initial configuration.

Armed with these auxiliary definitions, we are now ready to proceed with a precise formulation of the operational semantics $\mathcal{E}$ assigned to all well-formed Etude atoms. Predictably, $\mathcal{E}$ is formulated as a mapping of such atoms to their normalised representations, which, on MMIX, always depict appropriate constructs of the form "$\#x_{\mathrm{N.64}}$". Accordingly, the Haskell type signature of $\mathcal{E}$ can be specified as follows:

$$\mathcal{E}[\![\cdot]\!] :: (\mathrm{ord}\ v) \Rightarrow \mathit{atom}_v \rightarrow \mathit{atom}_v$$

In particular, every constant atom of the form "$\#x_\phi$" is always reducible directly to its encoded value "$\mathrm{enc}_\phi(x)$", provided that this encoding can be subsequently reversed without any loss of information. On the other hand, all of the remaining closed atomic forms "$op_\phi(\alpha)$" and "$\alpha_1\ op_\phi\ \alpha_2$" are evaluated systematically by obtaining the respective normal forms of the operands α, α_1 and α_2 and applying the resulting partially reduced atom to another function $\mathcal{E}'$, which performs further normalisation in accordance with the arithmetic semantics of the operator *op*. Formally:

$$\begin{array}{lll}\mathcal{E}[\![\#x_\phi]\!] & \mid x = \mathrm{dec}_\phi(\mathrm{enc}_\phi(x)) & = \mathrm{enc}_\phi(x)\\ \mathcal{E}[\![op_\phi(\alpha)]\!] & \mid \mathcal{E}(\alpha) = \mathcal{E}(\alpha) & = \mathcal{E}'[\![op_\phi[\![\mathcal{E}(\alpha)]\!]]\!]\\ \mathcal{E}[\![\alpha_1\ op_\phi\ \alpha_2]\!] & \mid \mathcal{E}(\alpha_1) = \mathcal{E}(\alpha_1) \wedge \mathcal{E}(\alpha_2) = \mathcal{E}(\alpha_2) & = \mathcal{E}'[\![[\![\mathcal{E}(\alpha_1)]\!]\ op_\phi\ [\![\mathcal{E}(\alpha_2)]\!]]\!]\end{array}$$

in which the cunning comparisons $\mathcal{E}(\alpha) = \mathcal{E}(\alpha)$ ensure that $\mathcal{E}$ remains undefined for all diverging and otherwise malformed operand values.

THEOREM 6-8: *(Atom compatibility)* Two atoms with pairwise-equivalent components are themselves equivalent to each other:

$$\text{EQV}_{\alpha 1} :: \forall\ \alpha_1,\ \alpha_2,\ \phi,\ op \Rightarrow (\alpha_1 \equiv \alpha_2) \rightarrow [\![op_\phi(\alpha_1)]\!] \equiv [\![op_\phi(\alpha_2)]\!]$$
$$\text{EQV}_{\alpha 2} :: \forall\ \alpha_{11},\ \alpha_{21},\ \alpha_{12},\ \alpha_{22},\ \phi,\ op \Rightarrow$$
$$(\alpha_{11} \equiv \alpha_{12}) \rightarrow (\alpha_{21} \equiv \alpha_{22}) \rightarrow [\![\alpha_{11}\ op_\phi\ \alpha_{21}]\!] \equiv [\![\alpha_{12}\ op_\phi\ \alpha_{22}]\!]$$

PROOF:

$$\text{EQV}_{\alpha 1}\ [\![\text{EQV}_\alpha\ (P :: [\![\mathcal{E}(\alpha_1) = \mathcal{E}(\alpha_2)]\!])]\!] = \text{EQV}_\alpha\ \text{L}_4$$

$$\begin{aligned}
\text{where } \text{L}_1 &:: [\![\mathcal{E}[\![op_\phi(\alpha_1)]\!] = \mathcal{E}'[\![op_\phi[\![\mathcal{E}(\alpha_1)]\!]]\!]]\!] && = \text{DEFN}\\
\text{L}_2 &:: [\![\mathcal{E}'[\![op_\phi[\![\mathcal{E}(\alpha_1)]\!]]\!] = \mathcal{E}'[\![op_\phi[\![\mathcal{E}(\alpha_2)]\!]]\!]]\!] && = \text{SUBST } P \text{ IN } \alpha \rightarrow \mathcal{E}'[\![op_\phi(\alpha)]\!]\\
\text{L}_3 &:: [\![\mathcal{E}[\![op_\phi(\alpha_2)]\!] = \mathcal{E}'[\![op_\phi[\![\mathcal{E}(\alpha_2)]\!]]\!]]\!] && = \text{DEFN}\\
\text{L}_4 &:: [\![\mathcal{E}[\![op_\phi(\alpha_1)]\!] = \mathcal{E}[\![op_\phi(\alpha_2)]\!]]\!] && = \text{TRANS L}_1\ (\text{TRANS L}_2\ (\text{SYMM L}_3))
\end{aligned}$$

The analogous proof for binary atomic forms follows a similar structure. □

The actual evaluation function $\mathcal{E}'$ can be defined for partially normalised unary and binary atomic forms as follows:

$$\mathcal{E}'[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow atom_\nu \rightarrow atom_\nu$$

First of all, in reality, an MMIX processor does not distinguish its functional and object formats from the two genres "F" and "O" as autonomous representations of data. Instead, every atomic operation performed under one of these formats is always treated exactly as if it was applied under the standard integer format "Z.64":

$$\begin{aligned}
&\mathcal{E}'[\![op_\phi(\alpha)]\!] \quad \mid \phi \in \{\text{F.64, O.61, O.62, O.63, O.64}\} = \mathcal{E}'[\![op_{\text{Z.64}}(\alpha)]\!]\\
&\mathcal{E}'[\![\alpha_1\ op_\phi\ \alpha_2]\!] \mid \phi \in \{\text{F.64, O.61, O.62, O.63, O.64}\} = \mathcal{E}'[\![\alpha_1\ op_{\text{Z.64}}\ \alpha_2]\!]
\end{aligned}$$

Next, it is convenient to dispense with a number of special cases in the definition of $\mathcal{E}'$ that pertain to all mathematically undefined applications of Etude's arithmetic operators. As already mentioned in Section 6.1, in the interest of conciseness I assume that our MMIX processor is permanently configured in such a way that all arithmetic traps for both integer and floating point operations remain disabled throughout the entire execution of a program. Under these conditions, if one of the two operands to the "$+_\phi$", "$-_\phi$", "$\times_\phi$" or "$\div_\phi$" operator applied under the rational format "R.64" represents a NaN, then the entire construction is reducible to that NaN, except that the bit $p-1$ is always set in the result. If both operands represent non-numeric quantities, then the value of the second operand is chosen in preference to the first. On the other hand, if a NaN is supplied to a relational operator "$=_\phi$", "$\neq_\phi$", "$<_\phi$", "$>_\phi$", "$\leq_\phi$" or "$\geq_\phi$", then the result is always 0, so that no NaN is ever deemed equal to itself or any other well-formed floating point quantity. Formally:

$$\begin{aligned}
&\mathcal{E}'[\![\#x_{\text{N.64}}\ op_{\text{R.64}}\ \#y_{\text{N.64}}]\!]\\
&\quad \mid op \in \{+, -, \times, \div\} \wedge \text{NaN}[\![\#y_{\text{N.64}}]\!] = [\![\#[\![y - 2^{p-1}y[p-1] + 2^{p-1}]\!]_{\text{N.64}}]\!]\\
&\quad \mid op \in \{+, -, \times, \div\} \wedge \text{NaN}[\![\#x_{\text{N.64}}]\!] = [\![\#[\![x - 2^{p-1}x[p-1] + 2^{p-1}]\!]_{\text{N.64}}]\!]\\
&\quad \mid op \in \{=, \neq, <, >, \leq, \geq\} \wedge (\text{NaN}[\![\#x_{\text{N.64}}]\!] \vee \text{NaN}[\![\#y_{\text{N.64}}]\!]) = [\![\#0_{\text{N.64}}]\!]
\end{aligned}$$

Further, if one or both of the two operands α_1 and α_2 involved in a floating point addition operation represents $\pm\infty$, then the entire atom is reducible to an infinity of

the same sign, except that the sum of two infinite floating point values with opposite signs is, by definition, deemed equal to ±NaN with the sign of α_2. Further, the IEEE Standard proclaims $-0+-0$ to have the value of -0 rather than the expected $+0$. These three special cases are captured by $\mathcal{E}'$ as follows:

$$\begin{array}{l|ll} \mathcal{E}'[\![\alpha_1 +_{\text{R.64}} \alpha_2]\!] & (\alpha_1 = -\infty \wedge \alpha_2 = +\infty) & = +\text{NaN} \\ & (\alpha_1 = +\infty \wedge \alpha_2 = -\infty) & = -\text{NaN} \\ & (\alpha_1 = +\infty \vee \alpha_2 = +\infty) & = +\infty \\ & (\alpha_1 = -\infty \vee \alpha_2 = -\infty) & = -\infty \\ & (\alpha_1 = \alpha_2 = -0) & = -0 \end{array}$$

Similarly, in a floating point subtraction operation, the same five special cases have equivalent definitions, after flipping the sign bit of the second operand as follows:

$$\begin{array}{l|ll} \mathcal{E}'[\![\alpha_1 -_{\text{R.64}} \alpha_2]\!] & (\alpha_1 = \alpha_2 = -\infty) & = +\text{NaN} \\ & (\alpha_1 = \alpha_2 = +\infty) & = -\text{NaN} \\ & (\alpha_1 = +\infty \vee \alpha_2 = -\infty) & = +\infty \\ & (\alpha_1 = -\infty \vee \alpha_2 = +\infty) & = -\infty \\ & (\alpha_1 = -0 \ \wedge \alpha_2 = +0) & = -0 \end{array}$$

On the other hand, floating point multiplication identifies only three special cases. First of all, the product of ±0 with ±∞ (in any order) is equal to +NaN if both operands have identical signs, or −NaN otherwise. All other multiplications involving infinite or zero values result in ±∞ or ±0 of an appropriate sign, which, in the following Haskell definition, is captured by adding 2^{63} to the encoding of +NaN, +∞ or +0 whenever the sign bits of the two operands have opposite values:

$$\begin{array}{l|ll} \mathcal{E}'[\![\alpha_1 \times_{\text{R.64}} \alpha_2]\!] & (\alpha_1 \in \{\pm\infty\} \wedge \alpha_2 \in \{\pm 0\}) \ \vee & \\ & \quad (\alpha_1 \in \{\pm 0\} \ \wedge \alpha_2 \in \{\pm\infty\}) & = [\![\#[\![\text{dec}_{[\![\text{N.64}]\!]}(+\text{NaN}) + 2^{63}s]\!]_{\text{N.64}}]\!] \\ & (\alpha_1 \in \{\pm\infty\} \vee \alpha_2 \in \{\pm\infty\}) & = [\![\#[\![\text{dec}_{[\![\text{N.64}]\!]}(+\infty) + 2^{63}s]\!]_{\text{N.64}}]\!] \\ & (\alpha_1 \in \{\pm 0\} \ \vee \alpha_2 \in \{\pm 0\}) & = [\![\#[\![\text{dec}_{[\![\text{N.64}]\!]}(+0) + 2^{63}s]\!]_{\text{N.64}}]\!] \end{array}$$

where $s = \text{dec}_{[\![\text{N.64}]\!]}(\alpha_1)[63] \neq \text{dec}_{[\![\text{N.64}]\!]}(\alpha_2)[63]$

In the same manner, the floating point division operations $\pm 0 \div \pm 0$ and $\pm\infty \div \pm\infty$ are semantically equivalent to ±NaN of an appropriate sign. Otherwise, divisions of the form $\pm\infty \div y$ and $x \div \pm 0$ result in $\pm\infty$, while $\pm 0 \div y$ and $x \div \pm\infty$ are reducible to ±0. Formally:

$$\begin{array}{l|ll} \mathcal{E}'[\![\alpha_1 \div_{\text{R.64}} \alpha_2]\!] & (\alpha_1 \in \{\pm 0\} \ \wedge \alpha_2 \in \{\pm 0\}) \ \vee & \\ & \quad (\alpha_1 \in \{\pm\infty\} \wedge \alpha_2 \in \{\pm\infty\}) & = [\![\#[\![\text{dec}_{[\![\text{N.64}]\!]}(+\text{NaN}) + 2^{63}s]\!]_{\text{N.64}}]\!] \\ & (\alpha_1 \in \{\pm\infty\} \vee \alpha_2 \in \{\pm 0\}) & = [\![\#[\![\text{dec}_{[\![\text{N.64}]\!]}(+\infty) + 2^{63}s]\!]_{\text{N.64}}]\!] \\ & (\alpha_1 \in \{\pm 0\} \ \vee \alpha_2 \in \{\pm\infty\}) & = [\![\#[\![\text{dec}_{[\![\text{N.64}]\!]}(+0) + 2^{63}s]\!]_{\text{N.64}}]\!] \end{array}$$

where $s = \text{dec}_{[\![\text{N.64}]\!]}(\alpha_1)[63] \neq \text{dec}_{[\![\text{N.64}]\!]}(\alpha_2)[63]$

Division by zero also crops up in integer arithmetic. If ϕ represents a non-rational format, then, with all arithmetic traps disabled, the MMIX processor reduces all atoms of the form "$\alpha_1 \div_\phi \#0_{\text{N.64}}$" and "$\alpha_1 \cdot|\cdot_\phi \#0_{\text{N.64}}$" to α_2 or α_1, respectively:

$$\begin{array}{ll} \mathcal{E}'[\![\alpha_1 \div_\phi \alpha_2]\!] & (\gamma(\phi) \in \{\text{N}, \text{Z}\} \wedge 1 \leq \varepsilon(\phi) \leq 64 \wedge \text{dec}_\phi(\alpha_2) = 0) = \alpha_2 \\ \mathcal{E}'[\![\alpha_1 \cdot|\cdot_\phi \alpha_2]\!] & (\gamma(\phi) \in \{\text{N}, \text{Z}\} \wedge 1 \leq \varepsilon(\phi) \leq 64 \wedge \text{dec}_\phi(\alpha_2) = 0) = \alpha_1 \end{array}$$

The final special case pertains to those applications of the six relational operators "$=_\phi$", "$\neq_\phi$", "$<_\phi$", "$>_\phi$", "$\leq_\phi$" and "$\geq_\phi$" under the rational format "R.64", in which one or both of the two atomic operands has an infinite value. In this case, the result is always equal to "$\#1_{\text{N.64}}$" or "$\#0_{\text{N.64}}$" whenever the corresponding mathematical relation does or does not hold asymptotically for the supplied operands. Formally:

$$\begin{aligned}
\mathcal{E}'[\![\alpha_1 =_{\text{R.64}} \alpha_2]\!] &\mid \alpha_1 \in \{\pm\infty\} \vee \alpha_2 \in \{\pm\infty\} = [\![\#[\![\alpha_1 = \alpha_2]\!]_{\text{N.64}}]\!] \\
\mathcal{E}'[\![\alpha_1 \neq_{\text{R.64}} \alpha_2]\!] &\mid \alpha_1 \in \{\pm\infty\} \vee \alpha_2 \in \{\pm\infty\} = [\![\#[\![\alpha_1 \neq \alpha_2]\!]_{\text{N.64}}]\!] \\
\mathcal{E}'[\![\alpha_1 <_{\text{R.64}} \alpha_2]\!] &\mid \alpha_1 \in \{\pm\infty\} \vee \alpha_2 \in \{\pm\infty\} = [\![\#[\![\alpha_1 \neq +\infty \wedge \alpha_2 \neq -\infty]\!]_{\text{N.64}}]\!] \\
\mathcal{E}'[\![\alpha_1 >_{\text{R.64}} \alpha_2]\!] &\mid \alpha_1 \in \{\pm\infty\} \vee \alpha_2 \in \{\pm\infty\} = [\![\#[\![\alpha_1 \neq -\infty \wedge \alpha_2 \neq +\infty]\!]_{\text{N.64}}]\!] \\
\mathcal{E}'[\![\alpha_1 \leq_{\text{R.64}} \alpha_2]\!] &\mid \alpha_1 \in \{\pm\infty\} \vee \alpha_2 \in \{\pm\infty\} = [\![\#[\![\alpha_1 = -\infty \vee \alpha_2 = +\infty]\!]_{\text{N.64}}]\!] \\
\mathcal{E}'[\![\alpha_1 \geq_{\text{R.64}} \alpha_2]\!] &\mid \alpha_1 \in \{\pm\infty\} \vee \alpha_2 \in \{\pm\infty\} = [\![\#[\![\alpha_1 = +\infty \vee \alpha_2 = -\infty]\!]_{\text{N.64}}]\!]
\end{aligned}$$

Otherwise, every ordinary application of the binary Etude operator "$+_\phi$", "$-_\phi$" and "$\times_\phi$" is equivalent to the true arithmetic value of the corresponding mathematical operation "$+$", "$-$" or "$\times$", encoded under the specified format ϕ as follows:

$$\begin{aligned}
\mathcal{E}'[\![\alpha_1 +_\phi \alpha_2]\!] &= \text{enc}_\phi(\text{dec}_\phi(\alpha_1) + \text{dec}_\phi(\alpha_2)) \\
\mathcal{E}'[\![\alpha_1 -_\phi \alpha_2]\!] &= \text{enc}_\phi(\text{dec}_\phi(\alpha_1) - \text{dec}_\phi(\alpha_2)) \\
\mathcal{E}'[\![\alpha_1 \times_\phi \alpha_2]\!] &= \text{enc}_\phi(\text{dec}_\phi(\alpha_1) \times \text{dec}_\phi(\alpha_2))
\end{aligned}$$

More so, the division operation "$\alpha_1 \div_\phi \alpha_2$" reduces into "$\#[\![\text{dec}_\phi(\alpha_1)/\text{dec}_\phi(\alpha_2)]\!]_\phi$" if ϕ represents a rational format, or "$\#[\![\lfloor \text{dec}(\phi)\ (\alpha_1)/\text{dec}(\phi)\ (\alpha_2) \rfloor]\!]_\phi$" under any other arithmetic genre, while the dual remainder operation "$\alpha_1 \cdot|\cdot_\phi \alpha_2$" produces an appropriate binary encoding of the numeric quantity "$\text{dec}_\phi(x) - \lfloor \text{dec}_\phi(x)/\text{dec}_\phi(y) \rfloor \times \text{dec}_\phi(x)$". The later operation is, however, applicable only under integral formats and always remains undefined if ϕ is equal to "R.64":

$$\begin{aligned}
\mathcal{E}'[\![\alpha_1 \div_\phi \alpha_2]\!] &\mid \gamma(\phi) = [\![\text{R}]\!] = \text{enc}_\phi(\text{dec}_\phi(\alpha_1)/\text{dec}_\phi(\alpha_2)) \\
&\mid \gamma(\phi) \neq [\![\text{R}]\!] = \text{enc}_\phi \lfloor \text{dec}_\phi(\alpha_1)/\text{dec}_\phi(\alpha_2) \rfloor \\
\mathcal{E}'[\![\alpha_1 \cdot|\cdot_\phi \alpha_2]\!] &\mid \gamma(\phi) \neq [\![\text{R}]\!] = \text{enc}_\phi(\text{dec}_\phi(\alpha_1) - \lfloor \text{dec}_\phi(\alpha_1)/\text{dec}_\phi(\alpha_2) \rfloor \times \text{dec}_\phi(\alpha_1))
\end{aligned}$$

In the bitwise operations "$\alpha_1 \triangle_\phi \alpha_2$", "$\alpha_1 \bar{\triangle}_\phi \alpha_2$" and "$\alpha_1 \nabla_\phi \alpha_2$", the format component ϕ constitutes a syntactic noise and is always ignored by the MMIX processor, even if it happens to have an invalid value. Instead, each of these three atomic forms is reducible into the bitwise product, difference or sum of the binary encodings x and y, as captured by the sum of a series $[2^k(\alpha_1[k]\ op\ \alpha_2[k]) \mid k \leftarrow [0 \mathinner{..} 63]]$, in which op is equal to one of the three boolean Haskell operators "$\wedge$", "$\neq$" or "$\vee$" for "$\triangle_\phi$", "$\bar{\triangle}_\phi$" and "∇_ϕ", respectively. In other words, "$\alpha_1 \triangle_\phi \alpha_2$" is always equivalent to a natural number n, whose kth bit is set in its binary representation if and only if the same bit is set in both of the operands α_1 and α_2. Similarly, the "$\bar{\triangle}_\phi$" operator sets this result bit if the corresponding indices of α_1 and α_2 have opposite values, while "∇_ϕ" sets it whenever at least one of these input bits is set. Formally:

$$\begin{aligned}
\mathcal{E}'[\![\#x_{\text{N.64}} \triangle_\phi \#y_{\text{N.64}}]\!] &= [\![\#[\![\textstyle\sum [2^k(x[k] \wedge y[k]) \mid k \leftarrow [0 \mathinner{..} 63]]]\!]_{\text{N.64}}]\!] \\
\mathcal{E}'[\![\#x_{\text{N.64}} \bar{\triangle}_\phi \#y_{\text{N.64}}]\!] &= [\![\#[\![\textstyle\sum [2^k(x[k] \neq y[k]) \mid k \leftarrow [0 \mathinner{..} 63]]]\!]_{\text{N.64}}]\!] \\
\mathcal{E}'[\![\#x_{\text{N.64}} \nabla_\phi \#y_{\text{N.64}}]\!] &= [\![\#[\![\textstyle\sum [2^k(x[k] \vee y[k]) \mid k \leftarrow [0 \mathinner{..} 63]]]\!]_{\text{N.64}}]\!]
\end{aligned}$$

Every application of the "$\ll_\phi$" or "$\gg_\phi$" operator under a non-rational format is equivalent to multiplying or dividing the decoded value of its first operand by 2^n, where n is the natural number derived from the second operand under the natural format "N.64". For "$\gg_\phi$", the result is rounded towards $-\infty$ in the same way as was done earlier by the ordinary integer division operator "$\div_\phi$". Formally:

$$\begin{aligned}
&\mathcal{E}'[\![\alpha_1 \ll_\phi \alpha_2]\!] \mid \gamma(\phi) \neq [\![\mathsf{R}]\!] = \mathrm{enc}_\phi(\mathrm{dec}_\phi(\alpha_1) \times 2^{\mathrm{dec}_{[\![\mathsf{N.64}]\!]}(\alpha_2)}) \\
&\mathcal{E}'[\![\alpha_1 \gg_\phi \alpha_2]\!] \mid \gamma(\phi) \neq [\![\mathsf{R}]\!] = \mathrm{enc}_\phi\lfloor \mathrm{dec}_\phi(\alpha_1)/2^{\mathrm{dec}_{[\![\mathsf{N.64}]\!]}(\alpha_2)} \rfloor
\end{aligned}$$

On the other hand, when one of the six relational Etude operators "$=_\phi$", "$\neq_\phi$", "$<_\phi$", "$>_\phi$", "$\leq_\phi$" or "$\geq_\phi$" is applied to a pair of finite numeric values, the entire operation is always equivalent to a natural atom of the form "$\#n_{\mathsf{N.64}}$", in which the integer n has the value of 1 or 0, obtained from the numeric representation of the corresponding Haskell boolean expression "$x = y$", "$x \neq y$", "$x < y$", "$x > y$", "$x \leq y$" or "$x \geq y$", in which x and y represent the respective decoded numeric values of the construct's two operands. In Haskell:

$$\begin{aligned}
&\mathcal{E}'[\![\alpha_1 =_\phi \alpha_2]\!] = \mathrm{enc}_{[\![\mathsf{N.64}]\!]}(\mathrm{dec}_\phi(\alpha_1) = \mathrm{dec}_\phi(\alpha_2)) \\
&\mathcal{E}'[\![\alpha_1 \neq_\phi \alpha_2]\!] = \mathrm{enc}_{[\![\mathsf{N.64}]\!]}(\mathrm{dec}_\phi(\alpha_1) \neq \mathrm{dec}_\phi(\alpha_2)) \\
&\mathcal{E}'[\![\alpha_1 <_\phi \alpha_2]\!] = \mathrm{enc}_{[\![\mathsf{N.64}]\!]}(\mathrm{dec}_\phi(\alpha_1) < \mathrm{dec}_\phi(\alpha_2)) \\
&\mathcal{E}'[\![\alpha_1 >_\phi \alpha_2]\!] = \mathrm{enc}_{[\![\mathsf{N.64}]\!]}(\mathrm{dec}_\phi(\alpha_1) > \mathrm{dec}_\phi(\alpha_2)) \\
&\mathcal{E}'[\![\alpha_1 \leq_\phi \alpha_2]\!] = \mathrm{enc}_{[\![\mathsf{N.64}]\!]}(\mathrm{dec}_\phi(\alpha_1) \leq \mathrm{dec}_\phi(\alpha_2)) \\
&\mathcal{E}'[\![\alpha_1 \geq_\phi \alpha_2]\!] = \mathrm{enc}_{[\![\mathsf{N.64}]\!]}(\mathrm{dec}_\phi(\alpha_1) \geq \mathrm{dec}_\phi(\alpha_2))
\end{aligned}$$

The only cases remaining in the definition of $\mathcal{E}'$ pertain to the three unary Etude operators "$-_\phi$", "$\sim_\phi$" and "ϕ'_ϕ". First of all, every atom of the form "$-_\phi(\alpha)$" is essentially equivalent to "$[\![-0]\!]\ -_\phi\ \alpha$" if ϕ belongs to the rational genre, or else to "$[\![+0]\!]\ -_\phi\ \alpha$" if it represents any other well-formed Etude format. Further, the *bit complement operation* "$\sim_\phi(\alpha)$" stands for a subtraction of α's value from $2^{64} - 1$ under every Etude format other than "R.64":

$$\begin{aligned}
\mathcal{E}'[\![-_\phi(\alpha)]\!] &\mid \phi = [\![\mathsf{R.64}]\!] = \mathcal{E}'[\![[\![-0]\!]\ -_\phi\ \alpha]\!] \\
&\mid \phi \neq [\![\mathsf{R.64}]\!] = \mathcal{E}'[\![[\![+0]\!]\ -_\phi\ \alpha]\!] \\
\mathcal{E}'[\![\sim_\phi(\alpha)]\!] &\mid \phi \neq [\![\mathsf{R.64}]\!] = \mathcal{E}'[\![\#[\![2^{64} - 1]\!]_{\mathsf{N.64}}\ -_\phi\ \alpha]\!]
\end{aligned}$$

Last but not least, all Etude *conversion operations* of the form "$\phi'_\phi(\alpha)$" denote the numeric value of α decoded under the given format ϕ and reinterpreted under the target format ϕ', except that, if α represents a non-finite floating point quantity, then its binary encoding is reinterpreted directly without further analysis and, if the source and target formats are identical, then the operation is always reducible directly to α, even if ϕ does not represent a well-formed Etude format. Formally:

$$\begin{aligned}
\mathcal{E}'[\![\phi'_\phi(\alpha)]\!] &\mid \phi = \phi' &&= [\![\alpha]\!] \\
&\mid \phi = [\![\mathsf{R.64}]\!] \wedge \neg\mathrm{FIN}(\alpha) &&= \mathrm{enc}_{\phi'}(\mathrm{dec}_\phi(\alpha)) \\
&\mid \text{otherwise} &&= \mathrm{enc}_{\phi'}(\mathrm{dec}_{[\![\mathsf{N.64}]\!]}(\alpha))
\end{aligned}$$

COROLLARY 6-9: *(Reduction of atoms)* Under $\mathcal{E}$, every well-formed atom is reducible into a valid atomic constant:

$$\begin{aligned}&\mathrm{WF}_{\mathcal{E}} \;::\; \forall\, \alpha \Rightarrow \mathrm{WF}(\alpha) \to \mathrm{WF}[\![\mathcal{E}(\alpha)]\!]\\ &\mathrm{IMM}_{\mathcal{E}} :: \forall\, \alpha \Rightarrow \mathrm{WF}(\alpha) \to [\![\mathrm{IMM}(\mathcal{E}(\alpha))]\!]\end{aligned}$$

THEOREM 6-10: *(Generic validity of constant atoms)* The following trivial atomic forms are always well-formed:

$$\mathrm{WF}_0 :: \forall\, \phi \Rightarrow \mathrm{WF}(\phi) \to \mathrm{WF}[\![\#0_\phi]\!]$$

$$\begin{aligned}&\mathrm{WF}_{\mathrm{I}} :: \forall\, n :: \mathit{integer},\, \phi \Rightarrow\\ &\quad \mathrm{WF}(\phi) \to\\ &\quad [\![\gamma(\phi) \in \{\mathsf{N}, \mathsf{Z}\} \wedge \mathrm{glb}(\phi) \le n \le \mathrm{lub}(\phi)]\!] \to\\ &\quad \mathrm{WF}[\![\#x_\phi]\!]\end{aligned}$$

$$\begin{aligned}&\mathrm{WF}_{\mathrm{R}} :: \forall\, s, m, e :: \mathit{integer},\, \phi \Rightarrow\\ &\quad \mathrm{WF}(\phi) \to\\ &\quad [\![\gamma(\phi) = [\![\mathsf{R}]\!] \wedge 0 \le s \le 1 \wedge 0 \le m < \mathrm{r}(\phi)^{\mathrm{p}(\phi)} \wedge \mathrm{E}_{\min}(\phi) \le e \le \mathrm{E}_{\max}(\phi)]\!] \to\\ &\quad \mathrm{WF}[\![\#[\![(-1)^s \times m \times \mathrm{r}(\phi)^{e\,-\,\mathrm{p}(\phi)}]\!]_\phi]\!]\end{aligned}$$

PROOF: From the various cases in the definition of $\mathcal{E}$, we have $\mathcal{E}[\![\#0_\phi]\!] = [\![\#0_{\mathrm{N}.64}]\!]$, which is obviously well-formed by $\mathrm{WF}_{\mathrm{NC}}$, since $0 \le 0 < 2^{64}$. Accordingly, WF_0 can be established by a simple case inspection of all well-formed Etude formats ϕ, while utilising the guarantees provides by the supplied validity assumption for satisfaction of the required preconditions featured in the definition of $\mathcal{E}$.

Similarly, for all well-formed integral formats ϕ and $\mathrm{glb}(\phi) \le n \le \mathrm{lub}(\phi)$, we can establish the reduction chain $\mathcal{E}[\![\#n_\phi]\!] = \mathrm{enc}_\phi(x) = [\![\#[\![\lfloor x \rfloor \bmod 2^\varepsilon]\!]_{\mathrm{N}.64}]\!]$ if ϕ belongs to the natural genre "N" and $\mathcal{E}[\![\#n_\phi]\!] = [\![\#[\![\lfloor x \rfloor \bmod 2^\varepsilon + (2^{64} - 2^\varepsilon) \times \lfloor x \rfloor[\varepsilon - 1]]\!]_{\mathrm{N}.64}]\!]$ if it represents an integer format. Either way, by definition of the "mod" operator, $n \bmod m < m$ for all possible integer values of n and m, and, by well-formedness of ϕ, $\varepsilon < 64$, so that, in both cases, the ensuing natural constant $\lfloor x \rfloor \bmod 2^\varepsilon$ or, for signed integers, $\lfloor x \rfloor \bmod 2^\varepsilon + (2^{64} - 2^\varepsilon) \times \lfloor x \rfloor[\varepsilon - 1]$, must always represent a non-negative integer less than 2^{64} in magnitude, which is sufficient to establish a validity of the supplied atom directly from the axiom $\mathrm{WF}_{\mathrm{NC}}$.

The proof of WF_{R} has a similar structure, using the supplied constraints on the integers s, m and e to satisfy the corresponding preconditions of $\mathcal{E}$. $\mathrm{WF}_{\mathrm{NC}}$ can then be established directly by a simple arithmetic reasoning about the range of all possible natural numbers $2^{63}s + 2^p e + \mathrm{round}(2^{b\,+\,p\,-\,e}|x| - 2^p)$ that are featured in the definition of "enc". □

An actual symbolic rendition of these proofs in the framework of the Haskell extensions from Chapter 4 would require an uncomplicated but rather extensive reasoning that, in Fermat's famous words, is simply too long to fit in the margin of this book, given that it would contribute little to our understanding of the critical verification processes at play within a linearly correct program translation system.

THEOREM 6-11: *(Addition of rational values)* Every well-formed Etude atom of the form "$\#x_\phi +_\phi \#y_\phi$" is equivalent to "$\#[\![x+y]\!]_\phi$", up to the rounding error of the specified rational format ϕ, provided that $\mathrm{glb}(\phi) \leq x+y \leq \mathrm{lub}(\phi)$:

$$\begin{aligned}
&\mathrm{EQV}_{+\mathrm{A}} :: \forall\, x,\, y,\, z,\, \phi \Rightarrow \\
&\quad \mathrm{WF}[\![\#x_\phi]\!] \rightarrow \mathrm{WF}[\![\#y_\phi]\!] \rightarrow \\
&\quad [\![\gamma(\phi) \in \{\mathsf{Z},\, \mathsf{R}\} \wedge \mathrm{glb}(\phi) \leq x+y \leq \mathrm{lub}(\phi)]\!] \rightarrow \\
&\quad [\![\#x_\phi +_\phi \#y_\phi]\!] \equiv [\![\#z_\phi]\!] \rightarrow \\
&\quad [\![|z-(x+y)| < \mathrm{ulp}_\phi(x+y)]\!]
\end{aligned}$$

PROOF: First of all, we observe that, given the assumption $\mathrm{glb}(\phi) \leq x+y \leq \mathrm{lub}(\phi)$, in the following proof we only need to concern ourselves with finite operand values, safely discarding all infinities and NaNs. Accordingly, let us begin with a detailed analysis of the common reduction chain of $\mathcal{E}$ for all such operands:

$$\begin{aligned}
&\mathcal{E}[\![\#x_\phi +_\phi \#y_\phi]\!] \\
&\quad = \mathcal{E}'[\![[\![\mathcal{E}[\![\#x_\phi]\!]]\!] +_\phi [\![\mathcal{E}[\![\#y_\phi]\!]]\!]]\!] && \textit{(reduce } \mathcal{E}\textit{)} \\
&\quad = \mathcal{E}'[\![[\![\mathrm{enc}_{[\![\phi]\!]}[\![\#x_\phi]\!]]\!] +_\phi [\![\mathrm{enc}_{[\![\phi]\!]}[\![\#y_\phi]\!]]\!]]\!] && \textit{(reduce } \mathcal{E}\textit{)} \\
&\quad = \mathrm{enc}_{[\![\phi]\!]}(\mathrm{dec}_{[\![\phi]\!]}(\mathrm{enc}_{[\![\phi]\!]}[\![\#x_\phi]\!]) + \mathrm{dec}_{[\![\phi]\!]}(\mathrm{enc}_{[\![\phi]\!]}[\![\#y_\phi]\!])) && \textit{(reduce } \mathcal{E}\textit{)} \\
&\quad = \mathrm{enc}_{[\![\mathsf{R}.64]\!]}(x+y) && \textit{(expand } \mathcal{E}\textit{)} \\
&\quad = z'
\end{aligned}$$

where the third reduction step follows from the precondition $x = \mathrm{dec}_{[\![\phi]\!]}(\mathrm{enc}_{[\![\phi]\!]}[\![\#x_\phi]\!])$ found in the definition of $\mathcal{E}$ over Etude constants.

Now, if $z = 0$, then, by definition of "enc", the resulting atom is equivalent to "$\#0_{\mathrm{N}.64}$". Under the IEEE floating point model, the decoded value of such an atom, represented in this work by the notation "$\mathrm{dec}_\phi[\![\#0_{\mathrm{N}.64}]\!]$", is equal to the rational number 0, so that $|0-0| = 0 < \mathrm{ulp}_\phi(0) = 2^{-1021}$ as required.

Otherwise, let $z = x+y$, with the biased exponent e equal to $\lfloor \log_2|z| \rfloor + b$ and let s be the sign of z, defined by the numeric value of the relation $z < 0$. Further, let $s' = z'[63]$, $e' = z'[52\ ...\ 62]$ and $m' = z'[0\ ...\ 51]$, so that these three variables represent the sign, exponent and mantissa of the encoded constant z'.

If z is normalised, so that $0 < e \leq 2b$, then, by definition of "ulp", the required margin of error is equal to $2^{\lfloor \log_2|z| \rfloor - \mathrm{p}(\phi)} = 2^{e-b-\mathrm{p}(\phi)} = 2^{e-b-p+1}$. Further, the "enc" function gives us $z = 2^{63}s + 2^p e + \mathrm{round}(2^{b+p-e}|z|) - 2^p$, so that $s' = s$, $e' = e$ and $m' = \mathrm{round}(2^{b+p-e}|z|) - 2^p$ by definition of bit extraction. Accordingly:

$$\begin{aligned}
&|\mathrm{dec}_\phi[\![\#z_{\mathrm{N}.64}]\!] - z| \\
&\quad = |(-1)^{s'} \times (m'+2^p)/2^{b+p-e'} - z| && \textit{(reduce ``dec'')} \\
&\quad = |(-1)^{s} \times (m'+2^p)/2^{b+p-e} - z| && \textit{(substitute } s' \textit{ and } e'\textit{)} \\
&\quad = |(-1)^{s} \times (m'+2^p)/2^{b+p-e} - (-1)^s \times |z|| && \textit{(properties of ``}|\cdot|\textit{'')} \\
&\quad = |(-1)^{s} \times ((m'+2^p)/2^{b+p-e} - |z|)| && \textit{(distribution of } (-1)^{z<0}\textit{)} \\
&\quad = (m'+2^p)/2^{b+p-e} - |z| && \textit{(since } |(-1)^x| = x \textit{ if } x > 0\textit{)} \\
&\quad = (\mathrm{round}(2^{b+p-e}|z|) - 2^p + 2^p)/2^{b+p-e} - |z| && \textit{(substitute } m'\textit{)} \\
&\quad = \mathrm{round}(2^{b+p-e}|z|)/2^{b+p-e} - |z| && \textit{(algebraic simplifications)} \\
&\quad < (2^{b+p-e}|z| + 1)/2^{b+p-e} - |z| && \textit{(properties of ``round'')} \\
&\quad = 2^{e-b-p} && \textit{(algebraic simplifications)} \\
&\quad < 2^{e-b-p+1}
\end{aligned}$$

as required. The increased precision of our result is due to the fact that our MMIX processor always performs exact rounding of all floating point quantities, so that none of its arithmetic operations can ever introduce more than half an ULP of error.

Finally, if z represents a denormalised quantity, i.e., if $e \leq 0$, then the result is encoded as $z' = 2^{63}s + \text{round}(2^{b+p-1}|z|)$, so that $m' = \text{round}(2^{b+p-1}|z|)$, $s' = s$ and $e' = 0$ by definition of the bit extraction operator. Further, for all such denormalised numbers, the required unit of least precision $\text{ulp}_\phi(z)$ is defined as $2^{E_{\min}(\phi)-1} = 2^{-1022}$. Keeping these facts in mind, the exact amount of rounding error introduced by the operation can be computed as follows:

$$
\begin{aligned}
&|\text{dec}_\phi[\![\#z_{\text{N.64}}]\!] - z| \\
&\quad = |(-1)^{s'} \times m'/2^{b+p-1} - z| && \textit{(reduce ``dec'')} \\
&\quad = |(-1)^{s} \times m'/2^{b+p-1} - z| && \textit{(substitute } s'\textit{)} \\
&\quad = |(-1)^{s} \times m'/2^{b+p-1} - (-1)^{z<0} \times |z|| && \textit{(properties of ``}|\cdot|\textit{'')} \\
&\quad = |(-1)^{s} \times m'/2^{b+p-1} - (-1)^{s} \times |z|| && \textit{(substitute } s\textit{)} \\
&\quad = |(-1)^{s} \times (m'/2^{b+p-1} - |z|)| && \textit{(distribution of } (-1)^{z<0}\textit{)} \\
&\quad = m'/2^{b+p-1} - |z| && \textit{(since } |(-1)^x| = x \textit{ if } x > 0\textit{)} \\
&\quad = \text{round}(2^{b+p-1}|x|)/2^{b+p-1} - |z| && \textit{(substitute } m'\textit{)} \\
&\quad < (2^{b+p-1}|z| + 1)/2^{b+p-1} - |z| && \textit{(properties of ``round'')} \\
&\quad = |z| + 2^{1-b-p} - |z| && \textit{(distribution of } 2^{b+p-1}\textit{)} \\
&\quad = 2^{1-1023-52} && \textit{(simplify, substitute } b \textit{ and } p\textit{)} \\
&\quad < 2^{-1022}
\end{aligned}
$$

as required. Observe that, for denormalised numbers, MMIX buys us a whooping 52 bits of precision, as a result of its complete support for gradual underflow performed during such operations, that is permitted but not mandated by Etude. In the above definitions, steps 3 and 7 follow from the well-known identities "$\text{round}(x) < x + 1$" and "$x = (-1)^{x<0} \times |x|$", which hold for all rational numbers x. □

Section 4.4 lists 45 additional theorems that are used to capture the generic algebraic semantics of all arithmetic Etude operations. However, for brevity, in this chapter I omit their detailed justifications, since all of the corresponding proofs follow a structure similar to, and generally much simpler than the above reasoning about the properties of rational addition, save for the less interesting details of Haskell arithmetic.

6.3.2 Evaluation State

In Section 4.5, the object environment of an Etude program was represented by an abstract type "*o-env*", whose properties were described solely through a set of suitable operations on such environments. It is now time to rectify this situation and specify the actual structure of the address space made available to all MMIX programs. Formally, this structure is defined by the following three Haskell types:

o-env:	*M*, *A*, *envelope*, *envelope*, *stack*, *integer*, *integer*, *integer*
M:	*integer* $\mapsto$ *integer*
A:	*integer* $\mapsto$ *bool*

In other words, every object environment depicts a tuple $(M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f)$, in which the finite map M, known as the program's *memory image*, defines the actual

bindings of individual addresses σ to their respective byte values stored at the corresponding memory locations. The current atomic content of a given object (σ, ϕ) can be retrieved from M by the construction $\bar{\alpha}(\Delta \triangleright \sigma, \phi)$, which, on MMIX, is defined as follows:

$$\bar{\alpha}[\![\cdot]\!]\ [\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow (\textit{o-env} \triangleright \textit{integer}, \textit{format}) \rightarrow \textit{atom}_\nu$$

$$\bar{\alpha}[\![M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f \triangleright \sigma, \phi]\!] \mid \phi \in \{\text{N.8}, \text{N.16}, \text{N.32}, \text{N.64}, \text{R.64}\} = [\![\#x_{\text{N.64}}]\!] \mid \text{otherwise} = \mathcal{E}[\![\phi_{\text{N.64}}(\#x_{\text{N.64}})]\!]$$

$$\text{where } x = \sum[2^{\omega k} M(\sigma' + \mathcal{S}(\phi) - 1 - k) \mid k \leftarrow [0 \ldots \mathcal{S}(\phi) - 1]]$$
$$\sigma' = \sigma - \sigma \bmod \mathcal{S}(\phi)$$

Intuitively, the above definition renders MMIX into a big-endian architecture, whereby the proper encoded value of the memory-resident object requested by $\bar{\alpha}(\Delta \triangleright \sigma, \phi)$ is assembled from the individual bytes of the corresponding memory image M, with the most significant 8 bits of the object fetched from the least address in the region. Unless ϕ specifies one of five blessed formats "N.8", "N.16", "N.32", "N.64" or "R.64", the resulting natural atom is always converted into ϕ, as if by the reduction of an appropriate term of the form "$\phi_{\text{N.64}}(\#x_{\text{N.64}})$". The reader should note that, unlike most other modern instruction set architectures, MMIX does not assume that the supplied address σ represents a multiple of the format's size and always adjusts any misaligned addresses, implicitly discarding the least-significant 3, 2, 1 or 0 bits of σ if required.

Further, the two envelopes $\bar{\xi}$ and $\bar{\xi}_i$ found in every MMIX object environment $(M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f)$ correspond directly to the similarly-named functions from Section 4.5:

$$\bar{\xi}\ [\![\cdot]\!], \bar{\xi}_i[\![\cdot]\!] :: \textit{o-env} \rightarrow \textit{envelope}$$

$$\bar{\xi}\ [\![M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f]\!] = \bar{\xi}$$
$$\bar{\xi}_i[\![M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f]\!] = \bar{\xi}_i$$

while the environment's stack $\bar{\psi}(\Delta)$ is, by definition, equal to its component $\bar{\psi}$:

$$\bar{\psi}[\![\cdot]\!] :: \textit{o-env} \rightarrow \textit{stack}$$

$$\bar{\psi}[\![M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f]\!] = \bar{\psi}$$

In practice, none of these three object environment components are used directly on the MMIX architecture, so that their inclusion in the structure of "*o-env*" has been effected only to facilitate identification of various memory allocation patterns that are considered illegal by the standard of the ultimate mapping between Etude programs and their MMIX representations in Section 6.4. Instead, the actual structure of the program's address space is defined solely by the environment's *address space* component A, which associates every byte address σ accessible to the program with a boolean flag $A(\sigma)$, set to a true value if and only if the underlying operating system has granted our program the right to modify the content of the byte object located at that address. Within the domain of A, a smaller range of consecutive addresses $[\sigma_f \ldots \sigma_i]$ is reserved for sole use by the program's data stack $\bar{\psi}(\Delta)$ and, further, the *current stack pointer* component σ_c

determines the least stack address that is currently utilised by the program. Accordingly, the construction $\sigma_c(\Delta \triangleright \bar{\xi})$ that, in Section 4.5, determined the location of a new envelope $\bar{\xi}$ upon its introduction into the address space by the "NEW ($\bar{\xi}$)" term form, is defined on MMIX as follows:

$$\sigma_c[\![\cdot]\!] :: (\textit{o-env} \triangleright \textit{envelope}) \to \textit{integer}$$
$$\sigma_c[\![M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f \triangleright \bar{\xi}']\!] = \sigma_c - \lceil \mathcal{S}(\bar{\xi}')/8 \rceil \times 8$$

so that each newly introduced stack frame is always placed below any that are already present in the program's address space and is assigned a memory location aligned to the architecture's word boundary of 8.

In Section 4.7, numeric address values were derived from their atomic representations using the implementation-defined constructs $\mathcal{L}_{\mathsf{O}}(\alpha)$ and $\mathcal{L}_{\mathsf{F}}(\alpha)$. On MMIX, both of these functions can be implemented by decoding the normalised form of α as follows:

$$\mathcal{L}_{\mathsf{O}}[\![\cdot]\!], \mathcal{L}_{\mathsf{F}}[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow \textit{atom}_\nu \to \textit{integer}$$
$$\mathcal{L}_{\mathsf{O}}[\![\alpha]\!] = \text{dec}_{[\![\mathsf{O}.64]\!]}(\mathcal{E}(\alpha))$$
$$\mathcal{L}_{\mathsf{F}}[\![\alpha]\!] = \text{dec}_{[\![\mathsf{F}.64]\!]}(\mathcal{E}(\alpha))$$

Throughout its entire execution, every well-formed Etude program must ensure that its environment $(M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f)$ always satisfies the following nine invariants:

① The envelope $\bar{\xi}$ must be well-formed.
② The initialiser envelope $\bar{\xi}_i$ must represent a subset of $\bar{\xi}$.
③ All frames of the stack $\bar{\psi}$ must be allocated in $\bar{\xi}$.
④ All frames of the stack $\bar{\psi}$ must be allocated within the region $[\sigma_f \ldots \sigma_i]$ that is reserved for this purpose by the operating system.
⑤ The stack pointer σ_c must also fall within that address region, with $\sigma_c \bmod 8 = 0$, so that σ_c represents a value of a well-formed atom under the object format "O.61".
⑥ Every address $\sigma \in \text{dom}(A)$ must have a value in the range $0 \leq \sigma < 2^{63}$, so that σ depicts a well-formed numeric value of an Etude atom under the "O.64" format. If $A(\sigma)$ is false, then the address must also appear in the domain of M, in order to ensure that the contents of all immutable memory objects are always prepopulated in the program's memory image M.
⑦ Every address $\sigma \in \text{dom}(M)$ must be also mapped in A, with $0 \leq M(\sigma) \leq 255$.
⑧ Every address $\sigma \in [\sigma_f \ldots \sigma_i]$ must be identified by A as a modifiable.
⑨ Finally, for each envelope element $(\sigma_k, \phi_k, \bar{\mu}_k) \in \bar{\xi}$, every address σ in the set $\{\sigma_k \ldots \sigma_k + \mathcal{S}(\phi_k) - 1\}$ must always appear in the domain of the address space A and, if $A(\sigma)$ is false for at least one such value of σ, then the set of memory access attributes $\bar{\mu}_k$ associated with σ must contain the access attribute "C".

Intuitively, these nine requirements are intended to provide the compiler with sufficient information about the behaviour of well-formed programs to facilitate an efficient transition between their Etude and native MMIX representations, as required for establishment of the linear correctness property in Section 6.4. The reader should observe that,

under the above conditions, the three environment components $\bar{\xi}$, $\bar{\xi}_i$ $\bar{\psi}$ echo all of the useful information stored in M and I. In fact, as we shall soon discover, well-formed MMIX programs may, under certain conditions, discard some of the less interesting data from $\bar{\xi}$ and $\bar{\psi}$, in order to admit various useful optimising transformations of many common Etude term forms. However, any program that invokes such behaviour is inherently unportable and must be always scrutinised in the context of the specific MMIX implementation of Etude. Formally, the well-formedness property "WF:: *o-env* $\to \star$" is defined for MMIX object environments as follows:

$$\begin{aligned}
&\text{data } \mathrm{WF}[\![\cdot]\!] :: \textit{o-env} \to \star \\
&\quad \text{where } \mathrm{WF}_\Delta :: \forall\, M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f \Rightarrow \\
&\qquad\qquad \mathrm{WF}(\bar{\xi}) \to \\
&\qquad\qquad [\![\bar{\xi}_i \subseteq \bar{\xi}]\!] \to \\
&\qquad\qquad [\![\bigcup[\bar{\xi}_k \oplus \sigma_k \mid (\sigma_k, \bar{\xi}_k) \leftarrow \bar{\psi}] \subseteq \bar{\xi}]\!] \to \\
&\qquad\qquad [\![\bigcup[\{\sigma_k \mathinner{..} \mathcal{S}(\bar{\xi}_k) - 1\} \mid (\sigma_k, \bar{\xi}_k) \leftarrow \bar{\psi}] \subseteq \{\sigma_f \mathinner{..} \sigma_i\}]\!] \to \\
&\qquad\qquad [\![\sigma_f \le \sigma_c \le \sigma_i \wedge \sigma_c \bmod 8 = 0]\!] \to \\
&\qquad\qquad (\forall\, \sigma \Rightarrow [\![\sigma \in \mathrm{dom}(A)]\!] \to [\![0 \le \sigma < 2^{63} \wedge (A(\sigma) \vee \sigma \in \mathrm{dom}(M))]\!]) \to \\
&\qquad\qquad (\forall\, \sigma \Rightarrow [\![\sigma \in \mathrm{dom}(M)]\!] \to [\![\sigma \in \mathrm{dom}(A) \wedge 0 \le M(\sigma) \le 255]\!]) \to \\
&\qquad\qquad (\forall\, \sigma \Rightarrow [\![\sigma_f \le \sigma \le \sigma_i]\!] \to [\![A(\sigma)]\!]) \to \\
&\qquad\qquad (\forall\, \sigma_k, \phi_k, \bar{\mu}_k \Rightarrow \\
&\qquad\qquad\quad [\![(\sigma_k, \phi_k, \bar{\mu}_k) \in \bar{\xi} \wedge \sigma_k \le \sigma < \sigma_k + \mathcal{S}(\phi_k)]\!] \to \\
&\qquad\qquad\quad (\sigma \in \mathrm{dom}(A) \wedge (A(\sigma) \vee [\![\mathsf{c}]\!] \in \bar{\mu}_k))) \to \\
&\qquad\qquad \mathrm{WF}[\![M, A, \bar{\xi}, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f]\!]
\end{aligned}$$

THEOREM 6-12: *(Validity of object environment envelopes and stacks)* Every well-formed object environment Δ must always specify a valid envelope $\bar{\xi}(\Delta)$ that includes at least all of the envelope elements from $\bar{\xi}_i(\Delta)$, as well as any addresses that are mentioned by the individual stack frames of $\bar{\psi}(\Delta)$:

$$\begin{aligned}
&\mathrm{ENV_E} :: \forall\, \Delta \Rightarrow \mathrm{WF}(\Delta) \to \mathrm{WF}[\![\bar{\xi}(\Delta)]\!] \\
&\mathrm{ENV_I} :: \forall\, \Delta \Rightarrow \mathrm{WF}(\Delta) \to [\![\bar{\xi}_i(\Delta) \subseteq \bar{\xi}(\Delta)]\!] \\
&\mathrm{ENV_S} :: \forall\, \Delta \Rightarrow \mathrm{WF}(\Delta) \to [\![\bigcup[\bar{\xi}_k \oplus \sigma_k \mid (\sigma_k, \bar{\xi}_k) \leftarrow \bar{\psi}(\Delta)] \subseteq \bar{\xi}(\Delta)]\!]
\end{aligned}$$

PROOF: Directly, by the earlier definition of WF:: *o-env* $\to \star$. □

The environment extension operation $\Delta/\bar{\xi}$ can be modelled quite easily on MMIX, since it never affects the actual configuration of the address space A or the associated memory image M. Instead, the current stack pointer σ_c is assigned directly the required value of $\sigma_c(\Delta \triangleright \bar{\xi})$ and, in order to satisfy the requirements from Section 4.5, $\bar{\xi}$ is also pushed onto the stack $\bar{\psi}(\Delta)$ and incorporated into the address space envelopes $\bar{\xi}(\Delta)$ and $\bar{\xi}_i(\Delta)$ as follows:

$$\begin{aligned}
&[\![\cdot]\!]/[\![\cdot]\!] :: \textit{o-env} \to \textit{envelope} \to \textit{o-env} \\
&[\![M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f]\!]/[\![\bar{\xi}']\!] \\
&\qquad = [\![M, A, \bar{\xi} \cup (\bar{\xi}' \oplus \sigma_c{}'), \bar{\xi}_i \cup (\bar{\xi}' \oplus \sigma_c{}'), (\sigma_c{}', \bar{\xi}') +\!\!+ \bar{\psi}, \sigma_i, \sigma_c{}', \sigma_f]\!] \\
&\quad \text{where } \sigma_c{}' = \sigma_c[\![M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f \triangleright \bar{\xi}']\!]
\end{aligned}$$

THEOREM 6-13: *(Algebraic semantics of object environment extensions)* The environment extension operation $\Delta/\bar{\xi}$ preserves the semantics of Δ as follows:

$$\begin{aligned}
&\text{EXT}_\text{S} :: \forall\, \Delta,\, \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow \text{WF}(\Delta/\bar{\xi}) \rightarrow [\![\bar{\psi}(\Delta/\bar{\xi}) = (\sigma_c(\Delta \triangleright \bar{\xi}),\, \bar{\xi}) +\!\!+ \bar{\psi}(\Delta)]\!] \\
&\text{EXT}_\text{E} :: \forall\, \Delta,\, \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow \text{WF}(\Delta/\bar{\xi}) \rightarrow [\![\bar{\xi}\,(\Delta/\bar{\xi}) = \bar{\xi}\,(\Delta) \cup (\bar{\xi} \oplus \sigma_c(\Delta \triangleright \bar{\xi}))]\!] \\
&\text{EXT}_\text{I} :: \forall\, \Delta,\, \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow \text{WF}(\Delta/\bar{\xi}) \rightarrow [\![\bar{\xi}_i(\Delta/\bar{\xi}) = \bar{\xi}_i(\Delta) \cup (\bar{\xi} \oplus \sigma_c(\Delta \triangleright \bar{\xi}))]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{EXT}_\text{A} :: \forall\, \Delta,\, \bar{\xi},\, \sigma_k,\, \phi_k,\, \bar{\mu}_k \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \text{WF}(\Delta/\bar{\xi}) \rightarrow [\![(\sigma_k,\, \phi_k,\, \bar{\mu}_k) \in \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad [\![\bar{\alpha}(\Delta/\bar{\xi} \triangleright \sigma_k,\, \phi_k)]\!] \equiv [\![\bar{\alpha}(\Delta \triangleright \sigma_k,\, \phi_k)]\!]
\end{aligned}$$

PROOF: The first three theorems are justified immediately from our Haskell definition of the environment extension operation, whereby all of the respective environment components directly assume the precise forms required above. On the other hand, the fourth theorem EXT_A follows from the fact that the MMIX definition of the environment inspection operation $\bar{\alpha}(\Delta \triangleright \sigma,\, \phi)$ relies only on the memory image $M(\Delta)$, which is clearly preserved under all of its well-formed extensions. □

In contrast, the dual contraction operation $\Delta\backslash\bar{\xi}$ may involve some additional processing whenever the precise envelope $\bar{\xi}$ does not appear at the top of the environment's data stack $\bar{\psi}(\Delta)$. On MMIX, such operations are still deemed valid, provided that the resulting stack pointer $\sigma_c + \lceil \mathcal{S}(\bar{\xi}')/8 \rceil \times 8$ remains within its allotted region $[\sigma_f \ldots \sigma_i]$, but, for the remainder of the program's execution, the $\bar{\xi}(\Delta)$, $\bar{\xi}_i(\Delta)$ and $\bar{\psi}(\Delta)$ components are all reset to $\varnothing$, so that no guarantees can be ever made about such programs' semantics within the portable fragment of the Etude language described in Chapter 4:

$$[\![\cdot]\!]\backslash[\![\cdot]\!] :: \textit{o-env} \rightarrow \textit{envelope} \rightarrow \textit{o-env}$$

$$\begin{aligned}
&[\![M,\, A,\, \bar{\xi},\, \bar{\xi}_i,\, \bar{\psi},\, \sigma_i,\, \sigma_c,\, \sigma_f]\!]\backslash[\![\bar{\xi}']\!] \\
&\quad \mid \bar{\psi} \neq \varnothing \wedge \text{head}(\bar{\psi}) = (\sigma_c{}',\, \bar{\xi}') = [\![M,\, A,\, \bar{\xi}\backslash(\bar{\xi}' \oplus \sigma_c{}'),\, \bar{\xi}_i\backslash(\bar{\xi}' \oplus \sigma_c{}'),\, \text{tail}(\bar{\psi}),\, \sigma_i,\, \sigma_c{}',\, \sigma_f]\!] \\
&\quad \mid \text{otherwise} \qquad\qquad\qquad\qquad\quad = [\![M,\, A,\, \varnothing,\, \varnothing,\, \varnothing,\, \sigma_i,\, \sigma_c{}',\, \sigma_f]\!] \\
&\text{where } \sigma_c{}' = \sigma_c + \lceil \mathcal{S}(\bar{\xi}')/8 \rceil \times 8
\end{aligned}$$

THEOREM 6-14: *(Algebraic semantics of object environment contractions)* The environment contraction operation $\Delta\backslash\bar{\xi}$ preserves the semantics of Δ as follows:

$$\begin{aligned}
&\text{CON}_\Delta :: \forall\, \Delta,\, \sigma_c,\, \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\, \bar{\xi}))]\!] \rightarrow \text{WF}[\![\Delta\backslash\bar{\xi}]\!] \\
&\text{CON}_\text{S} :: \forall\, \Delta,\, \sigma_c,\, \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\, \bar{\xi}))]\!] \rightarrow [\![\bar{\psi}(\Delta\backslash\bar{\xi}) = \text{tail}(\bar{\psi}(\Delta))]\!] \\
&\text{CON}_\text{E} :: \forall\, \Delta,\, \sigma_c,\, \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\, \bar{\xi}))]\!] \rightarrow [\![\bar{\xi}\,(\Delta\backslash\bar{\xi}) = \bar{\xi}\,(\Delta)\backslash(\bar{\xi} \oplus \sigma_c)]\!] \\
&\text{CON}_\text{I} :: \forall\, \Delta,\, \sigma_c,\, \bar{\xi} \Rightarrow \text{WF}(\Delta) \rightarrow [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\, \bar{\xi}))]\!] \rightarrow [\![\bar{\xi}_i(\Delta\backslash\bar{\xi}) = \bar{\xi}_i(\Delta)\backslash(\bar{\xi} \oplus \sigma_c)]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{CON}_\text{A} :: \forall\, \Delta,\, \sigma_c,\, \bar{\xi},\, \sigma_k,\, \phi_k,\, \bar{\mu}_k \Rightarrow \\
&\qquad \text{WF}(\Delta) \rightarrow \\
&\qquad [\![\text{head}(\bar{\psi}(\Delta) = (\sigma_c,\, \bar{\xi}))]\!] \rightarrow \\
&\qquad [\![(\sigma_k,\, \phi_k,\, \bar{\mu}_k) \in \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad [\![\sigma_k + \mathcal{S}(\phi_k) \leq \sigma_c \vee \sigma_k \geq \sigma_c + \mathcal{S}(\bar{\xi})]\!] \rightarrow \\
&\qquad [\![\bar{\alpha}(\Delta\backslash\bar{\xi} \triangleright \sigma_k,\, \phi_k)]\!] \equiv [\![\bar{\alpha}(\Delta \triangleright \sigma_k,\, \phi_k)]\!]
\end{aligned}$$

PROOF: Once again, all five theorems follow directly from the MMIX implementation of environment contractions, observing that only the portable case of that definition is ever applicable under the supplied precondition "head($\bar{\psi}(\Delta) = (\sigma_c, \bar{\xi})$)", that every subset of a well-formed Etude envelope is also well-formed itself and that the actual memory image M of the program is never affected by the operation. □

Finally, on MMIX, an update to the content of a given memory object (σ, ϕ) with the new value of α is modelled by discarding any alignment bits of σ, then setting each byte $\sigma + k \in [\sigma' \,..\, \sigma' + \mathcal{S}(\phi) - 1]$ in the program's memory image M to the corresponding 8 bits $x[\omega k \,..\, \omega(k+1) - 1]$ of α's encoded value $\mathrm{dec}_{[\![\mathrm{N.64}]\!]}(\mathcal{E}(\alpha))$ as follows:

$$[\![\cdot]\!]/[\![\cdot]\!] :: (\mathrm{ord}\ \nu) \Rightarrow \textit{o-env} \rightarrow (\textit{integer}, \textit{format}, \textit{atom}_\nu) \rightarrow \textit{o-env}$$

$$\begin{aligned}
&[\![M, A, \bar{\xi}, \bar{\xi}_i, \bar{\psi}, \sigma_i, \sigma_c, \sigma_f]\!]/[\![\sigma, \phi, \alpha]\!] \mid \textstyle\bigwedge [A(\sigma' + k) \mid k \leftarrow [0 \,..\, \mathcal{S}(\phi) - 1]] \\
&\qquad = [\![M', A, \bar{\xi}, \bar{\xi}_i', \bar{\psi}, \sigma_i, \sigma_c, \sigma_f]\!] \\
&\text{where } \sigma' = \sigma - \sigma \bmod \mathcal{S}(\phi) \\
&\qquad x = \mathrm{dec}_{[\![\mathrm{N.64}]\!]}(\mathcal{E}(\alpha)) \\
&\qquad M' = M/\{(\sigma' + \mathcal{S}(\phi) - 1 - k){:}x[\omega k \,..\, \omega(k+1) - 1] \mid k \leftarrow [0 \,..\, \mathcal{S}(\phi) - 1]\} \\
&\qquad \bar{\xi}_i' = \bar{\xi}_i \backslash (\sigma', \sigma' + \mathcal{S}(\phi))
\end{aligned}$$

THEOREM 6-15: *(Validity of object environment updates)* Well-formedness of every environment Δ is preserved under a valid environment update operation $\Delta/(\sigma, \phi, \alpha)$:

$$\begin{aligned}
&\mathrm{SET}_\Delta :: \forall\, \Delta, \sigma, \phi, x, \bar{\mu} \Rightarrow \\
&\qquad \mathrm{WF}(\Delta) \rightarrow \mathrm{WF}[\![\#x_\phi]\!] \rightarrow [\![(\sigma, \phi, \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad \mathrm{WF}[\![\Delta/(\sigma, \phi, [\![\#x_\phi]\!])]\!]
\end{aligned}$$

PROOF: Recalling the earlier definition of the environment well-formedness property "WF:: *o-env* $\rightarrow \star$" and, further, observing that the above definition of environment updates only ever modifies the two environment components M and $\bar{\xi}_i$, we are only required to establish three of the nine preconditions of WF.

In particular, we must first demonstrate that, in the updated environment, $\bar{\xi}_i'$ always represents a subset of $\bar{\xi}$. Given the definitions of the finite map extension and contraction operators from Appendix A, we observe that $\mathrm{dom}(M_1) \subseteq \mathrm{dom}(M_1/M_2)$ for all well-defined finite maps M_1 and M_2. Accordingly, $\bar{\xi}_i' = \bar{\xi}_i \backslash (\sigma', \sigma' + \mathcal{S}(\phi)) \subseteq \bar{\xi}_i \subseteq \bar{\xi}$.

For all $\sigma \in \mathrm{dom}(A)$, we also require that $0 \leq \sigma < 2^{63} \wedge (A(\sigma) \vee \sigma \in \mathrm{dom}(M))$. Observing that, for every valid map M and set S, the relation $\mathrm{dom}(M \backslash S) \subseteq \mathrm{dom}(M)$ is always satisfied by the finite map difference operator from Appendix A, we have $\mathrm{dom}(M') \subseteq \mathrm{dom}(M/\{\sigma_k'{:}n_k'\}$, so that the second of the required preconditions is likewise guaranteed to hold in the resulting environment.

Finally, well-formedness of the resulting object environment also requires that, for every $\sigma \in \mathrm{dom}(M)$, $0 \leq M(\sigma) \leq 255$. If the address $\sigma + k$ falls outside of the region affected by the update, then $M'(\sigma + k) = M(\sigma + k)$, so the property holds by definition. Otherwise, we have $M'(\sigma + k) = x[\omega k \,..\, \omega(k+1) - 1] \leq 2^{\omega(k+1) - 1 - \omega k + 1}$, which, as required, is always equal to 2^ω or 2^8, since the byte offset k cannot be greater than 7 in magnitude. □

THEOREM 6-16: *(Object environment binding after an update)*

$$\begin{aligned}&\text{SET}_\alpha :: \forall\ \Delta,\ \sigma,\ \phi,\ \alpha,\ \bar{\mu} \Rightarrow\\ &\quad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma,\ \phi,\ \alpha)]\!] \rightarrow [\![(\sigma,\ \phi,\ \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow\\ &\quad [\![\bar{\alpha}(\Delta/(\sigma,\ \phi,\ \alpha) \triangleright \sigma,\ \phi)]\!] \equiv [\![\alpha]\!]\end{aligned}$$

PROOF: Given the earlier definition of the atom equivalence relation in Section 6.3.1, it will suffice to show that the above theorem holds for the four distinguished natural formats N.8, N.16, N.32 or N.64, since every other well-formed Etude atom is always equivalent under reduction by $\mathcal{E}$ to some constant "$\#n_{\text{N}.\varepsilon}$", in which the integer n falls within the range of values $0 \leq n < 2^{\mathcal{S}(\phi)\omega}$ that are representable under the format ϕ. In order to establish theorem SET_α, it is therefore sufficient to show that every such value of n is always preserved under its insertion and a subsequent extraction from the same address. In particular, let m be the value stored at some address σ in a memory image M, which has been derived by an update of that address with the value of n. Then:

$$\begin{aligned}m &= \textstyle\sum[2^{\omega k} M(\sigma' + \mathcal{S}(\phi) - 1 - k) \mid k \leftarrow [0 \ldots \mathcal{S}(\phi) - 1]]\\ &= \textstyle\sum[2^{\omega k} n[\omega k \ldots \omega(k+1) - 1] \mid k \leftarrow [0 \ldots \mathcal{S}(\phi) - 1]]\\ &= \textstyle\sum[2^{\omega k} \sum[2^{j - \omega k}(\lfloor n/2^j \rfloor \bmod 2) \mid j \leftarrow [\omega k \ldots \omega(k+1) - 1]] \mid k \leftarrow [0 \ldots \mathcal{S}(\phi) - 1]]\\ &= \textstyle\sum[\sum[2^{\omega k} 2^{j - \omega k}(\lfloor n/2^j \rfloor \bmod 2) \mid j \leftarrow [\omega k \ldots \omega(k+1) - 1]] \mid k \leftarrow [0 \ldots \mathcal{S}(\phi) - 1]]\\ &= \textstyle\sum[2^{\omega k} 2^{j - \omega k}(\lfloor n/2^j \rfloor \bmod 2) \mid k \leftarrow [0 \ldots \mathcal{S}(\phi) - 1],\ j \leftarrow [\omega k \ldots \omega(k+1) - 1]]\\ &= \textstyle\sum[2^j(\lfloor n/2^j \rfloor \bmod 2) \mid k \leftarrow [0 \ldots \mathcal{S}(\phi) - 1],\ j \leftarrow [\omega k \ldots \omega(k+1) - 1]]\\ &= \textstyle\sum[2^j(\lfloor n/2^j \rfloor \bmod 2) \mid j \leftarrow [0 \ldots \omega \times \mathcal{S}(\phi) - 1]]\\ &= n\end{aligned}$$

as required. The first three steps in the above reduction sequence are justified, respectively, by the earlier definition of the environment inspection function $\bar{\alpha}$, the values of $M(\sigma' + \mathcal{S}(\phi) - 1 - k)$ inserted into the address space in accordance with the definition of environment update operations and, finally, by the definition of bit extraction from Appendix A. The remaining five steps, on the other hand, represent simple arithmetic transformations justified directly by the axioms of the Haskell "*integer*" type. □

THEOREM 6-17: *(Object environment data after an update)*

$$\begin{aligned}&\text{SET}_{\bar{\alpha}} :: \forall\ \Delta,\ \sigma,\ \phi,\ \alpha,\ \bar{\xi},\ \sigma_k,\ \phi_k,\ \bar{\mu}_k \Rightarrow\\ &\quad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma,\ \phi,\ \alpha)]\!] \rightarrow\\ &\quad [\![(\sigma,\ \phi,\ \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow\\ &\quad [\![(\sigma_k,\ \phi_k,\ \bar{\mu}_k) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow\\ &\quad [\![\sigma_k + \mathcal{S}(\phi_k) \leq \sigma \vee \sigma_k \geq \sigma + \mathcal{S}(\phi)]\!] \rightarrow\\ &\quad [\![\bar{\alpha}(\Delta/(\sigma,\ \phi,\ \alpha) \triangleright \sigma_k,\ \phi_k)]\!] \equiv [\![\bar{\alpha}(\Delta \triangleright \sigma_k,\ \phi_k)]\!]\end{aligned}$$

PROOF: Trivial, observing that none of the relevant address values are ever affected in M by the above Haskell definition of object environment updates. □

THEOREM 6-18: *(Compatibility of object environment updates)*

$$\text{SET}_\equiv :: \forall\ \Delta,\ \sigma,\ \phi,\ \alpha_1,\ \alpha_2 \Rightarrow [\![\alpha_1]\!] \equiv [\![\alpha_2]\!] \rightarrow [\![\Delta/(\sigma,\ \phi,\ \alpha_1) = \Delta/(\sigma,\ \phi,\ \alpha_2)]\!]$$

PROOF: In the definition of environment extension $\Delta/(\sigma,\ \phi,\ \alpha_i)$, the value n_i inserted into the memory image M at each affected address $\sigma' + k$ is defined as $\text{dec}_{[\![\text{N}.64]\!]}(\mathcal{E}(\alpha_i))$,

so that, given the equivalence of α_1 and α_2 or $\mathcal{E}(\alpha_1) = \mathcal{E}(\alpha_2)$, we have $x_1 = x_2$ and $\Delta/(\sigma, \phi, \alpha_1) = \Delta/(\sigma, \phi, \alpha_2)$ as required. □

THEOREM 6-19: *(Algebraic invariants of object environment updates)* Every well-formed environment update operation preserves the following properties of the program's address space:

$$\begin{aligned}
&\text{SET}_\text{I} :: \forall\, \Delta, \sigma, \phi, \alpha, \bar{\mu} \Rightarrow \\
&\quad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma, \phi, \alpha)]\!] \rightarrow [\![(\sigma, \phi, \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\quad [\![\bar{\xi}_i(\Delta/(\sigma, \phi, \alpha) \triangleright \sigma, \phi) = \bar{\xi}_i(\Delta) \backslash (\sigma, \sigma + \mathcal{S}(\phi))]\!] \\
&\text{SET}_\text{E} :: \forall\, \Delta, \sigma, \phi, \alpha, \bar{\mu} \Rightarrow \\
&\quad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma, \phi, \alpha)]\!] \rightarrow [\![(\sigma, \phi, \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\quad [\![\bar{\xi}(\Delta/(\sigma, \phi, \alpha)) = \bar{\xi}(\Delta)]\!] \\
&\text{SET}_\text{S} :: \forall\, \Delta, \sigma, \phi, \alpha, \bar{\mu} \Rightarrow \\
&\quad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma, \phi, \alpha)]\!] \rightarrow [\![(\sigma, \phi, \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\quad [\![\bar{\psi}(\Delta/(\sigma, \phi, \alpha)) = \bar{\psi}(\Delta)]\!] \\
&\text{SET}_\text{X} :: \forall\, \Delta, \sigma, \phi, \alpha, \bar{\mu}, \bar{\xi} \Rightarrow \\
&\quad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma, \phi, \alpha)]\!] \rightarrow [\![(\sigma, \phi, \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow \text{WF}[\![\Delta/\bar{\xi}]\!] \rightarrow \\
&\quad \text{WF}[\![(\Delta/(\sigma, \phi, \alpha))/\bar{\xi}]\!] \\
&\text{SET}_\text{N} :: \forall\, \Delta, \sigma, \phi, \alpha, \bar{\mu}, \bar{\xi} \Rightarrow \\
&\quad \text{WF}(\Delta) \rightarrow \text{WF}[\![\Delta/(\sigma, \phi, \alpha)]\!] \rightarrow [\![(\sigma, \phi, \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow \text{WF}[\![\Delta/\bar{\xi}]\!] \rightarrow \\
&\quad [\![\sigma_c(\Delta/(\sigma, \phi, \alpha) \triangleright \bar{\xi}) = \sigma_c(\Delta \triangleright \bar{\xi})]\!]
\end{aligned}$$

PROOF: Our Haskell implementation of object environment updates always returns the original value of each environment component scrutinised above, so that all five of these theorems hold directly by the definition of $\Delta/(\sigma, \phi, \alpha_i)$. □

6.3.3 Terms

The final piece of Etude's operational model that must be specialised for the MMIX architecture is the precise semantics of all monadic term constructs provided by the language. In a nutshell, the meaning of Etude terms is formalised similarly to atoms, using a function $\mathcal{E}_\tau$ akin to the earlier atom evaluation algorithm $\mathcal{E}$. In particular, in the context of a given evaluation environment (Λ, Δ), $\mathcal{E}_\tau(\Lambda, \Delta \triangleright \tau)$ reduces the specified term τ into a list of result atoms $\bar{\alpha}$, an updated object environment Δ' and a list of zero or more abstract entities known as *actions*, which, intuitively, describe the exact sequence of system calls invoked by the program during evaluation of τ. Formally:

$$\mathcal{E}_\tau[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow (\textit{f-env}_\nu, \textit{o-env} \triangleright \textit{term}_\nu) \rightarrow (\textit{o-env}, \textit{actions}_\nu, \textit{atoms}_\nu)$$

Each of these actions is depicted simply by a pair $(\pi, \bar{\alpha})$, in which π represents one of the operating system facilities described by the Haskell type "*prim*" and $\bar{\alpha}$ specifies the precise list of input arguments for the underlying system call. Formally:

$$\begin{aligned}
&\textit{action}_\nu : && (\textit{prim}, \textit{atoms}_\nu) \\
&\textit{actions}_\nu : && [\textit{action}_\nu]
\end{aligned}$$

The evaluation results obtained by $\mathcal{E}_\tau$ are used directly for determination of the actual term equivalence and well-formedness properties utilised in the generic algebraic description of Etude term semantics from Chapter 4. Specifically, these term properties capture precisely the notions of definitional equality and well-formedness of the respective evaluation results obtained from the entity's syntax and context by $\mathcal{E}_\tau$, as expressed by the following pair of Haskell property definitions:

$$\text{data } [\![\cdot]\!] \equiv [\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow (\textit{f-env}_\nu, \textit{o-env} \triangleright \textit{term}_\nu) \rightarrow (\textit{f-env}_\nu, \textit{o-env} \triangleright \textit{term}_\nu) \rightarrow \star$$

$$\begin{aligned}&\text{where } \text{EQV}_\tau :: \forall\ \Lambda_1, \Delta_1, \alpha_1, \Lambda_2, \Delta_2, \alpha_2 \Rightarrow \\ &\qquad [\![\mathcal{E}_\tau(\Lambda_1, \Delta_1 \triangleright \alpha_1) = \mathcal{E}_\tau(\Lambda_2, \Delta_2 \triangleright \alpha_2)]\!] \rightarrow \\ &\qquad [\![\Lambda_1, \Delta_1 \triangleright \alpha_1]\!] \equiv [\![\Lambda_2, \Delta_2 \triangleright \alpha_2]\!]\end{aligned}$$

$$\text{data } \text{WF}[\![\cdot]\!] :: (\text{ord } \nu) \Rightarrow (\textit{f-env}_\nu, \textit{o-env} \triangleright \textit{term}_\nu) \rightarrow \star$$

$$\text{where } \text{WF}_\tau :: \forall\ \Lambda, \Delta, \alpha \Rightarrow \text{WF}[\![\mathcal{E}_\tau(\Lambda, \Delta \triangleright \alpha)]\!] \rightarrow \text{WF}[\![\Lambda, \Delta \triangleright \alpha]\!]$$

THEOREM 6-20: *(Standard equivalence properties of Etude terms)* According to the above definition, "$\equiv$" constitutes a reflexive, symmetric and transitive relation over all Etude term forms:

$$\begin{aligned}&\text{REFL}_\tau :: \forall\ \Lambda, \Delta, \tau \Rightarrow \\ &\qquad [\![\Lambda,\ \Delta\ \triangleright \tau\ ]\!] \equiv [\![\Lambda,\ \Delta\ \triangleright \tau]\!]\end{aligned}$$

$$\begin{aligned}&\text{SYMM}_\tau :: \forall\ \Lambda_1, \Delta_1, \tau_1, \Lambda_2, \Delta_2, \tau_2 \Rightarrow \\ &\qquad [\![\Lambda_1, \Delta_1 \triangleright \tau_1]\!] \equiv [\![\Lambda_2, \Delta_2 \triangleright \tau_2]\!] \rightarrow \\ &\qquad [\![\Lambda_2, \Delta_2 \triangleright \tau_2]\!] \equiv [\![\Lambda_1, \Delta_1 \triangleright \tau_1]\!]\end{aligned}$$

$$\begin{aligned}&\text{TRANS}_\tau :: \forall\ \Lambda_1, \Delta_1, \tau_1, \Lambda_2, \Delta_2, \tau_2, \Lambda_3, \Delta_3, \tau_3 \Rightarrow \\ &\qquad [\![\Lambda_1, \Delta_1 \triangleright \tau_1]\!] \equiv [\![\Lambda_2, \Delta_2 \triangleright \tau_2]\!] \rightarrow \\ &\qquad [\![\Lambda_2, \Delta_2 \triangleright \tau_2]\!] \equiv [\![\Lambda_3, \Delta_3 \triangleright \tau_3]\!] \rightarrow \\ &\qquad [\![\Lambda_1, \Delta_1 \triangleright \tau_1]\!] \equiv [\![\Lambda_3, \Delta_3 \triangleright \tau_3]\!]\end{aligned}$$

PROOF: Trivial, from the corresponding properties of definitional equality "=". □

In particular, simple terms of the form "RET $(\bar{\alpha})$" never deliver any actions or affect the program's memory image. Instead, the supplied list of atoms $\bar{\alpha}$ is reduced by $\mathcal{E}$ and returned directly as part of the term's evaluated result:

$$\mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \text{RET } (\bar{\alpha})]\!] = (\Delta, \varnothing, \mathcal{E}(\bar{\alpha}))$$

THEOREM 6-21: *(Properties of "RET" terms)* All Etude "RET" term forms satisfy the following properties under the MMIX architecture:

$$\begin{aligned}&\text{WF}_{\text{RET}} :: \forall\ \Lambda, \Delta, \bar{\alpha} \Rightarrow \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}(\bar{\alpha}) \rightarrow \text{WF}[\![\Lambda, \Delta \triangleright \text{RET } (\bar{\alpha})]\!] \\ &\text{EQV}_{\text{RET}} :: \forall\ \Lambda, \Delta, \bar{\alpha}_1, \bar{\alpha}_2 \Rightarrow (\bar{\alpha}_1 \equiv \bar{\alpha}_2) \rightarrow [\![\Lambda, \Delta \triangleright \text{RET } (\bar{\alpha}_1)]\!] \equiv [\![\Lambda, \Delta \triangleright \text{RET } (\bar{\alpha}_2)]\!]\end{aligned}$$

PROOF: From the corresponding properties of atom well-formedness $\text{WF}_\equiv$ and equivalence EQV_α. □

In every application term of the form "$\alpha(\bar{\alpha})$", α must depict a well-formed atom that, in Λ, is bound to some Etude function $\lambda\bar{\nu}.\tau$, in which the parameter list $\bar{\nu}$ has the same length as the number of argument atoms in $\bar{\alpha}$. The construction simply evaluates

the body term τ, after substituting the normal form of each $\alpha_k \in \mathcal{E}(\bar{\alpha})$ for any free occurrences of the corresponding variable $v_k \in \bar{v}$ in τ. Formally:

$$\mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \alpha(\bar{\alpha})]\!] \mid (\text{length}(\bar{\alpha}) = \text{length}(\bar{v})) = \mathcal{E}_\tau(\Lambda, \Delta \triangleright \tau/(\bar{v}|\mathcal{E}(\bar{\alpha})))$$
$$\text{where } [\![\lambda \bar{v}.\tau]\!] = \Lambda(\text{dec}_{\text{F.64}}(\mathcal{E}(\alpha)))$$

THEOREM 6-22: *(Properties of application terms)* Every Etude application term satisfies the following properties under the MMIX architecture:

$$\text{WF}_{\text{APP}} :: \forall\ \Lambda, \Delta, x, \bar{\alpha}, \bar{v}, \tau \Rightarrow$$
$$\text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{F.}\Phi}]\!] \rightarrow \text{WF}(\bar{\alpha}) \rightarrow$$
$$[\![\Lambda(x) = [\![\lambda \bar{v}.\tau]\!] \wedge \text{length}(\bar{v}) = \text{length}(\bar{\alpha})]\!] \rightarrow$$
$$\text{WF}[\![\Lambda, \Delta \triangleright \tau/(\bar{v}|\bar{\alpha})]\!] \rightarrow$$
$$\text{WF}[\![\Lambda, \Delta \triangleright \#x_{\text{F.}\Phi}(\bar{\alpha})]\!]$$

$$\text{EQV}_{\text{APP}} :: \forall\ \Lambda, \Delta, \alpha_1, \bar{\alpha}_1, \alpha_2, \bar{\alpha}_2 \Rightarrow$$
$$(\alpha_1 \equiv \alpha_2) \rightarrow (\bar{\alpha}_1 \equiv \bar{\alpha}_2) \rightarrow$$
$$[\![\Lambda, \Delta \triangleright \alpha_1(\bar{\alpha}_1)]\!] \equiv [\![\Lambda, \Delta \triangleright \alpha_2(\bar{\alpha}_2)]\!]$$

$$\text{EQV}_{\beta} :: \forall\ \Lambda, \Delta, x, \bar{\alpha}, \bar{v}, \tau \Rightarrow$$
$$\text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{F.}\Phi}]\!] \rightarrow \text{WF}(\bar{\alpha}) \rightarrow$$
$$[\![\Lambda(x) = [\![\lambda \bar{v}.\tau]\!] \wedge \text{length}(\bar{v}) = \text{length}(\bar{\alpha})]\!] \rightarrow$$
$$\text{WF}[\![\Lambda, \Delta \triangleright \tau/(\bar{v}|\bar{\alpha})]\!] \rightarrow$$
$$[\![\Lambda, \Delta \triangleright \#x_{\text{F.}\Phi}(\bar{\alpha})]\!] \equiv [\![\Lambda, \Delta \triangleright \tau/(\bar{v}|\bar{\alpha})]\!]$$

PROOF: Since, by definition, every application term evaluates to the appropriately substituted function body $\tau/(\bar{v}|\bar{\alpha})$, theorem WF_{APP} holds trivially by the virtue of its own precondition $\text{WF}[\![\Lambda, \Delta \triangleright \tau/(\bar{v}|\bar{\alpha})]\!]$. Further, EQV_{APP} can be established by observing that $\mathcal{E}_\tau$ always reduces all of the atomic values involved in the operation into their respective normal forms, which, by EQV_α, must be structurally identical if the theorem's two preconditions are to be satisfied. Accordingly, the entire theorem is justified by reflexivity of the term equivalence relation. Finally, the beta-equivalence theorem EQV_β is established directly by the definition $\mathcal{E}_\tau$, which prescribes precisely the required behaviour to all reductions of Etude application expressions. □

A complete operational description of Etude system call operations is, regrettably, impossible without a detailed scrutiny of the underlying operating system facilities. Fortunately, neither is it required for a successful establishment of the linear correctness property for our compiler, whereby it will suffice to represent meanings of all well-formed system primitives by the following unspecified Haskell function:

$$\mathcal{E}_\delta[\![\cdot]\!] :: (\text{ord } v) \Rightarrow (\textit{f-env}_v, \textit{o-env}, \textit{prim} \triangleright \textit{atoms}_v) \rightarrow (\textit{o-env}, \textit{atoms}_v)$$

Using this auxiliary definition, the semantic significance of each system call operation "$\pi(\bar{\alpha})$" can be represented by the object environment and result atoms derived by $\mathcal{E}_\delta$ from an application of π to the normal forms of the arguments $\bar{\alpha}$. In addition, the evaluation result always includes the invoked primitive in its action list as a pair of the form $(\pi, \mathcal{E}(\bar{\alpha}))$. Formally:

$$\mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \pi(\bar{\alpha})]\!] = (\Delta', [(\pi, \mathcal{E}(\bar{\alpha}))], \bar{\alpha}')$$
$$\text{where } (\Delta', \bar{\alpha}') = \mathcal{E}_\delta(\Lambda, \Delta, \pi \triangleright \mathcal{E}(\bar{\alpha}))$$

THEOREM 6-23: *(Properties of system calls)* All Etude system calls satisfy the following compatibility law under the MMIX architecture:

$$\text{EQV}_{\text{SYS}} :: \forall\ \Lambda, \Delta, \pi, \bar{\alpha}_1, \bar{\alpha}_2 \Rightarrow (\bar{\alpha}_1 \equiv \bar{\alpha}_2) \rightarrow [\![\Lambda, \Delta \triangleright \pi(\bar{\alpha}_1)]\!] \equiv [\![\Lambda, \Delta \triangleright \pi(\bar{\alpha}_2)]\!]$$

PROOF: By symmetry of the equivalence relation, after reduction of all argument atoms into their respective normal forms. □

Every conditional Etude expression of the form "IF α THEN τ_1 ELSE τ_2" evaluates to either τ_1 or τ_2, whenever the specified atom α has a non-zero or zero normal form, respectively. In Haskell:

$$\begin{array}{l|l} \mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \text{IF } \alpha \text{ THEN } \tau_1 \text{ ELSE } \tau_2]\!] & (\text{dec}_{\text{N.64}}(\mathcal{E}(\alpha)) \neq 0) = \mathcal{E}_\tau(\Lambda, \Delta \triangleright \tau_1) \\ & (\text{dec}_{\text{N.64}}(\mathcal{E}(\alpha)) = 0) = \mathcal{E}_\tau(\Lambda, \Delta \triangleright \tau_2) \end{array}$$

THEOREM 6-24: *(Properties of conditional terms)* All conditional terms satisfy the following properties under the MMIX architecture:

$$\begin{aligned} &\text{WF}_{\text{TT}} :: \forall\ \Lambda, \Delta, x, \tau_1, \tau_2 \Rightarrow \\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}(\tau_1) \rightarrow [\![x \neq 0]\!] \rightarrow \\ &\quad \text{WF}[\![\text{IF } \#x_{\text{Z.}\Phi} \text{ THEN } \tau_1 \text{ ELSE } \tau_2]\!] \\ &\text{WF}_{\text{FF}} :: \forall\ \Lambda, \Delta, x, \tau_1, \tau_2 \Rightarrow \\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}(\tau_2) \rightarrow [\![x = 0]\!] \rightarrow \\ &\quad \text{WF}[\![\text{IF } \#x_{\text{Z.}\Phi} \text{ THEN } \tau_1 \text{ ELSE } \tau_2]\!] \\ &\text{EQV}_{\text{IF}} :: \forall\ \Lambda_1, \Delta_1, \alpha_1, \tau_{11}, \tau_{21}, \Lambda_2, \Delta_2, \alpha_2, \tau_{12}, \tau_{22} \Rightarrow \\ &\quad (\alpha_1 \equiv \alpha_2) \rightarrow \\ &\quad (\Lambda_1, \Delta_1 \triangleright \tau_{11}) \equiv (\Lambda_2, \Delta_2 \triangleright \tau_{12}) \rightarrow (\Lambda_1, \Delta_1 \triangleright \tau_{12}) \equiv (\Lambda_2, \Delta_2 \triangleright \tau_{22}) \rightarrow \\ &\quad [\![\Lambda_1, \Delta_1 \triangleright \text{IF } \alpha_1 \text{ THEN } \tau_{11} \text{ ELSE } \tau_{21}]\!] \equiv [\![\Lambda_2, \Delta_2 \triangleright \text{IF } \alpha_2 \text{ THEN } \tau_{12} \text{ ELSE } \tau_{22}]\!] \\ &\text{EQV}_{\text{TT}} :: \forall\ \Lambda, \Delta, x, \tau_1, \tau_2 \Rightarrow \\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}(\tau_1) \rightarrow [\![x \neq 0]\!] \rightarrow \\ &\quad [\![\text{IF } \#x_{\text{Z.}\Phi} \text{ THEN } \tau_1 \text{ ELSE } \tau_2]\!] \equiv [\![\Lambda, \Delta \triangleright \tau_1]\!] \\ &\text{EQV}_{\text{FF}} :: \forall\ \Lambda, \Delta, x, \tau_1, \tau_2 \Rightarrow \\ &\quad \text{WF}(\Lambda) \rightarrow \text{WF}(\Delta) \rightarrow \text{WF}[\![\#x_{\text{Z.}\Phi}]\!] \rightarrow \text{WF}(\tau_2) \rightarrow [\![x = 0]\!] \rightarrow \\ &\quad [\![\text{IF } \#x_{\text{Z.}\Phi} \text{ THEN } \tau_1 \text{ ELSE } \tau_2]\!] \equiv [\![\Lambda, \Delta \triangleright \tau_2]\!] \end{aligned}$$

PROOF: Once again, all of the above theorems follow directly from the definition of $\mathcal{E}_\tau$ over Etude's conditional term forms, which always obtains the normal form of the supplied atomic operand α, thus justifying the compatibility law EQV_{IF}, and proceeds by reducing the conditional term into one of its two branches τ_1 or τ_2, precisely as required by the above theorems. □

Further, the five Etude term forms dedicated to the task of address space and memory image management correspond directly to the analogous object environment operations from Section 6.3.2, so that their respective reductions can be formalised in Haskell as follows:

$$\begin{aligned} \mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \text{NEW } (\bar{\xi})]\!] &= (\Delta/\bar{\xi}, \varnothing, [\![\#[\![\sigma_c(\Delta \triangleright \bar{\xi})]\!]_{\text{O.}\Phi}]\!]) \\ \mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \text{DEL } (\bar{\xi})]\!] &= (\Delta\backslash\bar{\xi}, \varnothing, \varnothing) \\ \mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \text{GET } [\alpha\ , \bar{\mu}]_\phi]\!] &= (\Delta, \varnothing, [\![\bar{\alpha}(\Delta \triangleright \text{dec}_{\text{O.64}}(\mathcal{E}(\alpha)), \phi)]\!]) \\ \mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \text{SET } [\alpha_1, \bar{\mu}]_\phi \text{ TO } \alpha_2]\!] &= (\Delta/(\text{dec}_{\text{O.64}}(\mathcal{E}(\alpha_1)), \phi, \mathcal{E}(\alpha_2)), \varnothing, \varnothing) \\ \mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \text{SET}_{\text{I}} [\alpha_1, \bar{\mu}]_\phi \text{ TO } \alpha_2]\!] &= (\Delta/(\text{dec}_{\text{O.64}}(\mathcal{E}(\alpha_1)), \phi, \mathcal{E}(\alpha_2)), \varnothing, \varnothing) \end{aligned}$$

The reader should observe that, on MMIX, both of the "SET" and "SET_I" term constructors assume an identical semantic interpretation. Our architecture shares this peculiarity with almost all other modern instruction set designs, which rarely implement the constant access attributes "C" for memory-resident objects that are allocated dynamically during a program's execution.

THEOREM 6-25: *(Properties of object environment management terms)* On MMIX, all Etude terms of the form "NEW $(\bar{\xi})$" and "DEL $(\bar{\xi})$" satisfy the following algebraic theorems from Section 4.6:

$$\begin{aligned}
&\text{WF}_{\text{NEW}} :: \forall\ \Lambda,\ \Delta,\ \bar{\xi} \Rightarrow \text{WF}(\Lambda) \rightarrow \text{WF}[\![\Delta/\bar{\xi}]\!] \rightarrow \text{WF}[\![\Lambda,\ \Delta \triangleright \text{NEW}\ (\bar{\xi})]\!]\\
&\text{WF}_{\text{DEL}}\ :: \forall\ \Lambda,\ \Delta,\ \bar{\xi} \Rightarrow \text{WF}(\Lambda) \rightarrow \text{WF}[\![\Delta\backslash\bar{\xi}]\!] \rightarrow \text{WF}[\![\Lambda,\ \Delta \triangleright \text{DEL}\ \ (\bar{\xi})]\!]\\
&\\
&\text{EQV}_{\text{NEW}} :: \forall\ \Lambda,\ \Delta,\ \bar{\xi} \Rightarrow\\
&\qquad \text{WF}(\Lambda) \rightarrow \text{WF}[\![\Delta/\bar{\xi}]\!] \rightarrow\\
&\qquad [\![\Lambda,\ \Delta \triangleright \text{NEW}\ (\bar{\xi})]\!] \equiv [\![\Lambda,\ \Delta/\bar{\xi} \triangleright \text{RET}\ (\#[\![\sigma_c(\Delta \triangleright \bar{\xi})]\!]_{\text{O}.\Phi})]\!]\\
&\\
&\text{EQV}_{\text{DEL}}\ :: \forall\ \Lambda,\ \Delta,\ \bar{\xi} \Rightarrow\\
&\qquad \text{WF}(\Lambda) \rightarrow \text{WF}[\![\Delta\backslash\bar{\xi}]\!] \rightarrow\\
&\qquad [\![\Lambda,\ \Delta \triangleright \text{DEL}\ \ (\bar{\xi})]\!] \equiv [\![\Lambda,\ \Delta\backslash\bar{\xi} \triangleright \text{RET}\ ()]\!]
\end{aligned}$$

PROOF: By definition of $\mathcal{E}_\tau$, also utilising the analogous results ENV_S and EXT_S from Section 6.3.2. □

THEOREM 6-26: *(Properties of object environment inspection terms)* Every object environment inspection term of the form "GET $[\alpha,\ \bar{\mu}]_\phi$" satisfies the following three laws on the MMIX architecture:

$$\begin{aligned}
&\text{WF}_{\text{GET}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \phi,\ \bar{\mu} \Rightarrow\\
&\qquad \text{WF}(\Lambda) \rightarrow \text{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\text{Z}.\Phi}]\!] \rightarrow \text{WF}[\![\bar{\alpha}(\Delta \triangleright x+n,\ \phi)]\!] \rightarrow\\
&\qquad [\![(x+n,\ \phi,\ \bar{\mu}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow\\
&\qquad \text{WF}[\![\Lambda,\ \Delta \triangleright \text{GET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\text{Z}.\Phi},\ \bar{\mu}]_\phi]\!]\\
&\\
&\text{EQV}_{\text{GET}} :: \forall\ \Lambda,\ \Delta,\ \alpha_1,\ \alpha_2,\ \bar{\mu},\ \phi \Rightarrow\\
&\qquad (\alpha_1 \equiv \alpha_2) \rightarrow\\
&\qquad [\![\Lambda,\ \Delta \triangleright \text{GET}\ [\alpha_1,\ \bar{\mu}]_\phi]\!] \equiv [\![\Lambda,\ \Delta \triangleright \text{GET}\ [\alpha_2,\ \bar{\mu}]_\phi]\!]\\
&\\
&\text{EQV}_{\text{GETO}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \phi,\ \bar{\mu} \Rightarrow\\
&\qquad \text{WF}(\Lambda) \rightarrow \text{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\text{Z}.\Phi}]\!] \rightarrow \text{WF}[\![\bar{\alpha}(\Delta \triangleright x+n,\ \phi)]\!] \rightarrow\\
&\qquad [\![(x+n,\ \phi,\ \bar{\mu}\backslash\{\text{V}\}) \in_\text{A} \bar{\xi}(\Delta)]\!] \rightarrow\\
&\qquad [\![\Lambda,\ \Delta \triangleright \text{GET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\text{Z}.\Phi},\ \bar{\mu}]_\phi]\!] \equiv [\![\Lambda,\ \Delta \triangleright \text{RET}\ [\![\bar{\alpha}(\Delta \triangleright x+n,\ \phi)]\!]]\!]
\end{aligned}$$

PROOF: First of all, we observe that, on MMIX, theorems WF_{GET} and EQV_{GETO} are not constrained to the specific structures of the address atom α that are described above. After reduction by $\mathcal{E}$ within the definition of $\mathcal{E}_\tau$, all such terms are rendered equivalent to their appropriate normal forms "$\#n_{\text{N}.64}$", so that theorem EQV_{GET} holds trivially by the earlier reflexivity of term equivalence relation REFL_τ. Similarly, theorems WF_{GET} and EQV_{GETO} are justified by the definitional equality of the atoms delivered by $\mathcal{E}_\tau$ from "GET" using the object inspection construct $\bar{\alpha}(\Delta \triangleright x+n,\ \phi)$, which both of these constraints require to be well-formed as part of their respective preconditions. □

THEOREM 6-27: *(Properties of object environment update terms)* Under the MMIX architecture, every object environment update term of the form "$\mathsf{SET}\ [\alpha_1,\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha_2$" and "$\mathsf{SET}_\mathsf{I}\ [\alpha_1,\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha_2$" satisfies the following constraint theorems described earlier in Section 4.6:

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{SET}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \phi,\ \bar{\mu},\ \alpha \Rightarrow \\
&\qquad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}(\Delta/(x+n,\ \phi,\ \alpha)) \rightarrow \mathrm{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}]\!] \rightarrow \\
&\qquad [\![(x+n,\ \phi,\ \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad \mathrm{WF}[\![\Lambda,\ \Delta \triangleright \mathsf{SET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{WF}_{\mathrm{INI}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \phi,\ \bar{\mu},\ \alpha \Rightarrow \\
&\qquad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}(\Delta/(x+n,\ \phi,\ \alpha)) \rightarrow \mathrm{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}]\!] \rightarrow \\
&\qquad [\![(x+n,\ \phi,\ \bar{\mu}) \in_{\mathrm{A}} \bar{\xi}_i(\Delta)]\!] \rightarrow \\
&\qquad \mathrm{WF}[\![\Lambda,\ \Delta \triangleright \mathsf{SET}_\mathsf{I}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{EQV}_{\mathrm{SET}} :: \forall\ \Lambda,\ \Delta,\ \alpha_{11},\ \alpha_{21},\ \alpha_{12},\ \alpha_{22},\ \bar{\mu},\ \phi \Rightarrow \\
&\qquad (\alpha_{11} \equiv \alpha_{12}) \rightarrow (\alpha_{21} \equiv \alpha_{22}) \rightarrow \\
&\qquad [\![\Lambda,\ \Delta \triangleright \mathsf{SET}\ [\alpha_{11},\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha_{21}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \mathsf{SET}\ [\alpha_{12},\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha_{22}]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{EQV}_{\mathrm{INI}} :: \forall\ \Lambda,\ \Delta,\ \alpha_{11},\ \alpha_{21},\ \alpha_{12},\ \alpha_{22},\ \bar{\mu},\ \phi \Rightarrow \\
&\qquad (\alpha_{11} \equiv \alpha_{12}) \rightarrow (\alpha_{21} \equiv \alpha_{22}) \rightarrow \\
&\qquad [\![\Lambda,\ \Delta \triangleright \mathsf{SET}_\mathsf{I}\ [\alpha_{11},\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha_{21}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \mathsf{SET}\ [\alpha_{12},\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha_{22}]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{EQV}_{\mathrm{SETO}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \bar{\mu},\ \phi,\ \alpha \Rightarrow \\
&\qquad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}(\Delta/(x+n,\ \phi,\ \alpha)) \rightarrow \mathrm{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}]\!] \rightarrow \\
&\qquad [\![(x+n,\ \phi,\ \varnothing) \in_{\mathrm{A}} \bar{\xi}(\Delta)]\!] \rightarrow \\
&\qquad [\![\Lambda,\ \Delta \triangleright \mathsf{SET}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha]\!] \equiv [\![\Lambda,\ \Delta/(x+n,\ \phi,\ \alpha) \triangleright \mathsf{RET}\ ()]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{EQV}_{\mathrm{INIO}} :: \forall\ \Lambda,\ \Delta,\ x,\ n,\ \bar{\mu},\ \phi,\ \alpha \Rightarrow \\
&\qquad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}(\Delta/(x+n,\ \phi,\ \alpha)) \rightarrow \mathrm{WF}[\![\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi}]\!] \rightarrow \\
&\qquad [\![(x+n,\ \phi,\ \{\mathsf{C}\}) \in_{\mathrm{A}} \bar{\xi}_i(\Delta)]\!] \rightarrow \\
&\qquad [\![\Lambda,\ \Delta \triangleright \mathsf{SET}_\mathsf{I}\ [\#x_{[\![O(\phi)]\!]} +_{[\![O(\phi)]\!]} \#n_{\mathrm{Z}.\Phi},\ \bar{\mu}]_\phi\ \mathsf{TO}\ \alpha]\!] \equiv [\![\Lambda,\ \Delta/(x+n,\ \phi,\ \alpha) \triangleright \mathsf{RET}\ ()]\!]
\end{aligned}$$

PROOF: A formal justification of these theorems follows a general pattern analogous to that outlined earlier for the object environment inspection operations, observing that, on MMIX, no specific restrictions are imposed on these constructs' access attribute sets $\bar{\mu}$ or the precise structure of their address terms α. □

Last but not least, an Etude operation of the form "$\mathsf{LET}\ \bar{\nu} = \tau_1;\ \tau_2$" is essentially reducible into its body term τ_2, except that any actions executed by τ_1 are prefixed to those produced by τ_2 itself and that all free occurrences of any variables from $\bar{\nu}$ in τ_2 are replaced with the respective atomic values delivered by τ_1:

$$\mathcal{E}_\tau[\![\Lambda,\ \Delta \triangleright \mathsf{LET}\ \bar{\nu} = \tau_1;\ \tau_2]\!] \mid (\mathrm{length}(\bar{\alpha}_1) = \mathrm{length}(\bar{\nu})) = (\Delta_2,\ \bar{a}_1 +\!\!+ \bar{a}_2,\ \bar{\alpha}_2)$$

$$\begin{aligned}
&\text{where } (\Delta_1,\ \bar{a}_1,\ \bar{\alpha}_1) = \mathcal{E}_\tau(\Lambda,\ \Delta\ \triangleright \tau_1) \\
&\qquad\quad (\Delta_2,\ \bar{a}_2,\ \bar{\alpha}_2) = \mathcal{E}_\tau(\Lambda,\ \Delta_1 \triangleright \tau_2/(\bar{\nu}|\bar{\alpha}_1))
\end{aligned}$$

THEOREM 6-28: *(Properties of atom bindings)* The binding of a trivial well-formed Etude term "$\mathsf{RET}\ (\bar{\alpha})$" to a list of variables $\bar{\nu}$ of the same length as $\bar{\alpha}$ is equivalent to the

substitution of $\bar{\alpha}$ for $\bar{\nu}$ within the associated body term τ:

$$\begin{aligned}&\text{EQV}_{\text{LETA}} :: \forall\ \Lambda,\ \Delta,\ \bar{\nu},\ \bar{\alpha} \Rightarrow\\ &\quad \text{WF}(\bar{\nu}) \rightarrow \text{WF}(\bar{\alpha}) \rightarrow\\ &\quad [\![\text{length}(\bar{\nu}) = \text{length}(\bar{\alpha})]\!] \rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\nu}\ \texttt{=}\ \texttt{RET}\ (\bar{\alpha});\ \tau]\!] \equiv (\Lambda,\ \Delta \triangleright \tau/(\bar{\nu}|\bar{\alpha}))\end{aligned}$$

PROOF: According to the earlier definition of $\mathcal{E}_\tau(\Lambda,\ \Delta \triangleright \texttt{RET}\ (\bar{\alpha}))$, all such terms deliver an empty list of actions $\bar{a}_1$, together with the normal forms of the specified atoms $\bar{\alpha}$. The object environment Δ is not affected by such reduction. In effect, we have:

$$\begin{aligned}&\mathcal{E}_\tau(\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\nu}\ \texttt{=}\ \texttt{RET}\ (\bar{\alpha});\ \tau)\\ &\quad = \mathcal{E}_\tau(\Lambda,\ \Delta \triangleright \tau/(\bar{\nu}|\mathcal{E}(\bar{\alpha})))\\ &\quad \equiv (\Lambda,\ \Delta \triangleright \tau/(\bar{\nu}|\bar{\alpha}))\end{aligned}$$

as required. □

THEOREM 6-29: *(Properties of monadic term bindings)* Given a pair of Etude constructs "LET $\bar{\nu}_1$ = τ_{11}; τ_{21}" and "LET $\bar{\nu}_2$ = τ_{12}; τ_{22}", in which the respective variable lists have equal lengths and the two terms τ_{11} and τ_{12} are equivalent under the prevailing evaluation environments, the entire constructs are also equivalent, provided that, for every list of well-formed atoms $\bar{\alpha}'$ of an appropriate length, a substitution of these atoms for $\bar{\nu}_1$ in τ_{21} and, in τ_{22} for the variable list $\bar{\nu}_2$ delivers a pair of Etude term forms that are equivalent under any well-formed function and object environment (Λ', Δ'). Formally:

$$\begin{aligned}&\text{EQV}_{\text{LETS}} :: \forall\ \Lambda,\ \Delta,\ \bar{\nu}_1,\ \tau_{11},\ \tau_{21},\ \bar{\nu}_2,\ \tau_{12},\ \tau_{22} \Rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \tau_{11}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \tau_{12}]\!] \rightarrow\\ &\quad [\![\text{length}(\bar{\nu}_1) = \text{length}(\bar{\nu}_2)]\!] \rightarrow\\ &\quad (\forall\ \Lambda',\ \Delta',\ \bar{\alpha}' \Rightarrow\\ &\qquad \text{WF}(\bar{\alpha}') \rightarrow\\ &\qquad [\![\text{length}(\bar{\alpha}') = \text{length}(\bar{\nu}_1)]\!] \rightarrow\\ &\qquad [\![\Lambda',\ \Delta' \triangleright \tau_{21}/(\bar{\nu}_1|\bar{\alpha}')]\!] \equiv [\![\Lambda',\ \Delta' \triangleright \tau_{22}/(\bar{\nu}_2|\bar{\alpha}')]\!]) \rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\nu}_1\ \texttt{=}\ \tau_{11};\ \tau_{21}]\!] \equiv [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\nu}_2\ \texttt{=}\ \tau_{12};\ \tau_{22}]\!]\end{aligned}$$

PROOF: First of all, observe that, from the equivalence of τ_{11} and τ_{12}, we have $\mathcal{E}_\tau(\Lambda,\ \Delta \triangleright \tau_{11}) = \mathcal{E}_\tau(\Lambda,\ \Delta \triangleright \tau_{12}) = (\Delta',\ \bar{a},\ \bar{\alpha}')$, for some well-formed value of Δ' and $\bar{\alpha}'$. Further, an application of the delivered atom list $\bar{\alpha}'$ to the second assumption of EQV_{LETS} provides us with the equivalence of the substituted terms $(\Lambda,\ \Delta' \triangleright \tau_{21}/(\bar{\nu}_1|\bar{\alpha}'))$ and $(\Lambda,\ \Delta' \triangleright \tau_{22}/(\bar{\nu}_2|\bar{\alpha}'))$, or the definitional equality of $\mathcal{E}_\tau(\Lambda,\ \Delta' \triangleright \tau_{21}/(\bar{\nu}_1|\bar{\alpha}'))$ and $\mathcal{E}_\tau(\Lambda,\ \Delta' \triangleright \tau_{22}/(\bar{\nu}_2|\bar{\alpha}'))$, from which the equivalence of the entire corresponding "LET" forms can be obtained directly by the virtue of theorem EQV_τ. □

THEOREM 6-30: *(Properties of nested bindings)* A nested "LET" term of the form "LET ($\bar{\nu}$ = LET $\bar{\beta}$; τ_1); τ_2" is always guaranteed to be equivalent to its flattened variant "LET $\bar{\beta}$; (LET $\bar{\nu}$ = τ_1; τ_2)", provided that no variable bound in $\bar{\beta}$ appears free within the body term τ_2, with the possible exception of any variables that are also bound in $\bar{\nu}$:

$$\begin{aligned}&\text{EQV}_{\text{LETN}} :: \forall\ \Lambda,\ \Delta,\ \bar{\nu},\ \bar{\beta},\ \tau_1,\ \tau_2 \Rightarrow\\ &\quad [\![\text{BV}(\bar{\beta}) \cap (\text{FV}(\tau_2) \backslash \bar{\nu}) = \varnothing]\!] \rightarrow\\ &\quad [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ (\bar{\nu}\ \texttt{=}\ \texttt{LET}\ \bar{\beta};\ \tau_1);\ \tau_2]\!] \equiv [\![\Lambda,\ \Delta \triangleright \texttt{LET}\ \bar{\beta};\ (\texttt{LET}\ \bar{\nu}\ \texttt{=}\ \tau_1;\ \tau_2)]\!]\end{aligned}$$

PROOF: For simplicity, let us only consider the case of $\bar{\beta} = [\![\bar{v}' = \tau']\!]$, since all more complex binding groups are always reduced by $\mathcal{E}_\tau$ into a sequence of such term forms. From the definition of $\mathcal{E}_\tau(\Lambda, \Delta \triangleright \mathsf{LET}\ (\bar{v} = \mathsf{LET}\ \bar{v}' = \tau';\ \tau_1);\ \tau_2)$, we have:

$$\begin{aligned}&\mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \mathsf{LET}\ (\bar{v} = \mathsf{LET}\ \bar{v}' = \tau';\ \tau_1);\ \tau_2]\!] = (\Delta_2, \bar{a}_1 +\!\!+ \bar{a}_2, \bar{\alpha}_2)\\ &\quad\text{where } (\Delta_1, \bar{a}_1, \bar{\alpha}_1) = \mathcal{E}_\tau[\![\Lambda, \Delta\ \triangleright \mathsf{LET}\ \bar{v}' = \tau';\ \tau_1]\!])\ (\tau_2))\\ &\qquad\quad (\Delta_2, \bar{a}_2, \bar{\alpha}_2) = \mathcal{E}_\tau[\![\Lambda, \Delta_1 \triangleright \tau_2/(\bar{v}|\bar{\alpha}_1)]\!]\end{aligned}$$

and, for the inner "LET" binding:

$$\begin{aligned}&\mathcal{E}_\tau(\Lambda, \Delta \triangleright \mathsf{LET}\ \bar{v}' = \tau';\ \tau_1) = (\Delta_1, \bar{a}' +\!\!+ \bar{a}'_1, \bar{\alpha}_1)\\ &\quad\text{where } (\Delta', \bar{a}', \bar{\alpha}') = \mathcal{E}_\tau[\![\Lambda, \Delta\ \triangleright \tau']\!]\\ &\qquad\quad (\Delta_1, \bar{a}'_1, \bar{\alpha}_1) = \mathcal{E}_\tau[\![\Lambda, \Delta'\ \triangleright \tau_1/(\bar{v}'|\bar{\alpha}')]\!]\end{aligned}$$

where $\bar{a}_1 = \bar{a}' +\!\!+ \bar{a}'_1$. Similarly, the reduction of "$\mathsf{LET}\ \bar{v}' = \tau';\ (\mathsf{LET}\ \bar{v} = \tau_1;\ \tau_2)$" proceeds as follows:

$$\begin{aligned}&\mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \mathsf{LET}\ \bar{v}' = \tau';\ (\mathsf{LET}\ \bar{v} = \tau_1;\ \tau_2)]\!] = (\Delta_2, \bar{a}' +\!\!+ \bar{a}'_2, \bar{\alpha}_2)\\ &\quad\text{where } (\Delta', \bar{a}', \bar{\alpha}') = \mathcal{E}_\tau[\![\Lambda, \Delta\ \triangleright \tau']\!]\\ &\qquad\quad (\Delta_2, \bar{a}'_2, \bar{\alpha}_2) = \mathcal{E}_\tau[\![\Lambda, \Delta'\ \triangleright [\![\mathsf{LET}\ \bar{v} = \tau_1;\ \tau_2]\!]/(\bar{v}'|\bar{\alpha}')]\!]\end{aligned}$$

Given that, by assumption, $\bar{v}'$ does not appear free within τ_2, $\tau_2/(\bar{v}'|\bar{\alpha}') = \tau_2$, so that the later "LET" form can be simplified as follows:

$$\begin{aligned}&\mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \mathsf{LET}\ \bar{v} = [\![\tau_1/(\bar{v}'|\bar{\alpha}')]\!];\ \tau_2]\!] = (\Delta_2, \bar{a}'_1 +\!\!+ \bar{a}_2, \bar{\alpha}_2)\\ &\quad\text{where } (\Delta_1, \bar{a}'_1, \bar{\alpha}_1) = \mathcal{E}_\tau[\![\Lambda, \Delta\ \triangleright \tau_1/(\bar{v}'|\bar{\alpha}')]\!]\\ &\qquad\quad (\Delta_2, \bar{a}_2, \bar{\alpha}_2) = \mathcal{E}_\tau[\![\Lambda, \Delta_1 \triangleright \tau_2/(\bar{v}|\bar{\alpha}_1)]\!]\end{aligned}$$

so that both the nested and the flattened variant of the "LET" statement are reducible to the common result value $(\Delta_2, \bar{a}' +\!\!+ \bar{a}'_1 +\!\!+ \bar{a}_2, \bar{\alpha}_2)$ as required. □

On MMIX, all bindings that have been grouped into a single "LET" term are always evaluated precisely in the order their appearance within the group, so that the reductions of all remaining "LET" term forms can be formalised trivially as follows:

$$\begin{aligned}&\mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \mathsf{LET};\ \tau]\!] \quad = \mathcal{E}_\tau(\Lambda, \Delta \triangleright \tau)\\ &\mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \mathsf{LET}\ \bar{\beta};\ \tau]\!] = \mathcal{E}_\tau[\![\Lambda, \Delta \triangleright \mathsf{LET}\ [\![\mathrm{head}(\bar{\beta})]\!];\ \mathsf{LET}\ [\![\mathrm{tail}(\bar{\beta})]\!];\ \tau]\!]\end{aligned}$$

THEOREM 6-31: *(Properties of binding groups)* All Etude binding groups satisfy the following requirements on the MMIX architecture:

$$\begin{aligned}&\mathrm{WF}_{\mathrm{LETM}} :: \forall\ \Lambda, \Delta, \bar{\beta}, \tau \Rightarrow\\ &\quad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}(\Delta) \rightarrow\\ &\quad \mathrm{WF}[\![\Lambda, \Delta \triangleright \mathsf{LET}\ [\![\mathrm{head}(\bar{\beta})]\!];\ \mathsf{LET}\ [\![\mathrm{tail}(\bar{\beta})]\!];\ \tau]\!] \rightarrow\\ &\quad [\![[\mathrm{FV}(\tau_k)\backslash\bar{v}_k \mid [\![\bar{v}_k = \tau_k]\!] \leftarrow \bar{\beta}] \cup \cap\, \mathrm{BV}(\bar{\beta}) = \varnothing]\!] \rightarrow\\ &\quad (\forall\ k \Rightarrow [\![0 \le k < \mathrm{length}(\bar{\beta})!]\!] \rightarrow [\![\Lambda, \Delta \triangleright \mathsf{LET}\ \bar{\beta};\ \tau]\!] \equiv [\![\Lambda, \Delta \triangleright \mathsf{LET}\ [\![\mathcal{P}(k, \bar{\beta})]\!];\ \tau]\!]) \rightarrow\\ &\quad \mathrm{WF}[\![\Lambda, \Delta \triangleright \mathsf{LET}\ \bar{\beta};\ \tau]\!]\end{aligned}$$

$$\mathrm{EQV}_{\mathrm{LETE}} :: \forall\ \Lambda, \Delta, \tau \Rightarrow [\![\Lambda, \Delta \triangleright \mathsf{LET};\ \tau]\!] \equiv [\![\Lambda, \Delta \triangleright \tau]\!]$$

$$\begin{aligned}&\mathrm{EQV}_{\mathrm{LETM}} :: \forall\ \Lambda, \Delta, \bar{v}, \bar{\beta}, \tau \Rightarrow\\ &\quad [\![\mathrm{length}(\bar{\beta}) > 1]\!] \rightarrow\\ &\quad [\![\Lambda, \Delta \triangleright \mathsf{LET}\ \bar{\beta};\ \tau]\!] \equiv [\![\Lambda, \Delta \triangleright \mathsf{LET}\ [\![\mathrm{head}(\bar{\beta})]\!];\ \mathsf{LET}\ [\![\mathrm{tail}(\bar{\beta})]\!];\ \tau]\!]\end{aligned}$$

PROOF: By definition of the $\mathcal{E}_\tau$ function. The convoluted precondition of WF_{LETM}, which was provided in the earlier algebraic semantics of Etude in order to forbid overlapping of side effects within a binding group, is irrelevant on MMIX, which always imposes a predetermined order of term reduction within such constructs. □

Finally, one more auxiliary theorem must be established in order to complete our formalisation of Etude terms under the MMIX instruction set architecture:

THEOREM 6-32: *(Validity of equivalent term forms)* Every Etude term from a given equivalent class is well-formed whenever at least one of the terms in that class is deemed valid:

$$\begin{aligned}&\text{WF}_{\text{EQV}} :: \forall\ \Lambda_1,\ \Delta_1,\ \tau_1,\ \Lambda_2,\ \Delta_2,\ \tau_2 \Rightarrow\\ &\qquad \text{WF}[\![\Lambda_1,\ \Delta_1 \triangleright \tau_1]\!] \rightarrow [\![\Lambda_1,\ \Delta_1 \triangleright \tau_1]\!] \equiv [\![\Lambda_2,\ \Delta_2 \triangleright \tau_2]\!] \rightarrow \text{WF}[\![\Lambda_2,\ \Delta_2 \triangleright \tau_2]\!]\end{aligned}$$

PROOF: Trivial, from the analogous property $\text{WF}_{\equiv}$ that was established for Etude atoms in Section 6.3.1. □

A complete operational description of Etude terms is, regrettably, impossible without a detailed scrutiny of the underlying operating system facilities. In particular, most of such facilities operate in the real world of hardware devices and user interactions, so that their complete formalisation would probably mandate an extension of the earlier object environment structure from Section 6.3.2 with some suitable abstract rendition of that real world. Fortunately, this added complexity is not required for a successful establishment of the linear correctness property for our compiler, whereby the above algebraic treatment of system calls turns out to be sufficiently detailed as to satisfy all of the proof obligations raised in the following section of this work. Accordingly, without further ado, let us now proceed to the final code generation stage of the entire project.

6.4 Code Generation

As described in Chapter 2, the ultimate validity of our entire compiler rests on a single property of *linear correctness*. In Section 2.4, this property was formulated as the judgement $\rho \circ \hat{\psi} \circ \psi = \rho$, in which ρ and ψ represent the translation semantics of C and MMIX. In Sections 5.10 and 6.2, these two functions were implemented by the respective Haskell definitions $\mathcal{D}_{\text{TU}}$ and $\mathcal{D}_{\mathbf{M}}$. We are now ready to define the final piece of the entire puzzle $\hat{\psi}$, in a form of the reverse mapping of Etude modules into their MMIX counterparts. Formally, $\hat{\psi}$ is depicted by the following algorithm $\mathcal{C}$:

$$\mathcal{C}[\![\cdot]\!] :: (\text{ord}\ \upsilon) \Rightarrow \mathit{module}_\upsilon \rightarrow \mathit{MMIX\text{-}module}$$

$$\begin{aligned}&\mathcal{C}[\![\textsf{MODULE}\ \tau\ \textsf{EXPORT}\ \bar{\xi}\ \textsf{WHERE}\ \bar{\iota}]\!]\\ &\qquad = [\![\textbf{LET}\ S\ \textbf{TEXT}\ T\ \textbf{DATA}\ D\ \textbf{IN}\ \#0]\!]\\ &\text{where}\ S = \{\Sigma(\upsilon_k) : x_k \mid [\![x_k = \upsilon_k]\!] \leftarrow \bar{\xi}\} \cup \{\Sigma(\upsilon_k) : x_k \mid [\![\upsilon_k = \textsf{IMP}\ x_k]\!] \leftarrow \bar{\iota}\}\\ &\qquad T = \{\Sigma(\upsilon_k) : \mathcal{C}_\lambda[\![\Sigma' \triangleright \lambda\bar{\upsilon}_k.\tau_k]\!] \mid [\![\upsilon_k = \lambda\bar{\upsilon}_k.\tau_k]\!] \leftarrow \bar{\iota}\} \cup \{\#0 : \mathcal{C}_\lambda[\![\Sigma' \triangleright \lambda.\tau]\!]\}\\ &\qquad D = \{\Sigma(\upsilon_k) : [\![\textsf{OBJ}\ ([\![\delta_k/\Sigma']\!])\ \textsf{OF}\ (\bar{\xi}_k)]\!] \mid [\![\upsilon_k = \textsf{OBJ}\ (\delta_k)\ \textsf{OF}\ (\bar{\xi}_k)]\!] \leftarrow \bar{\iota}\}\\ &\qquad \Sigma = \text{dom}(\bar{\iota}) \mid [\#1\ ...]\\ &\qquad \Sigma' = \{\upsilon_k : [\![\sigma_k.0]\!] \mid \upsilon_k : \sigma_k \leftarrow \Sigma\}\end{aligned}$$

In other words, an Etude module of the form "MODULE τ EXPORT $\bar{\xi}$ WHERE $\bar{\imath}$" is compiled into the symbolic MMIX construct "**LET** *S* **TEXT** *T* **DATA** *D* **IN #**0". The supplied set of item definitions $\bar{\imath}$ is split into the text segment *T*, data segment *D* and symbol table *S* that are formed, respectively, from all function, data and imported items found in $\bar{\imath}$. Further, the symbol table is also extended with bindings of any symbols that are made available to the remainder of the entire program through the source module's export list $\bar{\xi}$. The precise translation of Etude functions, depicted in the above definition by the algorithm $\mathcal{C}_\lambda$, is discussed later in Section 6.4.3.

THEOREM 6-33: *(Linear correctness)* Collectively, the three algorithms $\mathcal{D}_{\text{TU}}$, $\mathcal{D}_{\mathbf{M}}$ and $\mathcal{C}$ satisfy the linear correctness property $\rho \circ \hat{\psi} \circ \psi = \rho$. In particular, for every set of translation units $t\bar{u}$ that forms an autonomous C program, their collective image produced by the system's linking algorithm $\mathcal{L}$ is equivalent to the term obtained by the same algorithm from the collection of meanings assigned to their respective MMIX renditions. Formally:

$$\text{LC} :: \forall\, t\bar{u} \Rightarrow \mathcal{L}[\mathcal{D}_{\text{TU}}(tu_k) \mid tu_k \leftarrow t\bar{u}] \equiv \mathcal{L}[\mathcal{D}_{\mathbf{M}}(\mathcal{C}(\mathcal{D}_{\text{TU}}(tu_k))) \mid tu_k \leftarrow t\bar{u}]$$

PROOF: Recall from Section 4.7 that $\mathcal{L}$ produces a single Etude term τ, together with a context (Λ, Δ) that, collectively, describes the initial address space of the entire program. Accordingly, the above theorem follows directly from a similar result derived for individual terms in Section 6.4.3 below. □

6.4.1 Variables and Registers

In the above definition of the module compilation algorithm $\mathcal{C}$, the reader will doubtless notice the two finite maps Σ and Σ'. In a nutshell, these maps are used to rename any input variables found in the source module into their appropriate MMIX counterparts that, in the resulting program, are depicted by the respective symbols **#**0, **#**1, **#**2 and so on. The designated symbol **#**0 is reserved for the module's initialiser term, while all other MMIX variables are bound systematically to any global Etude item definitions found in the supplied list $\bar{\imath}$. For data items, the renaming process is represented by the standard Etude substitution function δ_k/Σ that was defined earlier in Section 4.7. For all other Etude entities, however, variable renaming is integrated directly into the compilation process, whereby it facilitates implementation of an important translation stage known as *register allocation*.

Intuitively, register allocation converts all Etude variables found in the source program into appropriate general purpose MMIX registers. Although our target architecture provides an abundance of register resources, in principle, it is always possible to define Etude programs with more variables than those supported directly by the hardware. To this end, in the following translation, only the first 250 Etude variables are assigned to concrete hardware registers **$**0 ... **$**249, with all remaining entities bound to *pseudo-registers* **$**256, **$**257 and so on. As will be revealed shortly, in the translated program the values of all such pseudo-registers are stored in memory-resident objects that are set aside for that specific purpose by our compiler.

For conciseness, the sequence of all MMIX registers available for depiction of Etude variables is represented by the notation $\bar{v}_{\mathrm{G}}$. Further, the list of all variables assigned by our compiler to the individual parameters of every Etude function is depicted by the symbol $\bar{v}_{\mathrm{P}}$. Formally, both $\bar{v}_{\mathrm{G}}$ and $\bar{v}_{\mathrm{P}}$ are defined in Haskell as the following infinite list structures:

$$\bar{v}_{\mathrm{G}},\ \bar{v}_{\mathrm{P}} :: [\mathit{MMIX\text{-}var}]$$

$$\begin{aligned}\bar{v}_{\mathrm{G}} &= [\$249,\ \$248\ ...\ \$0\ \ ] +\!\!+ [\$256\ ...]\\ \bar{v}_{\mathrm{P}} &= [\$0,\ \ \ \$1\ \ \ ...\ \$249] +\!\!+ [\$256\ ...]\end{aligned}$$

A careful reader will observe that the order in which true MMIX registers appear in these lists is inverted within $\bar{v}_{\mathrm{G}}$. In Section 6.4.3, this simple technique serves to reduce the possibility of a single variable being associated with both a function argument and a temporary entity within that function's caller. Under the naïve register allocation algorithm defined in this section, such conflicts would result in a significant performance degradation of the generated MMIX programs, but, given the multitude of registers provided by our target architecture, they are unlikely to occur in practice.

The remaining six general purpose hardware register $250 ... $255 are reserved by our code generator for various specific purposes described throughout the remainder of this section. For clarity, in the following presentation they are always depicted by the following symbolic names:

$$\texttt{\$S},\ \texttt{\$K},\ \texttt{\$T},\ \texttt{\$U},\ \texttt{\$V},\ \texttt{\$W} :: \mathit{MMIX\text{-}var}$$

`$S` = $255	*(stack pointer)*
`$K` = $254	*(continuation pointer)*
`$T` = $253	*(temporary variable T)*
`$U` = $252	*(temporary variable U)*
`$V` = $251	*(temporary variable V)*
`$W` = $250	*(temporary variable W)*

Intuitively, the two register `$S` and `$K` (or $255 and $254) are dedicated for storage of the current stack pointer and the continuation address of the Etude function under execution. The following translation always ensures that `$S` invariably echoes the current value of the object environment component σ_c, in order to relieve the processor from maintaining that component explicitly within its hardware state. Further `$K` will always contain the address of the current function's caller, or, more specifically, the precise location of the instruction that the program intends to execute upon return from the presently evaluated function. The remaining four registers `$T`, `$U`, `$V` and `$W` provide our compiler with a temporary scratch space that is required by our register allocator for a transparent implementation of any pseudo-register variables.

In particular, the process of allocating a given MMIX variable v to an appropriate general purpose hardware register can be depicted in Haskell as follows:

$$\mathcal{A}_v[\![\cdot]\!] :: (\mathit{MMIX\text{-}var} \triangleright \mathit{MMIX\text{-}var}) \to \mathit{MMIX\text{-}var}$$

$$\begin{aligned}\mathcal{A}_v[\![\$N \triangleright v]\!] \ &\mid\ \$0 \le v \le \$255 = v\\ &\mid\ \text{otherwise} \qquad\ = \$N\end{aligned}$$

If the specified Etude variable already corresponds to one of the hardware registers $\texttt{\$}0$ ... $\texttt{\$}255$, then $\mathcal{A}_v(\texttt{\$}N \triangleright v)$ always allocates v directly to that register. Otherwise, the construction returns the *backup register* $\texttt{\$}N$, which, in all cases, must be supplied by the surrounding algorithm with a value disjoint from any other such registers already allocated for one of the Etude variables that remain free within the remainder of the program under translation.

For convenience, we also provide an implicit coercion function, which maps all hardware-provided registers to their concrete numeric encodings, suitable for use within the operand fields of an appropriate MMIX instruction form. Specifically:

$$[\![\cdot]\!] :: \textit{MMIX-var} \rightarrow \textit{integer}$$

$$[\![\texttt{\$}N]\!] = N$$
$$[\![\texttt{rR}]\!] = 6$$

Finally, the notation "load($\texttt{\$}D \triangleright v$)" represents a *variable load operation*, which, in the translated program, can be utilised to place the value of an arbitrary MMIX variable into the specified general purpose hardware register $\texttt{\$}D$. In all cases, load($\texttt{\$}D \triangleright v$) returns a sequence of zero or more MMIX instructions that achieve the desired effect as part of their execution. In particular, if v and $\texttt{\$}D$ represent the same hardware register, then the resulting sequence will be empty, since no actual movement of data is required. Otherwise, if v represents a hardware register, then the construction delivers a single *register copy operation*, depicted by an MMIX instruction of the form "**OR** $\texttt{\$}D$**,**$\texttt{\$}S$**,**$\texttt{\$}S$" or "**GET** $\texttt{\$}D$**,**$0$**,rR**" for all general purpose registers and the remainder register, respectively. If, on the other hand, the source variable assumes a symbolic form "$\sigma\texttt{.}\delta$", then the symbol's numeric value is placed in the target register using a sequence of four MMIX instructions "**SETL**", "**INCML**", "**INCMH**" and "**INCH**", which were dedicated to the specific task of constant synthesis earlier in Section 6.2. Finally, if v represents one of the pseudo-registers $\texttt{\$}S$ in which $S \geq 256$, then the variable's value is stored in a memory-resident object found at the predetermined offset $8 \times S$ into the dedicated MMIX data segment "**LOC**". In the resulting instruction sequence, the content of that object is placed into $\texttt{\$}D$ by synthesising the target address "**LOC.**$8S$" into the dedicated temporary register **$V** and fetching the required word of data using an appropriate MMIX instruction "$\texttt{LDOU}_\texttt{I}$". Formally:

$$\text{load}[\![\cdot]\!] :: (\textit{MMIX-var} \triangleright \textit{MMIX-var}) \rightarrow [\textit{MMIX-instr}]$$

$$\begin{array}{lll}
\text{load}[\![\texttt{\$}D \triangleright \texttt{\$}S]\!] & \mid \texttt{\$}S = \texttt{\$}D & = \varnothing \\
& \mid \texttt{\$}S \leq \texttt{\$}255 & = [\![\texttt{OR \$}D\texttt{, \$}S\texttt{, \$}S]\!] \\
& \mid \texttt{\$}S \geq \texttt{\$}256 & = \text{load}[\![\texttt{\$V} \triangleright \texttt{LOC.}8S]\!] +\!\!+ [\![\texttt{LDOU}_\texttt{I}\ \texttt{\$}D\texttt{, \$V, }0]\!] \\
\text{load}[\![\texttt{\$}D \triangleright \texttt{rR}]\!] & & = [\![\texttt{GET \ \ \$}D\texttt{, }0\texttt{, rR}]\!] \\
\text{load}[\![\texttt{\$}D \triangleright \sigma\texttt{.}\delta]\!] & & = [\![\texttt{SETL \ \$}D\texttt{, }[\![\delta[8 \ldots 15]]\!]\texttt{, \ }[\![\delta[0 \ldots 7]]\!] \ \ \texttt{@ }\sigma \\
& & \quad \texttt{INCML \$}D\texttt{, }[\![\delta[24 \ldots 31]]\!]\texttt{, }[\![\delta[16 \ldots 23]]\!]\texttt{ @ }\sigma \\
& & \quad \texttt{INCMH \$}D\texttt{, }[\![\delta[40 \ldots 47]]\!]\texttt{, }[\![\delta[32 \ldots 39]]\!]\texttt{ @ }\sigma \\
& & \quad \texttt{INCH \ \$}D\texttt{, }[\![\delta[56 \ldots 63]]\!]\texttt{, }[\![\delta[48 \ldots 55]]\!]\texttt{ @ }\sigma]\!]
\end{array}$$

Further, the dual operation "store($v \triangleright \texttt{\$}S$)" binds the specified MMIX variable v to the

current value of a given general purpose hardware register $\texttt{\$}S$. Specifically, if $v = \texttt{\$}S$, or if it represents a general purpose hardware register, then the construction behaves identically to the earlier "load" operation with the same arguments. On the other hand, if v refers to some pseudo-register $\texttt{\$}D$ for $D \geq 256$, then the corresponding memory-resident object "$\texttt{LOC.}8D$" is updated with the value of $\texttt{\$}S$ using an appropriate "$\texttt{STOU}_\texttt{I}$" instruction form. Finally, an update of the **rR** register can be performed using the MMIX instruction "**PUT**" as follows:

$$\begin{aligned}
&\text{store}[\![\cdot]\!] :: (\textit{MMIX-var} \triangleright \textit{MMIX-var}) \rightarrow [\textit{MMIX-instr}] \\
&\\
&\text{store}[\![\texttt{\$}D \triangleright \texttt{\$}S]\!] \mid \texttt{\$}D = \texttt{\$}S \quad = \varnothing \\
&\qquad\qquad\quad \mid \texttt{\$}D \leq \texttt{\$}255 = [\![\texttt{OR \$}D\texttt{, \$}S\texttt{, \$}S]\!] \\
&\qquad\qquad\quad \mid \texttt{\$}D \geq \texttt{\$}256 = \text{load}[\![\texttt{\$V} \triangleright \texttt{LOC.}8D]\!] +\!\!+ [\![\texttt{STOU}_\texttt{I}\ \texttt{\$}S\texttt{, \$V, }0]\!] \\
&\text{store}[\![\texttt{rR} \triangleright \texttt{\$}S]\!] \qquad\qquad = [\![\texttt{PUT rR, }0\texttt{, \$}S]\!]
\end{aligned}$$

The reader should observe that "store" is left undefined whenever its target operand represents a symbolic MMIX variable, since symbol bindings cannot be redefined in a well-formed Etude program.

Finally, two additional constructs facilitate direct movement of information between pairs of MMIX variables or between two sets of such variables. Formally, $\mathcal{C}_v(v_\text{D} \triangleright v_\text{S})$ begins by loading the value of v_S into an appropriate general purpose hardware register $\texttt{\$}R$, set to the scratch register **$W** if v_S does not already have the appropriate form. The resulting value is then placed in the targeted variable v_D using the earlier "store" combinator. In Haskell:

$$\begin{aligned}
&\mathcal{C}_v[\![\cdot]\!] :: (\textit{MMIX-var} \triangleright \textit{MMIX-var}) \rightarrow [\textit{MMIX-instr}] \\
&\\
&\mathcal{C}_v[\![v_\text{D} \triangleright v_\text{S}]\!] \;= \text{load}(\texttt{\$}R \triangleright v_\text{S}) +\!\!+ \text{store}(v_\text{D} \triangleright \texttt{\$}R) \\
&\;\text{where } [\![\texttt{\$}R]\!] = \mathcal{A}_v(\mathcal{A}_v(\texttt{\$W} \triangleright v_\text{D}) \triangleright v_\text{S})
\end{aligned}$$

Further, $\mathcal{C}_{\bar{v}}(\bar{v}_\text{D} \triangleright \bar{v}_\text{S})$ makes it possible to shuffle data between two potentially overlapping sets of MMIX variables. Formally, this construction is implemented as follows:

$$\begin{aligned}
&\mathcal{C}_{\bar{v}}[\![\cdot]\!] :: ([\textit{MMIX-var}] \triangleright [\textit{MMIX-var}]) \rightarrow [\textit{MMIX-instr}] \\
&\\
&\mathcal{C}_{\bar{v}}[\![\bar{v}_\text{D} \triangleright \bar{v}_\text{S}]\!] \\
&\quad \mid \bar{v}_\text{D} = \varnothing \vee \bar{v}_\text{S} = \varnothing = [\![]\!] \\
&\quad \mid \text{otherwise} \qquad\qquad = \mathcal{C}_v(v'_\text{D} \triangleright v_\text{D}) +\!\!+ \mathcal{C}_v(v_\text{D} \triangleright v_\text{S}) +\!\!+ \mathcal{C}_{\bar{v}}(\text{tail}(\bar{v}_\text{D}) \triangleright \text{tail}(\bar{v}'_\text{S})) \\
&\\
&\;\text{where } v_\text{S} = \text{head}(\bar{v}_\text{S}) \\
&\qquad\;\; v_\text{D} = \text{head}(\bar{v}_\text{D}) \\
&\qquad\;\; v'_\text{D} = \text{head}[v_k \mid v_k \leftarrow v_\text{D} +\!\!+ \bar{v}_\text{G},\ v_k \notin \text{tail}(\bar{v}_\text{S})] \\
&\qquad\;\; \bar{v}'_\text{S} = [\Sigma(v_k) \mid v_k \leftarrow \bar{v}_\text{S}] \\
&\qquad\;\; \Sigma = (\bar{v}_\text{S} | \bar{v}_\text{S})/\{v_\text{D}{:}v'_\text{D}\}
\end{aligned}$$

In summary, $\mathcal{C}_{\bar{v}}(\bar{v}_\text{D} \triangleright \bar{v}_\text{S})$ simply binds every variable $v_\text{D} \in \bar{v}_\text{D}$ to the corresponding value $v_\text{S} \in \bar{v}_\text{S}$. However, in order to preserve the semantics of any still unprocessed variables in $\bar{v}_\text{S}$, the present value of v_D is first saved in some temporary scratch space v'_D, taken to represent the first unallocated variable from the list $[v_\text{D}] +\!\!+ \bar{v}_\text{G}$ that is not mentioned in the remainder of the source list $\bar{v}_\text{S}$. Once v_S has been assigned its new

value, $\mathcal{C}_{\bar{v}}$ proceeds with any remaining entries in $\bar{v}_S$, after renaming any occurrences of v_D in that list to its new binding v'_D.

In order to simplify presentation of the following linear correctness theorems, let us introduce one additional auxiliary definition, which presents the meaning of an entire instruction section that was produced by our compiler as an appropriate function environment structure Λ. Formally:

$$\mathcal{D}_{\mathtt{\bar{I}}}[\![\cdot]\!] :: (\textit{MMIX-var} \mapsto \textit{integer}, \textit{MMIX-var} \triangleright \textit{MMIX-section}) \to \textit{f-env}_{\textit{MMIX-var}}$$
$$\mathcal{D}_{\mathtt{\bar{I}}}[\![\Gamma, v \triangleright \bar{\imath}]\!] = \{(\Gamma(v) + 4k : \mathcal{D}_{\mathtt{I}}(v \oplus 4k \triangleright \imath_k)) \mid (\imath_k, k) \leftarrow \bar{\imath}|[0 \ldots]\}$$

THEOREM 6-34: *(Preservation of atom semantics under compilation)* Let v_S and v_D represent a pair of MMIX variables involved in a given copy operation and let Γ represent the mapping of all relevant MMIX variables to their respective atomic values. Further, let Λ' represent a function environment updated with bindings of n successive instruction addresses, beginning at some Etude function variable v, to the respective n instructions generated from the compilation of "$\mathcal{C}_v(v_D \triangleright v_S)$". If v_S represents a pseudo-register $\mathtt{\$}N$, then let Δ' represent an object environment that is likewise updated with a binding of the corresponding address $\mathtt{LOC.}8N$ to $\Gamma(\mathtt{\$}N)$. Then, every application of v to a the standard parameter list π under a substitution by Γ is always equivalent to an application of the function $v \oplus 4n$ to the same parameter list under a substitution by an updated finite map $\Gamma/\{\Gamma(v_D) : v_S\}$. Formally:

$$\begin{aligned}
&\mathrm{LC}_v :: \forall\, \Gamma, \Gamma', \Lambda, \Lambda', \Delta, \Delta', v, v_S, v_D \Rightarrow \\
&\quad \mathrm{WF}(\Lambda) \to \mathrm{WF}(\Delta) \to \\
&\quad [\![\Gamma' = \Gamma/\{\Gamma(v_D) : v_S\}]\!] \to \\
&\quad [\![\Lambda' = \Lambda/\mathcal{D}_{\mathtt{\bar{I}}}(\Gamma, v, \mathcal{C}_v(v_D \triangleright v_S))]\!] \to \\
&\quad [\![\Delta' = \Delta/\{\Gamma(\mathtt{LOC.}8k) : \Gamma(\mathtt{\$}k) \mid [\![\mathtt{\$}k]\!] \leftarrow [v_S], k \geq 256]\!]) \to \\
&\quad [\![\Lambda', \Delta' \triangleright [\![\Gamma(v)(\pi)]\!]/\Gamma]\!] \equiv [\![\Lambda, \Delta \triangleright [\![[\![\Gamma(v \oplus \mathrm{length}(\mathcal{C}_v(v_D \triangleright v_S)))]\!](\pi)]\!]/\Gamma']\!]
\end{aligned}$$

PROOF: If $v_S = v_D$, then $\mathcal{C}_v$ produces an empty instruction sequence. Accordingly, $v \oplus \mathrm{length}(\mathcal{C}_v(v_D \triangleright v_S)) = v \oplus 0 = v$ and $\Gamma' = \Gamma/\{v_D : v_S\} = \Gamma/\{v_D : v_D\} = \Gamma$, so that the above theorem holds trivially in such cases.

Otherwise, if both v_S and v_D represent general purpose hardware registers, then the single "$\mathtt{OR}\ v_R\mathtt{,}\ \alpha\mathtt{,}\ \alpha$" instruction produced by the compiler is, by the definition of $\mathcal{D}_{\mathtt{I}}$, equivalent to "$\lambda\pi.\mathtt{LET}\ v_R = \alpha \bigtriangledown_{\mathrm{N.64}} \alpha; [\![\Gamma(v) \oplus 4]\!](\pi))$", which can be easily simplified into "$\lambda\pi.\mathtt{LET}\ v_R = \alpha; [\![\Gamma(v) \oplus 4]\!](\pi))$", given that, under the earlier operational interpretation of the "$\bigtriangledown_\phi$" operator in Section 6.3.1, we can easily establish the identity $\alpha \bigtriangledown_\phi \alpha \equiv \alpha$ for all Etude atoms α. Accordingly, an application of $\Gamma(v)$ to π is reducible by $\mathcal{E}_\tau$ into "$[\![\sigma \oplus 4]\!](\pi)$" as required.

Finally, if α depicts one of the MMIX pseudo-registers, then the resulting sequence of constant synthesis instructions can be likewise demonstrated to be equivalent to the register's corresponding address $\mathtt{LOC.}n$, after substituting each function's body into its caller in accordance with the operational model of Etude terms that was described in Section 6.3.3 and applying a simple arithmetic reasoning to the resulting sum of

constant operands in order to arrive at the required conclusion. Accordingly, the LC_v's assumption about the bindings of such address locations in the extended environment Δ' provides us with all the information required to complete the proof. □

6.4.2 Atoms

We are now ready to proceed with the actual translation of Etude atoms into appropriate MMIX instruction sequences. A proof of this translation's correctness is included at the end of the section, once we have completed the compiler's design for all valid atomic forms.

To begin with, we observe that, under all sequential program representations such as that provided by the MMIX instruction set architectures, every intermediate result of an arithmetic computation must be bound to a variable before its value can be utilised in the remainder of the program. Accordingly, we commence our translation of Etude into MMIX by describing a register allocation algorithm that selects an appropriate MMIX register for every well-formed atomic expression. A naïve but sufficient implementation of this algorithm can be defined as follows:

$$\mathcal{A}_\alpha[\![\cdot]\!] :: (\mathrm{ord}\ \upsilon) \Rightarrow (\upsilon \mapsto \textit{MMIX-var},\ \{\textit{MMIX-var}\} \triangleright \textit{atom}_\upsilon) \to \textit{MMIX-var}$$

$$\begin{aligned}
&\mathcal{A}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K} \triangleright \upsilon]\!] && = \Sigma(\upsilon)\\
&\mathcal{A}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K} \triangleright \phi'_\phi(\alpha)]\!] \mid (\phi \subseteq \phi') && = \mathcal{A}_\alpha(\Sigma,\ \bar{v}_\mathrm{K} \triangleright \alpha)\\
&\mathcal{A}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K} \triangleright \alpha]\!] && = \mathrm{head}[v_k \mid v_k \leftarrow \bar{v}_\mathrm{G},\ v_k \notin \bar{v}_\mathrm{K}]
\end{aligned}$$

In addition to the usual variable substitution mapping Σ, the above function also accepts a set of *live variables* $\bar{v}_\mathrm{K}$ that, intuitively, must be preserved for other uses later in the surrounding program. Accordingly, $\mathcal{A}_\alpha(\Sigma,\ \bar{v}_\mathrm{K} \triangleright \alpha)$ allocates the specified Etude atom α to the first variable $v_k \in \bar{v}_\mathrm{G}$ whose name does not appear in $\bar{v}_\mathrm{K}$. However, if α represents a simple Etude variable υ, then υ is always associated with the MMIX register $\Sigma(\upsilon)$ that was previously allocated for that variable. Further, as a special case, any operation of the form "$\phi'_\phi(\alpha)$" that, under the earlier semantics of Etude atoms defined in Section 6.3.1, is reducible to α itself, is allocated to the same register as its operand α, since such redundant conversions appear frequently in programs generated by automatic translations from other sources. Formally, the "$\subseteq$" operator used above for identification of these atomic forms is defined as follows:

$$[\![\cdot]\!] \subseteq [\![\cdot]\!] :: \textit{format} \to \textit{format} \to \textit{bool}$$

$$\begin{aligned}
[\![\phi_1]\!] \subseteq [\![\phi_2]\!] = {} & (\gamma(\phi_1) = \gamma(\phi_2) \wedge \varepsilon(\phi_1) \leq \varepsilon(\phi_2)) \vee\\
& (\gamma(\phi_1) = [\![\mathsf{N}]\!] \wedge \gamma(\phi_2) = [\![\mathsf{Z}]\!] \wedge \varepsilon(\phi_1) < \varepsilon(\phi_2)) \vee\\
& (\gamma(\phi_1) \neq [\![\mathsf{R}]\!] \wedge \gamma(\phi_2) \in \{\mathsf{N}, \mathsf{Z}\} \wedge \varepsilon(\phi_2) \geq 64) \vee\\
& (\gamma(\phi_1) \neq [\![\mathsf{R}]\!] \wedge \gamma(\phi_2) \in \{\mathsf{F}, \mathsf{O}\})
\end{aligned}$$

The actual sequence of MMIX instructions used to represent a given atom α in the translated program is then obtained by the following algorithm $\mathcal{C}_\alpha$:

$$\mathcal{C}_\alpha[\![\cdot]\!] :: (\mathrm{ord}\ \upsilon) \Rightarrow (\upsilon \mapsto \textit{MMIX-var},\ \{\textit{MMIX-var}\},\ \textit{MMIX-var} \triangleright \textit{atom}_\upsilon) \to [\textit{MMIX-instr}]$$

In every construction of the form "$\mathcal{C}_\alpha(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha)$", the *result variable* v_R should represent an MMIX register which has been previously allocated for the atom by

$\mathcal{A}_\alpha(\Sigma,\ \bar{\nu}_\mathrm{K} \triangleright \alpha)$. If α is formed entirely from a single Etude variable υ, or if it depicts some atomic constant "$\#x_\phi$", then the translation proceeds as follows:

$$
\begin{array}{lll}
\mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \upsilon]\!] & & = \mathcal{C}_\nu(\nu_\mathrm{R} \triangleright \Sigma(\upsilon)) \\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \#x_\phi]\!] & \mid\ (n < 2^{16}) & = [\![\texttt{SETL}\ \ \$R,\ [\![n[8\ldots15]]\!],\ \ [\![n[0\ldots7]]\!]]\!] \mathbin{+\!\!+} \\
 & & \quad \mathcal{C}_\nu(\nu_\mathrm{R} \triangleright \$R) \\
 & \mid\ (n < 2^{32}) & = [\![\texttt{SETL}\ \ \$R,\ [\![n[8\ldots15]]\!],\ \ [\![n[0\ldots7]]\!] \\
 & & \quad \texttt{INCML}\ \$R,\ [\![n[24\ldots31]]\!],\ [\![n[16\ldots23]]\!]]\!] \mathbin{+\!\!+} \\
 & & \quad \mathcal{C}_\nu(\nu_\mathrm{R} \triangleright \$R) \\
 & \mid\ (n < 2^{48}) & = [\![\texttt{SETL}\ \ \$R,\ [\![n[8\ldots15]]\!],\ \ [\![n[0\ldots7]]\!] \\
 & & \quad \texttt{INCML}\ \$R,\ [\![n[24\ldots31]]\!],\ [\![n[16\ldots23]]\!] \\
 & & \quad \texttt{INCMH}\ \$R,\ [\![n[40\ldots47]]\!],\ [\![n[32\ldots39]]\!]]\!] \mathbin{+\!\!+} \\
 & & \quad \mathcal{C}_\nu(\nu_\mathrm{R} \triangleright \$R) \\
 & \mid\ \text{otherwise} & = [\![\texttt{SETL}\ \ \$R,\ [\![n[8\ldots15]]\!],\ \ [\![n[0\ldots7]]\!] \\
 & & \quad \texttt{INCML}\ \$R,\ [\![n[24\ldots31]]\!],\ [\![n[16\ldots23]]\!] \\
 & & \quad \texttt{INCMH}\ \$R,\ [\![n[40\ldots47]]\!],\ [\![n[32\ldots39]]\!] \\
 & & \quad \texttt{INCH}\ \ \$R,\ [\![n[56\ldots63]]\!],\ [\![n[48\ldots55]]\!]]\!] \mathbin{+\!\!+} \\
 & & \quad \mathcal{C}_\nu(\nu_\mathrm{R} \triangleright \$R)
\end{array}
$$

where $[\![\#n_{\mathrm{N}.64}]\!] = \mathcal{E}[\![\#x_\phi ::]\!]$
$[\![\$R]\!] = \mathcal{A}_\nu(\texttt{\$T} \triangleright \nu_\mathrm{R})$

Intuitively, the above construction attempts to find the shortest sequence of MMIX instructions that are sufficient to bind ν_R to the atom's semantic interpretation. The reader should observe that, for many constants of the form "$\#n_\phi$", an even shorter sequence could be obtained using various MMIX instruction forms that were excluded from the discussion in Section 6.2. However, for conciseness, in this chapter I present only the simplest sufficient implementation of the MMIX code generator.

Next, we dispense with the translation of the following five simple atomic forms, whose meanings were defined in Section 6.3.1 in terms of some other Etude constructs:

$$
\begin{array}{lll}
\mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright op_\phi(\alpha)]\!] & \mid\ \gamma(\phi) \in \{\mathsf{F},\ \mathsf{O}\} & = \mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright op_{\mathrm{Z}.64}(\alpha)]\!] \\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \phi'_\phi(\alpha)]\!] & \mid\ \gamma(\phi') \in \{\mathsf{F},\ \mathsf{O}\} & = \mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \mathrm{Z}.64_\phi(\alpha)]\!] \\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \alpha_1\ op_\phi\ \alpha_2]\!] & \mid\ \gamma(\phi) \in \{\mathsf{F},\ \mathsf{O}\} & = \mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \alpha_1\ op_{\mathrm{Z}.64}\ \alpha_2]\!] \\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright -_\phi(\alpha)]\!] & \mid\ \gamma(\phi) = [\![\mathsf{R}]\!] & = \mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright [\![-0]\!] -_\phi \alpha]\!] \\
 & \mid\ \gamma(\phi) \neq [\![\mathsf{R}]\!] & = \mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright [\![+0]\!] -_\phi \alpha]\!] \\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \sim_\phi(\alpha)]\!] & & = \mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \#[\![2^{64}-1]\!]_{\mathrm{N}.64} -_\phi \alpha]\!]
\end{array}
$$

Similarly, an Etude conversion of the form "$\phi'_\phi(\alpha)$", in which $\phi \subseteq \phi'$, was defined in Section 6.3.1 to be synonymous with its atomic operand α. On the other hand, most other conversions that involve the rational format "R.64" must be performed in the resulting program using an appropriate application of the MMIX opcode "**FIXU**", "**FIX**", "**FLOTU**" or "**FLOT**" as follows:

$$
\begin{array}{lll}
\mathcal{C}_\alpha[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \phi'_\phi(\alpha)]\!] & \mid\ \phi \subseteq \phi' & = \mathcal{C}_\alpha(\Sigma,\ \bar{\nu}_\mathrm{K},\ \nu_\mathrm{R} \triangleright \alpha) \\
 & \mid\ \phi' = [\![\mathsf{N}.64]\!] \wedge \gamma(\phi) = [\![\mathsf{R}]\!] & = \mathcal{C}_\mathrm{U}(\Sigma,\ \bar{\nu}_\mathrm{K},\ \$A,\ \texttt{FIXU},\ \ 0 \triangleright \alpha) \\
 & \mid\ \phi' = [\![\mathsf{Z}.64]\!] \wedge \gamma(\phi) = [\![\mathsf{R}]\!] & = \mathcal{C}_\mathrm{U}(\Sigma,\ \bar{\nu}_\mathrm{K},\ \$A,\ \texttt{FIX},\ \ \ 0 \triangleright \alpha) \\
 & \mid\ \gamma(\phi') = [\![\mathsf{R}]\!] \wedge \gamma(\phi) = [\![\mathsf{N}]\!] & = \mathcal{C}_\mathrm{U}(\Sigma,\ \bar{\nu}_\mathrm{K},\ \$A,\ \texttt{FLOTU},\ 0 \triangleright \alpha) \\
 & \mid\ \gamma(\phi') = [\![\mathsf{R}]\!] \wedge \gamma(\phi) = [\![\mathsf{Z}]\!] & = \mathcal{C}_\mathrm{U}(\Sigma,\ \bar{\nu}_\mathrm{K},\ \$A,\ \texttt{FLOT},\ \ 0 \triangleright \alpha)
\end{array}
$$

where $[\![\$A]\!] = \mathcal{A}_\nu(\texttt{\$T} \triangleright \mathcal{A}_\alpha(\Sigma,\ \bar{\nu}_\mathrm{K} \triangleright \alpha))$
$[\![\$R]\!] = \mathcal{A}_\nu(\texttt{\$T} \triangleright \nu_\mathrm{R})$

In all of these cases, an MMIX rendition of the operand atom is first obtained using a recursive application of the algorithm $\mathcal{C}_\alpha$, binding that operand's value to a temporary MMIX variable chosen for the atom by the earlier definitions of $\mathcal{A}_\alpha$ and $\mathcal{A}_\nu$. The scratch register **$T** is used to hold both the value of α and the result of the entire operation if either of these happens to be allocated by $\mathcal{A}_\alpha$ to a pseudo-register. Once the desired result has been computed by the selected instruction, its output is copied into the specified MMIX variable ν_R, whereby it becomes available to the rest of the translated program. Formally:

$$\begin{aligned}
&\mathcal{C}_\mathrm{U}[\![\cdot]\!] :: (\text{ord } \upsilon) \Rightarrow \\
&\quad (\upsilon \mapsto \textit{MMIX-var}, \{\textit{MMIX-var}\}, \textit{MMIX-var}, \textit{MMIX-opcode}, \textit{integer} \triangleright \textit{atom}_\upsilon) \rightarrow \\
&\quad [\textit{MMIX-instr}]
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_\mathrm{U}[\![\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R}, \mathit{OP}, Y \triangleright \alpha]\!] = {} & \mathcal{C}_\alpha(\Sigma, \bar{\nu}_\mathrm{K}, \texttt{\$}A \triangleright \alpha) +\!\!+ \\
& [\![\mathit{OP}\ \texttt{\$}R\texttt{,}\ Y\texttt{,}\ \texttt{\$}A]\!] +\!\!+ \\
& \mathcal{C}_\nu(\nu_\mathrm{R} \triangleright \texttt{\$}R)
\end{aligned}$$

$$\begin{aligned}
\text{where } [\![\texttt{\$}A]\!] &= \mathcal{A}_\nu(\texttt{\$T} \triangleright \mathcal{A}_\alpha(\Sigma, \bar{\nu}_\mathrm{K} \triangleright \alpha)) \\
[\![\texttt{\$}R]\!] &= \mathcal{A}_\nu(\texttt{\$T} \triangleright \nu_\mathrm{R})
\end{aligned}$$

All other unary Etude operations of the form "$\phi'_\phi(\alpha)$" describe conversions of a rational number into some non-standard integral format, or else a conversion between a pair of such integral formats. In the first case, the operation is enacted in the translated program through an intermediate conversion of α into the corresponding standard format "N.64" or "Z.64", as depicted by the Etude term "$\phi'_{\phi''}(\phi''_\phi(\alpha))$", in which ϕ'' represents a standard format of the same genre as ϕ'. In the later case, we observe that, by the virtue of their semantic interpretation in Section 6.3.1, every such conversion is essentially equivalent to $\alpha \bmod 2^{\mathcal{W}(\phi')}$, which can be also rephrased as $\lfloor((\alpha \times 2^{64 - \mathcal{W}(\phi')}) \bmod 2^{64})/2^{64 - \mathcal{W}(\phi')}\rfloor$. The later variant of this expression can be easily compiled into an efficient MMIX instruction sequence through the following intermediate translation into a pattern of Etude bit shift operations:

$$\begin{aligned}
&\mathcal{C}_\alpha[\![\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R} \triangleright \phi'_\phi(\alpha)]\!] \\
&\quad \mid (\gamma' \in \{\mathsf{N}, \mathsf{Z}\} \wedge \gamma = [\![\mathsf{R}]\!]) = \mathcal{C}_\alpha[\![\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R} \triangleright \phi'_{\phi''}(\phi''_\phi(\alpha))]\!] \\
&\quad \mid \text{otherwise} = \mathcal{C}_\alpha[\![\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R} \triangleright (\alpha \ll_{\mathrm{N}.64} \#[\![64 - \varepsilon']\!]_{\mathrm{N}.64}) \gg_{\phi''} \#[\![64 - \varepsilon']\!]_{\mathrm{N}.64}]\!]
\end{aligned}$$

$$\begin{aligned}
\text{where } [\![\gamma.\varepsilon]\!] &= \phi \\
[\![\gamma'.\varepsilon']\!] &= \phi' \\
\phi'' &= [\![\gamma'.64]\!]
\end{aligned}$$

All of the remaining atomic forms that are defined by Etude pertain to applications of its various binary operators. Each of these operators is naturally associated with a predetermined MMIX opcode as captured by the following translation:

$$\begin{aligned}
\mathcal{C}_\alpha[\![\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R} \triangleright \alpha_1 +_{\mathrm{R}.\varepsilon} \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R}, \texttt{FADD} \triangleright \alpha_1, \alpha_2) \\
\mathcal{C}_\alpha[\![\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R} \triangleright \alpha_1 -_{\mathrm{R}.\varepsilon} \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R}, \texttt{FSUB} \triangleright \alpha_1, \alpha_2) \\
\mathcal{C}_\alpha[\![\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R} \triangleright \alpha_1 \times_{\mathrm{R}.\varepsilon} \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R}, \texttt{FMUL} \triangleright \alpha_1, \alpha_2) \\
\mathcal{C}_\alpha[\![\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R} \triangleright \alpha_1 \div_{\mathrm{R}.\varepsilon} \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma, \bar{\nu}_\mathrm{K}, \nu_\mathrm{R}, \texttt{FDIV} \triangleright \alpha_1, \alpha_2)
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ +_{\mathrm{N}.64}\ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{ADDU} \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ -_{\mathrm{N}.64}\ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{SUBU} \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ \times_{\mathrm{N}.64}\ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{MULU} \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ \div_{\mathrm{N}.\varepsilon}\ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{DIVU} \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ \div_{\mathrm{Z}.\varepsilon}\ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{DIV}\ \ \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ \ll_{\mathrm{N}.64}\ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{SLU}\ \ \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ \triangle_\phi\ \ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{AND}\ \ \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ \bar{\triangledown}_\phi\ \ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{XOR}\ \ \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ \triangledown_\phi\ \ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{OR}\ \ \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ \gg_{\mathrm{N}.\varepsilon}\ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{SRU}\ \ \triangleright \alpha_1,\ \alpha_2)\\
\mathcal{C}_\alpha[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R} \triangleright \alpha_1\ \gg_{\mathrm{Z}.\varepsilon}\ \alpha_2]\!] &= \mathcal{C}_\mathrm{A}(\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \texttt{SR}\ \ \triangleright \alpha_1,\ \alpha_2)
\end{aligned}$$

All of these atomic forms are invariably compiled into almost identical sequences of MMIX instructions, which are distinguished only in their choice of a specific opcode *OP* for enactment of the desired computational effect. In particular, the resulting program always beings by evaluating the first operand α_1 and binding its result to an appropriate MMIX variable v_A that has been designated for the purpose by $\mathcal{A}_\alpha$. In order to preserve the precise semantics of the following operand α_2, all Etude variables that appear free within α_2 are added to the set of live variables $\bar{v}_\mathrm{KA}$ during translation of α_1. Next, α_2 is likewise evaluated and bound to some general purpose hardware register $\texttt{\$}B$, equal to the atom's allotted variable v_B if v_B is implemented in hardware, or else to the scratch register $\texttt{\$}U$ if all such hardware registers have been already assigned to other rôles by the program. Within α_2, the set of live variables is extended with v_A, in order to preserve that variable's value during the atom's evaluation. Subsequently, the compiler ensures that the value of v_A is available in a hardware register, using the scratch space of $\texttt{\$}T$ if necessary. Finally, the specified MMIX instruction *OP* is applied to the values of the two operands and the result of the entire construction is copied into the desired MMIX variable v_R, mediating it through $\texttt{\$}T$ if deemed necessary by the register allocator. Formally:

$$\begin{aligned}
&\mathcal{C}_\mathrm{A}[\![\cdot]\!] :: (\mathrm{ord}\ \upsilon) \Rightarrow\\
&\qquad (\upsilon \mapsto \textit{MMIX-var},\ \{\textit{MMIX-var}\},\ \textit{MMIX-var},\ \textit{MMIX-opcode} \triangleright \textit{atom}_\upsilon,\ \textit{atom}_\upsilon) \rightarrow\\
&\qquad [\textit{MMIX-instr}]
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_\mathrm{A}[\![\Sigma,\ \bar{v}_\mathrm{K},\ v_\mathrm{R},\ \mathit{OP} \triangleright \alpha_1,\ \alpha_2]\!] = {}& \mathcal{C}_\alpha(\Sigma,\ \bar{v}_\mathrm{KA},\ v_\mathrm{A} \triangleright \alpha_1) +\!\!+\\
& \mathcal{C}_\alpha(\Sigma,\ \bar{v}_\mathrm{KB},\ \texttt{\$}B \triangleright \alpha_2) +\!\!+\\
& \mathcal{C}_v(\texttt{\$}A \triangleright v_\mathrm{A}) +\!\!+\\
& [\![\mathit{OP}\ \texttt{\$}R,\ \texttt{\$}A,\ \texttt{\$}B]\!] +\!\!+\\
& \mathcal{C}_v(v_\mathrm{R} \triangleright \texttt{\$}R)
\end{aligned}$$

$$\begin{aligned}
\text{where } [\![\texttt{\$}A]\!] &= \mathcal{A}_v(\texttt{\$T} \triangleright v_\mathrm{A})\\
[\![\texttt{\$}B]\!] &= \mathcal{A}_v(\texttt{\$U} \triangleright v_\mathrm{B})\\
[\![\texttt{\$}R]\!] &= \mathcal{A}_v(\texttt{\$T} \triangleright v_\mathrm{R})\\
v_\mathrm{A} &= \mathcal{A}_\alpha(\Sigma,\ \bar{v}_\mathrm{KA} \triangleright \alpha_1)\\
v_\mathrm{B} &= \mathcal{A}_\alpha(\Sigma,\ \bar{v}_\mathrm{KB} \triangleright \alpha_2)\\
\bar{v}_\mathrm{KA} &= \bar{v}_\mathrm{K} \cup \{\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \mathrm{FV}(\alpha_2)\}\\
\bar{v}_\mathrm{KB} &= \bar{v}_\mathrm{K} \cup \{v_\mathrm{A}\}
\end{aligned}$$

Similarly, the translation of signed and unsigned integer remainder operations is en-

acted using an appropriate MMIX division instruction "**DIVU**" or "**DIV**":

$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \cdot|\cdot_{\mathrm{N}.\varepsilon}\, \alpha_2]\!] = \mathcal{C}_\mathrm{B}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{DIVU} \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \cdot|\cdot_{\mathrm{Z}.\varepsilon}\, \alpha_2]\!] = \mathcal{C}_\mathrm{B}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{DIV}\ \ \triangleright \alpha_1,\, \alpha_2)$$

The precise translation of these two atomic forms is identical to the earlier treatment of other binary arithmetic operations, except that the ultimate result of the entire construct can be found in the special purpose register **rR** rather than its general purpose counterpart **$***R* that was featured in the definition of $\mathcal{C}_\mathrm{A}$:

$$\begin{aligned}
&\mathcal{C}_\mathrm{B}[\![\cdot]\!] :: (\mathrm{ord}\ \upsilon) \Rightarrow \\
&\qquad (\upsilon \mapsto \textit{MMIX-var},\, \{\textit{MMIX-var}\},\, \textit{MMIX-var},\, \textit{MMIX-opcode} \triangleright \textit{atom}_\upsilon,\, \textit{atom}_\upsilon) \to \\
&\qquad [\textit{MMIX-instr}]
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_\mathrm{B}[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \mathit{OP} \triangleright \alpha_1,\, \alpha_2]\!] = {}& \mathcal{C}_\alpha(\Sigma,\, \bar{\nu}_{\mathrm{KA}},\, \nu_\mathrm{A} \triangleright \alpha_1) +\!\!+ \\
& \mathcal{C}_\alpha(\Sigma,\, \bar{\nu}_{\mathrm{KB}},\, \texttt{\$}B \triangleright \alpha_2) +\!\!+ \\
& \mathcal{C}_\nu(\texttt{\$}A \triangleright \nu_\mathrm{A}) +\!\!+ \\
& [\![\mathit{OP}\ \texttt{\$T, \$}A\texttt{, \$}B]\!] +\!\!+ \\
& \mathcal{C}_\nu(\nu_\mathrm{R} \triangleright \texttt{rR})
\end{aligned}$$

$$\begin{aligned}
\text{where } [\![\texttt{\$}A]\!] &= \mathcal{A}_\nu(\texttt{\$T} \triangleright \nu_\mathrm{A}) \\
[\![\texttt{\$}B]\!] &= \mathcal{A}_\nu(\texttt{\$U} \triangleright \nu_\mathrm{B}) \\
\nu_\mathrm{A} &= \mathcal{A}_\alpha(\Sigma,\, \bar{\nu}_{\mathrm{KA}} \triangleright \alpha_1) \\
\nu_\mathrm{B} &= \mathcal{A}_\alpha(\Sigma,\, \bar{\nu}_{\mathrm{KB}} \triangleright \alpha_2) \\
\bar{\nu}_{\mathrm{KA}} &= \bar{\nu}_\mathrm{K} \cup \{\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \mathrm{FV}(\alpha_2)\} \\
\bar{\nu}_{\mathrm{KB}} &= \bar{\nu}_\mathrm{K} \cup \{\nu_\mathrm{A}\}
\end{aligned}$$

On the other hand, every application of the Etude comparison operator "$=_\phi$", "$\neq_\phi$", "$<_\phi$", "$>_\phi$", "$\leq_\phi$" or "$\geq_\phi$" must be translated into a sequence of two MMIX instructions. First, the actual comparison is performed using the operation "**FCMP**", "**CMPU**" or "**CMP**", as appropriate for the underlying Etude format ϕ. Next, the value of -1, 0 or 1 that each of these instruction forms delivers is converted into the required boolean constant 1 or 0 using one of the six conditional copy instructions "$\texttt{ZSZ}_\texttt{I}$", "$\texttt{ZSNZ}_\texttt{I}$", "$\texttt{ZSN}_\texttt{I}$", "$\texttt{ZSP}_\texttt{I}$", "$\texttt{ZSNP}_\texttt{I}$" or "$\texttt{ZSNN}_\texttt{I}$", as required to achieve the overall effect of the specified relational operation. In particular:

$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 =_{\mathrm{R}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{FCMP},\, \texttt{ZSZ}_\texttt{I}\ \ \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \neq_{\mathrm{R}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{FCMP},\, \texttt{ZSNZ}_\texttt{I} \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 <_{\mathrm{R}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{FCMP},\, \texttt{ZSN}_\texttt{I}\ \ \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 >_{\mathrm{R}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{FCMP},\, \texttt{ZSP}_\texttt{I}\ \ \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \leq_{\mathrm{R}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{FCMP},\, \texttt{ZSNP}_\texttt{I} \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \geq_{\mathrm{R}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{FCMP},\, \texttt{ZSNN}_\texttt{I} \triangleright \alpha_1,\, \alpha_2)$$

$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 =_{\mathrm{N}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMPU},\, \texttt{ZSZ}_\texttt{I}\ \ \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \neq_{\mathrm{N}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMPU},\, \texttt{ZSNZ}_\texttt{I} \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 <_{\mathrm{N}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMPU},\, \texttt{ZSN}_\texttt{I}\ \ \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 >_{\mathrm{N}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMPU},\, \texttt{ZSP}_\texttt{I}\ \ \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \leq_{\mathrm{N}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMPU},\, \texttt{ZSNP}_\texttt{I} \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \geq_{\mathrm{N}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMPU},\, \texttt{ZSNN}_\texttt{I} \triangleright \alpha_1,\, \alpha_2)$$

$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 =_{\mathrm{Z}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMP},\ \ \texttt{ZSZ}_\texttt{I}\ \ \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \neq_{\mathrm{Z}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMP},\ \ \texttt{ZSNZ}_\texttt{I} \triangleright \alpha_1,\, \alpha_2)$$
$$\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 <_{\mathrm{Z}.\varepsilon} \alpha_2]\!] = \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMP},\ \ \texttt{ZSN}_\texttt{I}\ \ \triangleright \alpha_1,\, \alpha_2)$$

$$\begin{aligned}
\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 >_{\mathrm{Z}.\varepsilon} \alpha_2]\!] &= \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMP},\quad \texttt{ZSP}_\texttt{I} \triangleright \alpha_1,\, \alpha_2) \\
\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \leq_{\mathrm{Z}.\varepsilon} \alpha_2]\!] &= \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMP},\quad \texttt{ZSNP}_\texttt{I} \triangleright \alpha_1,\, \alpha_2) \\
\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1 \geq_{\mathrm{Z}.\varepsilon} \alpha_2]\!] &= \mathcal{C}_\mathrm{C}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, \texttt{CMP},\quad \texttt{ZSNN}_\texttt{I} \triangleright \alpha_1,\, \alpha_2)
\end{aligned}$$

where the actual compilation of these atoms is performed by the following Haskell construction $\mathcal{C}_\mathrm{C}$, whose structure follows the same general principles as its earlier cousins $\mathcal{C}_\mathrm{A}$ and $\mathcal{C}_\mathrm{B}$:

$$\begin{aligned}
&\mathcal{C}_\mathrm{C}[\![\cdot]\!] :: (\mathrm{ord}\ \upsilon) \Rightarrow \\
&\qquad (\upsilon \mapsto \textit{MMIX-var},\, \{\textit{MMIX-var}\},\, \textit{MMIX-var}, \\
&\qquad\quad \textit{MMIX-opcode},\, \textit{MMIX-opcode} \triangleright \textit{atom}_\upsilon,\, \textit{atom}_\upsilon) \rightarrow \\
&\qquad [\textit{MMIX-instr}]
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_\mathrm{C}[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R},\, OP_1,\, OP_2 \triangleright \alpha_1,\, \alpha_2]\!] = {}& \mathcal{C}_\alpha(\Sigma,\, \bar{\nu}_{\mathrm{KA}},\, \nu_\mathrm{A} \triangleright \alpha_1) +\!\!+ \\
& \mathcal{C}_\alpha(\Sigma,\, \bar{\nu}_{\mathrm{KB}},\, \texttt{\$}B \triangleright \alpha_2) +\!\!+ \\
& \mathcal{C}_\nu(\texttt{\$}A \triangleright \nu_\mathrm{A}) +\!\!+ \\
& [\![OP_1\ \texttt{\$}R,\ \texttt{\$}A,\ \texttt{\$}B]\!] +\!\!+ \\
& [\![OP_2\ \texttt{\$}R,\ \texttt{\$}R,\ 1]\!] +\!\!+ \\
& \mathcal{C}_\nu(\nu_\mathrm{R} \triangleright \texttt{\$}R)
\end{aligned}$$

$$\begin{aligned}
\text{where } [\![\texttt{\$}A]\!] &= \mathcal{A}_\nu(\texttt{\$T} \triangleright \nu_\mathrm{A}) \\
[\![\texttt{\$}B]\!] &= \mathcal{A}_\nu(\texttt{\$U} \triangleright \nu_\mathrm{B}) \\
[\![\texttt{\$}R]\!] &= \mathcal{A}_\nu(\texttt{\$T} \triangleright \nu_\mathrm{R}) \\
\nu_\mathrm{A} &= \mathcal{A}_\alpha(\Sigma,\, \bar{\nu}_{\mathrm{KA}} \triangleright \alpha_1) \\
\nu_\mathrm{B} &= \mathcal{A}_\alpha(\Sigma,\, \bar{\nu}_{\mathrm{KB}} \triangleright \alpha_2) \\
\bar{\nu}_{\mathrm{KA}} &= \bar{\nu}_\mathrm{K} \cup \{\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \mathrm{FV}(\alpha_2)\} \\
\bar{\nu}_{\mathrm{KB}} &= \bar{\nu}_\mathrm{K} \cup \{\nu_\mathrm{A}\}
\end{aligned}$$

Finally, the four Etude operators "$+_\phi$", "$-_\phi$", "$\times_\phi$" and "$\ll_\phi$" can be also applied under non-standard integral formats. Observing that all of these operators implement pure modulo arithmetic, their MMIX representation can be obtained trivially by evaluating the operation under the standard natural format "N.64" and converting the resulting value into the desired precision of ϕ:

$$\begin{aligned}
&\mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \alpha_1\ op_\phi\ \alpha_2]\!] \mid op \in \{+,\, -,\, \times,\, \ll\} \wedge \gamma(\phi) \in \{\mathsf{N},\, \mathsf{Z}\} \\
&\qquad = \mathcal{C}_\alpha[\![\Sigma,\, \bar{\nu}_\mathrm{K},\, \nu_\mathrm{R} \triangleright \phi_{\mathrm{N}.64}(\alpha_1\ op_{\mathrm{N}.64}\ \alpha_2)]\!]
\end{aligned}$$

Last but not least, a list of Etude atoms can be compiled into a sequence of semantically equivalent MMIX instructions by the following recursive algorithm $\mathcal{C}_{\bar{\alpha}}$:

$$\begin{aligned}
&\mathcal{C}_{\bar{\alpha}}[\![\cdot]\!] :: (\mathrm{ord}\ \upsilon) \Rightarrow \\
&\qquad (\upsilon \mapsto \textit{MMIX-var},\, \{\textit{MMIX-var}\},\, [\textit{MMIX-var}] \triangleright \textit{atoms}_\upsilon) \rightarrow \\
&\qquad [\textit{MMIX-instr}]
\end{aligned}$$

Specifically, $\mathcal{C}_{\bar{\alpha}}(\Sigma,\, \bar{\nu}_\mathrm{K},\, \bar{\nu}_\mathrm{R} \triangleright \bar{\alpha})$ successively compiles each of the supplied atoms α_k in $\bar{\alpha}$, binding their respective results to the corresponding MMIX variables from $\bar{\nu}_\mathrm{R}$. During translation of each α_k, the existing value of the targeted register ν_R is saved in a temporary MMIX variable ν'_R if ν_R appears free within the remainder of the list, in which case the binding of the corresponding Etude variable υ_k is also updated in the substitution function Σ. The chosen temporary variable is also added to the atom's set of live register $\bar{\nu}_\mathrm{K}$, except that the actual target of the assignment ν_R is never included

in $\bar{v}_K$, in order to allow subsequent atoms in the list to reuse its preallocated value. Formally:

$$\begin{aligned}
\mathcal{C}_{\bar{\alpha}}[\![\Sigma,\ \bar{v}_K,\ \bar{v}_R \triangleright \bar{\alpha}]\!] \mid \bar{\alpha} \neq \varnothing \quad &= \mathcal{C}_v(v'_R \triangleright v_R) +\!\!+ \\
&\quad \mathcal{C}_\alpha(\Sigma,\ \bar{v}'_K \cup \{v'_R\} \backslash \{v_R\},\ v_R \triangleright \mathrm{head}(\bar{\alpha})) +\!\!+ \\
&\quad \mathcal{C}_{\bar{\alpha}}(\Sigma',\ \bar{v}_K,\ \mathrm{tail}(\bar{v}_R) \triangleright \mathrm{tail}(\bar{\alpha})) \\
\mid \text{otherwise} = [\![]\!] & \\
\text{where } v_R = \mathrm{head}(\bar{v}_R) & \\
v'_R = \mathrm{head}[v_k \mid v_k \leftarrow v_R +\!\!+ \bar{v}_G,\ v_k \notin \bar{v}'_K] & \\
\bar{v}'_K = \bar{v}_K \cup \{\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \mathrm{FV}(\mathrm{tail}(\bar{\alpha}))\} & \\
\Sigma' = \Sigma / \{\upsilon_k : v'_R \mid \upsilon_k \leftarrow \mathrm{FV}(\mathrm{tail}(\bar{\alpha})),\ \Sigma(\upsilon_k) = v_R\} &
\end{aligned}$$

THEOREM 6-35: *(Preservation of atom semantics under compilation)* Let Σ be a variable renaming function and Γ represent the mapping of all relevant MMIX variables to their respective atomic values. Let α be an Etude atom that is closed and well-formed under the substitution by $\Gamma \circ \Sigma$. Further, let Λ' represent a function environment updated with bindings of n successive instruction addresses that begin at some Etude function variable v to the respective n instructions generated from the compilation of α and let Δ' represent an object environment that is likewise updated with bindings of any symbolic addresses `LOC`.$8k$ to the current values of all pseudo-registers `$`k which appear free within α. Then, every application of v to a the standard parameter list π under a substitution by Γ is always equivalent to an application of the function $v \oplus 4n$ to the same parameter list under a substitution by an updated finite map $\Gamma / \{\Gamma(\Sigma(\upsilon)) : \alpha\}$. Formally:

$$\begin{aligned}
&\mathrm{LC}_\alpha :: \forall\ \Sigma,\ \Gamma,\ \Gamma',\ \Lambda,\ \Lambda',\ \Delta,\ \Delta',\ v,\ \upsilon,\ \alpha,\ \bar{v}_K,\ \upsilon \Rightarrow \\
&\quad \mathrm{WF}(\Lambda) \rightarrow \mathrm{WF}(\Delta) \rightarrow \mathrm{WF}(\alpha / (\Gamma \circ \Sigma)) \rightarrow \\
&\quad [\![\Gamma' = \Gamma / \{\Gamma(\Sigma(\upsilon)) : \alpha\}]\!] \rightarrow \\
&\quad [\![\Lambda' = \Lambda / \mathcal{D}_{\mathtt{I}}(\Gamma,\ v \triangleright \mathcal{C}_\alpha(\Sigma,\ \bar{v}_K,\ \Sigma(\upsilon) \triangleright \alpha))]\!] \rightarrow \\
&\quad [\![\Delta' = \Delta / \{(\Gamma(\mathtt{LOC}.8k) : \Gamma(\$k)) \mid [\![\$k]\!] \leftarrow [\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \mathrm{FV}(\alpha)],\ k \geq 256\}]\!] \rightarrow \\
&\quad [\![\Lambda',\ \Delta' \triangleright [\![\Gamma(v)(\pi)]\!] / \Gamma]\!] \equiv [\![\Lambda,\ \Delta \triangleright [\![[\![\Gamma(v \oplus \mathrm{length}(\mathcal{C}_\alpha(\Sigma,\ \bar{v}_K,\ \Sigma(\upsilon) \triangleright \alpha)))]\!](\pi)]\!] / \Gamma']\!]
\end{aligned}$$

PROOF: If α represents a single Etude register or a constant, then the above theorem follows directly from the earlier result LC_v, which provides us with a convenient base case for the subsequent induction proof. Otherwise, if the atom represents one of the fifteen binary operations compiled through the construction $\mathcal{C}_A$, then:

$$\begin{aligned}
\mathcal{C}_\alpha(\Sigma,\ \bar{v}_K,\ \Sigma(\upsilon) \triangleright \alpha) = \mathcal{C}_\alpha(\Sigma,\ \bar{v}_{KA},\ v_A \triangleright \alpha_1) +\!\!+ \\
\mathcal{C}_\alpha(\Sigma,\ \bar{v}_{KB},\ \$B \triangleright \alpha_2) +\!\!+ \\
\mathcal{C}_v(\$A \triangleright v_A) +\!\!+ \\
[\![OP\ \$R,\ \$A,\ \$B]\!] +\!\!+ \\
\mathcal{C}_v(v_R \triangleright \$R))
\end{aligned}$$

If we now apply LC_α inductively to the first two constructs in the above term and, further, utilise LC_v to similarly eliminate the third construct, then the resulting instruction sequence can be simplified into "$[\![OP\ \$R,\ \$A,\ \$B]\!] +\!\!+ \mathcal{C}_v(v_R \triangleright \$R)$", to be taken in the context of Γ updated with the set of bindings $\{v_A : \alpha_1,\ \$A : \alpha_1,\ \$B : \alpha_2\}$. Observing that the above instruction sequence would be translated by $\mathcal{D}_{\mathtt{I}}$ into an Etude function with

the body of "LET $R = $A $+_{R.64}$ $B; ... $[\![\nu \oplus n]\!](\pi)$", an application of this term to π under $\pi/\{\nu_A{:}\alpha_1, \$A{:}\alpha_1, \$B{:}\alpha_2\}$ is clearly equivalent to the term's body "$[\![\nu \oplus n]\!](\pi)$", which matches the right-hand side of the desired equivalence form. □

6.4.3 Functions and Terms

Armed with the above atom compiler, we are now ready to proceed with the translation of actual Etude functions. As already mentioned at the beginning of this section, every Etude construct of the form "$\lambda\bar{\upsilon}.\tau$" materialises in the eventual MMIX program as a collection of one or more MMIX instructions, which are generated from the function's body using the term compiler $\mathcal{C}_\kappa$ that will be described shortly. In particular, at the beginning of every such entity, the supplied substitution function Σ is updated with bindings of all Etude variables $\bar{\upsilon}$ to their respective MMIX analogues found in the standard parameter list $\bar{\nu}_P$ as defined earlier in Section 6.4.1. Formally:

$$\mathcal{C}_\lambda[\![\cdot]\!] :: (\text{ord } \upsilon) \Rightarrow (\upsilon \mapsto \textit{MMIX-var} \triangleright \textit{function}_\upsilon) \to [\textit{MMIX-instr}]$$
$$\mathcal{C}_\lambda[\![\Sigma \triangleright \lambda\bar{\upsilon}.\tau]\!] = \mathcal{C}_\kappa(\Sigma/\bar{\upsilon}|\bar{\nu}_P \triangleright \tau)$$

In the following translation, a term which appears outside of any "LET" bindings is known as a *tail term*. In the eventual MMIX program, the concrete rendition of all such terms is obtained by the following Haskell function $\mathcal{C}_\kappa$:

$$\mathcal{C}_\kappa[\![\cdot]\!] :: (\text{ord } \upsilon) \Rightarrow (\upsilon \mapsto \textit{MMIX-var} \triangleright \textit{term}_\upsilon) \to [\textit{MMIX-instr}]$$

In particular, every tail term ends in a jump to the function's continuation address, which, by construction, is always made available to the program as a value of the designated variable **$K**. Under the following translation, such jumps are always depicted by the MMIX instruction "**GO $X, $Y,** 0", which, as described earlier in Section 6.2, transfers control to the location within the program's address space specified by the general purpose register $Y, after saving the address of the following instruction in $X. Since, in tail terms, this later address is of no use to the function's caller, we will always place it in the scratch variable **$T**, whereby it will be promptly discarded by some following instruction which finds a better use for that register.

The actual results of the entire Etude function are always placed in the successive registers $0, $1, $2 and so on, as represented by the MMIX variable list $\bar{\nu}_P$. Accordingly, a simple tail term of the form "RET ($\bar{\alpha}$)" is compiled directly into an MMIX representation of the supplied atoms $\bar{\alpha}$, whose respective values are placed into the corresponding variables from $\bar{\nu}_P$. On the other hand, the computational effect of all "NEW", "DEL", "GET", "SET" and "SET$_I$" term forms must be obtained using another algorithm $\mathcal{C}_\tau$ that is defined later in this section. Their individual translations are then completed with the same operation "**GO $T, $K,** 0". In Haskell:

$$\begin{aligned}
\mathcal{C}_\kappa[\![\Sigma \triangleright \text{RET } (\bar{\alpha})]\!] &= \mathcal{C}_\tau[\![\Sigma, \varnothing, \text{take}(\text{length}(\bar{\alpha}), \bar{\nu}_P) \triangleright \text{RET } (\bar{\alpha})]\!] +\!\!+ [\![\textbf{GO}_I\ \textbf{\$T, \$K,}\ 0]\!] \\
\mathcal{C}_\kappa[\![\Sigma \triangleright \text{NEW } (\bar{\xi})]\!] &= \mathcal{C}_\tau[\![\Sigma, \varnothing, \$0 \triangleright \text{NEW } (\bar{\xi})]\!] +\!\!+ [\![\textbf{GO}_I\ \textbf{\$T, \$K,}\ 0]\!] \\
\mathcal{C}_\kappa[\![\Sigma \triangleright \text{DEL } (\bar{\xi})]\!] &= \mathcal{C}_\tau[\![\Sigma, \varnothing, \varnothing \triangleright \text{DEL } (\bar{\xi})]\!] +\!\!+ [\![\textbf{GO}_I\ \textbf{\$T, \$K,}\ 0]\!] \\
\mathcal{C}_\kappa[\![\Sigma \triangleright \text{GET } [\alpha, \bar{\mu}]_\phi]\!] &= \mathcal{C}_\tau[\![\Sigma, \varnothing, \$0 \triangleright \text{GET } [\alpha, \bar{\mu}]_\phi]\!] +\!\!+ [\![\textbf{GO}_I\ \textbf{\$T, \$K,}\ 0]\!]
\end{aligned}$$

$$\mathcal{C}_\kappa[\![\Sigma \triangleright \textsc{set}\ [\alpha_1,\ \bar{\mu}]_\phi\ \textsc{to}\ \alpha_2]\!] = \mathcal{C}_\tau[\![\Sigma,\ \varnothing,\ \varnothing\ \triangleright \textsc{set}\ [\alpha_1,\ \bar{\mu}]_\phi\ \textsc{to}\ \alpha_2]\!] +\!\!+ [\![\texttt{GO}_\texttt{I}\ \texttt{\$T,}\ \texttt{\$K,}\ 0]\!]$$
$$\mathcal{C}_\kappa[\![\Sigma \triangleright \textsc{set}_\text{i}\ [\alpha_1,\ \bar{\mu}]_\phi\ \textsc{to}\ \alpha_2]\!] = \mathcal{C}_\tau[\![\Sigma,\ \varnothing,\ \varnothing\ \triangleright \textsc{set}_\text{i}\ [\alpha_1,\ \bar{\mu}]_\phi\ \textsc{to}\ \alpha_2]\!] +\!\!+ [\![\texttt{GO}_\texttt{I}\ \texttt{\$T,}\ \texttt{\$K,}\ 0]\!]$$

The core meaning of an Etude tail call operation "$\alpha(\bar{\alpha})$" is likewise represented by a similar "$\texttt{GO}_\texttt{I}$" instruction. However, in this case, the targeted Etude function is to be found at a memory location depicted by the construct's initial operand α. Accordingly, we begin by compiling all of the supplied atoms and allocating them to the successive variables form $\bar{v}_\text{P}$, where the eventual program expects to find their respective values. Next, we ensure that the actual address of the target instruction has been placed in a real hardware register (say, $\texttt{\$}R$), after which the "$\texttt{GO}_\texttt{I}$" instruction can be applied directly to that register as follows:

$$\begin{aligned}\mathcal{C}_\kappa[\![\Sigma \triangleright \alpha(\bar{\alpha})]\!] &= \mathcal{C}_{\bar{\alpha}}(\Sigma,\ \varnothing,\ \bar{v}_\text{R} \triangleright \bar{\alpha} +\!\!+ [\alpha]) +\!\!+ \mathcal{C}_v(\texttt{\$}R \triangleright \text{last}(\bar{v}_\text{R})) +\!\!+ [\![\texttt{GO}_\texttt{I}\ \texttt{\$T,}\ \texttt{\$}R\texttt{,}\ 0]\!]\\ \text{where } [\![\texttt{\$}R]\!] &= \mathcal{A}_v(\texttt{\$T} \triangleright \text{last}(\bar{v}_\text{R}))\\ \bar{v}_\text{R} &= \text{take}(\text{length}(\bar{\alpha}) + 1,\ \bar{v}_\text{P})\end{aligned}$$

A suitable MMIX rendition of a system call operation "$\pi(\bar{\alpha})$" is obtained similarly, except that, instead of an actual memory address, the jump materialises as an MMIX "$\texttt{TRAP}$" instruction, whose three operands X, Y and Z are used to encode the 24-bit system call number π. Once control has been relinquished by the operating system, the program proceeds with a jump to the prevailing continuation address $\texttt{\$K}$:

$$\begin{aligned}\mathcal{C}_\kappa[\![\Sigma \triangleright \pi(\bar{\alpha})]\!] = \mathcal{C}_{\bar{\alpha}}(\Sigma,\ \varnothing,\ \text{take}(\text{length}(\bar{\alpha}),\ \bar{v}_\text{P}) \triangleright \bar{\alpha}) +\!\!+\\ [\![\texttt{TRAP}\ [\![\pi[16\ldots23]]\!]\texttt{,}\ [\![\pi[8\ldots15]]\!]\texttt{,}\ [\![\pi[0\ldots7]]\!]\texttt{;}\\ \texttt{GO}_\texttt{I}\ \texttt{\$T,}\ \texttt{\$K,}\ 0]\!]\end{aligned}$$

On the other hand, every conditional Etude term of the form "$\textsc{if}\ \alpha\ \textsc{then}\ \tau_1\ \textsc{else}\ \tau_2$" must choose between one of the two available targets τ_1 or τ_2. Either way, both of these terms are compiled into their respective instruction sequences $\bar{\imath}_1$ and $\bar{\imath}_2$. If the first of these sequences is short enough (specifically, if it contains fewer than 2^{16} instructions), then the branch target can be encoded verbatim within the Y and Z operands of an MMIX "$\texttt{BZ}$" operation. Otherwise, the desired target address must be synthesised indirectly, which, in the following translation, is performed by the Haskell function $\mathcal{C}_\text{J}$. Formally:

$$\begin{aligned}\mathcal{C}_\kappa[\![\Sigma \triangleright \textsc{if}\ \alpha\ \textsc{then}\ \tau_1\ \textsc{else}\ \tau_2]\!] \mid (\delta < 2^{18}) &= \mathcal{C}_\alpha(\Sigma,\ \bar{v}_\text{K},\ \texttt{\$}R \triangleright \alpha) +\!\!+\\ &\quad [\![\texttt{BZ}\ \texttt{\$}R\texttt{,}\ [\![\delta[10\ldots17]]\!]\texttt{,}\ [\![\delta[2\ldots9]]\!]]\!] +\!\!+\\ &\quad \bar{\imath}_1 +\!\!+ \bar{\imath}_2\\ \mid \text{otherwise} &= \mathcal{C}_\alpha(\Sigma,\ \bar{v}_\text{K},\ \texttt{\$}R \triangleright \alpha) +\!\!+\\ &\quad [\![\texttt{BZ}\ \texttt{\$}R\texttt{,}\ 0\texttt{,}\ 3]\!] +\!\!+\\ &\quad \mathcal{C}_\text{J}(4(\text{length}(\bar{\imath}_2) + 1)) +\!\!+\\ &\quad \bar{\imath}_2 +\!\!+ \bar{\imath}_1\end{aligned}$$
$$\begin{aligned}\text{where } [\![\texttt{\$}R]\!] &= \mathcal{A}_v(\texttt{\$T} \triangleright \mathcal{A}_\alpha(\Sigma,\ \bar{v}_\text{K} \triangleright \alpha))\\ \bar{v}_\text{K} &= \{\Sigma(v_k) \mid v_k \leftarrow \text{FV}(\tau_1) \cup \text{FV}(\tau_2)\}\\ \bar{\imath}_1 &= \mathcal{C}_\kappa(\Sigma \triangleright \tau_1)\\ \bar{\imath}_2 &= \mathcal{C}_\kappa(\Sigma \triangleright \tau_2)\\ \delta &= 4(\text{length}(\bar{\imath}_1) + 1)\end{aligned}$$

In order to compile a jump to such a remote address that is found n bytes away from its source, $\mathcal{C}_\text{J}(n)$ begins by obtaining the location of the current instruction, which,

on MMIX, can be achieved easily enough using the "**GETA**" opcode. Conveniently, "**GETA**" also allows us to increment that address by the 16 least significant bits of n. The remaining bits must be added to the resulting value using the familiar sequence of "**INCML**", "**INCMH**" and "**INCH**" operations. Once the entire address has been prepared in **$T**, the program proceeds with the unconditional control transfer to that location, enacted in the usual manner by an appropriate "**GO**$_{\mathtt{I}}$" instruction. The Z operand of "**GO**$_{\mathtt{I}}$", which, in the semantic translation from Section 6.2, was always added to the supplied target register value, is used to perform the final adjustment of the synthesised address by the number of instructions introduced internally within $\mathcal{C}_\mathrm{J}$ itself:

$$\mathcal{C}_\mathrm{J}[\![\cdot]\!] :: \mathit{integer} \to [\mathit{MMIX\text{-}instr}]$$

$$\begin{aligned}
\mathcal{C}_\mathrm{J}[\![\delta]\!] \mid (\delta < 2^{32}) = [\![&\mathtt{GETA}\ \ \mathtt{\$T},\ [\![\delta[10 \mathinner{..} 15]]\!],\ [\![\delta[2 \mathinner{..} 9]]\!]\\
&\mathtt{INCML}\ \mathtt{\$T},\ [\![\delta[24 \mathinner{..} 31]]\!],\ [\![\delta[16 \mathinner{..} 23]]\!]\\
&\mathtt{GO}_{\mathtt{I}}\ \ \ \mathtt{\$T},\ \mathtt{\$T},\ 12]\!]\\
\mid (\delta < 2^{48}) = [\![&\mathtt{GETA}\ \ \mathtt{\$T},\ [\![\delta[10 \mathinner{..} 15]]\!],\ [\![\delta[2 \mathinner{..} 9]]\!]\\
&\mathtt{INCML}\ \mathtt{\$T},\ [\![\delta[24 \mathinner{..} 31]]\!],\ [\![\delta[16 \mathinner{..} 23]]\!]\\
&\mathtt{INCMH}\ \mathtt{\$T},\ [\![\delta[40 \mathinner{..} 47]]\!],\ [\![\delta[32 \mathinner{..} 39]]\!]\\
&\mathtt{GO}_{\mathtt{I}}\ \ \ \mathtt{\$T},\ \mathtt{\$T},\ 16]\!]\\
\mid \text{otherwise} = [\![&\mathtt{GETA}\ \ \mathtt{\$T},\ [\![\delta[10 \mathinner{..} 15]]\!],\ [\![\delta[2 \mathinner{..} 9]]\!]\\
&\mathtt{INCML}\ \mathtt{\$T},\ [\![\delta[24 \mathinner{..} 31]]\!],\ [\![\delta[16 \mathinner{..} 23]]\!]\\
&\mathtt{INCMH}\ \mathtt{\$T},\ [\![\delta[40 \mathinner{..} 47]]\!],\ [\![\delta[32 \mathinner{..} 39]]\!]\\
&\mathtt{INCH}\ \ \mathtt{\$T},\ [\![\delta[56 \mathinner{..} 63]]\!],\ [\![\delta[48 \mathinner{..} 55]]\!]\\
&\mathtt{GO}_{\mathtt{I}}\ \ \ \mathtt{\$T},\ \mathtt{\$T},\ 20]\!]
\end{aligned}$$

Finally, a tail term of the form "LET $\bar{\upsilon} = \tau_1; \tau_2$" is depicted by a sequence of MMIX instructions that represent the inner Etude term τ_1, followed by the rendition of the tail's body τ_2. Any results of τ_1 are bound to fresh MMIX variables which otherwise cannot appear free within τ_2. Predictably, during compilation of the actual "LET" body τ_2, all of these variables are substituted for their respective Etude equivalents from $\bar{\upsilon}$, which, in Haskell, can be represented concisely as follows:

$$\begin{aligned}
\mathcal{C}_\kappa[\![\Sigma \triangleright \mathtt{LET}\ \bar{\upsilon} = \tau_1; \tau_2]\!] &= \mathcal{C}_\tau(\Sigma,\ \bar{\nu}_\mathrm{K},\ \mathrm{take}(\mathrm{length}(\bar{\upsilon}),\ \bar{\nu}) \triangleright \tau_1) +\!\!+ \mathcal{C}_\kappa(\Sigma/(\bar{\upsilon}|\bar{\nu}) \triangleright \tau_2)\\
\text{where } \bar{\nu}_\mathrm{K} &= \{\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \mathrm{FV}(\tau_2) \backslash \bar{\upsilon}\}\\
\bar{\nu} &= [\nu_k \mid \nu_k \leftarrow \bar{\nu}_\mathrm{G},\ \nu_k \notin \bar{\nu}_\mathrm{K}]
\end{aligned}$$

Last but not least, groups of "LET" bindings are compiled naturally through their standard semantic translation into monadic sequences of singular binding forms:

$$\begin{aligned}
\mathcal{C}_\kappa[\![\Sigma \triangleright \mathtt{LET};\ \tau]\!] &= \mathcal{C}_\kappa[\![\Sigma \triangleright \tau]\!]\\
\mathcal{C}_\kappa[\![\Sigma \triangleright \mathtt{LET}\ \bar{\beta};\ \tau]\!] &= \mathcal{C}_\kappa[\![\Sigma \triangleright \mathtt{LET}\ [\![\mathrm{head}(\bar{\beta})]\!];\ \mathtt{LET}\ [\![\mathrm{tail}(\bar{\beta})]\!];\ \tau]\!]
\end{aligned}$$

Within a "LET" binding, the above compilation algorithm must be adjusted slightly, in order to ensure that all Etude variables that are referenced in the remainder of the program retain their original semantic significance under the translation, and that the actual results delivered by the term are saved in the MMIX equivalents of the Etude variables to which these results have been assigned by the programmer. To this end, the translation of such terms in enacted by one final algorithm $\mathcal{C}_\tau$ in our compiler:

$$\mathcal{C}_\tau[\![\cdot]\!] :: (\mathrm{ord}\ \upsilon) \Rightarrow (\upsilon \mapsto \mathit{MMIX\text{-}var},\ \{\mathit{MMIX\text{-}var}\},\ [\mathit{MMIX\text{-}var}] \triangleright \mathit{term}_\upsilon) \to [\mathit{MMIX\text{-}instr}]$$

In the present context, a trivial "RET ($\bar{\alpha}$)" term simply compiles its atoms into the supplied list of MMIX variables $\bar{v}_{\mathrm{R}}$ as follows:

$$\mathcal{C}_\tau[\![\Sigma,\ \bar{v}_{\mathrm{K}},\ \bar{v}_{\mathrm{R}} \triangleright \mathsf{RET}\ (\bar{\alpha})]\!] = \mathcal{C}_{\bar{\alpha}}(\Sigma,\ \bar{v}_{\mathrm{K}},\ \bar{v}_{\mathrm{R}} \triangleright \bar{\alpha})$$

Further, an application term of the form "$\alpha(\bar{\alpha})$" proceeds similarly to its earlier tail variant, except that, prior to evaluating any of the Etude atoms found within the construct's syntax, the continuation variable **$K** and any other values $\bar{v}_{\mathrm{K}}$ that remain live somewhere in the remainder of the program must be placed into some temporary storage facilities, since any of these variables may be clobbered by the targeted function. As expected, the "**GO$_{\mathrm{I}}$**" instruction introduced by the term is followed with a binding operation $\mathcal{C}_{\bar{v}}$, which copies any delivered results to the respective registers from $\bar{v}_{\mathrm{R}}$. Once this has been achieved, any preserved variables are promptly restored to their original values. Formally:

$$\begin{aligned}\mathcal{C}_\tau[\![\Sigma,\ \bar{v}_{\mathrm{K}},\ \bar{v}_{\mathrm{R}} \triangleright \alpha(\bar{\alpha})]\!] = {} & \mathrm{save}[\mathtt{\$}K \mid \mathtt{\$}K \leftarrow \bar{v}_{\mathrm{K}} \cup \{\mathtt{\$K}\}] +\!\!+ \\ & \mathcal{C}_{\bar{\alpha}}(\Sigma,\ \varnothing,\ \bar{v}_{\mathrm{A}} \triangleright \bar{\alpha} +\!\!+ [\alpha]) +\!\!+ \\ & \mathcal{C}_{v}(\mathtt{\$}R \triangleright \mathrm{last}(\bar{v}_{\mathrm{A}})) +\!\!+ \\ & [\![\mathtt{GO_I\ \$K,\ \$}R,\ 0]\!] +\!\!+ \\ & \mathcal{C}_{\bar{v}}(\bar{v}_{\mathrm{R}} \triangleright \mathrm{take}(\mathrm{length}(\bar{v}_{\mathrm{R}}),\ \bar{v}_{\mathrm{P}})) +\!\!+ \\ & \mathrm{restore}[\mathtt{\$}K \mid \mathtt{\$}K \leftarrow \bar{v}_{\mathrm{K}} \cup \{\mathtt{\$K}\}]\end{aligned}$$

$$\begin{aligned}\text{where } [\![\mathtt{\$}R]\!] &= \mathcal{A}_v(\mathtt{\$T} \triangleright \mathrm{last}(\bar{v}_{\mathrm{A}})) \\ \bar{v}_{\mathrm{A}} &= \mathrm{take}(\mathrm{length}(\bar{\alpha}) + 1,\ \bar{v}_{\mathrm{P}})\end{aligned}$$

A convenient scratch space can be found for storage of any variables during an execution of a nested MMIX function call in the form of the data stack $\bar{\psi}(\Delta)$. As already mentioned earlier in Section 6.4.1, throughout the entire execution of all MMIX programs produced by our compiler, the current value of the stack pointer σ_c can be found in the designated register **$S**. Recalling from Section 6.3.2 that, on the present instruction set architecture, the data stack always grows downwards towards lower address values and that, under the MMIX semantics of Etude atoms from Section 6.3.1, the value of every register is always representable under the 64-bit natural format "N.64", the process of pushing the values of an arbitrary MMIX variable set $\bar{v}$ onto the data stack can be modelled by the following recursive algorithm:

$$\mathrm{save}[\![\cdot]\!] :: [\textit{MMIX-var}] \rightarrow [\textit{MMIX-instr}]$$

$$\begin{aligned}\mathrm{save}[\![\varnothing]\!] &= [\![]\!] \\ \mathrm{save}[\![\bar{v}]\!] &= \mathrm{load}(\mathtt{\$}R \triangleright \mathrm{head}(\bar{v})) +\!\!+ \\ &\quad\ [\![\mathtt{SETL\ \ \$T,\ 0,\ \ 8} \\ &\quad\ \ \mathtt{SUBU\ \ \$S,\ \$S,\ \$T} \\ &\quad\ \ \mathtt{STOU_I\ \$}R\mathtt{,\ \$S,\ 0}\ ]\!] +\!\!+ \\ &\quad\ \mathrm{save}(\mathrm{tail}(\bar{v}))\end{aligned}$$

$$\text{where } [\![\mathtt{\$}R]\!] = \mathcal{A}_v(\mathtt{\$T} \triangleright \mathrm{head}(\bar{v}))$$

A dual operation, which reverts these registers to their original values and restores the earlier configuration of the stack pointer σ_c, can be also implemented without further difficulties. The reader should observe, however, that during restoration of variables

from the data stack, their values must be retrieved in the reverse order of these variables' earlier allocation by "save", so that the following algorithm "restore" processes the tail of the supplied register list before proceeding with restoration of the initial entry in that sequence:

$$\begin{aligned}
&\text{restore}[\![\cdot]\!] :: [\textit{MMIX-var}] \to [\textit{MMIX-instr}] \\
&\text{restore}[\![\varnothing]\!] = [\![]\!] \\
&\text{restore}[\![\bar{v}]\!] \;= \text{restore}(\text{tail}(\bar{v})) +\!\!+ \\
&\qquad\qquad [\![\texttt{LDOU}_\texttt{I}\ \$R,\ \$S,\ 0 \\
&\qquad\qquad\ \texttt{SETL}\ \ \$T,\ 0,\ \ 8 \\
&\qquad\qquad\ \texttt{ADDU}\ \ \$S,\ \$S,\ \$T]\!] +\!\!+ \\
&\qquad\qquad \text{store}(\text{head}(\bar{v}) \triangleright \$R) \\
&\quad \text{where } [\![\$R]\!] = \mathcal{A}_v(\$T \triangleright \text{head}(\bar{v}))
\end{aligned}$$

Similarly to application expressions, in the context of a nested term form, an MMIX rendition of an Etude system call operation must be likewise surrounded in an appropriate pair of "save" and "restore" constructs:

$$\begin{aligned}
\mathcal{C}_\tau[\![\Sigma,\ \bar{v}_\text{K},\ \bar{v}_\text{R} \triangleright \pi(\bar{\alpha})]\!] = {} & \text{save}[\$K \mid \$K \leftarrow \bar{v}_\text{K}] +\!\!+ \\
& \mathcal{C}_{\bar{\alpha}}(\Sigma,\ \varnothing,\ \bar{v}_\text{P} \triangleright \bar{\alpha}) +\!\!+ \\
& [\![\texttt{TRAP}\ [\![\pi[16 \mathinner{..} 23]]\!],\ [\![\pi[8 \mathinner{..} 15]]\!],\ [\![\pi[0 \mathinner{..} 7]]\!]]\!] +\!\!+ \\
& \mathcal{C}_{\bar{v}}(\bar{v}_\text{R} \triangleright \text{take}(\text{length}(\bar{v}_\text{R}),\ \bar{v}_\text{P})) +\!\!+ \\
& \text{restore}[\$K \mid \$K \leftarrow \bar{v}_\text{K}]
\end{aligned}$$

Further, all conditional terms materialise in the resulting MMIX program almost identically to their earlier tail variants, except that, in the present scenario, we must ensure that execution proceeds at the following address within the surrounding function. To this end, one of the instruction sequences $\bar{\imath}_1$ and $\bar{\imath}_2$ derived from the respective nested terms τ_1 and τ_2 must be always amended with an additional jump past the entire block generated from the complete construct:

$$\begin{aligned}
&\mathcal{C}_\tau[\![\Sigma,\ \bar{v}_\text{K},\ \bar{v}_\text{R} \triangleright \textsf{IF}\ \alpha\ \textsf{THEN}\ \tau_1\ \textsf{ELSE}\ \tau_2]\!] \\
&\qquad \mid (\delta < 2^{18}) = \mathcal{C}_\alpha(\Sigma,\ \bar{v}'_\text{K},\ \$R \triangleright \alpha) +\!\!+ \\
&\qquad\qquad\qquad\qquad [\![\texttt{BZ}\ \$R,\ [\![\delta[10 \mathinner{..} 17]]\!],\ [\![\delta[2 \mathinner{..} 9]]\!]]\!] +\!\!+ \\
&\qquad\qquad\qquad\qquad \bar{\imath}_1 +\!\!+ \bar{\imath}'_1 +\!\!+ \bar{\imath}_2 \\
&\qquad \mid \text{otherwise} = \mathcal{C}_\alpha(\Sigma,\ \bar{v}'_\text{K},\ \$R \triangleright \alpha) +\!\!+ \\
&\qquad\qquad\qquad\qquad [\![\texttt{BZ}\ \$R,\ 0,\ 3]\!] +\!\!+ \\
&\qquad\qquad\qquad\qquad \mathcal{C}_\text{J}(4(\text{length}(\bar{\imath}_2) + \text{length}(\bar{\imath}'_2) + 1)) +\!\!+ \\
&\qquad\qquad\qquad\qquad \bar{\imath}_2 +\!\!+ \bar{\imath}'_2 +\!\!+ \bar{\imath}_1 \\
&\quad \text{where } [\![\$R]\!] = \mathcal{A}_v(\$T \triangleright \mathcal{A}_\alpha(\Sigma,\ \bar{v}'_\text{K} \triangleright \alpha)) \\
&\qquad\quad \bar{v}'_\text{K} \;= \bar{v}_\text{K} \cup \{\Sigma(v_k) \mid v_k \leftarrow \text{FV}(\tau_1) \cup \text{FV}(\tau_2)\} \\
&\qquad\quad \bar{\imath}_1 \;= \mathcal{C}_\tau(\Sigma,\ \bar{v}_\text{K},\ \bar{v}_\text{R} \triangleright \tau_1) \\
&\qquad\quad \bar{\imath}'_1 \;= \mathcal{C}_\text{J}(4(\text{length}(\bar{\imath}_2) + 1)) \\
&\qquad\quad \bar{\imath}_2 \;= \mathcal{C}_\tau(\Sigma,\ \bar{v}_\text{K},\ \bar{v}_\text{R} \triangleright \tau_2) \\
&\qquad\quad \bar{\imath}'_2 \;= \mathcal{C}_\text{J}(4(\text{length}(\bar{\imath}_1) + 1)) \\
&\qquad\quad \delta \;= 4(\text{length}(\bar{\imath}_1) + \text{length}(\bar{\imath}'_1) + 1)
\end{aligned}$$

Next, we must tackle the two object environment operations "NEW ($\bar{\xi}$)" and "DEL ($\bar{\xi}$)". As described earlier in Section 6.3.2, the entire effect of these term forms can be

captured by rounding the size of the specified envelope $\bar{\xi}$ to the nearest multiple of 8 and subtracting or adding the resulting integer to the existing value of the stack pointer σ_c which, in the translated program, is depicted simply by the general purpose hardware register **$S**. Accordingly:

$$\begin{aligned}
\mathcal{C}_\tau[\![\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R} \triangleright \textsc{new}\ (\bar{\xi})]\!] &= \mathcal{C}_\alpha[\![\Sigma,\, \bar{v}_\mathrm{K},\, \texttt{\$T} \triangleright \#[\![\lceil \mathcal{S}(\bar{\xi})/8 \rceil \times 8]\!]_{\mathrm{N}.64}]\!] +\!\!+ \\
&\quad [\![\texttt{SUBU \$S, \$S, \$T}]\!] +\!\!+ \\
&\quad \mathcal{C}_v(v_\mathrm{R} \triangleright \texttt{\$S}) \\
\mathcal{C}_\tau[\![\Sigma,\, \bar{v}_\mathrm{K},\, \varnothing \triangleright \textsc{del}\ (\bar{\xi})]\!] &= \mathcal{C}_\alpha[\![\Sigma,\, \bar{v}_\mathrm{K},\, \texttt{\$T} \triangleright \#[\![\lceil \mathcal{S}(\bar{\xi})/8 \rceil \times 8]\!]_{\mathrm{N}.64}]\!] +\!\!+ \\
&\quad [\![\texttt{ADDU \$S, \$S, \$T}]\!]
\end{aligned}$$

On the other hand, every memory inspection operation "$\textsc{get}\ [\alpha,\, \bar{\mu}]_\phi$" is translated into yet another common structure, with each variant of this term form differing only in the precise choice of an MMIX opcode, selected in accordance with the specified Etude format ϕ to perform the actual movement of data from the program's memory image. In particular, the three distinguished natural formats "N.8", "N.16" and "N.32" are implemented directly by the MMIX instruction forms "$\texttt{LDBU}_\mathtt{I}$", "$\texttt{LDWU}_\mathtt{I}$" and "$\texttt{LDTU}_\mathtt{I}$", while their signed integer cousins "Z.8", "Z.16" and "Z.32" correspond to "$\texttt{LDB}_\mathtt{I}$", "$\texttt{LDW}_\mathtt{I}$" and "$\texttt{LDT}_\mathtt{I}$", respectively. In every other case, the object's content is fetched invariably using the 64-bit load operation "$\texttt{LDOU}_\mathtt{I}$":

$$\begin{aligned}
\mathcal{C}_\tau[\![\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R} \triangleright \textsc{get}\ [\alpha,\, \bar{\mu}]_{\mathrm{N}.8}]\!] &= \mathcal{C}_\mathrm{LD}(\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R},\, \texttt{LDBU}_\mathtt{I} \triangleright \alpha) \\
\mathcal{C}_\tau[\![\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R} \triangleright \textsc{get}\ [\alpha,\, \bar{\mu}]_{\mathrm{N}.16}]\!] &= \mathcal{C}_\mathrm{LD}(\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R},\, \texttt{LDWU}_\mathtt{I} \triangleright \alpha) \\
\mathcal{C}_\tau[\![\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R} \triangleright \textsc{get}\ [\alpha,\, \bar{\mu}]_{\mathrm{N}.32}]\!] &= \mathcal{C}_\mathrm{LD}(\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R},\, \texttt{LDTU}_\mathtt{I} \triangleright \alpha) \\
\mathcal{C}_\tau[\![\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R} \triangleright \textsc{get}\ [\alpha,\, \bar{\mu}]_{\mathrm{Z}.8}]\!] &= \mathcal{C}_\mathrm{LD}(\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R},\, \texttt{LDB}_\mathtt{I} \triangleright \alpha) \\
\mathcal{C}_\tau[\![\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R} \triangleright \textsc{get}\ [\alpha,\, \bar{\mu}]_{\mathrm{Z}.16}]\!] &= \mathcal{C}_\mathrm{LD}(\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R},\, \texttt{LDW}_\mathtt{I} \triangleright \alpha) \\
\mathcal{C}_\tau[\![\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R} \triangleright \textsc{get}\ [\alpha,\, \bar{\mu}]_{\mathrm{Z}.32}]\!] &= \mathcal{C}_\mathrm{LD}(\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R},\, \texttt{LDT}_\mathtt{I} \triangleright \alpha) \\
\mathcal{C}_\tau[\![\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R} \triangleright \textsc{get}\ [\alpha,\, \bar{\mu}]_\phi]\!] &= \mathcal{C}_\mathrm{LD}(\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R},\, \texttt{LDOU}_\mathtt{I} \triangleright \alpha)
\end{aligned}$$

All such load operations are depicted in the translated program by a sequence of MMIX instructions which beings with the compiled representation of the supplied address atom α, followed by the actual load operation "*OP* **$***R*, **$***A*, 0" and ends in a copy operation that moves the fetched value **$***R* into its rightful place v_R:

$$\begin{aligned}
&\mathcal{C}_\mathrm{LD}[\![\cdot]\!] :: (\mathrm{ord}\ \upsilon) \Rightarrow \\
&\qquad (\upsilon \mapsto \textit{MMIX-var},\, \{\textit{MMIX-var}\},\, \textit{MMIX-var},\, \textit{MMIX-opcode} \triangleright \textit{atom}_\upsilon) \rightarrow \\
&\qquad [\textit{MMIX-instr}]
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_\mathrm{LD}[\![\Sigma,\, \bar{v}_\mathrm{K},\, v_\mathrm{R},\, \mathit{OP} \triangleright \alpha]\!] &= \mathcal{C}_\alpha(\Sigma,\, \bar{v}_\mathrm{K},\, \texttt{\$}A \triangleright \alpha) +\!\!+ \\
&\quad [\![\mathit{OP}\ \texttt{\$}R\texttt{, \$}A\texttt{, 0}]\!] +\!\!+ \\
&\quad \mathcal{C}_v(v_\mathrm{R} \triangleright \texttt{\$}R)
\end{aligned}$$

where $[\![\texttt{\$}A]\!] = \mathcal{A}_v(\texttt{\$T} \triangleright \mathcal{A}_\alpha(\Sigma,\, \bar{v}_\mathrm{K} \triangleright \alpha))$
$[\![\texttt{\$}R]\!] = \mathcal{A}_v(\texttt{\$T} \triangleright v_\mathrm{R})$

The partial MMIX instruction set that was formalised in Section 6.2 provides only four unsigned variants of the dual object update operations "$\texttt{STBU}_\mathtt{I}$", "$\texttt{STWU}_\mathtt{I}$", "$\texttt{STTU}_\mathtt{I}$" and "$\texttt{STOU}_\mathtt{I}$". Accordingly, all updates to the content of a signed memory-resident object must be first replaced with a semantically equivalent unsigned term form, observing that, either way, MMIX discards any insignificant bits of the object's value, so

that both Etude formats result in identical program behaviours. Once this adjustment has been enacted, the compiler proceeds with the term's translation as follows:

$$\begin{aligned}
\mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \varnothing \triangleright \textsc{set}\ [\alpha_1,\ \bar{\mu}]_{\mathrm{Z}.\varepsilon}\ \textsc{to}\ \alpha_2]\!] &= \mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \varnothing \triangleright \textsc{set}\ [\alpha_1,\ \bar{\mu}]_{\mathrm{N}.\varepsilon}\ \textsc{to}\ \alpha_2]\!] \\
\mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \varnothing \triangleright \textsc{set}\ [\alpha_1,\ \bar{\mu}]_{\mathrm{N}.8}\ \textsc{to}\ \alpha_2]\!] &= \mathcal{C}_{\mathrm{ST}}(\Sigma,\ \bar{\nu}_\mathrm{K},\ \mathtt{STBU_I} \triangleright \alpha_1,\ \alpha_2) \\
\mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \varnothing \triangleright \textsc{set}\ [\alpha_1,\ \bar{\mu}]_{\mathrm{N}.16}\ \textsc{to}\ \alpha_2]\!] &= \mathcal{C}_{\mathrm{ST}}(\Sigma,\ \bar{\nu}_\mathrm{K},\ \mathtt{STWU_I} \triangleright \alpha_1,\ \alpha_2) \\
\mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \varnothing \triangleright \textsc{set}\ [\alpha_1,\ \bar{\mu}]_{\mathrm{N}.32}\ \textsc{to}\ \alpha_2]\!] &= \mathcal{C}_{\mathrm{ST}}(\Sigma,\ \bar{\nu}_\mathrm{K},\ \mathtt{STTU_I} \triangleright \alpha_1,\ \alpha_2) \\
\mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \varnothing \triangleright \textsc{set}\ [\alpha_1,\ \bar{\mu}]_{\phi}\ \ \textsc{to}\ \alpha_2]\!] &= \mathcal{C}_{\mathrm{ST}}(\Sigma,\ \bar{\nu}_\mathrm{K},\ \mathtt{STOU_I} \triangleright \alpha_1,\ \alpha_2)
\end{aligned}$$

Further, the reader should recall that, on MMIX, every "$\textsc{set}_\mathrm{i}$" variant of an object update operation assumes a semantic meaning that is indistinguishable from the corresponding "SET" form of the term:

$$\mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \bar{\nu}_\mathrm{R} \triangleright \textsc{set}_\mathrm{i}\ [\alpha_1,\ \bar{\mu}]_\phi\ \textsc{to}\ \alpha_2]\!] = \mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \bar{\nu}_\mathrm{R} \triangleright \textsc{set}\ [\alpha_1,\ \bar{\mu}]_\phi\ \textsc{to}\ \alpha_2]\!]$$

In all cases, the resulting instruction sequence is prepared from the "SET" operation in the following manner:

$$\begin{aligned}
&\mathcal{C}_{\mathrm{ST}}[\![\cdot]\!] :: (\mathrm{ord}\ \upsilon) \Rightarrow \\
&\qquad (\upsilon \mapsto \textit{MMIX-var},\ \{\textit{MMIX-var}\},\ \textit{MMIX-opcode} \triangleright \textit{atom}_\upsilon,\ \textit{atom}_\upsilon) \rightarrow \\
&\qquad [\textit{MMIX-instr}]
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_{\mathrm{ST}}[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \mathit{OP} \triangleright \alpha_1,\ \alpha_2]\!] &= \mathcal{C}_\alpha(\Sigma,\ \bar{\nu}_{\mathrm{KA}},\ \nu_\mathrm{A} \triangleright \alpha_1) +\!\!+ \\
&\quad\ \mathcal{C}_\alpha(\Sigma,\ \bar{\nu}_{\mathrm{KB}},\ \mathtt{\$}B \triangleright \alpha_2) +\!\!+ \\
&\quad\ \mathcal{C}_\nu(\mathtt{\$}A \triangleright \nu_\mathrm{A}) +\!\!+ \\
&\quad\ [\![\mathit{OP}\ \mathtt{\$}B\mathtt{,}\ \mathtt{\$}A\mathtt{,}\ 0]\!] \\
\text{where } [\![\mathtt{\$}A]\!] &= \mathcal{A}_\nu(\mathtt{\$T} \triangleright \nu_\mathrm{A}) \\
[\![\mathtt{\$}B]\!] &= \mathcal{A}_\nu(\mathtt{\$U} \triangleright \nu_\mathrm{B}) \\
\nu_\mathrm{A} &= \mathcal{A}_\alpha(\Sigma,\ \bar{\nu}_{\mathrm{KA}} \triangleright \alpha_1) \\
\nu_\mathrm{B} &= \mathcal{A}_\alpha(\Sigma,\ \bar{\nu}_{\mathrm{KB}} \triangleright \alpha_2) \\
\bar{\nu}_{\mathrm{KA}} &= \bar{\nu}_\mathrm{K} \cup \{\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \mathrm{FV}(\alpha_2)\} \\
\bar{\nu}_{\mathrm{KB}} &= \bar{\nu}_\mathrm{K} \cup \{\nu_\mathrm{A}\}
\end{aligned}$$

Finally, in the context of a nested Etude term, a binding "LET" construct of the form "$\textsc{let}\ \bar{\upsilon} = \tau_1;\ \tau_2$" is compiled in an essentially the same way as its earlier tail variant, except that, during translation of τ_1, the set of preserved variables $\bar{\nu}_\mathrm{K}$ is extended with any entities that are found free within the following body term τ_2. Formally:

$$\begin{aligned}
\mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \bar{\nu}_\mathrm{R} \triangleright \textsc{let}\ \bar{\upsilon} = \tau_1;\ \tau_2]\!] &= \mathcal{C}_\tau(\Sigma,\ \bar{\nu}'_\mathrm{K},\ \mathrm{take}(\mathrm{length}(\bar{\upsilon}),\ \bar{\nu}) \triangleright \tau_1) +\!\!+ \\
&\quad\ \mathcal{C}_\tau(\Sigma/(\bar{\upsilon}|\bar{\nu}),\ \bar{\nu}_\mathrm{K},\ \bar{\nu}_\mathrm{R} \triangleright \tau_2) \\
\text{where } \bar{\nu}'_\mathrm{K} &= \bar{\nu}_\mathrm{K} \cup \{\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \mathrm{FV}(\tau_2) \backslash \bar{\upsilon}\} \\
\bar{\nu} &= [\nu_k \mid \nu_k \leftarrow \bar{\nu}_\mathrm{G},\ \nu_k \notin \bar{\nu}'_\mathrm{K}]
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \bar{\nu}_\mathrm{R} \triangleright \textsc{let};\ \tau]\!] &= \mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \bar{\nu}_\mathrm{R} \triangleright \tau]\!] \\
\mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \bar{\nu}_\mathrm{R} \triangleright \textsc{let}\ \bar{\beta};\ \tau]\!] &= \mathcal{C}_\tau[\![\Sigma,\ \bar{\nu}_\mathrm{K},\ \bar{\nu}_\mathrm{R} \triangleright \textsc{let}\ [\![\mathrm{head}(\bar{\beta})]\!];\ \textsc{let}\ [\![\mathrm{tail}(\bar{\beta})]\!];\ \tau]\!]
\end{aligned}$$

THEOREM 6-36: *(Preservation of term semantics under compilation)* Let Σ be a variable renaming function and Γ represent the mapping of all relevant MMIX variables to their respective atomic values. Let τ be an Etude term that is closed and well-formed under the substitution by $\Gamma \circ \Sigma$. Further, let Λ' represent a function environment updated with bindings of n successive instruction addresses that begin at some Etude function variable ν to the respective n instructions generated from the compilation of

τ and let Δ' represent an object environment that is likewise updated with bindings of any symbolic addresses **LOC.**$8k$ to the current values of all pseudo-registers **$**$k$ which appear free within τ. Then, taken in the context of Λ and Δ, a term of the form "LET $\bar{\upsilon} = \tau$; $[\![$**$**$\mathbf{K}(\pi)/\Gamma]\!]$" is equivalent to the function application term "$\nu(\pi)$" under substitution by Γ. Formally:

$$\begin{aligned}
&\mathrm{LC}_\tau :: \forall\, \Sigma, \Gamma, \Gamma', \Lambda, \Lambda', \Delta, \Delta', \nu, \upsilon, \alpha, \bar{\nu}_\mathrm{K}, \upsilon \Rightarrow \\
&\quad \mathrm{WF}(\Lambda) \to \mathrm{WF}(\Delta) \to \mathrm{WF}(\tau/(\Gamma \circ \Sigma)) \to \\
&\quad [\![\Lambda' = \Lambda/\mathcal{D}_\mathbf{I}(\Gamma, \nu \triangleright \mathcal{C}_\tau(\Sigma, \bar{\nu}_\mathrm{K}, [\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \bar{\upsilon}] \triangleright \alpha))]\!] \to \\
&\quad [\![\Delta' = \Delta/\{(\Gamma(\mathbf{LOC.}8k) : \Gamma(\$k)) \mid [\![\$k]\!] \leftarrow [\Sigma(\upsilon_k) \mid \upsilon_k \leftarrow \mathrm{FV}(\tau)], k \geq 256\}]\!] \to \\
&\quad [\![\Lambda, \Delta \triangleright [\![\mathsf{LET}\ \bar{\upsilon} = \tau;\ [\![\$\mathbf{K}(\pi)/\Gamma]\!]]\!]\!] \equiv [\![\Lambda', \Delta' \triangleright [\![[\![\Gamma(\Sigma(\nu))]\!](\pi)]\!]/\Gamma]\!]
\end{aligned}$$

PROOF: For all "RET", "NEW", "DEL", "GET", "SET" and "SET$_\mathrm{I}$" term forms of τ, "LET $\bar{\upsilon} = \tau$; **$**$\mathbf{K}(\pi)$" is always reducible under $\mathcal{E}_\tau$ into a direct call to **$**$\mathbf{K}$, provided that all variables found in both $\bar{\upsilon}$ and π are replaced in **$**$\mathbf{K}$'s argument list with their actual atomic values delivered by τ, since none of these term forms ever affects the evaluation state. From theorems LC_ν and LC_α, we also have an equivalence between such calls and their compiled representations, observing that the MMIX rendition of every such term is precisely identical to that obtained from the delivered atomic values. Accordingly, the desired equivalence relation "LET $\bar{\upsilon} = \tau$; **$**$\mathbf{K}(\pi) \equiv \nu(\pi)$" holds as required for all of these term forms.

If τ represents a function application term, then both "LET $\bar{\upsilon} = \tau$; **$**$\mathbf{K}(\pi)$" and "$\nu(\pi)$" are compiled into identical MMIX instruction sequences, save for an occasional copy operation that can be amortised into Γ using the earlier equivalence result provided by theorem LC_ν. Accordingly, for all such term forms, theorem LC_τ holds trivially by the virtue of a definitional equality between the respective function environments.

Finally, the induction step, which pertains to the preservation of nested "LET" bindings, follows directly from the equivalence between "LET $\bar{\upsilon}_2$ = (LET $\bar{\upsilon}_1 = \tau_1; \tau_2); \tau_3$" and its flattened variant "LET $\bar{\upsilon}_1 = \tau_1$; (LET $\bar{\upsilon}_2 = \tau_2; \tau_3$)". Accordingly, we can begin by establishing theorem LC_τ inductively for the inner "LET" term and, subsequently, utilising the resulting equivalence in order to establish the result for the entire construct.

The remaining two cases, which pertain to the system calls and conditional "IF" term forms, are no more difficult to scrutinise. However, a detailed analysis of their respective evaluation sequences is somewhat more elaborate, so that I leave the fulfilment of the associated proof obligations as an exercise for the reader. □

With this last set of Haskell definitions, we have now completed our implementation of a verified compiler for the C programming language and can proceed directly with an implementation and subsequent verification of individual C programs, using their semantic model defined in the previous three chapters. While space and time constraints prohibited me from presenting the full details of many lesser aspects of the described compiler implementation, or from elaborating fully on the required proof obligations, these omissions should in no way distract from the magnitude of our present success.

6.5 Example Translation

As an illustrative example of our compiler in action, let us present the MMIX rendition of the C translation unit discussed earlier in Section 5.11, which, as the reader may recall, provided a textbook implementation of the quick sort algorithm. The resulting MMIX module exports two external names *swap* and *sort*, which are bound to the text symbols #1 and #2, respectively. The data segment is empty, while the text segment defines eight distinct symbols, of which #0 represents the entry point into the entire program:

```
LET #1:swap
    #2:sort
TEXT ...
DATA
IN #0
```

First, symbol #0 represents the trivial initialiser term of the two C declarations found in the original source file:

```
#0: GO_I   $T,  $K,  0
```

The following section #1 implements the actual body of the C function *swap*:

```
#1: SETL  $T,  0,    8              SUBU  $S,  $S,  $T
    OR    $249, $S,  $S             OR    $248, $0,  $0
    LDO_I $248, $248, 0             LDT_I $248, $248, 0
    STTU_I $248, $249, 0            OR    $248, $0,  $0
    LDO_I $248, $248, 0             OR    $247, $1,  $1
    LDO_I $247, $247, 0             LDT_I $247, $247, 0
    STTU_I $247, $248, 0            OR    $248, $247, $247
    OR    $248, $1,  $1             LDO_I $248, $248, 0
    LDT_I $249, $249, 0             STTU_I $249, $248, 0
    SETL  $T,  0,    8              ADDU  $S,  $S,  $T
    SETL  $2,  0,    0 @ #3         INCML $2,  0,    0 @ #3
    INCMH $2,  0,    0 @ #3         INCH  $2,  0,    0 @ #3
    GO_I  $T,  $2,   0
```

Next, #2 provides an MMIX rendition of *sort* itself:

```
#2:OR    $249, $2,  $2              LDT_I $249, $249, 0
   OR    $248, $1,  $1              LDT_I $248, $248, 0
   SETL  $247, 0,   1               ADDU  $248, $248, $247
   SETL  $247, 0,   32              SLU   $248, $248, $247
   SETL  $247, 0,   32              SR    $248, $248, $247
   CMP   $249, $249, $248           ZSP_I $249, $249, 1
   SETL  $248, 0,   0               CMP   $249, $249, $248
   ZSNZ_I $249, $249, 1             BZ    $249, 0,    38
   SETL  $T,  0,    8               SUBU  $S,  $S,  $T
   OR    $249, $S,  $S              SETL  $T,  0,    8
   SUBU  $S,  $S,  $T               OR    $248, $S,  $S
   SETL  $T,  0,    8               SUBU  $S,  $S,  $T
   OR    $247, $S,  $S              OR    $246, $0,  $0
   OR    $245, $1,  $1              LDT_I $245, $245, 0
```

```
SETL  $244, 0,    4              MULU  $245, $245, $244
ADDU  $246, $246, $245           LDT_I $246, $246, 0
STTU_I $246, $249, 0             OR    $246, $1,   $1
LDT_I $246, $246, 0              SETL  $245, 0,    1
ADDU  $246, $246, $245           SETL  $245, 0,    32
SLU   $246, $246, $245           SETL  $245, 0,    32
SR    $246, $246, $245           STTU_I $246, $248, 0
OR    $246, $2,   $2             LDT_I $246, $246, 0
STTU_I $246, $247, 0             OR    $3,   $249, $249
OR    $4,   $248, $248           OR    $5,   $247, $247
SETL  $6,   0,    0 @ #4         INCML $6,   0,    0 @ #4
INCMH $6,   0,    0 @ #4         INCH  $6,   0,    0 @ #4
GO_I  $T,   $6,   0              SETL  $3,   0,    0 @ #7
INCML $3,   0,    0 @ #7         INCMH $3,   0,    0 @ #7
INCH  $3,   0,    0 @ #7         GO_I  $T,   $3,   0
```

The function #3 implements the trivial exit block of *swap*:

```
#3:  GO_I  $T,  $K,  0
```

while #4 depicts the loop body at the heart of the entire algorithm:

```
#4:  OR    $249, $4,   $4        LDT_I $249, $249, 0
     OR    $248, $5,   $5        LDT_I $248, $248, 0
     CMP   $249, $249, $248      ZSN_I $249, $249, 1
     SETL  $248, 0,    0         CMP   $249, $249, $248
     ZSNZ_I $249, $249, 1        BZ    $249, 0,    118
     OR    $249, $0,   $0        OR    $248, $4,   $4
     LDT_I $248, $248, 0         SETL  $247, 0,    4
     MULU  $248, $248, $247      ADDU  $249, $249, $248
     LDT_I $249, $249, 0         OR    $248, $3,   $3
     LDT_I $248, $248, 0         CMP   $249, $249, $248
     ZSNP_I $249, $249, 1        SETL  $248, 0,    0
     CMP   $249, $249, $248      ZSNZ_I $249, $249, 1
     BZ    $249, 0,    16        OR    $249, $4,   $4
     SETL  $248, 0,    1         LDT_I $247, $249, 0
     ADDU  $248, $247, $248      SETL  $247, 0,    32
     SLU   $248, $248, $247      SETL  $247, 0,    32
     SR    $248, $248, $247      STTU_I $248, $249, 0
     OR    $249, $248, $248      SETL  $6,   0,    0 @ #5
     INCML $6,   0,    0 @ #5    INCMH $6,   0,    0 @ #5
     INCH  $6,   0,    0 @ #5    GO_I  $T,   $6,   0
     SETL  $T,   0,    8         SUBU  $S,   $S,   $T
     OR    $249, $S,   $S        SETL  $T,   0,    8
     SUBU  $S,   $S,   $T        OR    $248, $S,   $S
     SETL  $247, 0,    0 @ #1    INCML $247, 0,    0 @ #1
     INCMH $247, 0,    0 @ #1    INCH  $247, 0,    0 @ #1
     OR    $246, $0,   $0        OR    $245, $4,   $4
     LDT_I $245, $245, 0         SETL  $244, 0,    4
     MULU  $245, $245, $244      ADDU  $246, $246, $245
     STOU_I $246, $249, 0        OR    $246, $0,   $0
     OR    $245, $5,   $5        SETL  $244, 0,    1
     LDT_I $243, $245, 0         SUBU  $244, $243, $244
```

```
SETL  $243, 0,    32          SLU   $244, $244, $243
SETL  $243, 0,    32          SR    $244, $244, $243
STTU_I $244, $245, 0          OR    $245, $244, $244
SETL  $244, 0,    4           MULU  $245, $245, $244
ADDU  $246, $246, $245        STOU_I $246, $248, 0
SETL  $T,   0,    8           SUBU  $S,   $S,   $T
STOU_I $0,  $S,   0           SETL  $T,   0,    8
SUBU  $S,   $S,   $T          STOU_I $1,  $S,   0
SETL  $T,   0,    8           SUBU  $S,   $S,   $T
STOU_I $2,  $S,   0           SETL  $T,   0,    8
SUBU  $S,   $S,   $T          STOU_I $3,  $S,   0
SETL  $T,   0,    8           SUBU  $S,   $S,   $T
STOU_I $4,  $S,   0           SETL  $T,   0,    8
SUBU  $S,   $S,   $T          STOU_I $5,  $S,   0
SETL  $T,   0,    8           SUBU  $S,   $S,   $T
STOU_I $K,  $S,   0           OR    $0,   $249, $249
OR    $1,   $248, $248        OR    $2,   $247, $247
GO_I  $K,   $2,   0           LDOU_I $K,  $S,   0
SETL  $T,   0,    8           ADDU  $S,   $S,   $T
LDOU_I $5,  $S,   0           SETL  $T,   0,    8
ADDU  $S,   $S,   $T          LDOU_I $4,  $S,   0
SETL  $T,   0,    8           ADDU  $S,   $S,   $T
LDOU_I $3,  $S,   0           SETL  $T,   0,    8
ADDU  $S,   $S,   $T          LDOU_I $2,  $S,   0
SETL  $T,   0,    8           ADDU  $S,   $S,   $T
LDOU_I $1,  $S,   0           SETL  $T,   0,    8
ADDU  $S,   $S,   $T          LDOU_I $0,  $S,   0
SETL  $T,   0,    8           ADDU  $S,   $S,   $T
SETL  $T,   0,    8           ADDU  $S,   $S,   $T
SETL  $T,   0,    8           ADDU  $S,   $S,   $T
SETL  $6,   0,    0 @ #5      INCML $6,   0,    0 @ #5
INCMH $6,   0,    0 @ #5      INCH  $6,   0,    0 @ #5
GO_I  $T,   $6,   0           SETL  $6,   0,    0 @ #6
INCML $6,   0,    0 @ #6      INCMH $6,   0,    0 @ #6
INCH  $6,   0,    0 @ #6      GO_I  $T,   $6,   0
```

The next function #5 facilitiates an exit from the inner "**if**" statement:

```
#5: SETL  $6,   0,    0 @ #4      INCML $6,   0,    0 @ #4
    INCMH $6,   0,    0 @ #4      INCH  $6,   0,    0 @ #4
    GO_I  $T,   $6,   0
```

and #6 provides a rather lengthy rendition of the remaining computations performed within the loop:

```
#6: SETL  $T,   0,    8           SUBU  $S,   $S,   $T
    OR    $249, $S,   $S          SETL  $T,   0,    8
    SUBU  $S,   $S,   $T          OR    $248, $S,   $S
    SETL  $247, 0,    0 @ #1      INCML $247, 0,    0 @ #1
    INCMH $247, 0,    0 @ #1      INCH  $247, 0,    0 @ #1
    OR    $246, $0,   $0          OR    $245, $4,   $4
    SETL  $244, 0,    1           LDT_I $243, $245, 0
```

```
SUBU  $244, $243, $244
SLU   $244, $244, $243
SR    $244, $244, $243
OR    $245, $244, $244
MULU  $245, $245, $244
STOU_I $246, $249, 0
OR    $245, $1,   $1
SETL  $244, 0,    4
ADDU  $246, $246, $245
SETL  $T,   0,    8
STOU_I $0,  $S,   0
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
STOU_I $2,  $S,   0
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
STOU_I $5,  $S,   0
SUBU  $S,   $S,   $T
OR    $0,   $249, $249
OR    $2,   $247, $247
LDOU_I $K,  $S,   0
ADDU  $S,   $S,   $T
SETL  $T,   0,    8
LDOU_I $4,  $S,   0
ADDU  $S,   $S,   $T
SETL  $T,   0,    8
LDOU_I $1,  $S,   0
ADDU  $S,   $S,   $T
SETL  $T,   0,    8
SETL  $T,   0,    8
SETL  $T,   0,    8
SETL  $T,   0,    8
OR    $249, $S,   $S
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
OR    $247, $S,   $S
INCML $246, 0,    0 @ #2
INCH  $246, 0,    0 @ #2
STOU_I $245, $249, 0
LDT_I $245, $245, 0
OR    $245, $4,   $4
STTU_I $245, $247, 0
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
STOU_I $1,  $S,   0
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
STOU_I $5,  $S,   0
SUBU  $S,   $S,   $T
OR    $0,   $249, $249
OR    $2,   $247, $247
```

```
SETL  $243, 0,    32
SETL  $243, 0,    32
STTU_I $244, $245, 0
SETL  $244, 0,    4
ADDU  $246, $246, $245
OR    $246, $0,   $0
LDT_I $245, $245, 0
MULU  $245, $245, $244
STOU_I $246, $248, 0
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
STOU_I $1,  $S,   0
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
STOU_I $4,  $S,   0
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
STOU_I $K,  $S,   0
OR    1,    $248, $248
GO_I  $K,   $2,   0
SETL  $T,   0,    8
LDOU_I $5,  $S,   0
ADDU  $S,   $S,   $T
SETL  $T,   0,    8
LDOU_I $2,  $S,   0
ADDU  $S,   $S,   $T
SETL  $T,   0,    8
LDOU_I $0,  $S,   0
ADDU  $S,   $S,   $T
ADDU  $S,   $S,   $T
ADDU  $S,   $S,   $T
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
OR    $248, $S,   $S
SUBU  $S,   $S,   $T
SETL  $246, 0,    0 @ #2
INCMH $246, 0,    0 @ #2
OR    $245, $0,   $0
OR    $245, $1,   $1
STTU_I $245, $248, 0
LDT_I $245, $245, 0
SETL  $T,   0,    8
STOU_I $0,  $S,   0
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
STOU_I $2,  $S,   0
SUBU  $S,   $S,   $T
SETL  $T,   0,    8
STOU_I $K,  $S,   0
OR    $1,   $248, $248
OR    $3,   $246, $246
```

```
GOI   $K, $3,   0
SETL  $T, 0,    8
LDOUI $5, $S,   0
ADDU  $S, $S,   $T
SETL  $T, 0,    8
LDOUI $1, $S,   0
ADDU  $S, $S,   $T
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $T, 0,    8
OR    $249, $S, $S
SUBU  $S, $S,   $T
SETL  $T, 0,    8
OR    $247, $S, $S
INCML $246, 0,    0 @ #2
INCH  $246, 0,    0 @ #2
STOUI $245, $249, 0
LDTI  $245, $245, 0
OR    $245, $1,   $1
STTUI $245, $247, 0
SUBU  $S, $S,   $T
SETL  $T, 0,    8
STOUI $1, $S,   0
SUBU  $S, $S,   $T
SETL  $T, 0,    8
STOUI $K, $S,   0
OR    $1, $248, $248
OR    $3, $246, $246
LDOUI $K, $S,   0
ADDU  $S, $S,   $T
SETL  $T, 0,    8
LDOUI $1, $S,   0
ADDU  $S, $S,   $T
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $T, 0,    8
SETL  $3, 0,    0 @ #7
INCMH $3, 0,    0 @ #7
GOI   $T, $3,   0
```

```
LDOUI $K, $S,   0
ADDU  $S, $S,   $T
SETL  $T, 0,    8
LDOUI $2, $S,   0
ADDU  $S, $S,   $T
SETL  $T, 0,    8
LDOUI $0, $S,   0
ADDU  $S, $S,   $T
ADDU  $S, $S,   $T
ADDU  $S, $S,   $T
ADDU  $S, $S,   $T
SUBU  $S, $S,   $T
SETL  $T, 0,    8
OR    $248, $S, $S
SUBU  $S, $S,   $T
SETL  $246, 0,    0 @ #2
INCMH $246, 0,    0 @ #2
OR    $245, $0,   $0
OR    $245, $5,   $5
STTUI $245, $248, 0
LDTI  $245, $245, 0
SETL  $T, 0,    8
STOUI $0, $S,   0
SUBU  $S, $S,   $T
SETL  $T, 0,    8
STOUI $2, $S,   0
SUBU  $S, $S,   $T
OR    $0, $249, $249
OR    $2, $247, $247
GOI   $K, $3,   0
SETL  $T, 0,    8
LDOUI $2, $S,   0
ADDU  $S, $S,   $T
SETL  $T, 0,    8
LDOUI $0, $S,   0
ADDU  $S, $S,   $T
ADDU  $S, $S,   $T
ADDU  $S, $S,   $T
ADDU  $S, $S,   $T
ADDU  $S, $S,   $T
ADDU  $S, $S,   $T
ADDU  $S, $S,   $T
INCML $3, 0,    0 @ #7
INCH  $3, 0,    0 @ #7
```

Finally, symbols **#**7 and **#**8 describe the exit blocks of the outer "**if**" statement and the *sort* function itself:

```
#7: SETL  $3,  0,  0 @ #8          INCML $3,  0,  0 @ #8
    INCMH $3,  0,  0 @ #8          INCH  $3,  0,  0 @ #8
    GO_I  $T,  $3,  0

#8: GO_I  $T,  $K,  0
```

The reader will undoubtedly observe the naïveté of the above translation, whereby constants such as 8 are repeatedly placed into scratch registers during allocation of objects on the program's data stack. Given a smarter register allocation algorithm and a more elaborate constant synthesis strategy, the above program could be easily reduced into 20-odd individual MMIX operations. Nevertheless, I chose to present the above unoptimised version of the program in order to expose the actual behaviour of our compiler in its present form that was described throughout the last three chapters.

7

CONCLUSION

Man is still the most extraordinary computer of all.
— John F. Kennedy

When, over seven years ago, I embarked on the road towards a formal verification of a C compiler, I held little hope of ever arriving at the ultimate destination and expected to content myself with a partial solution to only one or two aspects of the entire project. Yet here it is: not just a complete, formally verified translation of the standard C programming language into a sequence of realistic, executable machine instructions, but, more so, an entire framework ready to facilitate similar designs for many other source and target languages. Although, due to space restrictions, I was forced to omit many of the less interesting details such as the lexical analysis, parsing and linking stages of the translation process and I was only able to present a casual and sometimes schematic outlines of the required proofs, it should be clear that all of these remaining tasks pale in comparison with the complexity of the original project at its outset. With the help of capable theorem proving tools such as Coq and Twelf, it should be possible to refine the informal reasoning presented in Chapter 6 into a stringent mechanically verified proof of correctness for our compiler. Nevertheless, in the interest of professional honesty, it is now time to summarise both the successes and the limitations of the linear correctness approach advocated in Chapter 4, as well as the few conspicuous omissions from the present work.

First and foremost, my approach rests crucially on a novel and hence necessarily controversial perception of a compiler as a functor between the categories of its source and target languages. It assumes that the system's correctness can be reduced solely to that functor's isomorphism, without recourse to any external semantic authority for mediation of meanings during the various stages of a program's translation. As such, my approach challenges the very foundations of all existing work on compiler verification, so that, notwithstanding the philosophical argument from Chapter 2, I fully expect the linear correctness principle to raise eyebrows and a certain degree of academic opposition. Accordingly, more than an authoritative solution to the challenge of compiler verification, I indent the present work to constitute opening remarks in a debate on the philosophical substratum of program correctness.

I should also point out that the formalisation of C presented in Chapter 5 is not all that different from the existing denotational approaches to language semantics. Nevertheless, its successful incorporation into an actual translation system refutes the folk

wisdom which has it that such translation semantics do not work in practice due to an overwhelming amount of engineering trickery found in an effective implementation of every industrial compiler. In particular, most of the incidental implementation details found in Chapter 5 have direct counterparts in more conventional operational specifications of a realistic programming language [Milner 90].

A particularly interesting aspect of linear correctness, which represents both its cardinal reward and a potential focus of many philosophical objections, is the apparent scarcity of formal proof obligations imposed by the design. It stems from the fact that every translation of a language into the compiler's intermediate representation, including both the system's front end and the semantic description of the target language, is taken to be true by definition and, accordingly requires no formal justification. The only proof that cannot be avoided under the linear correctness approach is a commutativity of the mapping between the intermediate and target portrayal of the source program, which, as shown in Chapter 6 of this work, turns out to be far simpler than the more complex commutativity relation of the Morris diagram [Morris 73]. Of course, in practice, the system would benefit greatly from an inclusion of additional results, such as the precise termination criteria for the compiler outlined in Section 5.11, or the positive magnitude of all C types. Here, the foundational nature of this work raises as many questions as it answers, since further research and experience is required to determine which, if any, useful program properties can be derived from the semantic definition provided by a linearly correct compiler for a language in which the program has been described.

The second conspicuous omission in this work pertains to the issues of lexical analysis and parsing of a textual representation found in the original source files supplied to the compiler by its users. By the time I came to the realisation that these topics could not be covered adequately within the available time and space, I had completed large portions of the rather non-trivial specification of a standard C preprocessor and was understandably reluctant to conclude that the work had to be discarded from the ultimate dissertation. However, these topics are already treated thoroughly by other authors, certainly in more detail than I could ever afford without distracting the reader from the primary goals of the present project. Nevertheless, in order to retain as much explicitness in the specification of C as possible without dwelling into the awkward issues of manipulating individual characters or lexeme sequences, I chose to present my compiler as a translation of a program's parse tree, following the precise BNF grammar of the language as prescribed by the C Standard. Although, initially, the decision seemed like a necessary evil, I have since come to believe that the forfeiture of a convenient abstract syntax tree representation is vital for a compiler's ability to serve as an effective formal specification of its source language, since it avoids a certain degree of detachment between the designers of a translation system and its eventual users. In a similar way, although the translation semantics presented in Chapter 5 ought to avoid details of a complex error diagnostic infrastructure found in typical industrial compilers, the

model should, in principle, accommodate such diagnostics in its eventual practical implementation. For example, the unspecified Haskell exception monad M ubiquitous in Chapter 5 could be, in principle, extended with any such diagnostics, without mandating any adjustments to the actual semantic translation which underpins the system's correctness.

Incidentally, there is nothing intrinsically new about my reliance on the program's parse tree in a formal specification of a programming language. For example, LISP has been defined by a translation of LISP source files into the language of LISP itself from the beginning [McCarthy 62, Smith 84], while Haskell is formalised as a translation of its concrete grammar into a subset of the language known affectionately as "the core", whose syntax is itself captured by Haskell's own concrete grammar [Peyton Jones 03]. Further, most theorem proving frameworks such as Twelf and Coq are formalised solely in terms of their abstract, rather than concrete syntax [Twelf 98, INRIA 02].

A much more significant omission of this work pertains to the reasoning about properties of primitive definitions found in the standard Haskell libraries, such as those of the arithmetic, list and set operations described in Appendix A, whose mathematical underpinning is taken for granted in the proofs from Chapter 6. In most, but certainly not all cases, these properties can be inferred easily from the corresponding Haskell definitions. However, the precise amount and difficulty of the work required for a complete formalisation of the Haskell prelude remains uncertain and open for future research. In the meantime, the formal reasoning presented in this dissertation should be taken with the usual degree of caution appropriate for the traditional "pen and paper" proofs of mathematics. All of the above limitations would be lifted by a concrete implementation of my extensions to the Haskell programming language proposed in Chapter 3. To this end, I believe that a formal development of the underlying theory and an effective execution of these extensions represents an essential continuation of my present work and the next logical step on the road towards a fully verified implementation of a C compiler.

In my original vision for the project, I intended to focus on the issues of applying conventional optimising transformations within the framework of a purely functional intermediate program representation. Only one or two chapters were to be devoted to the actual translation of C programs. However, when the volume of material began to approach some four hundred pages, it became clear that this preliminary scope was far too broad for a single monograph and that it had to be restricted considerably if the work was to be completed within the allotted time frame. First to go was the goal of presenting an exhaustive treatment of modern program optimisation techniques within the framework, a topic that, although fascinating and tremendously important, could easily span multiple volumes on its own, while remaining essentially orthogonal to the actual issues of a verified compiler design. Accordingly, I elected to present a system that performs the absolute minimum of work required for a faithful translation a C program into a sequence of executable machine instructions, disregarding any issues of that pro-

gram's operational efficiency. Nevertheless, the performance of programs during their execution remains a primary issue of concern to any compiler design methodology and, while I have exercised every care to ensure that arbitrary optimising program transformations can be incorporated posteriorly into the prescribed framework without invalidating any of the results presented herein, more work in the area is required to establish the sufficiency of my approach for representation of many such transformations. Some preliminary results on the effectiveness of purely functional program representations for capturing of the information essential to such transformations has been presented in my earlier publication on the topic [Chakravarty 03]. Perhaps the only limitation to the expressiveness of Etude is its inability to accommodate certain popular optimisations such as sparse conditional constant propagation, which do not always preserve the semantics of non-terminating programs [Wegman 91]. Since the linear correctness approach from Chapter 2 relies crucially on transitivity of the program equivalence relation, such optimisations are inherently inexpressible within the framework. Further experience is required to estimate the exact impact of this deficiency on the effective performance of programs produced by our compiler.

A possibility of exhaustion of any bounded resources, such as the program's stack space, raises another challenge for my compiler verification approach. While, on modern computers, most such resources are so abundant that, in practice, this problem is no more interesting than a formalisation of the precise effects a power failure would have on the meaning of an otherwise well-behaved program, the intricacies of semantic transparency proofs, such as those involved in the final translation of Etude programs into their MMIX counterparts in Section 6.4, artificially exasperate these issues to a rank of a serious obstacle. In Chapter 6, I have given this problem only a very superficial treatment, since I was unable to arrive at a satisfactory resolution of the underlying limitation in the course of my own research. However, future work on linear compiler design will need to establish a suitable formal model of resource exhaustion, to serve as a panacea for all program transformations which affect their memory usage, both in the course of the essential instruction stream generation and any auxiliary improvements to the operational performance of resulting programs. I can only hope that such future resolution of this issue will not invalidate the central premises of my approach.

However, as the project developed, its scope has not always contracted, but also expanded into a number of originally unanticipated areas. In particular, initially I held little hope of applying the linear correctness technique to a verification of the final stream of machine instructions produced by the compiler. However, in the end, incorporation of that ultimate translation stage into the system proved not only natural, but, more so, pivotal to the complete approach, so that the validity of the entire compiler rests crucially on proving commutativity of that phase with respect to the semantics of its intermediate program representation. Nevertheless, code generation has introduced the project to an entirely new level of complexity, necessitating certain simplifications in order to present all of the critical ideas within a reasonable amount of material. In

particular, the translation unveiled in Chapter 6 includes only a very rudimentary register allocation algorithm, since even the most naïve treatment of the topic would require a significant amount of effort, all of which would remain largely orthogonal to the actual topic of compiler verification. It is rather unfortunate that the need for register allocation in the final translation of Etude into a target architecture blurs the otherwise clear boundary between a canonical language translation and its pragmatic optimising variant. However, this confusion of concerns cannot be avoided for as long as real hardware architectures continue to provide only a finite amount of register storage space.

Further, certain additional issues in generation of actual executable programs remain unresolved at the present time. In particular, the compiler described in this work includes only a fairly abstract model of the linkage stage of translation, or the semantics of all operating system primitives. Nevertheless, given the existing research in the area, especially that conducted as part of monadic foundations for the treatment of side effects in the Haskell language, promise to alleviate these issues in an elegant and unobtrusive fashion [Peyton Jones 87, Peyton Jones 93].

However, to confuse the matter further, in the real world, every machine language comes with a well-established and rigid operational interpretation in the form of its actual hardware implementation. Although these operational models are rarely specified formally, they should nevertheless be recognised somewhere within my framework, in order to preserve the expected behaviour of the program under its translation by the compiler. Although the issues of hardware verification itself are beyond the scope of the present work, a satisfactory design of a verified compiler must, at least in principle, enable verification of the semantic interpretation $\rho \circ \hat{\psi}$ utilised by its linearly correct implementation with respect to any other semantic model of the target language provided by its designers. Although I do not resolve this issue at the present time, the reader should observe that the natural and well-understood operational semantics of lambda calculi promise to serve as a very capable tool for the task.

Finally, in the original proposal, I had no intention of capturing any under-specifications of the C programming language prescribed the ANSI/ISO committee. However, it soon became obvious that a faithful model of those linguistic features which the C Standard deems undefined, unspecified and implementation-defined is not only interesting, but also highly demonstrative of the flexibility with which the linear correctness approach is able to tackle the challenges posed by modern programming language designs. Accordingly, much of the material in Chapters 4, 5 and 6 is shaped by a desire to retain these aspects of C within the presented compilation framework. To this end, the specification of C and MMIX defined in this work contains certain conspicuous omissions, such as a formal description of the underlying memory model. A proper separation between the portable and non-portable C constructs proved to be one of the most sensitive tasks of the entire project, and I have only achieved the presented results after numerous unsuccessful attempts at the exercise. In this, my work confirms the folklore belief that, when it comes to language semantics, saying less is often harder

than saying too much. The stratification of meaning favoured by my approach sits well with the practicalities of software implementation, allowing programmers to elect the precise layer within the resulting "semantic onion" which best serves the purpose and structure of their particular software.

Perhaps the only feature of the C programming language, whose unspecified semantics are not recognised correctly by its formalisation in Chapter 5, is the precise behaviour of statically linked array objects that are introduced implicitly by string literal expressions. In the C Standards, such objects are permitted to occupy overlapping memory regions whenever the corresponding string literals have suitably overlapping contents. Although this optional linguistic feature is not recognised in Chapter 5, it should not pose too many difficulties in a future revision of the system. One particularly compelling means of admitting overlapping objects into the generic Etude model from Section 4.5 is to extend the set of memory access attributes with a new family of *aliasing tags*, which would include a separate value for each potentially overlapping object in the program. Any environment updates to one such object would then automatically invalidate the contents of all other objects with the same aliasing access attribute found anywhere else in the address space. This technique seems especially attractive, as it could also prove useful during later optimising program transformations, by allowing many other important object aliasing information to be captured within the constraints of the generic Etude language.

While on the topic of memory access attributes, it must be acknowledged that my treatment of volatile objects, and, for that matter, all other issues that stem from an asynchronous or otherwise non-deterministic execution of C programs is, in general, inexpressible within the current incarnation of Etude. In all likelihood, an entirely different kind of intermediate program representation would be required in order to provide a proper formalisation of non-determinism and concurrency. It could, for example, be based on the pi calculus of Milner, Parrow and Walker [Milner 92a, Milner 92b]. Notwithstanding these limitations, the reader should observe that, by happy coincidence, the actual MMIX programs produced by the present version of my compiler are ready and able to be executed in concurrent environments.

Sometimes, however, in order to blur the distinction between unspecified and undefined behaviours of C programs, the translation semantics presented in Chapters 4 and 5 must deviate from the strict letter of the C Standard. This effect is particularly visible in the area of unspecified evaluation order, whereby C expressions such as "$f\,(a) + f\,(b)$" are deemed undefined by our semantics of Etude groups, whenever the evaluation of f produces some non-trivial side effects, despite the fact that the C Standard brands such expressions as merely unspecified. To the best of my knowledge, this issue cannot be resolved adequately without introducing some degree of support for non-deterministic evaluation into the language. However, the reader should observe that the current algebraic semantics of Etude are quite capable to correctly modelling even the most intricate forms of interactions between memory access operations in such expression groups, so

that, even in its present form, the translation semantics of C capture the notion of unspecified evaluation order with an accuracy and clarity that is unparalleled in all past research on the topic.

Three further improvements to my intermediate program representation suggest themselves naturally. First and foremost, in this work I do not advocate for a single universal representation of programs within all linearly correct compilers, so that various intricacies of a particular source and target language must be captured by a stratified semantic model, in a manner epitomised by Chapters 5 and 6. Nevertheless, given the experience of projects such as TenDRA and C-- [Macrakis 92, Currie 95, Peyton Jones 98b], it should not be difficult to arrive at a fully portable variant of Etude, so that, in principle, all unspecified and implementation-defined aspects of the source language could be captured by permitting some leeway in the actual translation process itself. For example, in Chapter 5 this approach was already adopted in description of structure and union assignments, whose semantics were modelled by concrete but otherwise unspecified Etude terms depicted by the implementation-defined constructions "set" and "set_I". However, given that extending this approach to the remainder of the language would significantly complicate the final translation of the resulting program into its ultimate binary representation, I leave all such enhancements open for future research.

In the meantime, a simple but useful improvement to our intermediate program representation would involve clarification of the proper distinction between programs that are *well-formed*, in that they conform to the various syntactic constraints imposed on the abstract syntax of Etude, and those which are *well-defined*, i.e., denote computations other than "$\perp$". Although, for conciseness, in this work I consolidate both semantic properties into a single predicate "WF", such simplification was only possible due to an absence of a more sophisticated concrete syntax in my presentation and, in a more pragmatic scenarios, the distinction will have to be reinstated, in order to allow a precise characterisation of those Etude programs which can be actually handled by the translation from Chapter 6.

Last but not least, it would be tempting to extend Etude with a strong type system, in order to extend the project into the area of *compiler-aided program verification*, envisioned by Anthony Hoare in his 2003 Grand Challenge address in Warsaw [Hoare 03]. Besides its obvious benefits to the design of reliable software, a suitably expressive dependant type system would also, in all likelihood, enable future compilers to incorporate many useful semantic properties of C programs directly into the framework of Etude, instead of formalising them explicitly within the translation process. However, the actual design of such a type system and its precise expressive power remain uncertain, so that, at the present time, they constitute perhaps the most exciting area for future research stemming from my work.

John F. Kennedy once said in his commencement address at Yale University that the great enemy of the truth is very often not the lie —— deliberate, contrived and dis-

honest — but the myth —– persistent, persuasive, and unrealistic. This insight is as true in computer science as it is in politics, and nowhere does it manifest itself more than in the history of compiler verification. For over fifty years, the quest for formal certification of a program translation system has centred around the single premise that a language such as C constitutes an autonomous notation for a description of algorithms rather than a mapping between such notations. This is despite the fact that most of the significant successes in the area have been achieved through recognition of some translational aspects of the system, such as Blum's and Kannan's work on program checking [Blum 89, Blum 95, Wasserman 97] or the staged compiler verification effort of Blazy, Dargaye and Leroy [Blazy 06]. By making explicit the definition of a programming language as the subject rather than an object of a translation between a programmer's intentions and a mathematical depiction of computation, I was able to achieve in the space of one monograph what has alluded computer scientists for half a century: a complete verified implementation of a compiler for a practical high level programming language. Accordingly, more than any other of its contributions, my work demonstrates conclusively that no programming language can be ever extricated from its translation into some more fundamental calculus of computation.

This thesis describes only one verified compiler, which targets a rather utopian instruction set architecture and a venerable source language that has already seen better days and whose popularity will only diminish further with age. But I have no doubt that many more linearly correct systems will be devised in the future, opening the stage for the next great adventure in the science of programming languages, the grand challenge of a compiler that verifies its work, so that, once and for all, the reality of a provably correct software can be brought to the everyday experience of the information technology industry.

ASSUMED NOTATION

As discussed in Chapter 3, the presentation of all Haskell source code included in this work omits the implementation details of many simple definitions. Most of these omitted constructs can be found in the standard Haskell library, in which case only their type signatures are presented below. For all remaining entities that are applied but not defined in the actual body of this work, this appendix contains a detailed description of both the type and the implementation of the entity in the standard syntax of the Haskell programming language.

A.1 Standard Types

The standard Haskell library provides a plethora of predefined scalar types, but only four of them are actually used in this work: "Bool", "Integer", "Rational" and "String". For consistency with the conventions adopted by the C standard for presentation of language grammars, the names of these types are presented simply as "*bool*", "*integer*", "*rational*" and "*string*", respectively. The two data constructors of the "*bool*" type are written as "true" and "false", while integers are depicted simply by numeric constants such as "0", "-1" or "7". Rational numbers are always introduced by the arithmetic operators described later in Section A.5, while concrete string values never appear in the presented compiler implementation at all. As mentioned in Chapter 3, the "undefined" value which appears as an implicit member of every Haskell type is written as "$\perp$".

The standard prelude also supports a number of type classes, but only seven of these are of any interest to us. The "eq" class provides the two standard structural equality operators "$=$" and "$\neq$", while "ord" implements total ordering of values, as exemplified by the four operators "$<$", "$>$", "$\leq$" and "$\geq$". Further, the "num" class introduces the four arithmetic operators "$+$", "$-$", "$\times$" and "$/$" common to both integers and rational numbers, while "show" describes types whose values can be formatted into human-readable strings. Finally, "enum" represents enumerated types such as "*bool*", all of which can be happily confused with integer values by the virtue of two predefined functions "fromEnum" and "toEnum". For conciseness, in this work both of these functions are treated as implicit coercions, whose names are hidden from the presentation. Their type signatures are introduced as follows:

$[\![\cdot]\!] :: (\text{enum } T) \Rightarrow T \rightarrow \mathit{integer}$
$[\![\cdot]\!] :: (\text{enum } T) \Rightarrow \mathit{integer} \rightarrow T$

Most often, these implicit conversions are applied to the values of boolean quantities, whereby the "true" and "false" constructors are rendered isomorphic to the integers 1 and 0, respectively. At any rate, if both enumeration coercions represent total functions, then the notation "succ(n)" is equal to the following value $n+1$ in the underlying enumeration sequence:

$$\text{succ}[\![\cdot]\!] :: (\text{enum } T) \Rightarrow T \rightarrow T$$
$$\text{succ}[\![n]\!] = n + 1$$

In similar vain, in this work we also confuse integer and rational numbers, allowing the earlier to be used freely whenever a rational number is actually expected by the compiler. This implicit promotions of integers into the realm of fractions can be formalised by a coercion function of the following form:

$$[\![\cdot]\!] :: \mathit{integer} \rightarrow \mathit{rational}$$
$$[\![x]\!] = x/1$$

assuming that the division operator "/" described later in Section A.5 always produces a rational number. The reverse demotion of rationals into the realm of integers is also possible, provided that the rational's value does not contain a fractional part. Formally:

$$[\![\cdot]\!] :: \mathit{rational} \rightarrow \mathit{integer}$$
$$[\![x]\!] \mid ([x] = x) = [x]$$

where the truncation operation "$\lfloor x \rfloor$" is also defined later in Section A.5.

A.2 Type Combinators

In Haskell, type-valued functions are represented by polymorphic data types. Perhaps the simplest of such constructions is epitomised by the standard "Maybe" type. Since, in this work, this type combinator is used almost exclusively for depiction of optional parse tree components, both of its constructors "Just" and "Nothing" are almost always suppressed from the presentation. In particular, an application of the standard Haskell type combinator "Maybe" to T, is represented by the concise notation "T_{opt}", formatted in line with the conventions adopted by the C standardisation committee [ANSI 89]. In those contexts where an omission of the data constructor "Nothing" could lead to a confusion, the constructor is written as "ϵ". Accordingly, the Haskell definition of "Maybe" can be presented as follows:

$$[\![T]\!]_{opt} :$$
$$\quad T$$
$$\quad \epsilon$$

For clarity, all variables of a "Maybe" type are generally assigned names that are decorated with the "*opt*" subscript, such as "x_{opt}" and "$\mathit{identifier}_{opt}$". To complete the annihilation of the "Just" construction from our presentation, we also introduce the following coercion function, which is always undefined on "Nothing" values:

$$[\![\cdot]\!] :: T_{opt} \rightarrow T$$
$$[\![x]\!] = x$$

In Haskell, stateful constructs are often represented as *monadic computations*, whose structure is described by members of the standard classes "monad" and "monad-fix". In this work, such monads appear solely in the translation of C entities, whereby they are used to identify and reject any invalid inputs to the compiler. To this end, the notation "require (P)" represents a monadic term that always evaluates to the Haskell unit value "$()$" lifted into some monad M, provided that the supplied boolean predicate P is true. Formally:

$$\text{require } [\![\cdot]\!] :: (\text{monad } M) \Rightarrow \mathit{bool} \rightarrow M()$$

$$\begin{aligned}\text{require } [\![\text{true}]\!] &= \text{return}\\ \text{require } [\![\text{false}]\!] &= \text{reject "Semantic error"}\end{aligned}$$

which relies on two additional standard monadic combinators "return" and "reject" that are provided by every Haskell monad. In the actual Haskell prelude, the "reject" combinator actually appears under the name "fail":

$$\begin{aligned}\text{return } [\![\cdot]\!] &:: (\text{monad } M) \Rightarrow T \rightarrow M(T)\\ \text{reject} \quad &:: (\text{monad } M) \Rightarrow M(T)\end{aligned}$$

The final monadic operation "||" combines two or more computations into a list of the form "$M_1 \,||\, M_2 \,||\, ... \,||\, M_n$". It evaluates to the first of these terms which delivers a well-formed Haskell value. Formally:

$$[\![\cdot]\!] \;||\; [\![\cdot]\!] :: (\text{monad } M) \Rightarrow T_{opt} \rightarrow T_{opt} \rightarrow M(T)$$

$$\begin{aligned}[\![x]\!] \;||\; [\![M_2]\!] &= \text{return } (x)\\ [\![\epsilon]\!] \;||\; [\![\ y\ ]\!] &= \text{return } (y)\\ [\![\epsilon]\!] \;||\; [\![\ \epsilon\ ]\!] &= \text{reject}\end{aligned}$$

observing that the standard "Maybe" type already belongs to the "monad" class.

Last but not least, no treatment of type combinators would be complete without a mention of *tuples*, or n-products of predetermined Haskell types. In this work, a collection of n Haskell values $x_1 :: T_1, x_2 :: T_2 \,...\, x_n :: T_n$ is usually written as a term $(x_1, x_2 \,...\, x_n)$ of the type $(T_1, T_2 \,...\, T_n)$, although the surrounding parentheses may be omitted whenever they are deemed visually redundant by the expression's contents. For clarity, the "," separator in a tuple expression or type is often replaced with the symbol ":" or "$\triangleright$". In other words, both of the terms $(A{:}B)$ and $(A \triangleright B)$ should be always considered as semantically identical to (A, B), distinguished from the later form only in the interest of presentation.

A.3 Logical Operations

The three Haskell operators "&&", "||" and "not" are typeset in this work as the well-known logical notations "$\wedge$", "$\vee$" and "$\neg$", respectively. In the standard library, they assume the following Haskell signatures:

$$\begin{aligned}&[\![\cdot]\!] \wedge [\![\cdot]\!],\ [\![\cdot]\!] \vee [\![\cdot]\!] :: \mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool}\\ &\neg[\![\cdot]\!] :: \mathit{bool} \rightarrow \mathit{bool}\end{aligned}$$

It is important to observe that the standard definitions of these functions are strict only in their first boolean argument.

For convenience, the "$\wedge$" and "$\vee$" operators may be also applied to an entire list of operands using the standard Haskell functions "and" and "or". In the presentation, these two list operations are always typeset as "$\bigwedge$" and "$\bigvee$", respectively and their Haskell signatures are defined as follows:

$$\bigwedge[\![\cdot]\!],\ \bigvee[\![\cdot]\!] :: [bool] \to bool$$

where the notation "$[T]$" represents the standard Haskell polymorphic list type. Both of these operations are particularly useful when combined with the list comprehension notation. For example, the expression "$\bigwedge[P(x_k) \mid x_k \leftarrow \bar{x}]$" evaluates to "true" if and only if every element of the list $\bar{x}$ satisfies the predicate function P.

A.4 Relational Operations

Boolean values may be also constructed using the two Haskell operators "==" and "/=", typeset as "$=$" and "$\neq$" in our presentation. In this work, these operators are always assumed to represent the structural identity relation that can be derived implicitly for every non-functional type T using an appropriate "deriving" clause. Intuitively, these operators have the following Haskell signatures:

$$[\![\cdot]\!] = [\![\cdot]\!],\ [\![\cdot]\!] \neq [\![\cdot]\!] :: (\text{eq}\ T) \Rightarrow T \to T \to bool$$

A pair of values whose type is subject to an appropriate total ordering may be also compared using the standard Haskell operators "<", "<=", ">" and ">=", predictably typeset as "$<$", "$\leq$", "$>$" and "$\geq$", respectively:

$$[\![\cdot]\!] < [\![\cdot]\!],\ [\![\cdot]\!] \leq [\![\cdot]\!],\ [\![\cdot]\!] > [\![\cdot]\!],\ [\![\cdot]\!] \geq [\![\cdot]\!] :: (\text{ord}\ T) \Rightarrow T \to T \to bool$$

Given a pair of ordered values (x, y), the lesser and greater of its members may be retrieved using the respective notations "$\min(x, y)$" and "$\max(x, y)$". Both of these functions are provided in the standard Haskell library with the following types:

$$\min[\![\cdot]\!],\ \max[\![\cdot]\!] :: (\text{ord}\ T) \Rightarrow (T, T) \to T$$

As with the "$\wedge$" and "$\vee$" operators, "min" and "max" can be also applied to entire lists of values. The relevant functions are known in the standard Haskell library under the names of "minimum" and "maximum", but, for clarity, we will typeset them simply as follows:

$$\min[\![\cdot]\!],\ \max[\![\cdot]\!] :: (\text{ord}\ T) \Rightarrow [T] \to T$$

It should be observed that both of these constructions have undefined semantics when applied to an empty list of values.

Comparisons of numeric quantities often give rise to mathematical notations of the form "$x < y < z$". In this work, such constructs are given a formal mandate by the following Haskell definitions:

$$[\![\cdot]\!]\ [\![\cdot]\!]\ [\![\cdot]\!]\ [\![\cdot]\!]\ [\![\cdot]\!] :: T \to (T \to U \to bool) \to U \to (U \to V \to bool) \to V \to bool$$
$$[\![x]\!]\ op_1\ [\![y]\!]\ op_2\ [\![z]\!] = (x\ op_1\ y \wedge y\ op_2\ z)$$

A.5 Arithmetic Operations

Predictably, the four standard binary arithmetic operators "+", "-", "*" and "/" are always typeset as the familiar mathematical symbols "$+$", "$-$", "$\times$" and "$/$". Further, the unary operator "negate" is also written as "$-$". Each of these five operators may be applied to either integer or rational operands, but, in both cases, it always reflects the true arithmetic result of the corresponding mathematical construction:

$$[\![\cdot]\!] + [\![\cdot]\!],\ [\![\cdot]\!] - [\![\cdot]\!],\ [\![\cdot]\!] \times [\![\cdot]\!] :: (\text{num}\ T) \Rightarrow T \to T \to T$$
$$[\![\cdot]\!] / [\![\cdot]\!] :: (\text{num}\ T) \Rightarrow T \to T \to \mathit{rational}$$
$$-[\![\cdot]\!] :: (\text{num}\ T) \Rightarrow T \to T$$

The reader should observe that, in this work, the division operator "/" always produces a rational number, even when applied to integer arguments. To convert the resulting fraction back into an integer, one of the following four rounding operations must be applied explicitly to its value:

$$\text{round}[\![\cdot]\!] :: \mathit{rational} \to \mathit{integer}$$
$$\lceil [\![\cdot]\!] \rfloor :: \mathit{rational} \to \mathit{integer}$$
$$\lfloor [\![\cdot]\!] \rfloor :: \mathit{rational} \to \mathit{integer}$$
$$\lfloor [\![\cdot]\!] \rfloor :: \mathit{rational} \to \mathit{integer}$$

All four of these operations are implemented for us in the standard Haskell library. The "round" function converts its operand to the nearest integer value, while "$\lfloor x \rfloor$", "$\lceil x \rceil$" and "$\lfloor x \rfloor$" round x towards zero, infinity and negative infinity, respectively. On the other hand, the superficially similar notation "$|x|$" has nothing to do with rounding at all. Instead, it returns the absolute value of a numeric quantity. Once again, it is already provided for us in the standard Haskell library with the following type signature:

$$|[\![\cdot]\!]| :: (\text{num}\ T) \Rightarrow T \to T$$

Mathematicians often like to apply the $+$ and $\times$ operators to entire lists of values. In the standard Haskell library, the relevant functions are written as "sum" and "product" but, in this work, they are typeset using the familiar symbols "$\sum$" and "$\prod$", giving rise to the following declarations of their types:

$$\textstyle\sum [\![\cdot]\!],\ \prod [\![\cdot]\!] :: (\text{num}\ T) \Rightarrow [T] \to T$$

Occasionally, when the two operands of the "$\times$" operator are represented by short variable names or parenthesised expressions, the operator symbol is omitted from the presentation for conciseness. No ambiguity is generally possible, since the resulting presentation essentially resembles the syntax of a function application, which is of course meaningless on numeric quantities. Formally, the notation is enacted by the following unusual Haskell definition:

$$[\![\cdot]\!]\ \ [\![\cdot]\!] :: (\text{num}\ T) \Rightarrow T \to T \to T$$
$$[\![x]\!]\ \ [\![y]\!] = x \times y$$

The standard library also supports a number of exponentiation operators. In this work, all of these are presented using the same familiar mathematical formatting, which is

exemplified by the following Haskell type signature:

$$[\![\cdot]\!]^{[\![\cdot]\!]} :: (\text{num } T) \Rightarrow T \to T \to T$$

While, in our compiler, we do not require a complete mathematical implementation of the dual "log" function, in a number of places we rely on its integer approximations "$\lceil \log_2(x) \rceil$" and "$\lfloor \log_2(x) \rfloor$". Fortunately, both of these can be obtained easily enough using the following recursive algorithms:

$$\lceil \log_2[\![\cdot]\!] \rceil, \lfloor \log_2[\![\cdot]\!] \rfloor :: \textit{rational} \to \textit{integer}$$

$$\begin{array}{lll} \lceil \log_2[\![x]\!] \rceil & \mid (x \leq 1/2) & = \lceil \log_2(x \times 2) \rceil - 1 \\ & \mid (1/2 < x \leq 1) & = 0 \\ & \mid (x > 1) & = \lceil \log_2(x/2) \rceil + 1 \\ \lfloor \log_2[\![x]\!] \rfloor & \mid (x < 1) & = \lfloor \log_2(x \times 2) \rfloor - 1 \\ & \mid (1 \leq x < 2) & = 0 \\ & \mid (x \geq 2) & = \lfloor \log_2(x/2) \rfloor + 1 \end{array}$$

For integers only, we must also provide three additional functions "gcd", "lcm" and "mod" borrowed from number theory. The first two construct the greatest common divisor and the least common multiple of two integers, as obtained using the well-known Euclid's algorithm. In this work, their type signature are represented as follows:

$$\text{gcd}[\![\cdot]\!], \text{lcm}[\![\cdot]\!] :: (\textit{integer}, \textit{integer}) \to \textit{integer}$$

The third, "mod", is used to implement proper modulo arithmetic. Given an arbitrary integer x and a positive integer y, "x mod y" returns the least non-negative integer that is congruent to x in arithmetic modulo y. An efficient implementation of this function is readily available in the standard Haskell library and comes with the following type signature:

$$[\![\cdot]\!] \text{ mod } [\![\cdot]\!] :: \textit{integer} \to \textit{integer} \to \textit{integer}$$

In Appendix C, we also find it necessary to apply the "lcm" operator to an entire list of values. Using the built-in Haskell function "foldl" described in Section A.6, this construction can be implemented as follows:

$$\text{lcm}[\![\cdot]\!] :: [\textit{integer}] \to \textit{integer}$$
$$\text{lcm}[\![\bar{x}]\!] = \text{fold } \textit{lcm } 1 \ \bar{x}$$

A careful use of the "mod" operator also permits us to extract individual bits of data from a given integer value x. In particular, the ith bit in the two's complement representation of such an integer can be obtained using the following formula:

$$[\![\cdot]\!] \ [\![\cdot]\!] :: \textit{integer} \to \textit{integer} \to \textit{integer}$$
$$[\![x]\!] \ [\![i]\!] = \lfloor x/2^i \rfloor \bmod 2$$

Finally, an arbitrary subset of bits extracted in this fashion can be summed up backed into an integer value, giving rise to the following useful notation:

$$[\![\cdot]\!] \ [\![\cdot]\!] :: \textit{integer} \to [\textit{integer}] \to \textit{integer}$$
$$[\![x]\!] \ [\![\bar{i}]\!] = \sum [x(k) \times 2^{k \, - \, \min(\bar{i})} \mid k \leftarrow \bar{i}]$$

This completes the entire suite of arithmetic functions required for a satisfactory implementation of our C compiler. Although many of the above definitions are clearly sub-optimal, they are nevertheless correct. Readers are encouraged to improve upon them and to apply the theorem proving capabilities from Chapter 3 in order to establish a semantic neutrality of any such improvements.

A.6 Function Combinators

Of course, numbers are not the only entities manipulated by our compiler. In particular, our implementation often operates on Haskell functions themselves. Perhaps the most useful of all such *function combinators* is the composition operator "$\circ$", whose well-known implementation is included in the standard Haskell library as follows:

$$[\![\cdot]\!] \circ [\![\cdot]\!] :: (U \to V) \to (T \to U) \to (T \to V)$$

$$[\![f]\!] \circ [\![g]\!] = \lambda x.f(g(x))$$

Intuitively, given a pair of unary functions f and g, $f \circ g$ represents a new unary function that begins by applying g to its operand and produces the value obtained from an application of f to the result of g. Such function composition may be also applied naturally to an entire list of functions of the Haskell type $T \to T$:

$$\odot[\![\cdot]\!] :: [T \to T] \to (T \to T)$$

$$\odot[\![\bar{f}]\!] = \text{fold} \circ \text{id}\ \bar{f}$$

in which "id" represents the following trivial identity function:

$$\text{id}[\![\cdot]\!] :: T \to T$$

$$\text{id}[\![x]\!] = x$$

and "fold $f\ y\ \bar{x}$" depicts the *left fold combinator*, which, beginning with the value y, systematically applies $f(y, x_k)$ to all $x_k \in \bar{x}$:

$$\text{fold}\ [\![\cdot]\!]\ [\![\cdot]\!]\ [\![\cdot]\!] :: (T \to U \to T) \to T \to [U] \to T$$

$$\begin{aligned}\text{fold}\ [\![f]\!]\ [\![y]\!]\ [\![\varnothing]\!] &= y \\ \text{fold}\ [\![f]\!]\ [\![y]\!]\ [\![x{:}\bar{x}]\!] &= \text{fold}\ f\ (f(y, x))\ \bar{x}\end{aligned}$$

This combinator is, of course, available as part of the standard Haskell library and serves the important rôle of combining list values into a single result. However, in this work, it is seldom used directly, other than through its implicit application in most list operations such as $\sum$, $\prod$ and $\odot$.

A.7 List Operations

The compiler implementation presented in this work relies heavily on the standard polymorphic list type "$[T]$". Individual values of this type include the empty list $\varnothing$, together with nodes of the form "$x{:}\bar{x}$". Most often, however, list values are represented by an explicit syntax such as $[1, 2, 3]$, $[1 \ldots 3]$ or $[f(x_k) \mid x_k \leftarrow \bar{x}]$. The later form represents a list obtained by applying the function f to every element of some other list $\bar{x}$

and all of these notations are supported natively by the Haskell programming language. Further, to make many list comprehension expressions such as $[x \times k \mid x_k \leftarrow \bar{x}]$ clearer, we will write the standard Haskell function "zip" as an infix operator "|". Intuitively, $\bar{a}_1|\bar{a}_2$ converts a pair of lists into a single list of pairs, in which each pair is assembled from the corresponding elements of its two operands, after truncating both lists to the length of the shorter argument. Its type signature assumes the following form:

$$[\![\cdot]\!] \mid [\![\cdot]\!] :: [T] \rightarrow [U] \rightarrow [T, U]$$

The actual length of a finite Haskell list can be obtained using the standard function "length", which is introduced with the following type signature:

$$\text{length}[\![\cdot]\!] :: [T] \rightarrow \textit{integer}$$

Further, a concatenation of two lists is represented by the familiar notation "$\bar{x} +\!\!+ \bar{y}$":

$$[\![\cdot]\!] +\!\!+ [\![\cdot]\!] :: [T] \rightarrow [T] \rightarrow [T]$$

The dual operation of deconstructing a list into two sub-lists is most commonly performed with the help of standard Haskell functions "head", "tail", "take" and "drop". In particular, given a non-empty list $\bar{x}$, head($\bar{x}$) and tail($\bar{x}$) return the list's first element and the remainder of the list, respectively, while last($\bar{x}$) and init($\bar{x}$) return the final element and all elements preceding it. Further, reverse($\bar{x}$) conveniently reverses the ordering of all elements in $\bar{x}$, so that head(reverse($\bar{x}$)) is always equal to last($\bar{x}$), unless $\bar{x}$ represents an empty list:

$$\text{head}[\![\cdot]\!], \text{last}[\![\cdot]\!] :: [T] \rightarrow T$$
$$\text{tail}[\![\cdot]\!], \text{init}[\![\cdot]\!], \text{reverse}[\![\cdot]\!] :: [T] \rightarrow [T]$$

None of these five functions is ever defined on an empty list $\varnothing$.

Finally, given an arbitrary list $\bar{x}$ and an integer n, take(n, $\bar{x}$) retrieves the first n elements of $\bar{x}$, while drop(n, $\bar{x}$) discards these elements, returning the remainder of $\bar{x}$. If n is 0 or negative, "take" will return an empty list while "drop" will return its operand unchanged. Conversely, if n is greater than the length of $\bar{x}$, "take" will return $\bar{x}$ unchanged and "drop" will return an empty list. The Haskell signatures of both functions are given as follows:

$$\text{take}[\![\cdot]\!], \text{drop}[\![\cdot]\!] :: (\textit{integer}, [T]) \rightarrow [T]$$

Finally, in Section 4.6, we rely on one more list combinator "$\mathcal{P}(k, \bar{x})$", which returns the kth permutation of the list $\bar{x}$:

$$\mathcal{P}[\![\cdot]\!] :: (\textit{integer}, [T]) \rightarrow [T]$$

A.8 Finite Sets

No treatment of algebraic data types would be complete without providing a convenient polymorphic implementation of *sets*. Fortunately, the standard Haskell library comes complete with a very capable implementation of this data structure, in the form of the standard "Data.Set" module. In this work, the set of n ordered Haskell expressions $x_1, x_2 \ldots x_n :: T$ is represented by the familiar notation "$\{x_1, x_2, x_n\}$" and its type

is written simply as "$\{T\}$". In this work, I also recognise the notion of set comprehensions, writing them as expression of the form $\{1 \ldots 7\}$ and even $\{f(x_k) \mid x_k \leftarrow \bar{x}\}$. Formally, such notation can be given a formal mandate by confusing the concepts of sets and lists whenever the context of the discussion provides sufficient disambiguation of the term, which gives rise to the following pair of implicit coercion functions:

$$[\![\cdot]\!] :: (\text{ord } T) \Rightarrow [T] \rightarrow \{T\}$$
$$[\![\cdot]\!] :: (\text{ord } T) \Rightarrow \{T\} \rightarrow [T]$$

The *cardinality* of a set, or the number of distinct elements stored within it, is represented by standard mathematical notation $|\bar{x}|$:

$$|[\![\cdot]\!]| :: (\text{ord } T) \Rightarrow \{T\} \rightarrow \textit{integer}$$

The Haskell library also provides an efficient direct access to the smallest and greatest element of a set of ordered values using the following pair of functions:

$$\min[\![\cdot]\!], \max[\![\cdot]\!] :: (\text{ord } T) \Rightarrow \{T\} \rightarrow T$$

Most often, however, we are merely concerned with the question of set membership. Given a value x and a set $\bar{x}$, the Haskell expression "$x \in \bar{x}$" returns "true" if x can be found in the supplied set and "false" otherwise. This common operator (and its dual "$\notin$") are readily available in the standard Haskell library:

$$[\![\cdot]\!] \in [\![\cdot]\!], [\![\cdot]\!] \notin [\![\cdot]\!] :: (\text{ord } T) \Rightarrow T \rightarrow \{T\} \rightarrow \textit{bool}$$

Similarly, the subset operator "$\subseteq$" and its strict variant "$\subset$" have the following Haskell declarations:

$$[\![\cdot]\!] \subset [\![\cdot]\!], [\![\cdot]\!] \subseteq [\![\cdot]\!] :: (\text{ord } T) \Rightarrow \{T\} \rightarrow \{T\} \rightarrow \textit{bool}$$

Further, a union, intersection and difference of two sets can be obtained using the standard mathematical notations "$\bar{x}_1 \cup \bar{x}_2$", "$\bar{x}_1 \cap \bar{x}_2$" and "$\bar{x}_1 \backslash \bar{x}_2$", respectively. They are implemented by the following Haskell functions:

$$[\![\cdot]\!] \cup [\![\cdot]\!], [\![\cdot]\!] \cap [\![\cdot]\!], [\![\cdot]\!] \setminus [\![\cdot]\!] :: (\text{ord } T) \Rightarrow \{T\} \rightarrow \{T\} \rightarrow \{T\}$$

Similarly, the cross-product of two sets can be obtained using the following standard notation "$\bar{x}_1 \times \bar{x}_2$":

$$[\![\cdot]\!] \times [\![\cdot]\!] :: (\text{ord } T, \text{ord } U) \Rightarrow \{T\} \rightarrow \{U\} \rightarrow \{T, U\}$$

Last but not least, the "$\cup$" and "$\cap$" operators may be also applied wholesale to an entire list of sets:

$$\bigcup[\![\cdot]\!], \bigcap[\![\cdot]\!] :: (\text{ord } T) \Rightarrow [\{T\}] \rightarrow \{T\}$$

Highly optimised implementations of the above set operations are readily available in the standard Haskell library. However, the reader should observe that most of the set functions described in this section are defined only for types that are a member of the Haskell class "ord". When no sensible total ordering is available for a type whose sets are constructed by our compiler implementation, the reader should assume that an adequate instance of the "ord" class has been constructed implicitly using the standard instance derivation mechanism supported by the language.

A.9 Finite Maps

A *finite map* is a unary function defined over a finite domain of distinct values in such a way that it can be *extended* with additional definitions during execution of the program. The standard Haskell library conveniently provides us with an efficient implementation of such constructs for ordered domain types, in the form of the "Data.Map" module. In this work, the type of a map from an ordered domain type T to U is written as "$T \mapsto U$". The most primitive value of such a type is the *empty map* $\varnothing$, whose domain is guaranteed to be empty. Other map values may be constructed using the familiar set comprehension notation such as $\{x_1{:}y_1,\, x_2{:}y_2,\, x_3{:}y_3\}$ and $\{f(x_k,\, y_k) \mid x_k{:}y_k \leftarrow M\}$, in which every value x_k represents an element of the resulting map's domain. In the resulting map, each x_k is bound to the corresponding value y_k. Similarly to the earlier set type, these notations are mandated by the following pair of implicit coercions between finite maps and lists of tuples:

$$[\![\cdot]\!] :: (\text{ord } T) \Rightarrow [T,\, U] \rightarrow (T \mapsto U)$$
$$[\![\cdot]\!] :: (\text{ord } T) \Rightarrow (T \mapsto U) \rightarrow [T,\, U]$$

The cardinality or size of a finite map is depicted by the standard mathematical notation $|M|$, with a type similar to the corresponding set operation:

$$|[\![\cdot]\!]| :: \text{ord } T \Rightarrow (T \mapsto U) \rightarrow \textit{integer}$$

A set of all elements present in the domain or codomain of a given finite map M is represented by the respective notations "dom(M)" and "codom(M)". A native implementation of both constructions is provided in the standard library as the following pair of Haskell functions:

$$\text{dom}\quad [\![\cdot]\!] :: (\text{ord } T) \Rightarrow (T \mapsto U) \rightarrow \{T\}$$
$$\text{codom}[\![\cdot]\!] :: (\text{ord } U) \Rightarrow (T \mapsto U) \rightarrow \{U\}$$

In practice, finite maps are generally considered to represent partial functions. In this work, an application of a map $\bar{m}$ to a value x from the map's domain is typeset as like an ordinary function application $\bar{m}(x)$. In the real life of Haskell, such map applications are executed by a function with the following type signature:

$$[\![\cdot]\!]\ [\![\cdot]\!] :: (\text{ord } T) \Rightarrow (T \mapsto U) \rightarrow T \rightarrow U$$

To complete the partial function illusion, we can also define the standard composition operator "$\circ$" on finite maps as follows:

$$[\![\cdot]\!] \circ [\![\cdot]\!] :: (\text{ord } T,\, \text{ord } U) \Rightarrow (U \mapsto V) \rightarrow (T \mapsto U) \rightarrow (T \mapsto V)$$
$$[\![M]\!] \circ [\![N]\!] = \{x{:}M(y) \mid (x{:}y) \leftarrow N,\, y \in \text{dom}(M)\}$$

Given two or more finite maps with disjoint domain sets, a new map that consists of all the bindings found in any of the supplied operands can be constructed with one of the following familiar union operators:

$$[\![\cdot]\!] \cup [\![\cdot]\!] :: (\text{ord } T) \Rightarrow (T \mapsto U) \rightarrow (T \mapsto U) \rightarrow (T \mapsto U)$$

$$\bigcup [\![\cdot]\!] :: (\text{ord } T) \Rightarrow [T \mapsto U] \rightarrow (T \mapsto U)$$

Both operations are explicitly left undefined on arguments with overlapping domains. When combining such overlapping finite maps, a different kind of a binary map operator, known as a *map extension* or a *right-biased union*, is required. In this work, an extension of a given map $\bar{m}_1$ by $\bar{m}_2$ is represented by the notation "$\bar{m}_1 / \bar{m}_2$". Formally, it is implemented in Haskell as follows:

$$[\![\cdot]\!] \,/\, [\![\cdot]\!] :: (\text{ord } T) \Rightarrow (T \mapsto U) \rightarrow (T \mapsto U) \rightarrow (T \mapsto U)$$

$$[\![M]\!] \,/\, [\![N]\!] = (M \backslash \text{dom}(N)) \cup N$$

In other words, M/N maps every element $x \in \text{dom}(N)$ to $N(x)$ and every element $y \notin \text{dom}(N)$ to $M(y)$. Intuitively, the new bindings introduced by the second operand of "/" override any existing mappings from M. This popular map operation is critical to a successful implementation of most symbol tables encountered in our compiler.

A careful reader will observe that, in the above definition, the map difference operator "\" was applied with a set, rather than a map as its second argument. In fact, it is often convenient to define a similar asymmetric map-set intersection operator "∩". Both of these functions are implemented in the standard library with the following obvious type signatures:

$$[\![\cdot]\!] \cap [\![\cdot]\!],\ [\![\cdot]\!] \backslash [\![\cdot]\!] :: (\text{ord } T) \Rightarrow (T \mapsto U) \rightarrow \{T\} \rightarrow (T \mapsto U)$$

As for finite sets, a complete implementation of every data type and function described in this section is readily available in the standard Haskell module "Data.Map". Since the art of purely functional data type constructions is not the primary subject of this work, I refer an interested reader to the existing relevant literature for a further discussion of the topic [Adams 93].

BIBLIOGRAPHY

[ANSI 68] *USA Standard COBOL*, *American Standard USAS X3.23-1968*, American National Standards Institute, 1968.

[ANSI 89] *Programming Languages — C*, *International Standard ANSI/ISO 9899:1990*, The Institute of Electrical and Electronics Engineers, Inc, 1992.

[ANSI 99] *Programming Languages — C*, *International Standard ISO/IEC 9899:1999(E)*, American National Standards Institute, 2000.

[Abramsky 93] Samson Abramsky and C.H. Luke Ong. Full Abstraction in the Lazy Lambda Calculus. *Information and Computation*, Volume 105(2), 1993.

[Abramsky 94] Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, Volume 59, pp. 543–574, 1994.

[Abramsky 99] Samson Abramsky and Paul-André Melliès. Concurrent games and full completeness. In *Proceedings of the Fourteenth International Symposium on Logic in Computer Science*, pp. 431–442. Computer Science Press of the IEEE, 1999.

[Aczel 96] Amir D. Aczel. *Fermat's Last Theorem: Unlocking the Secret of an Ancient Mathematical Problem*, Delta.

[Adams 93] Stephen Adams. Efficient sets — a balancing act. *Journal of Functional Programming*, Volume 3(4), pp. 553–561, October 1993.

[Aho 86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers— Principles, Techniques and Tools*, Addison-Wesley, 1986.

[Appel 98] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, Volume 33(4), 1998.

[Appel 98ML] Andrew W. Appel. *Modern Compiler Implementation in ML*, Cambridge University Press, 1998.

[Attali 93] Isabelle Attali, Denis Caromel and Michael Oudshoorn. A Formal Definition of the Dynamic Semantics of the Eiffel Language. In *Proceedings of the 16th Australian Computer Science Conference*, pp. 109–120, Brisbane, Australia. Griffith University, 1993.

[Backus 54] John W. Backus, Harlan Herrick and Irving Ziller. *Preliminary Report: Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN*, Programming Research Group, Applied Science Division, International Business Machines Corporation, November 1954.

[Backus 59] John W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing*, pp. 125–132, UNESCO, 1959.

[Barendregt 84] Hendrik Pieter Barendregt. *Lambda Calculus, Its Syntax and Semantics*, North-Holland, 1984.

[Bauer-Mengelberg 67] Stefan Bauer-Mengelberg. On the building blocks of mathematical logic. In *A Source Book in Mathematical Logic, 1879-1931*, pp. 355–366. Harvard University Press, 1967.

[Bekic 74] Hans Bekic, Dines Bjørner, Wolfgang Henhapl, Cliff B. Jones and Peter Lucas. *A Formal Definition of a PL/I Subset*, IBM Laboratory, Vienna, Technical Report 25.139, September 1974.

[Berghofer 03] Stefan Berghofer and Martin Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In *Electronic Notes in Theoretical Computer Science*, Volume 82(2), pp. 377–394. Elsevier, April 2004.

[Bjørner 82a] Dines Bjørner and Cliff B. Jones. ALGOL 60. In *Formal Specification and Software Development, Chapter 6*, pp. 141–173. Prentice Hall, 1982.

[Bjørner 82b] Dines Bjørner and Cliff B. Jones. Pascal. In *Formal Specification and Software Development, Chapter 7*, pp. 175–251. Prentice Hall, 1982.

[Black 96] Paul E. Black and Phillip J. Windley. Inference Rules for Programming Languages with Side Effects In Expressions. In *Proceeding of the Ninth International Conference on Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science*, Volume 1125, pp. 51–60, Turku, Finland. Springer-Verlag, 1996.

[Blakley 92] George Robert Blakley. *A Smalltalk Evolving Algebra and its Uses*, PhD Thesis, University of Michigan, 1992.

[Blazy 06] Sandrine Blazy, Zaynah Dargaye and Xavier Leroy. Formal Verification of a C Compiler Front-End. In *International Symposium on Formal Methods*, *Lecture Notes in Computer Science*, Volume 4085, pp. 460–475. Springer-Verlag, 2006.

[Blech 04] Jan Olaf Blech and Sabine Glesner. A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL. In *Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft für Informatik*. Lecture Notes in Informatics, September 2004.

[Blum 89] Manuel Blum and Sampath Kanna. Designing programs that check their work. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pp. 86–97, Seattle, Washington, USA. ACM Press, 1989.

[Blum 95] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, Volume 42(1), pp. 269–291, 1995.

[Börger 94] Egon Börger, Igor Đurđanović and Dean Rosenzweig. Occam: Specification and Compiler Correctness. Part I: The Primary Model. In *Proceedings of PROCOMET'94, IFIP Working Conference on Programming Concepts, Methods and Calculi*, pp. 489–508. North-Holland, 1994.

[Börger 95a] Egon Börger and Dean Rosenzweig. A mathematical definition of Full Prolog. *Science of Computer Programming*, Volume 24(3), pp. 249–286, 1995.

[Börger 95b] Egon Börger and Dean Rosenzweig. The WAM — Definition and Compiler Correctness. *Logic Programming: Formal Methods and Practical Applications*, 1995.

[Börger 96] Egon Börger and Igor Đurđanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, Volume 39(1), pp. 52–92, 1996.

[Broy 80] Manfred Broy and Martin Wirsing. Programming Languages as Abstract Data Types, Université de Lille, 1980.

[Broy 87] Manfred Broy, Martin Wirsing and Peter Pepper. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, Volume 9(1), pp. 54–99, 1987.

[Burstall 69] Rod M. Burstall and Peter J. Landin. Programs and their proofs: an algebraic approach. In *Machine Intelligence*, Volume 4(2), pp. 17–44. Edinburgh University PRess, 1969.

[Chakravarty 03] Manuel M.T. Chakravarty, Gabriele Keller and Patryk Zadarnowski. A Functional Perspective on SSA Optimisation Algorithms. In *Electronic Notes in Theoretical Computer Science*, Volume 82(2), pp. 347–361. Elsevier, April 2004.

[Chirica 86] Laurian M. Chirica and David F. Martin. Toward compiler implementation correctness proofs. *ACM Transactions on Programming Languages and Systems*, Volume 8(2), pp. 185–214, 1986.

[Church 36] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, Volume 58(2), pp. 345–363, April 1936.

[Church 40] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, Volume 5(2), pp. 56–68, June 1940.

[Church 41] Alonzo Church. *The Calculi of Lambda Conversion*, Princeton University Press, 1941.

[Cleaveland 80] J. Craig Cleaveland. Programming languages considered as abstract data types. In *Proceedings of the ACM 1980 annual conference (ACM'80)*, pp. 236–245. ACM Press, 1980.

[Clemmensen 84] Geert B. Clemmensen and Ole N. Oest. Formal specification and development of an Ada compiler: a VDM case study. In *Proceedings of the Seventh International Conference on Software Engineering*, pp. 430–440, Orlando, FL, USA. IEEE Press, 1984.

[Cohn 78] Avra Cohn. *High level proof in LCF*, Department of Computer Science, University of Edinburgh, CSR-35-78, November 1978.

[Cohn 88] Avra Cohn. A proof of correctness of the Viper microprocessor: The first level. In *VLSI Specification, Verification and Synthesis*, pp. 1–91. Kluwer Academic Publishers, 1988.

[Cohn 89] Avra Cohn. Correctness properties of the Viper block model: The second level. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 27–72. Springer-Verlag, 1989.

[Conway 58] Melvin E. Conway. Proposal for an UNCOL. *Communications of the ACM*, Volume 1(10), pp. 5–8, 1958.

[Cook 94a] Jeffrey V. Cook, Eve Cohen and Tim Redmond. *A Formal Denotational Semantics for C*, Trusted Information Systems, Technical Report 409D, September 1994.

[Cook 94b] Jeffrey V. Cook and Sakthi Subramanian. *A Formal Semantics for C in Nqthm*, Trusted Information Systems, Technical Report 517D, October 1994.

[Coquand 86] Thierry Coquand and Gérard Huet. *The Calculus of Constructions*, INRIA, N°530, May 1986.

[Crary 03] Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pp. 198–212, New Orleans, Louisiana, USA. ACM Press, 2003.

[Curien 07] Pierre-Louis Curien. Definability and Full Abstraction. *Electronic Notes in Theoretical Computer Science*, Volume 172, pp. 301–310, April 2007.

[Currie 95] I.F. Currie. *TDF Specification, Issue 4.0*, DERA, DRA/CIS(SE2)/CR/94/36/4.0, June 1995.

[Curry 58] Haskell B. Curry and Robert Feys. *Combinatory Logic I*, North-Holland, 1972.

[Curry 72] Haskell B. Curry, J. Roger Hindley and Jonathan P. Seldin. *Combinatory Logic II*, North-Holland, 1972.

[Curzon 91] Paul Curzon. A Verified Compiler for a Structured Assembly Language. In *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pp. 253–262. IEEE Computer Society Press, 1991.

[Cytron 91] Ron K. Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and Kenneth F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, Volume 13(4), pp. 451–490, 1991.

[da Silva 92] Fabio Q. B. da Silva. *Observational Equivalence and Compiler Correctness*, University of Edinburgh, Laboratory for Foundations of Computer Science, ECS-LFCS-92-240, September 1992.

[Danvy 03] Olivier Danvy. A New One-Pass Transformation into Monadic Normal Form, Warsaw, Poland. Springer Verlag, April 2003.

[Denney 98] Ewen W.K.C. Denney. *A Theory of Program Refinement*, PhD Thesis, University of Edinburgh, 1998.

[Despeyroux 86] Joëlle Despeyroux. Proof of Translation in Natural Semantics. In *Proceedings of the First Symposium on Logic in Computer Science*, *LICS*, pp. 16–18, Cambridge, MA, USA. IEEE Computer Society, June 1986.

[Diehl 96] Stephan Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*, PhD Thesis, Universität des Saarlandes, 1996.

[Dijkstra 76] Edsger W. Dijkstra. *A Discipline of Programming*, Prentice Hall, 1976.

[Dold 98] Axel Dold, Thilo Gaul, Vincent Vialard and Wolf Zimmermann. ASM-Based Mechanized Verification of Compiler Backends. In *Proceedings of the 5th International Workshop on Abstract State Machines*, pp. 50–67, Magdeburg, Germany, 1998.

[Flanagan 93] Cormac Flanagan, Amr Sabry, Bruce F. Duba and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI'93)*, pp. 237–247, Albuquerque, New Mexico, USA. ACM Press, 1993.

[Floyd 67] Robert W. Floyd. Assigning Meanings to Programs. In *Proceedings of a Symposium in Applied Mathematics*, New York City. American Mathematical Society, April 1966.

[Gaul 99] Thilo Gaul, Andreas Heberle, Wolf Zimmermann and Wolfgang Goerigk. Construction of Verified Software Systems with Program-Checking: An Application To Compiler Back-Ends. In *Proceedings of the Workshop on Run-Time Result Verification, Federated Logic Conference*, Trento, Italy, 1999.

[Gettier 63] Edmund L. Gettier. Is Justified True Belief Knowledge?. *Analysis*, Volume 23(6), pp. 121–123, June 1963.

[Giesl 04] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp and Stephan Falke. Automated Termination Proofs with AProVE. In *RTA '2004: Rewriting Techniques and Applications*, *Lecture Notes in Computer Science*, Volume 3091, pp. 210–220. Springer-Verlag, 2004.

[Glesner 02] Sabine Glesner, Rubino Geiß and Boris Boesler. Verified Code Generation for Embedded Systems. In *Electronic Notes in Theoretical Computer Science*, Volume 65(2), pp. 19–36. Elsevier, 2002.

[Glesner 03] Sabine Glesner. Using Program Checking to Ensure the Correctness of Compiler Implementations. *Journal of Universal Computer Science*, Volume 9(3), pp. 191–222, 2003.

[Gödel 31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, Volume 38, pp. 173–198, 1931.

[Gödel 40] Kurt Gödel. *The Consistency of the Axiom of Choice and of the Generalized Continuum Hypothesis with the Axioms of Set Theory*, Princeton University Press, 1940.

[Goerigk 96] Wolfgang Goerigk, Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich W. von Henke, Ulrich Hoffmann, Hans Langmaack, Holger Pfeifer, Harald Ruess and Wolf Zimmermann. Compiler Correctness and Implementation Verification: The Verifix Approach. In *International Conference on Compiler Construction*, Linkøping, Sweden, 1996.

[Goos 99] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In *Correct System Design, Recent Insight and Advances (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, *Lecture Notes in Computer Science*, Volume 1710, pp. 201–230. Springer-Verlag, 1999.

[Goos 00] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In *Abstract State Machines: Theory and Applications*, *Lecture Notes in Computer Science*, Volume 1912, pp. 281–297, Monte Verità, Switzerland. Springer-Verlag, March 2000.

[Gordon 88] Michael J.C. Gordon. HOL: A proof generating system for higher order logic. In *VLSI Specification, Verification and Synthesis*, pp. 73–128. Kluwer Academic Publishers, 1988.

[Gurevich 88] Yuri Gurevich and James M. Morris. Algebraic Operational Semantics and Modula-2. In *Proceedings of the 1st Workshop on Computer Science Logic*, *Lecture Notes in Computer Science*, Volume 329, pp. 81–101. Springer-Verlag, 1988.

[Gurevich 90] Yuri Gurevich and Lawrence S. Moss. Algebraic Operational Semantics and Occam. In *Proceedings of the Third Workshop on Computer Science Logic*, *Lecture Notes in Computer Science*, Volume 440, pp. 176–192. Springer-Verlag, 1990.

[Gurevich 93] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language, February 1993.

[Hannan 93] John Hannan. Searching for semantics. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 1–12, Copenhagen, Denmark. ACM Press, 1993.

[Harper 87] Robert Harper, Furio Honsell and Gordon Plotkin. A Framework for Defining Logics. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, LICS'87*, pp. 194–204, Ithaca, NY, USA. IEEE Computer Society Press, July 1987.

[Hatcliff 94] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pp. 458–471, Portland, Oregon, USA. ACM Press, January 1994.

[Hoare 69] Sir Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Volume 12(10), pp. 576–580, 1969.

[Hoare 73] Sir Charles Antony Richard Hoare and Niklaus Wirth. An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica*, Volume 2, pp. 335–355, 1973.

[Hoare 03] Sir Charles Antony Richard Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, Volume 50(1), pp. 63–69, 2003.

[Hopper 57] Grace M. Hopper. Automatic Programming for Business Applications. In *Proceedings of the Fourth Annual Computer Applications Symposium*. Armour Research Foundation, Illinois Institute of Technology, October 1957.

[Howard 69] William A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pp. 479–490. Academic Press, 1980.

[Hudak 07] Paul Hudak, John Hughes, Simon Peyton Jones and Philip Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 12-1–12-55, San Diego, California. ACM Press, 2007.

[IEEE 91] *IEEE Standard for the Scheme Programming Language*, *IEEE Standard 1178-1990*, Institute of Electrical and Electronics Engineers, 1991.

[IEEE 754] *IEEE Standard for Binary Floating-Point Arithmetic*, *IEEE Std 754-1985*, American National Standards Institute, 1985.

[INRIA 02] *The Coq Proof Assistant Reference Manual, Version 7.2*, INRIA, Technical Report 255, 2002.

[Jones 00] Mark P. Jones. Type Classes with Functional Dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pp. 230–244. Springer-Verlag, 2000.

[Jones 03] Cliff Jones. Operational Semantics Revisited. In *Proceedings of the 2003 ETAPS SE-WMT Workshop on Structured Programming: The Hard Core of Software Engineering*, Warsaw, Poland, April 2003.

[Kaufmann 00] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, June 2000.

[Kelsey 95] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, pp. 13–22, San Francisco, CA, USA. ACM Press, 1995.

[Kelsey 98] Richard A. Kelsey *et al.*. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, Volume 11(1), August 1998.

[Kernighan 78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*, Prentice-Hall, 1978.

[Kleene 43] Stephen C. Kleene. Recursive Predicates and Quantifiers. In *Transactions of the American Mathematical Society*, Volume 53(1), pp. 41–73, January 1943.

[Knuth 84] Donald E. Knuth. *Computers & Typesetting*, Addison-Wesley, 1984–1986.

[Knuth 86] Donald E. Knuth. *The TEXbook*, Addison-Wesley, 1983.

[Knuth 05] Donald E. Knuth. *MMIX: A RISC Computer for the New Millennium*, Addison-Wesley, 2005.

[Kutter 96] Philipp W. Kutter. *Dynamic Semantics of the Oberon Programming Language*, ETH Zürich, TIK Report 25, February 1996.

[Kutter 97] Philipp W. Kutter and Alfonso Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, Volume 3(5), pp. 443–503, 1997.

[Landin 63] Peter J. Landin. Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. *Communications of the ACM*, Volume 8(2), pp. 89–101, 1965.

[Langmaack 89] H. Langmaack and M. Müller-Olm. *First Steps in Proven Correct Compiler Development in ProCoS*, Christian-Albrechts-Universität zu Kiel, October 1989.

[Langmaack 90] H. Langmaack, M. Fränzle and M. Müller-Olm. *Development of Proven Correct Compilers in ProCoS*, Christian-Albrechts-Universität zu Kiel, 1990.

[Lorenzen 61] Paul Lorenzen. Ein dialogisches Konstruktivitätskriterium. In *Infinitistic Methods*, pp. 193–200. Pergamon Press, 1961.

[Macrakis 92] Stavros Macrakis. *The Structure of ANDF: Principles and Examples*, Open Software Foundation, RI-ANDF-RP1-1, January 1992.

[Martin-Löf 71] Per Martin-Löf. *A Theory of Types*, University of Stockholm, Technical Report 71-3, 1971.

[Martin-Löf 83] Per Martin-Löf. On the Meanings of the Logical Constants and the Justifications of the Logical Laws. *Nordic Journal of Philosophical Logic*, Volume 1(1), pp. 11–60, 1996.

[Martin-Löf 96] Per Martin-Löf. On the meanings of the logical constants and the justifications of logical laws. *Nordic Journal of Philosophical Logic*, Volume 1(1), pp. 11–60, 1996.

[McCarthy 58] John McCarthy. Programs with Common Sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pp. 75–91. Her Majesty's Stationary Office, 1959.

[McCarthy 59] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, Volume 3(4), pp. 184–195, 1960.

[McCarthy 60] John McCarthy, Robert Brayton, Daniel J. Edwards, Phyllis A. Fox, Louis Hodes, David C. Luckham, Klim Maling, David M.R. Park and Steven R. Russell. *LISP I Programmer's Manual*, MIT Press, 1960.

[McCarthy 62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hard and Michael I. Levin. *LISP 1.5 Programmer's Manual*, MIT Press, 1962.

[McCarthy 67] John McCarthy and James Painter. Correctness of a Compiler for Arithmetic Expressions. In *Mathematical Aspects of Computer Science, Proceedings of the Symposium in Applied Mathematics*, Volume 19, pp. 33–41. American Mathematical Society, 1967.

[Milner 72a] Robin Milner. Implementation and Applications of Scott's Logic for Computable Functions. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, New Mexico State University. ACM Press, 1972.

[Milner 72b] Robin Milner and R. Weyhauch. Proving compiler correctness in a mechanised logic. In *Machine Intelligence*, Volume 7(3), pp. 51–70. Edinburgh University PRess, 1972.

[Milner 73] Robin Milner. *Models of LCF*, Stanford Artificial Intelligence Laboratory, STAN-CS-73-332, 1973.

[Milner 77] Robin Milner. Fully Abstract Models of Typed Lambda Calculi. *Theoretical Computer Science*, Volume 4(1), pp. 1–22, 1977.

[Milner 90] Robin Milner, Mads Tofte and Robert Harper. *The Definition of Standard ML*, MIT Press, February 1990.

[Milner 91] Robin Milner and Mads Tofte. *Commentary on Standard ML*, MIT Press, November 1990.

[Milner 92a] Robin Milner, Joachim Parrow and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, Volume 100(1), pp. 1–40, 1992.

[Milner 92b] Robin Milner, Joachim Parrow and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, Volume 100(1), pp. 41–77, 1992.

[Moggi 89] Eugenio Moggi. *An Abstract View of Programming Languages*, University of Edinburgh, Laboratory for Foundations of Computer Science, ECS-LFCS-90-113, 1990.

[Moggi 91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, Volume 93(1), pp. 55–92, July 1991.

[Moore 89] J Strother Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, Volume 5(4), pp. 461–492, 1989.

[Morris 73] F. Lockwood Morris. Advice on Structuring Compilers and Proving Them Correct. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'73)*, pp. 144–152, Boston, Massachusetts. ACM Press, 1973.

[Morris 88] James M. Morris. *Algebraic Operational Semantics for Modula-2*, PhD Thesis, University of Michigan, 1988.

[Morrisett 95] Greg Morrisett. *Compiling with Types*, PhD Thesis, Carnegie Mellon University, December 1995.

[Morrisett 99] Greg Morrisett, David Walker, Karl Crary and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, Volume 21(3), pp. 527–568, 1999.

[Mulmuley 85] Ketan D. Mulmuley. *Full Abstraction and Semantic Equivalence*, MIT Press, 1987.

[Necula 97] George C. Necula. Proof-Carrying Code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106–119, Paris, France, January 1997.

[Norrish 97] Michael Norrish. *An Abstract Dynamic Semantics for C*, University of Cambridge, TR421, May 1997.

[Okuma 03] Koji Okuma and Yasuhiko Minamide. Executing Verified Compiler Specification. In *Programming Languages and Systems*, *Lecture Notes in Computer Science*, Volume 2895, pp. 178–194. Springer, 2003.

[Orejas 81] Fernando Orejas. Even more on advice on structuring compilers and proving them correct: changing an arrow. *SIGPLAN Notices*, Volume 16(3), pp. 82–84, 1981.

[Owre 92] S. Owre, J.M. Rushby and N. Shankar. PVS: A Prototype Verification System. In *CADE 11: Proceedings of the 11th International Conference on Automated Deduction*, *Lecture Notes in Artificial Intelligence*, Volume 607, pp. 748–752, Saratoga, NY, USA. Springer-Verlag, June 1992.

[Palsberg 92] Jens Palsberg. A Provably Correct Compiler Generator. In *Proceedings of the Fourth European Symposium on Programming (ESOP'92)*, *Lecture Notes in Computer Science*, Volume 582, pp. 418–434, Rennes, France. Springer-Verlag, February 1992.

[Papaspyrou 98] Nikolaos S. Papaspyrou. *A Formal Semantics for the C Programming Language*, PhD Thesis, National Technical University of Athens, February 1998.

[Paulson 90] Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. *Logic and Computer Science*, 1990.

[Paulson 05] Lawrence C. Paulson with contributions by Tobias Nipkow and Markus Wenzel. *The Isabelle Reference Manual*, University of Cambridge Computer Laboratory, 2005.

[Pedersen 80] Jan Storbank Pedersen. A Formal Semantics Definition of Sequential Ada. In *Towards a Formal Description of Ada*, *Lecture Notes in Computer Science*, Volume 98, pp. 213–308. Springer-Verlag, 1980.

[Peyton Jones 87] Simon Peyton Jones. *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[Peyton Jones 93] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 71–84. ACM, 1993.

[Peyton Jones 96] Simon Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In *Proceedings of the Sixth European Symposium on Programming Languages and Systems*, *Lecture Notes in Computer Science*, Volume 1058, pp. 18–44. Springer-Verlag, January 1996.

[Peyton Jones 98] Simon Peyton Jones and André L.M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, Volume 32(1–3), pp. 3–47, September 1998.

[Peyton Jones 98b] Simon Peyton Jones, Thomas Nordin and Dino Oliva. C--: A Portable Assembly Language. In *Implementation of Functional Languages*, *Lecture Notes in Computer Science*, Volume 1467, pp. 1–19. Springer-Verlag, 1998.

[Peyton Jones 03] Simon Peyton Jones *et al.*. *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, 2003.

[Peyton Jones 06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich and Geoffrey Washburn. Simple unification-based type inference for GADTs. *SIGPLAN Notices*, Volume 41(9), pp. 50–61, 2006.

[Pitman 79] Kent Pitman. A Fortran → Lisp Translator. In *Proceedings of the 1979 Macsyma User's Conference*, Washington, D.C., USA, June 1979.

[Plotkin 77] Gordon D. Plotkin. LCF Considered as a Programming Language. *Theoretical Computer Science*, Volume 5(3), pp. 225–255, 1977.

[Plotkin 81] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*, University of Aarhus, DAIMI FN-19, 1981.

[Polak 81] Wolfgang Polak. *Compiler Specification and Verification*, *Lecture Notes in Computer Science*, Springer-Verlag, 1981.

[Reynolds 82] John C. Reynolds. The essence of ALGOL. In *International Symposium on Algorithmic Languages*, pp. 345–372, Amsterdam, The Netherlands. North-Holland, 1982.

[Rosser 36] Alonzo Church and J.B. Rosser. Some Properties of Conversion. In *Transactions of the American Mathematical Society*, Volume 39(3), pp. 472–482, May 1936.

[Sabry 92] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style.. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, pp. 288–298, San Francisco, California, USA. ACM Press, 1992.

[Sampaio 90] A. Sampaio. *A Comparative Study of Theorm Provers: Proving Correctness of Compiling Specifications*, Oxford University Computing Laboratory, Technical Report PRG-20-90, 1990.

[Schönfinkel 24] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, Volume 92, pp. 305–316, 1924.

[Scott 71] Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, Oxford University Computing Laboratory, Programming Research Group Technical Monograph PRG-6, 1971.

[Scott 82] Dana Scott. Domains for Denotational Semantics. In *International Colloquium on Automata, Languages and Programs*, *Lecture Notes in Computer Science*, Volume 140, pp. 577–613. Springer-Verlag, 1982.

[Sewell 07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar and Rok Strniša. Ott: Effective Tool Support for the Working Semanticist. *SIGPLAN Notices*, Volume 42(9), pp. 1–12, 2007.

[Shürmann 03] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In *TPHOL '03: Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science*, Volume 2758, pp. 120–135. Springer-Verlag, 2003.

[Simpson 90] Todd Simpson. *Correctness of a Compiler Specification for the SECD Machine*, Department of Computer Science, The University of Calgary, 1990-410-34, October 1990.

[Smith 84] Brian Cantwell Smith. Reflection and semantics in LISP. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 23–35, Salt Lake City, Utah, USA. ACM Press, January 1984.

[Stringer-Calvert 98] David W.J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness*, PhD Thesis, University of York, March 1998.

[Tarditi 96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone and Robert Harper. TIL: a type-directed, optimizing compiler for ML. *SIGPLAN Notices*, Volume 39(4), pp. 554–567, 2004.

[Thatcher 80] James W. Thatcher, Eric G. Wagner and Jesse B. Wright. More on advice on structuring compilers and proving them correct. In *Proceedings of a Workshop on Semantics-Directed Compiler Generation*, *Lecture Notes in Computer Science*, Volume 94, pp. 165–188. Springer-Verlag, 1980.

[Turing 36] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, Volume 2(42), pp. 230–265, 1936.

[Twelf 98] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide, Version 1.2*, Carnegie Mellon University, CMU-CS-98-173, 1998.

[Vale 93] Marc Vale. *The Evolving Algebra Semantics of COBOL. Part I: Programs and Control*, University of Michigan, Department of Electrical Engineering and Computer Science, CSE-TR-162-93, 1993.

[Wadler 92] Philip Wadler. The Essence of Functional Programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 1–14, Albequerque, New Mexico, USA, January 1992.

[Wallace 93] Charles Wallace. *The Semantics of the C++ Programming Language*, University of Michigan, Department of Electrical Engineering and Computer Science, CSE-TR-190-93, 1993.

[Wallace 95] Charles Wallace. The Semantics of the C++ Programming Language. In *Specification and validation methods*, pp. 131–164. Oxford University Press, 1995.

[Wallace 97] Charles Wallace. *The Semantics of the Java Programming Language*, University of Michigan, Department of Electrical Engineering and Computer Science, CSE-TR-355-97, 1997.

[Wasserman 97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, Volume 44(6), pp. 826–849, 1997.

[Wegman 91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 13(2), pp. 181–210, 1991.

[Wegner 72] Peter Wegner. The Vienna Definition Language. *ACM Computing Surveys (CSUR)*, Volume 4(1), pp. 5–63, 1972.

[Wolczko 87] Mario Wolczko. Semantics of Smalltalk-80. In *Proceedings of the European Conference on Object-Oriented Programming*, *Lecture Notes in Computer Science*, Volume 276, pp. 108–120. Springer-Verlag, 1987.

[Young 89] William D. Young. Verified Compilation in micro-Gypsy. In *Proceedings of the ACM SIGSOFT '89 third symposium on Software Testing, Analysis, and Verification*, pp. 20–26, Key West, FL, USA. ACM Press, 1989.

[Zimmermann 97] Wolf Zimmermann and Thilo Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science*, Volume 3(5), pp. 504–567, August 1997.

IMPLEMENTATION OF C COMPILERS

In order to complete the translation semantics of C programs presented in Chapter 5, we must now provide a concrete Haskell implementation of every language parameter that was left unspecified in the earlier generic fragment of the language. On the following pages, I present one such implementation that is suitable for the MMIX architecture described in Chapter 6. The design choices made below represent the typical interpretation of the C Standard common to virtually all compilers for the language that are deployed in the industry.

First and foremost, every C compiler must settle on a concrete mapping between C types and Etude formats. On MMIX, these are assigned as follows:

$\phi[\![\cdot]\!] :: \mathit{type} \to \mathit{format}$

$$
\begin{aligned}
\phi[\![\texttt{signed char}]\!] &= [\![\mathsf{Z}.8]\!] \\
\phi[\![\texttt{signed short}]\!] &= [\![\mathsf{Z}.16]\!] \\
\phi[\![\texttt{signed int}]\!] &= [\![\mathsf{Z}.32]\!] \\
\phi[\![\texttt{signed long}]\!] &= [\![\mathsf{Z}.64]\!] \\
\phi[\![\texttt{char}]\!] &= [\![\mathsf{Z}.8]\!] \\
\phi[\![\texttt{int}]\!] &= [\![\mathsf{Z}.32]\!] \\
\phi[\![\texttt{unsigned char}]\!] &= [\![\mathsf{N}.8]\!] \\
\phi[\![\texttt{unsigned short}]\!] &= [\![\mathsf{N}.16]\!] \\
\phi[\![\texttt{unsigned int}]\!] &= [\![\mathsf{N}.32]\!] \\
\phi[\![\texttt{unsigned long}]\!] &= [\![\mathsf{N}.64]\!] \\
\phi[\![\texttt{float}]\!] &= [\![\mathsf{R}.64]\!] \\
\phi[\![\texttt{double}]\!] &= [\![\mathsf{R}.64]\!] \\
\phi[\![\texttt{long double}]\!] &= [\![\mathsf{R}.64]\!] \\
\phi[\![t\ \texttt{(}p_{opt}\texttt{)}]\!] &= [\![\mathsf{F}.64]\!] \\
\phi[\![\texttt{enum}\ t\ v]\!] &= [\![\mathsf{Z}.32]\!] \\
\phi[\![\mathit{type\text{-}qualifier}\ t]\!] &= \phi(t)
\end{aligned}
$$

Observe that, since the MMIX architecture from Chapter 6 provides for only one representation of all floating point values, in our C compiler all three floating types "**`float`**", "**`double`**" and "**`long double`**" share the common rational format "R.64".

Otherwise every signed bit field type t of a given width n is associated with the format "Z.n" and every unsigned such bit field is assigned the format "N.n". Further, all pointers to an object or incomplete type t and all arrays of such a type are represented by the object format corresponding to $\phi(t)$, while function pointers depict the same

standard function format "F.64". Finally, the format of a structure or union type is always derived from its member list using the construction $\phi(\bar{m}(t))$ that will be described shortly. Formally:

$$\begin{array}{lll}
\phi[\![t]\!] & \mid \mathrm{BF}(t) \wedge \mathrm{ST}(t) & = [\![\mathsf{Z}.[\![\mathcal{W}(t)]\!]]\!] \\
& \mid \mathrm{BF}(t) \wedge \mathrm{UT}(t) & = [\![\mathsf{N}.[\![\mathcal{W}(t)]\!]]\!] \\
& \mid (\mathrm{PTR}(t) \vee \mathrm{ARR}(t)) \wedge (\mathrm{OBJ}(\mathcal{B}(t)) \vee \mathrm{INC}(\mathcal{B}(t))) & = O(\phi(\mathcal{B}(t))) \\
& \mid \mathrm{PTR}(t) \wedge \mathrm{FUN}(t) & = [\![\mathsf{F}.64]\!] \\
& \mid \mathrm{SU}(t) & = \phi(\bar{m}(t))
\end{array}$$

In particular, all incomplete structure or union types are always assigned the same standard object format "O.64", since their empty member list cannot provide any specific information about the alignment of such objects. However, if $\bar{m}$ contains at least one member, then the format is defined as the greatest object format corresponding to the type of any $m_k \in \bar{m}$, with all bit field members taken to have the bit field format Ψ. Formally:

$$\phi[\![\cdot]\!] :: \mathit{members} \rightarrow \mathit{format}$$

$$\begin{array}{l}
\phi[\![\varnothing]\!] = [\![\mathsf{O}.64]\!] \\
\phi[\![\bar{m}]\!] = \max(\{O(\phi(\mathcal{T}(m_k))) \mid m_k \leftarrow \bar{m},\ \neg\mathrm{BF}(\mathcal{T}(m_k))\} \\
\qquad\qquad \cup\ \{O(\Psi) \mid m_k \leftarrow \bar{m},\ \mathrm{BF}(\mathcal{T}(m_k)) \wedge \mathcal{W}(\mathcal{T}(m_k)) > 0\})
\end{array}$$

Further, in Chapter 5, certain properties of C programs were formalised with the help of five designated C types "***char_t***", "***wchar_t***", "***int_t***", "***ptrdiff_t***" and "***size_t***" that were originally introduced in Section 5.4.2. Traditionally, these types have been selected as follows:

$$\textbf{\textit{char\_t}}, \textbf{\textit{wchar\_t}}, \textbf{\textit{int\_t}}, \textbf{\textit{ptrdiff\_t}}, \textbf{\textit{size\_t}} :: \mathit{type}$$

$$\begin{array}{ll}
\textbf{\textit{char_t}} & = [\![\textbf{\textit{signed char}}]\!] \\
\textbf{\textit{wchar_t}} & = [\![\textbf{\textit{signed int}}]\!] \\
\textbf{\textit{int_t}} & = [\![\textbf{\textit{signed int}}]\!] \\
\textbf{\textit{ptrdiff_t}} & = [\![\textbf{\textit{signed long}}]\!] \\
\textbf{\textit{size_t}} & = [\![\textbf{\textit{unsigned long}}]\!]
\end{array}$$

Like most other compilers, we also choose to represent every object of an enumeration type uniformly as a plain integer, regardless of the precise set of enumerator values introduced by that type. In Section 5.8.5, the C type corresponding to a given enumerator list $\bar{x}$ was represented by the implementation-specified construct $\mathcal{T}_{\mathrm{ENL}}(t)$, which we can now settle as follows:

$$\mathcal{T}_{\mathrm{ENL}}[\![\cdot]\!] :: [\mathit{integer}] \rightarrow \mathit{type}$$

$$\mathcal{T}_{\mathrm{ENL}}[\![\mathit{el}]\!] = [\![\textbf{\textit{int}}]\!]$$

Next on the agenda is the member layout function $\mathcal{L}$, that was utilised in Section 5.8.4 for completion of the structure and union type definitions. However, before introducing a concrete definition of that function, we should associate every well-formed object type with one additional property known as *alignment*. Intuitively, such an alignment represents the greatest common divisor of all integer values that are

suitable for representation of pointers to objects of that type. Formally, the alignment of a given type t is written as $\mathcal{A}(t)$. Its almost universally accepted implementation is represented by the following simple algorithm:

$$\mathcal{A}[\![\cdot]\!] :: \mathit{type} \rightarrow \mathit{integer}$$

$$\begin{aligned}\mathcal{A}[\![t]\!] \;&|\; \mathrm{SCR}\,(t) = \mathcal{S}(t)\\ &|\; \mathrm{ARR}(t) = \mathcal{A}(\mathcal{B}(t))\\ &|\; \mathrm{SU}\quad(t) = \mathrm{lcm}[\mathcal{A}(\mathcal{T}(m_k)) \;|\; m_k \leftarrow \bar{m}(t)]\end{aligned}$$

In other words, the alignment of all scalar types is always equal to their size, while, for arrays, it is determined by the alignment of the element type $\mathcal{B}(t)$. For structure and union types, however, the alignment is defined as the least common multiple of the individual alignments of the type's members $\bar{m}(t)$, so that, by definition, the size of every object and bit field type always represents an integral multiple of its alignment.

Using this auxiliary definition, we can now specify the precise behaviour of the member layout function $\mathcal{L}$. First of all, we must ensure that the total size of the resulting member list, defined in Section 5.4.6 as the greatest value among the sums of each member's size $\mathcal{S}(\mathcal{T}(m_k))$ with its offset $O(m_k)$ within the entire entity, always represents an integral multiple of its alignment, as required for a proper implementation of C arrays containing structure or union objects. The most direct way of guaranteeing this property is to extend every member list $\bar{m}'$ with an additional anonymous member located at the offset 0, whose type represents an array of $\lceil s/a \rceil \times a$ characters, recalling that the "**`char`**" type always has the predetermined size of 1. The C standard explicitly allows for an implicit incorporation of such *padding* into a structure or union type, so that the following definition of $\mathcal{L}$ also reflects the indented behaviour of these constructs envisioned by the authors of the C Standard:

$$\mathcal{L}[\![\cdot]\!] :: (\mathit{struct\text{-}or\text{-}union} \triangleright \mathit{members}) \rightarrow \mathit{members}$$

$$\mathcal{L}[\![su \triangleright \bar{m}]\!] = \bar{m}' +\!\!+ [\![\texttt{char[}[\![\lceil s/a \rceil \times a]\!]\texttt{]} \mathbin{@} 0]\!]$$

$$\begin{aligned}\text{where } \bar{m}' &= \mathcal{L}'(su \triangleright \bar{m})\\ s &= \max[O(m_k) + \mathcal{S}(\mathcal{T}(m_k)) \;|\; m_k \leftarrow \bar{m}']\\ a &= \mathrm{lcm}\,[\mathcal{A}(\mathcal{T}(m_k)) \;|\; m_k \leftarrow \bar{m}']\end{aligned}$$

The actual assignment of offsets to the individual members of the list $\bar{m}$ is then performed by a separate function $\mathcal{L}'$. In particular, if the supplied member list $\bar{m}$ belongs to a union type, then $\mathcal{L}'$ assigns the same byte and bit offset of 0 to every member $m_k \in \bar{m}$, as if by the $\mathcal{Z}(m_k)$ function described earlier in Section 5.8.4, except that any bit field members of a zero or negative width are always purged from the result. However, the behaviour of $\mathcal{L}'$ is more complicated if the supplied list is intended for incorporation into a structure object. In that case, $\mathcal{L}'(\texttt{struct} \triangleright \bar{m})$ iterates over all members $m_k \in \bar{m}$, assigning to each m_k a successive byte and bit offset within the structure:

$$\mathcal{L}'[\![\cdot]\!] :: (\mathit{struct\text{-}or\text{-}union} \triangleright \mathit{members}) \rightarrow \mathit{members}$$

$$\begin{aligned}\mathcal{L}'[\![\texttt{union} \;\triangleright \bar{m}]\!] &= [\mathcal{Z}(m_k) \;|\; m_k \leftarrow \bar{m}, \neg\mathrm{BF}(\mathcal{T}(m_k)) \vee \mathcal{W}(\mathcal{T}(m_k)) > 0]\\ \mathcal{L}'[\![\texttt{struct} \triangleright \bar{m}]\!] &= \mathcal{L}''(0,\, 0,\, \bot \triangleright \bar{m})\end{aligned}$$

In particular, the construction $\mathcal{L}''(\sigma,\ \omega,\ \varsigma \triangleright \bar{m})$ performs the actual allocation of member offsets using three integer parameters σ, ω and ς, all of which are assigned the initial values of 0 at the beginning of the structure. Intuitively, σ specifies the total size of all members process thus far, ω determines the number of bits left vacant in the bit field container preceding the current member and ς represents the offset of that partially populated container whenever $\omega > 0$. Using these three parameters, the precise layout of a given list $\bar{m}$ can be obtained by the following recursive algorithm:

① If $\bar{m}$ is empty, then the entire structure has been allocated and the algorithm terminates with an empty member list $\varnothing$.

② Otherwise, if σ does not represent an integral multiple of the alignment appropriate for the type t of the first member in $\bar{m}$, then the supplied member list is reprocessed with a new value of that parameter equal to $\lceil \sigma/\mathcal{A}(t) \rceil \times \mathcal{A}(t)$, also setting ω to 0 and leaving ς undefined.

③ Once the correct alignment of σ has been attained, we can use its value directly as the offset for the first member of $\bar{m}$, provided that that member has a well-formed object type t. Any remaining members in $\bar{m}$ are then processed, after incrementing σ by the size $\mathcal{S}(t)$ of the newly-allocated member and resetting ω to 0.

④ On the other hand, if the first member has a bit field type of a positive width $\mathcal{W}(t)$ no greater than the current ω value, then, the member's type is adjusted with a bit offset equal to $\omega - \mathcal{W}(t)$, also placing the entire bit field member at the offset ς. The remainder of the list is then processed with the same values of the σ and ς parameters, but with ω decreased by $\mathcal{W}(t)$.

⑤ If, however, the bit field is too wide for the number of bits ω available in the previously allocated container member, then the current container, if any, is abandoned and a new one is introduced into the structure. In particular, we set ς to the original value of σ, increment σ itself by the size of the bit field container format Ψ and, finally, set ω to the width $\mathcal{W}(\Psi)$ of the newly allocated container member. The bit field member is then reprocessed under these new parameter values.

⑥ Finally, we always reset ω to 0 whenever a bit field container with a zero or negative width is encountered within the structure. The member itself is always dropped from the resulting list.

Formally:

$$\mathcal{L}''[\![\cdot]\!] :: (\mathit{integer},\ \mathit{integer},\ \mathit{integer} \triangleright \mathit{members}) \to \mathit{members}$$

$$\begin{array}{lll}
\mathcal{L}''[\![\sigma,\ \omega,\ \varsigma \triangleright \bar{m}]\!] & \mid \bar{m} = \varnothing & = \varnothing \\
& \mid \sigma \bmod \mathcal{A}(t) \neq 0 & = \mathcal{L}''(\lceil \sigma/\mathcal{A}(t) \rceil \times \mathcal{A}(t),\ 0,\ \bot \triangleright \bar{m}) \\
& \mid \mathrm{OBJ}(t) & = [\![t\ x\ @\ \sigma]\!] +\!\!+ \mathcal{L}''(\sigma + \mathcal{S}(t),\ 0,\ \bot \triangleright \mathrm{tail}(\bar{m})) \\
& \mid \mathrm{BF}(t) \wedge 0 < \mathcal{W}(t) \leq \omega & = [\![t'\ x\ @\ \varsigma]\!] +\!\!+ \mathcal{L}''(\sigma,\ \omega - \mathcal{W}(t'),\ \varsigma \triangleright \mathrm{tail}(\bar{m})) \\
& \mid \mathrm{BF}(t) \wedge \mathcal{W}(t) > \omega & = \mathcal{L}''(\sigma + \mathcal{S}[\![\Psi]\!],\ \mathcal{W}(\Psi),\ \sigma \triangleright \bar{m}) \\
& \mid \mathrm{BF}(t) \wedge \mathcal{W}(t) \leq 0 & = \mathcal{L}''(\sigma,\ 0,\ \bot \triangleright \mathrm{tail}(\bar{m}))
\end{array}$$

$$\begin{array}{lll}
\text{where } [\![t\ x\ @\ n]\!] & = \mathrm{head}(\bar{m}) \\
t' & = \mathrm{tq}(t) +\!\!+ [\![[\![\mathcal{B}(t)]\!] : [\![\mathcal{W}(t)]\!] \,.\, [\![\omega - \mathcal{W}(t)]\!]]\!]
\end{array}$$

In Section 5.7.2, a similar construction $\mathcal{L}(V)$ was also employed during preparation of the argument values for calls to C functions with a variable parameter list. In particular, given a set V of all temporary local variables allocated for storage of these arguments' values, $\mathcal{L}(V)$ produces a mapping of each variable $v_k \in \mathrm{dom}(V)$ to successive offsets within a conglomerated "**`va_list`**" object, with every argument allocated at an address aligned to the MMIX word boundary. Formally:

$$\mathcal{L}[\![\cdot]\!] :: V \to (v \mapsto \mathit{integer})$$

$$\begin{aligned}\mathcal{L}[\![V]\!] \mid\ & V = \varnothing = \varnothing \\ \mid\ & W' = \varnothing = \{v{:}0\} \\ \mid\ & \text{otherwise} = W' \cup \{v{:}\max[\lceil (n_k + \mathcal{S}(V(v_k)))/8 \rceil \times 8 \mid v_k{:}n_k \leftarrow W']\}\end{aligned}$$

$$\begin{aligned}\text{where } W' &= \mathcal{L}(V \backslash \{v\}) \\ v &= \max(\mathrm{dom}(V))\end{aligned}$$

A precise manipulation of bit field l-values also introduces a number of further implementation-defined aspects into the language. In Section 5.7.1.2, a pair of constructions $\mathcal{U}(t \triangleright \alpha)$ and $\mathcal{P}(t, \alpha \triangleright \alpha')$ were used to, respectively, extract the value of a given bit field t from its container atom α and to populate that container with a new value α' for that bit field. On MMIX, the first of these constructions can be represented as follows:

$$\mathcal{U}[\![\cdot]\!] :: (\mathit{type} \triangleright \mathit{atom}_v) \to \mathit{atom}_v$$

$$\mathcal{U}[\![t \triangleright \alpha]\!] = [\![(\alpha \ll_{\mathrm{N.64}} \#[\![64 - \mathcal{W}(t) - O(t)]\!]_{\mathrm{N.64}}) \gg_{[\![\gamma(\phi(t))]\!].64} \#[\![64 - \mathcal{W}(t)]\!]_{\mathrm{N.64}}]\!]$$

Intuitively, this expression discards the most significant $64 - \mathcal{W}(t) - O(t)$ bits of α using an appropriate "$\ll_{\mathrm{N.64}}$" operation, and shifts the resulting bit field value right by $64 - \mathcal{W}(t)$ bit positions using "$\gg_{\mathrm{N.64}}$" or "$\gg_{\mathrm{Z.64}}$", as appropriate for the genre of t, essentially sign-extending its value to 64 bits and placing the result at the bit offset of 0.

The dual operation $\mathcal{P}(t, \alpha \triangleright \alpha')$ begins by applying the supplied bit container value α to the bit mask $2^{64} - 2^{\mathcal{W}(t) + O(t)} + 2^{O(t)} - 1$, using an appropriate bitwise AND operation, effectively discarding the existing content of the specified bit field in α, then shifts α' into place, while also truncating any of its unnecessary most significant bits. It delivers a bitwise OR of the resulting two atomic values. Formally:

$$\mathcal{P}[\![\cdot]\!] :: (\mathit{type}, \mathit{atom}_v \triangleright \mathit{atom}_v) \to \mathit{atom}_v$$

$$\begin{aligned}\mathcal{P}[\![t, \alpha \triangleright \alpha']\!] = [\![&(\alpha \bigtriangleup_{\mathrm{N.64}} \#[\![2^{64} - 2^{\mathcal{W}(t) + O(t)} + 2^{O(t)} - 1]\!]_{\mathrm{N.64}}) \bigtriangledown_{\Psi} \\ &((\alpha' \ll_{\mathrm{N.64}} \#[\![64 - \mathcal{W}(t)]\!]_{\mathrm{N.64}}) \gg_{\mathrm{N.64}} \#[\![64 - \mathcal{W}(t) - O(t)]\!]_{\mathrm{N.64}})]\!]\end{aligned}$$

In Section 5.8.12, one more bit field assembly construction is required to combine the initial values of multiple bit fields that appear within a single memory-resident object, each pre-packed into the bit field container "$\#0_{\Psi}$". Intuitively, such feat can be achieved by performing an inclusive OR of all atomic values involved in the operation. Formally, this simple construction is implemented on MMIX as follows:

$$\mathcal{P}[\![\cdot]\!] :: \mathit{atoms}_v \to \mathit{atom}_v$$

$$\begin{aligned}\mathcal{P}[\![\bar{\alpha}]\!] \mid\ & (\bar{\alpha} = \varnothing) = [\![\#0_{\mathrm{N.64}}]\!] \\ \mid\ & (\bar{\alpha} \neq \varnothing) = [\![[\![\mathrm{head}(\bar{\alpha})]\!] \bigtriangledown_{\mathrm{N.64}} [\![\mathcal{P}(\mathrm{tail}(\bar{\alpha}))]\!]]\!]\end{aligned}$$

Finally, the two memory copy operations "set" and "set_I" that, in Sections 5.7 and 5.8 were used to depict assignments and initialisations of structure and union objects, can be represented naïvely by a sequence of one or more character copy operations, derived from the size of the underlying object type t as follows:

$$\text{set}\ \ [\![\cdot]\!]\ \text{to}\ [\![\cdot]\!],\ \text{set}_\text{I}\ [\![\cdot]\!]\ \text{to}\ [\![\cdot]\!] :: (\mathit{type} \triangleright \mathit{atom}_\nu) \to (\mathit{type} \triangleright \mathit{atom}_\nu) \to \mathit{term}_\nu$$

$$\begin{aligned}\text{set}\ \ [\![t_1 \triangleright \alpha_1]\!]\ \text{to}\ [\![t_2 \triangleright \alpha_2]\!] &= \text{copy}\ [\![\alpha_1,\ \alpha_2,\ \mathcal{S}(t_2)]\!]\\ \text{set}_\text{I}\ [\![t_1 \triangleright \alpha_1]\!]\ \text{to}\ [\![t_2 \triangleright \alpha_2]\!] &= \text{copy}\ [\![\alpha_1,\ \alpha_2,\ \mathcal{S}(t_2)]\!]\end{aligned}$$

In both cases, an actual Etude term that performs the required movement of data is constructed from the size of the source object t_2 by the following recursive algorithm:

$$\text{copy}\ [\![\cdot]\!] :: (\mathit{atom}_\nu,\ \mathit{atom}_\nu,\ \mathit{integer}) \to \mathit{term}_\nu$$

$$\begin{aligned}\text{copy}\ [\![\alpha_1,\ \alpha_2,\ n]\!]\ \mid\ n > 0 &= [\![\texttt{LET ()} = [\![\text{copy}(\alpha_1,\ \alpha_2,\ n-1)]\!];\\ &\qquad \texttt{LET T}_1 = \texttt{GET}\ [\alpha_2 +_{\text{O.64}} \#n_{\text{Z.64}}]_{\text{N.8}};\\ &\qquad \texttt{SET}\ [\alpha_1 +_{\text{O.64}} \#n_{\text{Z.64}}]_{\text{N.8}}\ \texttt{TO T}_1]\!]\\ \mid\ n \leq 0 &= [\![\texttt{RET ()}]\!]\end{aligned}$$

Observe that, since our target architecture does not implement any memory access attributes, there is no difference between the "set" and "set_I" variants of this operation. Further, it should be pointed out that the above definition of "copy" could be easily enhanced to take advantage of the precise address alignment information included in every Etude object format. For example, objects aligned on a word boundary could be copied using the "$\texttt{GET}\ [...]_{\text{N.64}}$" and "$\texttt{SET}\ [...]_{\text{N.64}}$" term forms instead of the above "$\texttt{GET}\ [...]_{\text{N.8}}$" and "$\texttt{SET}\ [...]_{\text{N.8}}$" variants, therefore reducing the length of the resulting term sequence by 87.5%. However, in the interest of exposition, I leave all such improvements open for a future work on optimising program transformations with the Etude framework.

Last but not least, in Section 5.7.3 one additional notation $\mathcal{V}_\phi(\alpha)$ was left open to interpretation by individual C compilers. In Chapter 5, this construction was used to obtain an actual numeric value corresponding to some closed Etude atom α that was derived from a constant C expression of an arithmetic type associated with the Etude format ϕ.

Although the C Standard admits a broad spectrum of different implementations for $\mathcal{V}_\phi$, including some that disagree with certain aspects of the true meanings of normal Etude atoms, like most modern C compilers, ours will always ensure that every arithmetic C constant has precisely the same meaning as a syntactically identical expression evaluated during the actual program execution. Accordingly, in this section, the function $\mathcal{V}_\phi$ is defined directly in terms of the actual operational semantics of Etude atoms $\mathcal{E}$ that were described in Section 6.3.1. It differs from $\mathcal{E}$ only insofar as its support for floating point arithmetic is restricted to operations over finite numeric quantities. Further, the reader should observe that, unlike $\mathcal{E}$, $\mathcal{V}_\phi$ is formulated as a monadic construction rather than a partial Haskell function, in order to facilitate proper diagnostics of malformed C programs. In particular, a typical implementation

of $\mathcal{V}_\phi$ can be derived from the evaluation function $\mathcal{E}$ of the underlying instruction set architecture as follows:

$\mathcal{V}_{[\![\cdot]\!]}[\![\cdot]\!]$ :: (monad-fix M) $\Rightarrow$ *format* $\rightarrow$ $atom_v$ $\rightarrow$ M(*rational*)

$\mathcal{V}_{[\![\phi']\!]}[\![\#x_\phi]\!]$ = do
  require $((\gamma(\phi\,) \in \{\mathsf{N}, \mathsf{Z}\} \wedge 1 \leq \varepsilon(\phi\,) \leq 64) \vee \phi\, = [\![\mathsf{R}.64]\!]) \wedge$
    $((\gamma(\phi') \in \{\mathsf{N}, \mathsf{Z}\} \wedge 1 \leq \varepsilon(\phi') \leq 64) \vee \phi' = [\![\mathsf{R}.64]\!]) \wedge$
    $(\phi' \neq [\![\mathsf{R}.64]\!] \vee \mathrm{FIN}(\mathrm{enc}_\phi(x)))$
  return $(\mathrm{dec}_{\phi'}(\mathrm{enc}_\phi(x)))$

$\mathcal{V}_{[\![\phi'']\!]}[\![{\phi'}_\phi(\alpha)]\!]$ = do
  require $((\gamma(\phi') \in \{\mathsf{N}, \mathsf{Z}\} \wedge 1 \leq \varepsilon(\phi') \leq 64) \vee \phi' = [\![\mathsf{R}.64]\!])$
  $x \leftarrow \mathcal{V}_\phi\ (\alpha)$
  $z \leftarrow \mathcal{V}_{\phi''}(\mathcal{E}[\![{\phi'}_\phi(\#x_\phi)]\!])$
  return (z)

$\mathcal{V}_{[\![\phi']\!]}[\![op_\phi(\alpha)]\!]$ = do
  require $(\phi \neq [\![\mathsf{R}.64]\!] \vee op \neq [\![\sim]\!])$
  $x \leftarrow \mathcal{V}_\phi\ (\alpha)$
  $z \leftarrow \mathcal{V}_{\phi'}\ (\mathcal{E}[\![op_\phi(\#x_\phi)]\!])$
  return (z)

$\mathcal{V}_{[\![\phi']\!]}[\![\alpha_1\ op_\phi\ \alpha_2]\!]$ = do
  require $(\phi \neq [\![\mathsf{R}.64]\!] \vee op \notin \{\cdot|\cdot, \triangle, \nabla, \bar{\nabla}, \ll, \gg\})$
  $x \leftarrow \mathcal{V}_\phi\ (\alpha_1)$
  $y \leftarrow \mathcal{V}_\phi\ (\alpha_2)$
  $z \leftarrow \mathcal{V}_{\phi'}\ (\mathcal{E}[\![\#x_\phi\ op_\phi\ \#y_\phi]\!])$
  return (z)

$\mathcal{V}_{[\![\phi']\!]}[\![v]\!]$ = reject

This final definition completes the entire specification of the C programming language under the MMIX architecture. While, in theory, no portable C program should ever rely on any of the details discussed in this section, most practical software projects tend to require varying degrees of knowledge about the precise behaviour of the underlying system, which, in most cases, goes beyond the somewhat relaxed guarantees that are provided by the C Standard. Fortunately, as demonstrated by the above set of Haskell definitions, the linear correctness approach to compiler design can readily facilitate such programs, by furnishing them with multiple layers of semantic specifications, each depicting the behaviour of actual computational hardware with an ever increasing amount of detail.

DEFINITION INDEX

D.1 Theorems

D.2 Data Types

D.3 Data Constructors

D.4 Term Forms