# A Formal Semantics for the C Programming Language

Nikolaos S. Papaspyrou

Doctoral Dissertation

February 1998

*National Technical University of Athens*
*Department of Electrical and Computer Engineering*
*Division of Computer Science*
*Software Engineering Laboratory*

National Technical University of Athens
Department of Electrical and Computer Engineering
Division of Computer Science
Software Engineering Laboratory



Nikolaos S. Papaspyrou, *A Formal Semantics for the C Programming Language*, Doctoral Dissertation, National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory, February 1998.

URL: `http://www.softlab.ntua.gr/~nickie/Thesis/`
Pages: xvi + 253

Also available as: Technical Report CSD-SW-TR-1-98, National Technical University of Athens, Department of Electrical and Computer Engineering, Software Engineering Laboratory, February 1998.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

# Abstract

C together with its descendants represents a strong and indisputable status quo in the current software industry. It is a very popular general-purpose programming language, characterized by its economy of expression, its large set of operators and data types, and its concern for source code portability. The current reference document for C is the international standard ISO/IEC 9899:1990. The semantics of C is informally defined in the standard, using natural language. This causes a number of ambiguities and problems of interpretation about the intended semantics of the language.

In this thesis, a formal denotational semantics for the ANSI C programming language is proposed, with emphasis on its accuracy and completeness with respect to the standard. It is demonstrated that a programming language as useful in practice and as inherently complicated as C can nonetheless be defined formally. The proposed semantics could be used as a precise, unambiguous, abstract and implementation-independent standard for the language. Moreover, it would be a basis for the formal reasoning about C programs and a valuable theoretical tool in the software development process.

In order to improve in modularity and elegance, the proposed semantics uses several monads and monad transformers to model different aspects of computations. Interesting results have been achieved in the attempt to accurately model complex characteristics of C, such as the unspecified order of evaluation and sequence points, using monad notation. These results may be useful in specifying the semantics of programming languages supporting non-deterministic features and parallelism.

An implementation of an abstract interpreter for C programs based on the proposed semantics has also been developed, using Haskell as the implementation language. The implementation has been used to evaluate the accuracy and completeness of the proposed semantics. Although this process is still under way, the results so far have been entirely satisfactory.

## Keywords

ANSI C programming language, ISO/IEC 9899:1990 standard, formal definition, denotational semantics, monads, monad transformers.

# Acknowledgements

I want to thank my supervisor, Manolis Skordalakis, for all the guidance he has constantly given me and for his confidence. In the ten years that I know him, he was always been accessible whenever I needed him, helpful and friendly. My sincere thanks are also due to the members of my defense committee, Giorgos Papakonstantinou, Andreas Stafylopatis, Stathis Zachos, Panagiotis Tsanakas, Timos Sellis and Panos Rondogiannis, for their prompt and effective help, their valuable advice and the inspiring conversations that we have had.

My way of thinking and the direction of my research has been greatly influenced by my studies at Cornell University. I would like to thank my advisor there, David Gries, and the other members of the CS department faculty for teaching me to combine theory and practice, directing me towards the formal study of programming languages and providing me access to plentiful literature. It took some time until I fully appreciated this lesson and I feel that I never had the chance to thank them.

The members of the Software Engineering Laboratory have always been good friends. In particular, I wish to thank Vassilis Vescoukis, Simos Retalis, Tassos Koutoumanos, Clio Sgouropoulou and Kostas Tavernarakis for their outstanding support, the long hours of conversation and the good times that we have shared. Also, my sincere thanks go to the laboratory's numerous administrators for the technical support that they have never refused in all these years.

Many thanks are due to Dragan Maćoš for introducing me to monads, making useful suggestions about this thesis and being a friend. Moreover, I want to thank Tasos Viglas and Vassilis Papadimos for their help during the early debugging of the developed semantics, Alkis Polyzotis, Giannis Sismanis, Manos Renieris and Katerina Potika for their suggestions regarding the implementation of the developed semantics.

I would also like to thank my friends who have always offered their love, encouragement and support. They made my studies more enjoyable. Last but not least, my wholehearted thanks go to my family and my companion Katerina for their endless patience, love, and faith, and for giving me the option to abstain from some activities that were incompatible with my studies.

Nikolaos S. Papaspyrou,

Athens, February 27, 1998.

---

This thesis was typeset with the LaTeX $2_\varepsilon$ document preparation system, using the `thesis` class written by Wenzel Matiaske. Paul Taylor's `diagrams` package for typesetting commutative diagrams and `QED` package for typesetting mathematical proofs were extremely useful, as well as Peter Møller Neergaard's `semantic` package for typesetting inference rules.

# Contents

# List of tables

# List of figures

# Part I

# Prelude

# Chapter 1

# Introduction

CHAPTER OVERVIEW

This chapter is an introduction to the present thesis. In Section 1.1 a brief presentation of the C programming language is attempted, with emphasis on its history and characteristics. Section 1.2 is a non-technical introduction to the semantics of programming languages, biased in favour of the denotational approach. Section 1.3 presents an overview of this thesis. The objectives, motivation and possible applications are first discussed, followed by an overview of the methodology that is used and a summary of the thesis' contribution. Section 1.4 contains an overview of the related work and the chapter concludes by presenting the structure of this thesis, in Section 1.5.

## 1.1 The C programming language

C is a well known and very popular general purpose programming language. It was developed in the years 1969-1973 at the AT&T Bell Labs as a system implementation language for the Unix operating system. The father of C is Dennis Ritchie who also developed the first compiler in 1971-1973. A detailed account of the development of the C language written by Dennis Ritchie himself can be found in [Ritc93].

ORIGINS OF C

The direct ancestor of C is a language called B [John73], designed by Ken Thompson in 1969-1970 as an implementation language for the DEC PDP-7 computer. B can be viewed as a limitation of BCPL [Rich79], a language designed by Martin Richards in the mid-1960s mainly as a compiler-writing tool. The main difference between C and its ancestors is the presence of a non-trivial type system. BCPL and B are typeless, featuring just one "word" type which represents both data and "pointers" to data. The introduction of other types in C was necessary in order to provide language support for characters and floating point numbers, that were supported by emerging hardware in the early 1970s. In 1973, the core of C as we now know it was complete and a compiler for DEC PDP-11 had been developed by Dennis Ritchie. The language featured integers and characters as base types, full arrays and pointers, special boolean operators and a powerful preprocessor.

In the years to follow, portability and type safety issues introduced a number of changes in the C language. Several new types were added to the type system and type casts were introduced. The first widely available description of the language, the book "The C Programming Language" also known as "K&R", appeared in 1978 [Kern78].[1] By that time, the kernel of the Unix operating system had been rewritten in C and both language and compilers gained significantly in confidence. During the 1980s, the use of the C language spread widely and compilers became available on nearly every machine architecture and operating system.

[1] A second edition incorporating later changes was published ten years later [Kern88]. The "K&R" book served as the language reference until a formal standard was adopted in 1989.

CHARACTERISTICS

C, as well as both its ancestors B and BCPL, belongs to the family of languages expressing the traditional procedural programming paradigm, typified by Fortran and Algol 60. It is particularly oriented towards system programming and its small, concise description allows the development of simple compilers. C is mainly characterized by its economy of expression, realized by a laconic syntax as well as a large set of operators and data types, and also by the fact that it provides access to the inner parts of the computer.

C can be characterized as a medium-level language. On the one hand, it is close to the machine. The abstractions that it introduces are founded in the concrete data types and operations that are supported by most conventional computers and, for this reason, programs in C are usually very effective. On the other hand, these abstractions are at a sufficiently high level to facilitate programming and lay the grounds for program portability between different machines. Portability is further enhanced by the fact that C programs rely on library routines for I/O and interaction with the operating system.

According to Dennis Ritchie, the ideas that mostly characterize C and differentiate it from its ancestors and other contemporary languages are two: the relationship between arrays and pointers and the syntax of declarations, mimicking the syntax of expressions. However, these are also among its most frequently criticized features and sources of misinterpretations, not only for the beginner but even for experienced C programmers.

STANDARDIZATION

By 1982 the changes that were introduced to the C language, as a result of adapting to the common practice, were numerous. Each compiler implemented a slightly different version of C. The first edition of "K&R" no longer described C in its actual use and, even when it did, it was not precise on a number of details. In an attempt to standardize the language, the American National Standards Institute (ANSI) established the X3J11 committee in the summer of 1983. Its goal was "to develop a clear, concise and unambiguous standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments". The committee was cautious and conservative with respect to language extensions. The main change that it introduced was the use of function prototypes, which was a significant step in the direction of a stronger type system for C. However, the committee decided to leave the old style as a compromise to the huge volume of existing software in C.

This process was complete in late 1989 and resulted in the standard document ANS X3.159-1989 [ANSI89a], which was later adopted by the International Organization for Standardization as standard ISO/IEC 9899:1990 [ANSI90]. The standard is complemented by a series of other documents, acting as clarifications [ANSI89b] or corrections [ANSI94]. The flavour of the C language that is specified in the standard is called ISO C, or usually ANSI C. Since 1990, a review process for the standard is under way. As a result of this process, a completely revised standard, nicknamed "C9X", is expected by the year 1999.

CURRENT SITUATION

Since its early years of development, C has been used to program a wide area of applications, including the biggest part of the Unix operating system. Compilers for C are currently available for almost any computer system and, although the language still allows the development of non portable applications, programs in C are generally portable, usually with small modifications. During the last twenty years, C has been used as the basis for, or at least strongly influenced, the development of a number of programming languages. Among these one should mention Concurrent C [Geha89], Objective C [Cox91] and especially C++ [Stro91, Elli90] and Java [Gosl96]. In the current software industry it could be argued that C and its descendants represent a strong and indisputable status quo. The standard for C is nowadays accepted as a common basis for the language and is taken as a point of reference by the developers and the users of implementations and other tools.

## 1.2 Programming language semantics

SYNTAX AND SEMANTICS

The study of programming languages invariably distinguishes between two fundamental features: *syntax* and *semantics.* Syntax refers to the appearance and structure of the well-formed sentences of the language, including programs themselves. Semantics refers to the meanings of these sentences, which must be preserved by compilers or other language implementations. The line that separates syntax from semantics is not always clearly marked. The etymology of the word "semantics" leads to the ancient Greek language and the verb "$\sigma\eta\mu\alpha\acute{\iota}\nu\omega$" (to mean). The original meaning of the word is the study of attachment between words and sentences of a language and their meanings.

The syntax of programming languages is usually formally specified. The area of formal syntax specification has been thoroughly studied and there are currently various standard formalisms for this purpose. The most widely used is context-free grammars, usually expressed in the Backus Naur Form (BNF) and its variations. Grammars allow, or even suggest, a direct connection between syntax and parser implementation and this strong connection is probably the major reason why formal syntax specification has been developed so much.

On the other hand, programming language semantics is most commonly specified in an informal way. This is mainly due to the complexity of the task, which becomes even worse considering that the simplicity of BNF representation can be partly attributed to the fact that the most intricate parts of syntax specification are "moved to the semantic level". As opposed to the case of syntax, there is a lack of standard, widely accepted and widely used methodologies for describing the semantics.

INFORMAL SEMANTICS

Every person who uses a programming language to develop programs must understand its semantics at some level of abstraction. Programmers usually understand the semantics by means of examples, intuition and descriptions in natural language. Such semantics descriptions are informal and are typically based on a set of assumptions about the reader's knowledge and understanding. Informal semantic descriptions are inherently ambiguous, as is always the case with natural languages. In the best case, a programmer's intuition fills the missing points in the description and leads to the correct understanding of a language's semantics. In the worst case, the description is fatally ambiguous or even misleading and the programmer is prone to misinterpretations, which often lead to programming errors.

FORMAL SEMANTICS

Research in the area of formal specification of programming language semantics started in the 1960s. With the rapid increase in the complexity of high-level programming languages, formal semantics solidly based on mathematical logic systems and precise rules of inference were sought as a possible way of overcoming ambiguities and enforcing discipline in this field. Since then, the product of more than three decades of research has been the development and thorough study of numerous methods and formalisms. An excellent introduction to the field is [Wins93], whereas [Mitc96] provides a more thorough presentation of methods and mathematical foundations.

For historical reasons, formal semantics are usually classified as following one of three main approaches:

- *Operational semantics*: Meanings are sequences of computation steps that result from the program's execution. Structural operational semantics are a more systematic variation. An elementary reference can be found in [Henn90]

- *Denotational semantics*: Meanings are mathematical objects, typically functions from inputs to outputs. This category of semantics explicitly constructs mathematical models of programming languages. It will be further discussed in the sequel, as it is the approach followed in this thesis.

- *Axiomatic semantics*: Meanings are expressed indirectly in terms of logical propositions stating properties of the programs. Such an approach is useful because it directly aims to support program verification. The seminal paper on axiomatic semantics is [Hoar69], whereas a classic text is [dBak80]. Following this approach, it has been suggested that proofs of correctness be developed at the same time with programs [Dijk76, Grie81].

It should be noted, however, that these three approaches must not be viewed in opposition to each other. They are in fact complementary and highly dependent on each other. Each has its uses and serves best a particular category of applications. Operational and denotational semantics can be used to specify an interpreter for the language under study, and thus help in defining or refining a language. Axiomatic semantics are helpful in developing proofs about program properties. Probably the most significant application of formal semantics is in rapid prototyping, using tools that translate language specifications to correct compilers or interpreters.

Formalisms sharing properties from more than one of these approaches also exist. Abstract state machines (formerly known as evolving algebras) are one such formalism, started by Gurevich as an attempt to bridge the gap between formal models of computation and practical specification methods [Gure93a, Gure95]. Action Semantics, developed by Mosses and Watt, is a practical framework for the formal description of programming languages combining features of all three traditional approaches [Moss92].

DENOTATIONAL SEMANTICS

Denotational semantics, or the mathematical approach to programming language semantics, is a formalism introduced by Scott and Strachey in the late 1960s. Since then, it has been widely studied by distinguished researchers and has been used as a method for the semantic analysis, description, evaluation as well as the implementation of various programming languages. The seminal paper on denotational semantics is [Scot71]. Other introductory papers including useful bibliography are [Tenn76] and [Moss90]. Introductory books presenting in more depth the underlying theory and the techniques that have been developed include [Miln76], [Stoy77], [Gord79], [Alli86] and [Schm86]. An graduate-level book with more mathematical depth is [Gunt92]. This thesis was strongly influenced by [Tenn91], an excellent book revealing the connections between syntax and the various flavours of semantics. The same is accomplished by [Mitc96], focusing mainly on the mathematical foundations of the various semantic approaches. An overview and survey of the research field is given in [Fior96].

According to the denotational approach, programming language semantics is described by attributing mathematical *denotations* to programs and program segments. Denotations are typically high-order functions over appropriate mathematical entities, such as *domains* whose theory is briefly presented in Chapter 3. One of the main properties of denotational semantics is *compositionality*, that is, the fact that the meaning of a sentence can be obtained by appropriately composing the meanings of its subparts, as these are determined by syntactic structure. Although researchers have not agreed on a standard meta-language for expressing denotations and there is considerable variation in the notational conventions used by various authors, it seems that variations of the $\lambda$-calculus over domains are very popular. This is the approach taken in this thesis.

MONADS

One of the most important drawbacks of classic denotational semantics is lack of modularity. Small changes or extensions in a language definition often imply a complete rewrite of its formal semantics. As a consequence, although denotational semantics is an appropriate and elegant formalism for moderately sized languages, it does not scale up easily to real programming languages.[2] Further-

[2] Throughout this thesis, the term "real programming languages" stands for high-level programming languages that are widely used in industry for software development, in contrast to languages that are designed and used in academic laboratories for experimental purposes. Of course, this does not mean that these languages are more real, or in fact any better, than the others. Quite the contrary is true in many cases.

more, it is not easy to reuse a part of a denotational description of one programming language into another. One would like to consider various features of programming languages in isolation, so that their study is easier. However, it should be possible later to put all the pieces together and form a complex denotational description for the whole programming language. This final composition is the point where classic denotational semantics fails.

The use of *category theory* and *monads* has been proposed as a remedy and has become quite popular in the denotational semantics community. The intuition behind the use of monads in semantics, suggested by Moggi, is that computations resulting in values from a domain $V$ can be represented as elements of a domain $\mathsf{M}(V)$, where $\mathsf{M}$ is an appropriate monad [Mogg89]. It is also suggested that programming language features can be studied independently in terms of relatively simple monads and later glued together to form a complete semantic description for the language. Furthermore, monads have recently been used as an elegant way of introducing imperative features in functional programming and many functional languages support them directly.

Monad notation is used in this thesis and it is demonstrated that, as a result, the semantics are significantly improved in terms of modularity and elegance. A brief introduction to monads, their applications with respect to denotational semantics, the exact notation used in this thesis and pointers to useful bibliography on this field are given in Chapter 3. Comprehensive introductions to monads and their use in denotational semantics can be found in [Mogg90] and [Wadl92].

## 1.3 Overview of this thesis

OBJECTIVES

The main objective of this thesis is to develop and evaluate a formal description for the semantics of the C programming language. The developed semantics should satisfy the following requirements:

- *Accuracy*: the formal description should be as close as possible to the informal semantics of ANSI C, as this is defined in the standard. Most of the formal descriptions of real programming language semantics that have been suggested in literature are inaccurate, to some extend, either because of intended simplifications or by mistake. Taking into consideration the complexity of these languages, one should admit that an inaccurate semantics is useful, as long as the inaccuracies are clearly documented.

- *Completeness*: the language described should be as large a subset of ANSI C as possible. In describing the formal semantics of a real programming language, it is common practice to exclude complicated aspects of the language which cannot be correctly described in a simple and elegant way. It is also common to treat some features of the language as syntactic sugar and define them in terms of other features. The first practice does not produce accurate formal descriptions and should be avoided as much as possible. The second practice is also avoided in the present thesis because, although it does not affect the accuracy of the semantics, it often deprives the semantic description from its direct connection with the syntax of the language.

- *Simplicity*: the formal description should be kept as simple as possible. The rationale behind this requirement is that simple formal systems are easier to develop, understand, keep under control when changes are needed, and (especially) use. This will be better understood after the applications for such a formal description are presented, later in this section. This requirement comes in direct conflict with the previous two.

**Table 1.1:** Research topics addressed in the present thesis, according to ACM CCS 1998.

D. SOFTWARE
- D.2 SOFTWARE ENGINEERING
  - D.2.4 Software/Program Verification
    - Formal methods
- D.3 PROGRAMMING LANGUAGES
  - D.3.0 General
    - Standards
  - ▶ **D.3.1 Formal Definitions and Theory**
    - **Semantics**

F. THEORY OF COMPUTATION
- F.3 LOGICS AND MEANINGS OF PROGRAMS
  - F.3.1 Specifying and Verifying and Reasoning about Programs
  - ▶ **F.3.2 Semantics of Programming Languages**
    - **Denotational semantics**
- F.4 MATHEMATICAL LOGIC AND FORMAL LANGUAGES
  - F.4.1 Mathematical Logic
    - Lambda calculus and related systems
  - F.4.3 Formal Languages

The first two requirements were considered as the most important throughout this doctoral research. That is, the developed semantics should be as complete and accurate as possible, with respect to the standard. Simplicity should be sought, as long as the other two are not affected.

RESEARCH TOPIC

Formal semantics of C and other real programming languages has always been a research topic of great interest both to theoreticians and practitioners, as is documented briefly in Section 1.4 and in more detail in Chapter 16. According to the ACM Computing Classification System,[3] the research topics that are addressed in the present thesis are shown in Table 1.1. In particular, the main research topics of this thesis are written in bold characters and marked with the symbol ▶.

### 1.3.1 Motivation

WHY C?

A reasonable question that can be asked at this point is whether the formal semantics of C presents a worthwhile object of study. The answer to this is affirmative, from two different perspectives. From a practical point of view, C is admittedly a widely spread programming language and a formal specification for its semantics is potentially very important and useful. On the other hand, from a theoretical point of view, C is characterized by a number of interesting features whose combination is worthwhile studying on its own right. As a first example, the presence of side effects in expressions combined with unspecified order of evaluation inevitably leads to non-determinism, which the standard makes a remarkable effort to restrain. As a second example, the control structures supported by C include a

[3] The 1998 version of the ACM CCS can be found at the URL `http://www.acm.org/class/1998/`.

variety of selection, iteration and unrestricted jump statements, and give rise to a number of interesting problems combined with C's block structure and the declaration of variables in compound statements.

MANIFESTATION OF PROBLEMS

Except for the description of the syntax, which has been specified formally using a grammar in BNF notation, the rest of the ANSI C standard, has been written in natural language, including the description of the semantics. This causes a number of ambiguities and problems of interpretation, as far as the semantics of C is concerned.

The newsgroup `comp.std.c` is a forum in which issues related to the ANSI C standard and its interpretation are discussed. In contrast to other newsgroups dealing with the C programming language, articles posted in `comp.std.c` are typically characterized by a highly scientific level. Many distinguished researchers and members of the standardization committee frequently participate in the discussions, as well as language implementors and experienced programmers. The following excerpts come from a thread of articles with the subject "On Pointer Arithmetic", initiated by the author of this thesis in early January 1997. The first article in the thread was simply asking whether two small pieces of C code, none of which was overly complicated or out-of-place with common practice, are conforming with respect to the standard.[4]

> Where in the standard does it say that...
>
> The standard doesn't say or imply anything like...
>
> The standard does say that... but this does not mean that...
>
> The more I look at it, the more it appears to me that...
>
> My interpretation is that...

The first group of excerpts indirectly reveals some points of confusion about what the standard states and its interpretation, as expressed by the verbs "imply" and "mean" that are used. The last two excerpts indicate that the authors recognize ambiguities in the standard and express their personal interpretations at various levels of certainty.

> Hmmm... nice to see someone else make the same "mistake" as I did a while back. The standard really should be better written if it is not only me who misreads it.
>
> The problem, of course, is that the standard is not as clearly worded as either of us would like. I have good reasons for favoring my interpretation...

The second group of excerpts expresses their authors' critical view towards the standard, by recognizing sources of misinterpretation beyond any doubt.

> As far as I can see, your reading is plain distortion of the standard. But then, perfectly reasonable things have been ruled illegal by the committee anyway... But if that is what is decided, nothing will convince me that the standard that is in front of me is written in any variety of English that I am familiar with.
>
> In any case, let us agree to disagree. I will grant you this much: Natural languages are vague enough that every document in a natural language can be misinterpreted by someone sufficiently motivated.

Finally, the third group of excerpts takes a more aggressive critical approach, attributing the standard's ambiguity to the natural language that is used for its description.

APPLICATIONS

With all this in mind, the necessity for a formal description of the semantics of C becomes quite apparent. The same could be argued in the case of other real programming languages which usually lack formal descriptions, the main reason for this being an aversion of the average programmer and the whole software industry towards formal methods in language specification. The applications of such a description are briefly summarized in the following:

- It would serve as a precise standard for compiler implementation, specifying a rigid mathematical model for the language without specifying or restricting the techniques used in implementations.
- It would provide a basis for reasoning about the correctness of programs.
- It would be useful as user documentation; trained programmers can use it as a language reference, in order to answer subtle questions about the language.
- It would be a valuable tool for language analysis, design and evaluation; semantic descriptions can be tuned in order to suggest elegant and efficient implementations.
- It could be used as input to a compiler generator. A relatively outdated survey on semantics-driven compiler construction is given in [Gaud81], whereas experimental systems for this purpose include SIS [Moss76, Bodw82], MESS [Lee87, Pleb88], PSP [Paul82], CERES [Jone80], Actress [Brow92], Cantor [Pals92], DML [Pett92]. A more recent approach, based on the use of monads, is described in [Lian95a, Lian96].

### 1.3.2 Methodology

ABSTRACT INTERPRETER

The present specification for the semantics of C can be best understood as part of an *abstract interpreter* for C programs. Such an interpreter is depicted in Figure 1.1 as a data process. The left part of the figure is a module diagram of the interpreter. It shows the chain of actions performed by the interpreter as well as the data that is processed by these actions. Each action takes as input the result of the previous action and, after processing it, passes it to the following action. Sometimes, previous results are also necessary for the processing. The initial piece of data is a *source program*, written in C, and the final result is a representation of this program's *meaning*, i.e. a description of what the program does when executed. The right part of the figure gives a small example, illustrating the function of the interpreter. This example will be discussed in detail gradually, until the end of this section.[5]

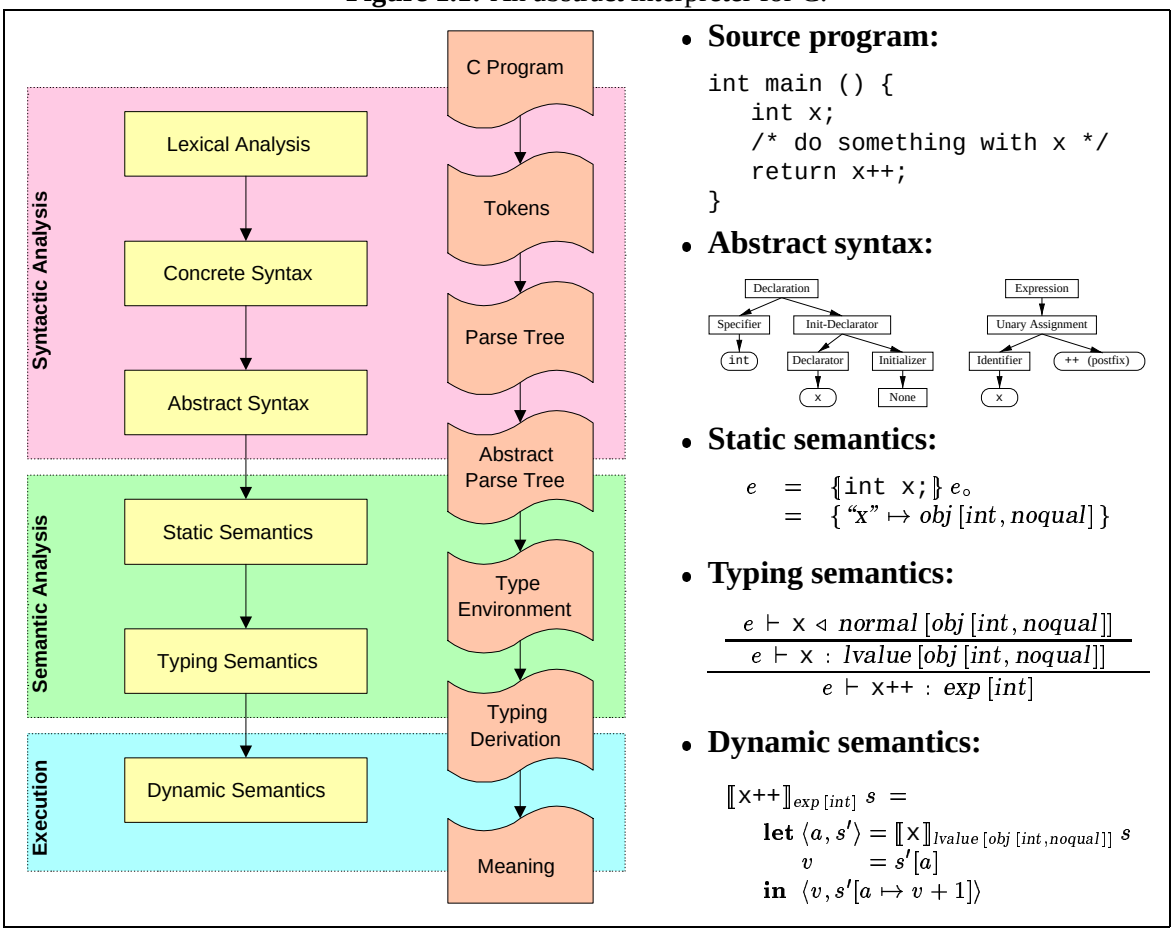
The interpreter consists of the following three layers, represented in Figure 1.1 as big boxes. Each layer contains a series of actions.

- *Syntactic analysis*: aims at checking the syntactic validity of the source program and signalling syntax errors. Syntactically correct programs are transformed to abstract parse trees, which represent their syntactic structure in detail.
- *Semantic analysis*: aims at checking the semantic validity of the program, as represented by the abstract parse tree, and signalling semantic errors, e.g. use of undeclared identifiers or type mismatches.

[4] The discussion did not reach a unanimous verdict, although many opinions were expressed.

[5] Notice that several simplifications have been made in this example, since its purpose is only to illustrate the methodology that is used. Therefore, the semantics attributed to this example is not accurate with respect to the rest of this thesis.

**Figure 1.1:** An abstract interpreter for C.




- *Execution*: aims at describing the meaning of programs, i.e. their execution behaviour. This is the main part of the interpreter.

SYNTACTIC ANALYSIS

Syntactic analysis of C programs presents some difficulties related to the complexity of the language. However, it is not a very interesting subject from a researcher's point of view. For this reason, it is only briefly discussed in the present thesis. A BNF grammar describing the syntax of the language is given in Appendix B of the standard. A number of ambiguities are caused by the definition of *typedefs* and can be resolved by the use of some sort of symbol table. The layer of syntactic analysis consists of three actions:

- *Lexical analysis*: transforms the source program to a sequence of lexical units, also called *tokens*. A formal description of tokens is given in §B.1 of the standard.
- *Concrete syntax*: checks the syntactic validity of the source program by grouping tokens together in more complex syntax entities. The BNF grammar describing the concrete syntax of C that is used for this purpose is taken from §B.2 of the standard. Provided that the source program is syntactically valid, the result of this action is the parse tree according to the given grammar.

- *Abstract syntax*: simplifies the parse tree by extracting superfluous information and results in an abstract syntax tree. The abstract syntax of C is formally described in Section 2.2 by means of a second BNF grammar. Probably the most characteristic example of information that can be abstracted out results from the grouping of expressions by means of parentheses and operator precedence rules. The concrete syntax that correctly describes this grouping is enormously more complicated than the abstract syntax.

STATIC SEMANTICS

Static semantics is the first action in the semantic analysis, which manages the identifiers that are defined in the source program. It aims at detecting static semantic errors, such as the redefinition of an identifier in the same scope, as well as associating identifiers with appropriate types or values. Static semantics is based solely on the abstract syntax. For each syntactically well-formed program phrase $P$, its static semantic meaning is denoted by $\{\!|P|\!\}$. Static semantic meanings are mathematical objects. Typically they are type environments, i.e. associations of identifiers to types, or functions handling type environments.

As an example, consider the simple C program that follows. A part of this program's abstract parse tree is shown in the right part of Figure 1.1.

```
int main () {
   int x;
   /* do something with x */
   return x++;
}
```

Consider now the syntactically well-formed phrase "`int x;`", that is, the declaration in the function's body. The static semantic meaning of such declaration is a function that updates the type environment by declaring an integer variable "`x`". If $e_\circ$ is taken to be the empty type environment, containing no declarations, then the result of $\{\!|$`int x;`$|\!\}$ applied to $e_\circ$ is an updated environment $e$ which contains a declaration for "`x`":

$$\begin{aligned} e &= \{\!|\texttt{int x;}|\!\}\, e_\circ \\ &= \{\, \text{“}\texttt{x}\text{”} \mapsto \mathit{obj}\,[\mathit{int}, \mathit{noqual}]\,\} \end{aligned}$$

TYPING SEMANTICS

The second action in semantic analysis is typing semantics, which manages the types of all program phrases. It aims at detecting type-mismatch errors, such as assignment to a constant value, and at associating syntactically well-formed phrases with appropriate phrase types. Such associations are given by means of typing derivations, that is, formal proofs that phrases are well-typed. Typing semantics is based not only on the abstract syntax of the program, but also on the static semantic environments that result from the previous action.

In the same example program, consider now the program phrase "`x++`" that is part of the function's body. Assume also that the static semantic analysis has finished and resulted in the type environment $e$ that was given above. Under this assumption, it is possible to derive that the phrase "`x++`" is an expression that computes an integer value.[6] A proof of this is given by the following derivation:

$$\dfrac{\dfrac{e \vdash \texttt{x} \triangleleft \mathit{normal}\,[\mathit{obj}\,[\mathit{int}, \mathit{noqual}]]}{e \vdash \texttt{x} : \mathit{lvalue}\,[\mathit{obj}\,[\mathit{int}, \mathit{noqual}]]}}{e \vdash \texttt{x++} : \mathit{exp}\,[\mathit{int}]}$$

[6] One should recall that expressions in C are allowed to generate side effects. This is the reason why the incrementing of "`x`" in "`x++`" does not deserve special mention here.

The notation that is used will be explained in detail in Part III. Two things should be mentioned, however. The first is that the conclusion of this derivation is the judgement "$e \vdash \texttt{x++} : exp\,[int]$" which states exactly that this phrase is an expression computing an integer value, given the type environment $e$. The second is that the derivation makes use of two typing rules, shown as horizontal lines. The first rule (upper line) states that an integer variable declared in the environment is an l-value designating an integer object. The second rule (lower line) states that l-values may be dereferenced by accessing the value that is stored in the designated objects.

DYNAMIC SEMANTICS

Dynamic semantics is the final action in the abstract interpreter, which is not considered a part of semantic analysis.[7] Its primary aim is the definition of a well-typed program's execution behaviour. As a useful side effect, run-time errors and other sources of undefined behaviour are detected at the same time. Dynamic semantics is based on the abstract syntax, the environments that result from static semantics and the typing derivations that result from typing semantics. For each well-typed program phrase $P$ of type $\theta$, its dynamic semantic meaning is denoted by $[\![P]\!]_\theta$. Such meanings are also mathematical objects. Typically they are functions describing aspects of the execution of the corresponding program phrases. The typing derivation for $P$ is important, as the same phrase is allowed to have different meanings when attributed different types.

Going once more back to the same example, consider again the well-typed program phrase "x++", whose type derivation was given above. Assume that we choose direct semantics as the execution model for expressions: the dynamic semantic meaning of an expression is a function that takes as an argument the initial program state $s$ and returns two results: the computed value of the expression and the final program state after the expression's evaluation. Following this model, the dynamic semantics for "x++" is given by the following equation:

$$\begin{array}{lcl} [\![\texttt{x++}]\!]_{exp\,[int]}\ s & = & \textbf{let}\ \langle a, s'\rangle = [\![\texttt{x}]\!]_{lvalue\,[obj\,[int,noqual]]}\ s \\ & & \quad\ v \quad\ = s'[a] \\ & & \textbf{in}\ \ \langle v, s'[a \mapsto v+1]\rangle \end{array}$$

Notice that the typing derivation dictates the types that are used for the dynamic semantics of "x++", on the one side, and of "x" on the other side of the equation. Just for the sake of clarity, $a$ represents the address of the object designated by the l-value "x", $s'$ is the program state after evaluating "x" and $v$ is the value stored in $a$ at the program state $s'$. The result of the evaluation is $v$, and the final program state is the same as $s'$, with the value stored in $a$ incremented by one.

### 1.3.3 Contribution

FORMAL SEMANTICS OF C

The main contribution of this thesis is a formal description for the semantics of the ANSI C programming language, following the denotational approach. The importance and possible applications of such a formal description have been outlined in Section 1.3.1. The developed semantics is satisfactorily complete and accurate, with respect to the standard, and compares very favourably to other approaches for the same purpose, as is further discussed in Chapter 16.

MONADS AND MONAD TRANSFORMERS

Another significant contribution of this thesis is the application of monad notation for the specification of the semantics of a real programming language. Monads [Mogg89, Wadl92, Wadl94] and monad transformers [Lian95b, Lian96] have been proposed as a method of improving the modularity

[7] Following the terminology used in compilers and other language implementations, analysis primarily focuses on the extraction of static program properties and error detection. The dynamic semantics, as defined here, correspond to the phase of code generation.

and elegance of classic denotational semantics. Although the former have been much used lately in functional programming practice, applications of monad notation in the denotational semantics of real programming languages have not been found in literature. The semantics that is presented in this thesis makes use of seven monads and a monad transformer, in order to represent different aspects of computations. Most of these monads have already been investigated in literature and are only important for the simplifications they achieve in the semantics. It should be noted however that standard domain theory is used and the monads are defined over the category of domains and continuous functions.

NON-DETERMINISM AND INTERLEAVING

The powerdomain monad is used in this thesis in order to allow multiple possible results in computations. Several monads have been used in literature for this reason, with the same properties. Although the monadic properties of the convex powerdomain has already been investigated in literature, a complete concise definition is given in this thesis. Furthermore, in the developed semantics execution interleaving is represented by means of the resumption monad transformer, which is defined in this thesis and whose properties are investigated here. The combination of the resumption monad transformer with the continuation and powerdomain monads produces interesting results that can be useful in specifying the semantics of programming languages with non-deterministic features and/or parallelism.

IMPLEMENTATION

A significant effort has been made to evaluate the developed semantics and assess its accuracy and completeness. For this reason, an implementation of the abstract interpreter was developed and tested, using improvised tests and parts of available test suites for C implementations. The interpreter implements the semantics directly and was written in Haskell. Standard ML and C++ were also considered as possible implementation languages. Experimentation with the latter has shown that object-oriented programming languages with generic types can imitate functional characteristics such as the lambda notation and high-order functions in a type-safe way. Therefore, they are adequate for the implementation of denotational semantics, although they are obviously not a natural choice.

## 1.4 Overview of related work

A detailed reference and comparison of this thesis to related work is given in Chapter 16. However, for the sake of completeness of this introduction, a brief summary is included here. An overview of related work in the field of implementing denotational semantics is given in Chapter 15.

SEMANTICS OF REAL LANGUAGES

The semantics of many popular programming languages have been formally specified in literature using various formalisms. However, in most cases it is not the semantics of the whole language that is specified, but that of a significantly large subset, leaving out usually the most tricky features. Few real languages have been given formal semantics as part of their definition. Among them one should mention Scheme [IEEE91, Abel91] and Standard ML [Miln90, Miln91, Kahr93], using denotational and operational semantics respectively. Denotational semantics have also been used for the formalization of Ada [Pede80], Algol 60 [Bjor82a], Pascal [Bjor82b], and Smalltalk-80 [Wolc87], whereas operational semantics have been used for Eiffel [Atta93] and Scheme [Hons95]. In a seminal paper, axiomatic semantics have been used to partially describe the semantics of Pascal [Hoar73]. Among other formalisms, action semantics have been used for Pascal [Moss93] and Standard ML [Watt87], and abstract state machines for Ada [Morr90], Cobol [Vale93], C++ [Wall93, Wall95], Modula-2 [Gure88, Morr88], Oberon [Kutt97b, Kutt97a], Occam [Gure90, Borg94a, Borg96], Prolog [Borg94b] and Smalltalk [Blak92].

MONADS

Monads have been proposed by Moggi as a ground-breaking attempt to structure denotational semantics. [Mogg90] In a short time, the idea of monads was popularized by in the functional programming community by the work of Wadler [Wadl92]. Since then, related research has focused on the combination of monads to structure semantic interpreters. Monad transformers, which were also first proposed by Moggi, have attracted the attention of many researchers. In the work of Liang, Hudak and Jones [Lian95b], monad transformers are demonstrated to successfully modularize semantic interpreters and the lifting of monad operations is investigated.

C SEMANTICS

Significant research has been conducted recently concerning semantic aspects of the C programming language, mainly because of the language's popularity and its wide applications. In what seems to be the earliest formal approach, Sethi addresses the semantics of pre-ANSI C, using the denotational approach [Seth80]. In the work of Gurevich and Huggins [Gure93b] a formal semantics for C is given using the formalism of evolving algebras. A higher-level axiomatic semantics is proposed by Black and Windley [Blac96], focusing on C's expression language with side effects under a number of simplifications. In the work of Cook and Subramanian [Cook94b] a semantics for C is developed in the theorem prover Nqthm. Cook et al. have also developed a denotational semantics for C based on temporal logic [Cook94a]. An operational semantics for C has been sketched, in terms of a random access machine, as a part of the MATHS project in California State University. Finally, in the work of Norrish [Norr97] a complete operational semantics for C is given using small-step reductions. To the best of the author's knowledge, this is the only description of the semantics of C that formalizes correctly some subtle points of the language, like unspecified order of evaluation and sequence points. The author knows of no similar denotational approach.

## 1.5 Structure of this thesis

The structure of this thesis follows largely the methodology that has been outlined in Section 1.3.2. The thesis is divided in five parts, each of which consists of a number of related chapters.

- **Part I: Prelude.**
  - *Chapter 1: Introduction.*
    The present chapter, an introduction to the thesis.
  - *Chapter 2: An overview of C.*
    Attempts to present important aspects of the syntax and semantics of C which affect the rest of the thesis. This chapter contains the definition of C's abstract syntax and explains how the developed semantics deviates from the standard.
  - *Chapter 3: Mathematical background.*
    Briefly presents the mathematical background that is used as a basis for this thesis, including basic category theory, domain theory, monads and monad transformers.
- **Part II: Static semantics.**
  - *Chapter 4: Static semantic domains.*
    Defines the domains that are used to describe the static semantics of C, together with the basic operations that can be performed on their elements.
  - *Chapter 5: Static semantics of declarations.*
    Defines the static semantics of C declarations, including external declarations and translation units.

  - *Chapter 6: Static semantics of recursively defined types.*
    Revises the definition of static semantic domains in Chapter 4 to allow for recursively defined data types.

- **Part III: Typing semantics.**

  - *Chapter 7: Typing judgements.*
    Introduces to the notions of typing semantics, typing judgements, rules and derivations. It also discusses issues specific to the typing semantics of C.
  - *Chapter 8: Typing semantics of expressions.*
    Defines the typing semantics of C expressions.
  - *Chapter 9: Typing semantics of declarations.*
    Defines the typing semantics of C declarations, external declarations and translation units.
  - *Chapter 10: Typing semantics of statements.*
    Defines the typing semantics of C statements.

- **Part IV: Dynamic semantics.**

  - *Chapter 11: Dynamic semantic domains.*
    Defines the domains that are used to describe the dynamic semantics of C and the basic operations on their elements. A number of monads representing different aspects of computations is also defined here.
  - *Chapter 12: Dynamic semantics of expressions.*
    Defines the dynamic semantics of C expressions, including unspecified evaluation order, unspecified order of side effects and sequence points.
  - *Chapter 13: Dynamic semantics of declarations.*
    Defines the dynamic semantics of C declarations, external declarations and translation units.
  - *Chapter 14: Dynamic semantics of statements.*
    Defines the dynamic semantics of all C statements, including jump statements such as *break;* and *goto.*

- **Part V: Epilogue.**

  - *Chapter 15: Implementation.*
    Outlines and the implementation of the developed semantics in Haskell and the obtained results from its evaluation. It also discusses issues arising in the implementation of denotational descriptions with functional and object-oriented languages.
  - *Chapter 16: Related work.*
    Presents related research in the fields of real programming language semantics, the use of monads and monad transformers in denotational semantics and the formal definition of C. It also attempts a comparison between this thesis and others' research, whenever applicable.
  - *Chapter 17: Conclusion.*
    Summarizes the accomplishments and the contribution of this work and discusses directions for future research.

# Chapter 2

# An overview of C

*"C is quirky, flawed and an enormous success."*
Dennis Ritchie, 1993, in [Ritc93]

CHAPTER OVERVIEW

This chapter begins with an informal presentation of selected issues related to the syntax and semantics of the C programming language, in Section 2.1. The rest of the chapter aims at defining the language whose semantics is described in this thesis, which differs slightly from ANSI C, and uses for this purpose a semi-formal approach. Section 2.2 defines the abstract syntax of the language, using the BNF notation, whereas Section 2.3 summarizes the known deviations between the specified language and the ANSI C standard in an informal way.

## 2.1 Selected issues from the syntax and semantics of C

COMPLEXITIES

Developing a formal definition for the C programming language is a hard task. The main sources of complexity are several syntactic and semantic issues particular to C, which fall roughly in the following three categories:

- *Complex type system*: The basic data types are machine oriented and the same is true for the whole philosophy of C's type system. The connection between pointers and arrays and pointer arithmetic are also characteristic of C. Incomplete types, recursively defined types and bit-fields are also sources of complexity.
- *Complex control flow*: C features complex control structures such as the *for* and *switch* statements, and various kinds of jump statements, either restricted (*break* and *continue*) or unrestricted (*goto*). An additional complexity is created by the presence of variable declarations in block statements, in conjunction with jump statements.
- *Complex expression evaluation*: The presence of side-effects in expressions is the major source of complexity here. Combined with unspecified evaluation order, this potentially leads to non-determinism. Expression evaluation is memory-oriented and the standard introduces the mechanism of sequence points, in order to enforce portability.

UNDER-SPECIFICATION

The semantics of C are under-specified by the standard, in an attempt not to overly restrict implementations. There are three types of under-specified behaviour distinguished by the standard:

- *Implementation-defined*: Behaviour of a correct program construct and correct data which depends on the characteristics of the implementation. The implementation must properly document such behaviour.

- *Unspecified* Behaviour of a correct program construct and correct data, for which the standard explicitly imposes no requirements. Implementations need not document such behaviour.
- *Undefined*: Behaviour of a non-portable or incorrect program construct, or of incorrect data. In such cases, implementations are allowed a wide range of actions, including program termination, documented or even unpredictable behaviour.

The rest of this section discusses the main characteristics of C. It is structured in three subsections: *declarations, expressions* and *statements.* The same structure is followed throughout this thesis, in the definition of the semantics. For a thorough introduction to the C programming language, the reader is referred to [Kern88, Harb95].

### 2.1.1 Declarations

SYNTAX OF DECLARATIONS

The syntax of C declarations is probably one of the most characteristic points of the language. Declarations are written in a way similar to way that the declared identifiers are used. This can be confusing at first and frequently requires the use of parentheses. As an example, consider the following two declarations:

```
int * f ();
int (*p) ();
```

The first declares a function "`f`" which returns a pointer to an integer. The second declares a pointer "`p`" to a function returning an integer. One can better understand the effect of the second declaration by regarding the phrase "`*p`" as a function returning an integer, as suggested by the parentheses.

TYPES

The type system of C supports the following types:

- *Character types*, that is, the type `char`, its `signed` version and its `unsigned` version, which are all different.
- *Integer types* in three sizes (`short int`, `int` and `long int`), each of which can be either `signed` or `unsigned`. Omission of sign information makes the type `signed`.
- *Floating types* in three sizes (`float`, `double` and `long double`).
- *Enumerated types*, declared by using `enum`.
- *Structure types*, declared by using `struct`, containing a sequentially allocated non-empty set of member objects.
- *Union types*, declared by using `union`, containing an overlapping non-empty set of member objects.
- *Array types*, containing a contiguously allocated non-empty set of objects with a specific element type.
- *Function types*, describing functions with a specific return type and parameters.
- *Pointer types*, pointing to objects or functions of a specific referenced type.
- The *empty type*, specified by `void`.

QUALIFIERS

*Qualified types* can be formed by applying the qualifiers `const` and `volatile` to an unqualified type. For each unqualified type, three qualified versions exist: one qualified with `const`, one with `volatile` and one with `const volatile`. The use of `const` declares that an object is constant and any attempt to modify it leads to undefined behaviour. Objects declared as `volatile` may be modified in ways unknown to the implementation, independently of program execution.

INCOMPLETE TYPES

Array types of unknown size, structure and union types of unknown contents, and the type `void` are *incomplete types.* Except for `void`, incomplete types can be completed by specifying the unknown information in a later declaration. In general, it is not possible to declare an object of an incomplete type. Incomplete types may be used for the recursive definition of structures and unions, as discussed in Chapter 6.

INITIALIZATION

Declared objects may be initialized as part of their declarations by expressions of an appropriate type. Special syntax is used for the initialization of arrays, structures and unions, requiring lists of initializers enclosed in braces. Initialization of objects is performed in the order that the initializers appear in the program source. In general, the initial values of objects that are not explicitly initialized are undefined.

TAGS AND TYPEDEFS

C supports two kinds of identifiers which denote types. *Tags* can be used to describe structures, unions and enumerations. They must always be used in conjunction with the keywords `struct`, `union` and `enum`. On the other hand, identifiers declared by using `typedef` may denote any kind of type. The syntax is the same as for the declaration of normal identifiers.

### 2.1.2 Expressions

EXPRESSIONS AND SIDE EFFECTS

In general, *expressions* are sequences of operands and operators. An expression may specify the computation of a value, designate an object or a function, generate side-effects, or combine any of the previous. Expressions that designate objects are called *l-values* and expressions that designate functions are called *function designators*. A *side effect* is a change in the execution environment, such as the modification of an object's value or the access to an object declared as `volatile`.

SEQUENCE POINTS

The standard defines the semantics of C in terms of an abstract machine. It is not required that implementations behave exactly as the abstract machine does and, for example, side effects need not take place at the same time when they are generated. However, at certain specified points during program execution, called *sequence points*, all side effects from previous evaluations must be complete and no side effects of subsequent evaluations must have taken place. The standard also forbids modification of an object more than once between the previous and the next sequence point and specifies that, in this case, the object's prior value may only be accessed to determine its new contents.

EVALUATION ORDER

In general, the order in which subexpressions are evaluated and the order in which side effects take place are unspecified by the standard. Evaluation order is affected by the precedence and associativity of operators, as well as the use of parentheses, but not completely determined by these factors.

OPERATORS

C supports a large number of operators. Among them, the most characteristic of the language are the pointer reference and dereference unary operators (`&` and `*`), the prefix and postfix unary assignments (`++` and `--`), the composite binary assignment operators (e.g. `+=`), the bitwise manipulation operators, the logical short-circuit operators (`&&` and `||`), the comma operator and the ternary conditional operator `?:`.

### 2.1.3 Statements

EXPRESSION AND COMPOUND STATEMENTS

The omission of an assignment statement from C is compromised by the presence of the expression statement and the fact that side effects can be generated by expressions. Compound statements may be used in order to group statements in blocks and may contain declarations of local objects. Such objects are allocated each time program execution enters the block and are destroyed when it leaves

the block, regardless of how the block is entered or leaved. However, object initializations only take place when the block is normally entered.

SELECTION STATEMENTS

The selection statements of C are `if` (with an optional `else` clause) and `switch`. The second is controlled by an integral expression. In its body, the labels `case` and `default` may be defined and the `break` statement may be used to interrupt execution of the body.

ITERATION STATEMENTS

Probably the most characteristic statement of C is the `for` statement, with a rather peculiar syntax. Other iteration statements are `while` and `do`. The `break` statement can be used for the interruption of an iteration statement, whereas the `continue` statement can be used for starting a new iteration.

UNRESTRICTED JUMPS

C allows unrestricted jumps in the body of a function, by means of the `goto` statement in conjunction with labeled statements.[1] The use of a `goto` statement potentially causes execution to leave one or more nested blocks and enter one or more other nested blocks.

RETURN STATEMENT

The `return` statement is used for determining a value that is returned from a function. If the function's return type is not `void` and the `return` statement is not followed by an expression, the returned value is undefined and must not be used.

## 2.2 Abstract syntax

NEED FOR AN ABSTRACT SYNTAX

A context-free grammar for the concrete syntax of the C programming language is given in Appendix B of the ANSI C standard, using a BNF-like notation. The grammar contains a number of ambiguities, mainly caused by the definition of *typedefs* and other context-sensitive features of the language. All these ambiguities are resolved in implementations by using appropriate symbol tables. However, the concrete syntax produces parse trees which contain redundant information and, for this reason, are not appropriate for the specification of program semantics. A more abstract syntax for C is defined in this section by means of a context-free grammar in BNF notation. Although the same ambiguities are present in the abstract syntax grammar, the produced parse trees are greatly simplified and, therefore, more manageable.

### 2.2.1 Declarations

EXTERNAL DECLARATIONS

External declarations are declarations that appear outside any function and therefore have file scope. A translation unit is a sequence of one or more external declarations. Each external declaration can in be either a normal declaration or a function definition.

- *translation-unit* ::= *external-declaration-list*
- *external-declaration-list* ::= *external-declaration* | *external-declaration external-declaration-list*
- *external-declaration* ::= *declaration*
  | *declaration-specifiers declarator* { *declaration-list statement-list* }

DECLARATIONS

A declaration list may contain zero or more declarations. Each declaration consists of a storage class specifier, a type qualifier, a type specifier and a list of declarations with possible initializations. The first two are optional.

[1] Non-local jumps are supported by the standard library of C.

- *declaration-list* ::= $\epsilon$ | *declaration declaration-list*
- *declaration* ::= *declaration-specifiers init-declarator-list* `;`
- *declaration-specifiers* ::= *storage-class-specifier type-qualifier type-specifier*

SPECIFIERS AND QUALIFIERS

The storage class specifiers defined in the C standard are *typedef*, *extern*, *static*, *auto* and *register*. In this thesis however, only *typedef* and the absence of a storage class specifier are allowed, as stated by deviation D-5 in Section 2.3. The latter is equivalent to the standard's *auto* specifier. Allowed type qualifiers are *const* and *volatile*, whereas type specifiers range over a wide variety of types.

- *storage-class-specifier* ::= $\epsilon$ | `typedef`
- *type-qualifier* ::= $\epsilon$ | `const` | `volatile` | `const volatile`
- *type-specifier* ::= `void` | `char` | `signed char` | `unsigned char`
  | `short int` | `unsigned short int` | `int` | `unsigned int`
  | `long int` | `unsigned long int` | `float` | `double` | `long double`
  | *struct-specifier* | *union-specifier* | *enum-specifier* | *typedef-name*

DECLARATORS AND INITIALIZATION

A list of declarators consists of zero or more declarators, each of which may be followed by an initializer. Initializers are expressions or bracketed lists of initializers.

- *init-declarator-list* ::= $\epsilon$ | *init-declarator init-declarator-list*
- *init-declarator* ::= *declarator* | *declarator* = *initializer*
- *initializer* ::= *expression* | { *initializer-list* }
- *initializer-list* ::= *initializer* | *initializer* `,` *initializer-list*

STRUCTURES, UNIONS AND ENUMERATIONS

Structure and union specifiers contain non-empty lists of member declarations. Each such declaration is similar to a normal declaration, with the absence of storage class specifiers and initializers. Furthermore, bit-fields may be defined as members. Enumerations are defined as non-empty lists of elements, possibly followed by their denoted values.

- *struct-specifier* ::= `struct` *I* { *struct-declaration-list* }
  | `struct` { *struct-declaration-list* }
  | `struct` *I*
- *union-specifier* ::= `union` *I* { *struct-declaration-list* }
  | `union` { *struct-declaration-list* }
  | `union` *I*
- *struct-declaration-list* ::= *struct-declaration* | *struct-declaration struct-declaration-list*
- *struct-declaration* ::= *struct-specifiers struct-declarator-list* `;`
- *struct-specifiers* ::= *type-qualifier type-specifier*
- *struct-declarator-list* ::= *struct-declarator* | *struct-declarator struct-declarator-list*
- *struct-declarator* ::= *declarator* | *declarator* `:` *constant-expression*
- *enum-specifier* ::= `enum` *I* { *enumerator-list* } | `enum` { *enumerator-list* } | `enum` *I*
- *enumerator-list* ::= *enumerator* | *enumerator enumerator-list*
- *enumerator* ::= *I* | *I* = *constant-expression*

DECLARATORS Declarators may denote ordinary identifiers, arrays of known or unknown size, identifiers corresponding to functions or pointers. Note that function prototypes are always required in function declarators, as stated by deviation D-3 in Section 2.3. Parameter type lists consist of zero or more parameter declarations, optionally followed by an ellipsis.[2] Parameter declarations do not necessarily name the declared parameters, as shown by the use of abstract declarators.

- *declarator* ::= *I* | *declarator* `[` *constant-expression* `]` | *declarator* `[ ]`
  | *declarator* `(` *parameter-type-list* `)` | `*` *type-qualifier declarator*
- *parameter-type-list* ::= $\epsilon$ | `...` | *parameter-declaration parameter-type-list*
- *parameter-declaration* ::= *declaration-specifiers declarator*
  | *declaration-specifiers abstract-declarator*
- *abstract-declarator* ::= $\epsilon$ | *abstract-declarator* `[` *constant-expression* `]` | *abstract-declarator* `[ ]`
  | *abstract-declarator* `(` *parameter-type-list* `)`
  | `*` *type-qualifier abstract-declarator*

TYPE NAMES Type synonyms defined by using the *typedef* storage class specifier are just identifiers. A full type name consists of an optional type qualifier, a type specifier and an abstract declarator.

- *typedef-name* ::= *I*
- *type-name* ::= *type-qualifier type-specifier abstract-declarator*

### 2.2.2 Expressions

Among all C program phrases, expressions are the most simplified by the use of an abstract syntax. Primary expressions are identifiers, constants or string literals. Postfix expressions include array subscripts, function calls and structure or union member dereferences. The actual arguments to a function call are a list of zero or more expressions. Prefix expressions allow the use of unary operators, unary assignments and type casts. Binary expressions, binary assignments and the conditional operator complete the picture. The non-terminal for constant expressions, equivalent to that for expressions, is only kept for the sake of clarity.

- *expression* ::= *I* | *n* | *f* | *c* | *s* | *expression* `[` *expression* `]` | *expression* `(` *arguments* `)`
  | *expression* `.` *I* | *expression* `->` *I* | `sizeof` *expression* | `sizeof` `(` *type-name* `)`
  | *unary-operator expression* | `(` *type-name* `)` *expression*
  | *expression binary-operator expression* | *unary-assignment expression*
  | *expression binary-assignment expression* | *expression* `?` *expression* `:` *expression*
- *arguments* ::= $\epsilon$ | *expression* `,` *arguments*
- *constant-expression* ::= *expression*

OPERATORS The large variety of operators is one of C's most characteristic features. Operators are distinguished on the one hand by their arity (unary, binary or ternary) and on the other hand in normal operators and assignment operators.

- *unary-operator* ::= `&` | `*` | `+` | `-` | ~ | `!`
- *binary-operator* ::= `*` | `/` | `%` | `+` | `-` | `<<` | `>>` | `<` | `>` | `<=` | `>=` | `==` | `!=`
  | `&` | `^` | `|` | `&&` | `||` | `,`
- *unary-assignment* ::= `++` (*prefix*) | `++` (*postfix*) | `--` (*prefix*) | `--` (*postfix*)
- *binary-assignment* ::= `=` | `*=` | `/=` | `%=` | `+=` | `-=` | `<<=` | `>>=` | `&=` | `^=` | `|=`

[2] See also deviation D-10 in Section 2.3.

### 2.2.3 Statements

The abstract syntax for statements is also relatively simple. A first thing to notice is the presence of a unique scope identifier *id* in block statements, which may be a simple non-zero natural number. Such identifiers can be easily generated automatically in the transition from concrete to abstract syntax and greatly simplify the definition of dynamic semantics, as will be seen in Part IV. Notice also the two variations for statements *if* and *return.*

- *statement-list* ::= $\epsilon$ | *statement statement-list*
- *statement* ::= `;` | *expression* `;` | `{` *id declaration-list statement-list* `}`
  | `if (` *expression* `)` *statement* | `if (` *expression* `)` *statement* `else` *statement*
  | `switch (` *expression* `)` *statement*
  | `case` *constant-expression* `:` *statement* | `default :` *statement*
  | `while (` *expression* `)` *statement* | `do` *statement* `while (` *expression* `) ;`
  | `for (` *expression-optional* `;` *expression-optional* `;` *expression-optional* `)` *statement*
  | `continue ;` | `break ;` | `return ;` | `return` *expression* `;`
  | *I* `:` *statement* | `goto` *I* `;`
- *expression-optional* ::= $\epsilon$ | *expression*

## 2.3 Deviations from the standard

REASONS FOR DEVIATIONS

The task of formally specifying the semantics of the C programming language is a very hard one, due to the high complexity of the language itself. A complete and accurate semantics, with respect to the standard, would inevitably be very complex if not altogether infeasible. In the developed semantics, several deviations from the standard have been allowed, thus reducing the value of the semantics in terms of completeness, accuracy or both. The author believes that these deviations only affect the language slightly and justifies their existence with one of the following two arguments:

1. In the case of some aspects of C that are not supported by the developed semantics, the increase in complexity of the semantics that would be required in order to accommodate them is disproportionate to the benefits from having them.
2. Some other features are considered as obsolescent by the current standard and are only supported for compatibility with old fashioned programming practice. Such features will probably be omitted in the forthcoming revised version of the standard and, for this reason, they were not included in the developed semantics if their inclusion meant a significant increase in complexity.

To the best of the author's knowledge, the deviations between the developed semantics and (the author's interpretation of) the ANSI C standard are the following. The most important deviations are mentioned in the beginning of the list, but this order of importance is only subjective.

SINGLE TRANSLATION UNIT

**Deviation D-1.** The C programs whose semantics is defined consist of a single translation unit. Therefore, the notion of a program in this thesis is equivalent to that of a translation unit. The author believes that this restriction does not impose significant problems, since programs spanning through multiple translation units can be easily concatenated into a single unit by a relatively simple preprocessor. The use of function libraries presents a problem here. However, on condition that a library's functions are themselves written in C the same concatenation process may be

assumed.[3] Following a different approach, the static and dynamic semantics of library functions can be externally specified.

FILE SCOPE **Deviation D-2.** Identifiers declared in the file scope, i.e. outside function definitions, are treated in the same way as the others. That is, file scope is treated as an external block scope. This is consistent with deviation D-1. As a side effect however, no implicit initialization of external objects takes place, as specified in §6.7.2 of the standard.

ENFORCED FUNCTION PROTOTYPES **Deviation D-3.** Function prototypes are required to exist for all called functions and the actual arguments are required to comply with the prototypes. This is a step towards a more strongly-typed C, which will probably be taken in the revised standard.

DECLARATION OF IDENTIFIERS **Deviation D-4.** No identifiers other than statement labels may be declared as a consequence of processing expressions or statements. That is, the static environment may only change as a result of processing declarations, excluding expressions that appear in initializers. The standard allows the declaration of identifiers in two more cases. First, an identifier of function type is indirectly declared whenever a call to an unknown function is encountered. In the developed semantics, this would also violate what has been discussed in deviation D-3. Second, the declaration of tags is possible in type names which may be present in expressions, as is the case of "`sizeof(struct tag { int whatever; })`". The author believes that such declarations should not be allowed and regards this deviation more as a correction to the standard than as an omission.

OMITTED STORAGE SPECIFIERS **Deviation D-5.** Storage specifiers other than *typedef* are not supported. The cases of *auto* and *extern* could be incorporated rather easily, but were left out as a side effect from deviation D-1, since they are not very useful in programs consisting of a single translation unit. The omission of *static* is the most important consequence of this deviation. Static variables may of course be preprocessed out, but the author believes that a solution integrated in the semantics should be investigated. Finally, the increase in complexity required by the inclusion of *register* is not clear. This specifier offers little to the programmer and it has been argued that a good optimizing implementation of C should be able to determine which variables should be allocated to registers without the programmer's help.

FULLY BRACKETED INITIALIZERS **Deviation D-6.** Initializers must be fully bracketed according to the types of the initialized objects, in contrast to the less strict bracketing that is allowed in §6.5.7 of the standard. String literals are however allowed as initializers for character arrays.

BIT-FIELD TYPES **Deviation D-7.** The types *int* and *signed int* are always considered identical, even in the case of bit-fields. This could be corrected with a number of changes in the abstract syntax, separating the declaration of bit-field members from that of ordinary members.

TYPEDEF IN PARAMETERS **Deviation D-8.** The use of the *typedef* storage class specifier in a parameter declaration, which is disallowed by the standard, is simply ignored by the developed semantics. The use of a storage class specifier in parameter declarations could have been altogether disallowed, since the only case allowed by the standard is *register*. This should be corrected.

[3] Obviously, the specification of functions written in assembly or other languages is not a responsibility of a formal semantics for C.

**Deviation D-9.** The use of the *typedef* storage class specifier is allowed in function definitions, while it is disallowed by the standard. In this case, the developed semantics simply ignores the function's body. This should also be corrected. TYPEDEF IN FUNCTIONS

**Deviation D-10.** The use of ellipsis in function prototypes does not require the presence of other named parameters, e.g. "`int f (...);`" which is not allowed by the standard is allowed in the developed semantics. The author considered unnecessary to comply to this requirement of the standard, which is imposed solely because of implementation practice. USE OF ELLIPSIS

**Deviation D-11.** In consistence with deviation D-1, two structure or union types are considered to be compatible by the developed semantics only if they are the same type. The standard also allows structure or union types defined in different translation units to be compatible if they agree in the order and types of members. STRUCTURE AND UNION TYPES

It should be noted also, more as a clarification than as a deviation from the standard, that the developed semantics covers only the ANSI C language. It is not the author's intention to specify the semantics of C's standard library, which is also informally defined in the C standard. The omission of everything included in the standard library allows the developed semantics to ignore sources of complexity originating from an underlying operating system, such as input/output devices, file management, dynamic allocation of memory, signals and interrupts, etc.

# Chapter 3

# Mathematical background

CHAPTER OVERVIEW

This chapter attempts to define the mathematical background that is required for understanding this thesis. A brief introduction to category theory is presented in Section 3.1. Monads and monad transformers are introduced in Section 3.2. In Section 3.3 an overview of domain theory is given and the chapter concludes in Section 3.4 with a definition of the meta-language for denotational semantics that is used in this thesis. Throughout this chapter, only definitions and useful theorems are stated. The reader is referred to the related literature for a more informative introduction and the proofs of the theorems.

## 3.1 Category theory

Category theory was developed in an attempt to unify simple abstract concepts that were applicable in many branches of mathematics. Excellent introductions to category theory and its application in Computer Science can be found in [Pier90, Gogu91, Pier91, Aspe91, Barr96].

### 3.1.1 Basic definitions

CATEGORIES

**Definition 3.1**. A *category* $\mathsf{C}$ is a collection of *objects* and a collection of *arrows*[1] satisfying the following properties:

- For each arrow $f$ there is a *domain* object $dom(f)$ and a *codomain* object $codom(f)$, and by writing $f : x \to y$ it is indicated that $x = dom(f)$ and $y = codom(f)$.
- For every pair of arrows $f : x \to y$ and $g : y \to z$ there is a *composite* arrow $g \circ f : x \to z$.
- Composition of arrows is associative, i.e. for all arrows $f : x \to y$, $g : y \to z$ and $h : z \to w$ it is $h \circ (g \circ f) = (h \circ g) \circ f$.
- For each object $x$ there is an *identityarrow!identity* arrow $id_x : x \to x$.
- Identity arrows are identities for arrow composition, i.e. for all arrows $f : x \to y$ it is $f \circ id_x = id_y \circ f = f$.

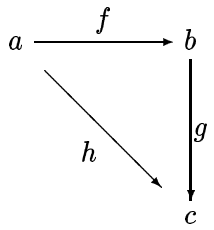
**Definition 3.2**. An object $x$ of category $\mathsf{C}$ is *initial* if for every object $y$ there is exactly one arrow $f : x \to y$. Dually, an object $x$ is *terminal* if for every object $y$ there is exactly one arrow $f : y \to x$.

**Definition 3.3**. Two objects $x$ and $y$ of category $\mathsf{C}$ are *isomorphic* if there are arrows $f : x \to y$ and $g : y \to x$ such that $f \circ g = id_y$ and $g \circ f = id_x$. Arrows $f$ and $g$ are called *isomorphisms.*

[1] Arrows are often called *morphisms* in literature.

COMMUTING DIAGRAMS

Properties of categories are commonly presented using *commuting diagrams*. A diagram is a graph whose nodes are objects and whose edges are arrows. A diagram *commutes* if for every pair of nodes and for every pair of paths connecting these two nodes the composition of arrows along the first path is equal to the composition of arrows along the second. An example of a commuting diagram, implying that $g \circ f = h$, is shown below.




### 3.1.2 Functors and natural transformations

FUNCTORS

**Definition 3.4**. A *functor* $F$ from category $\mathsf{C}$ to category $\mathsf{D}$, written as $F : \mathsf{C} \to \mathsf{D}$, is a pair of mappings. Every object $x$ in $\mathsf{C}$ is mapped to an object $F(x)$ in $\mathsf{D}$ and every arrow $f : x \to y$ in $\mathsf{C}$ is mapped to an arrow $F(f) : F(x) \to F(y)$ in $\mathsf{D}$. Moreoever, the following properties must be satisfied:

- $F(id_x) = id_{F(x)}$ for all objects $x$ in $\mathsf{C}$.
- $F(g \circ f) = F(g) \circ F(f)$ for all arrows $f : x \to y$ and $g : y \to z$ in $\mathsf{C}$.

**Definition 3.5**. An *endofunctor* on category $\mathsf{C}$ is a functor $F : \mathsf{C} \to \mathsf{C}$.

**Definition 3.6**. If $F : \mathsf{C} \to \mathsf{D}$ and $G : \mathsf{D} \to \mathsf{E}$ are functors, then their *composition* is a functor $G \circ F : \mathsf{C} \to \mathsf{E}$. It is defined by taking $(G \circ F)(x) = G(F(x))$ and $(G \circ F)(f) = G(F(f))$.

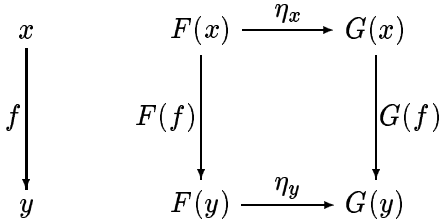
**Definition 3.7**. For every category $\mathsf{C}$, an *identity functor* $id_{\mathsf{C}} : \mathsf{C} \to \mathsf{C}$ can be defined by taking $id_{\mathsf{C}}(x) = x$ and $id_{\mathsf{C}}(f) = f$.

Note that if $F : \mathsf{C} \to \mathsf{C}$ is an endofunctor and $n$ is a positive natural number, the notation $F^n : \mathsf{C} \to \mathsf{C}$ can be used for the composition of $F$ with itself $n$ times. The notation can be extended so that $F^0 = id_{\mathsf{C}}$.

**Theorem 3.1**. Functors preserve isomorphisms.

**Theorem 3.2**. Identity functors are identities for functor composition, that is, if $F : \mathsf{C} \to \mathsf{D}$ is a functor, then $F \circ id_{\mathsf{C}} = id_{\mathsf{D}} \circ F = F$

**Definition 3.8**. If $F : \mathsf{C} \to \mathsf{D}$ and $G : \mathsf{C} \to \mathsf{D}$ are functors, then a *natural transformation* $\eta$ between $F$ and $G$, written as $\eta : F \dot{\to} G$ is a family of arrows in $\mathsf{D}$. In this family, an arrow $\eta_x : F(x) \to G(x)$ in $\mathsf{D}$ is defined for every object $x$ in $\mathsf{C}$. Moreover, the following diagram must commute: NATURAL TRANSFORMATIONS

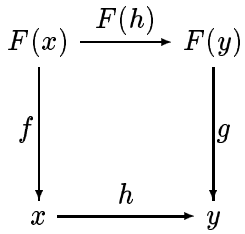



**Definition 3.9**. If $F : \mathsf{C} \to \mathsf{D}$, $G : \mathsf{C} \to \mathsf{D}$ and $H : \mathsf{C} \to \mathsf{D}$ are functors, $\alpha : F \dot{\to} G$ and $\beta : G \dot{\to} H$ are natural transformations, then the *composition* $\beta \circ \alpha : F \dot{\to} H$ is a natural transformation. It is defined by taking $(\beta \circ \alpha)_x = \beta_x \circ \alpha_x$

**Definition 3.10**. If $F : \mathsf{C} \to \mathsf{D}$, $G : \mathsf{C} \to \mathsf{D}$, $H : \mathsf{D} \to \mathsf{E}$ and $K : \mathsf{D} \to \mathsf{E}$ are functors, $\alpha : F \dot{\to} G$ and $\beta : H \dot{\to} K$ are natural transformations, then the compositions $\beta \circ F : H \circ F \dot{\to} K \circ F$ and $H \circ \alpha : H \circ F \dot{\to} H \circ G$ are natural transformations. They are defined by taking $(\beta \circ F)_x = \beta_{F(x)}$ and $(H \circ \alpha)_x = H(\alpha_x)$.

**Definition 3.11**. If $\mathsf{C}$ is a category and $F : \mathsf{C} \to \mathsf{C}$ is an endofunctor on $\mathsf{C}$, then a *$F$-algebra* is an arrow $f : F(x) \to x$. $F$-ALGEBRAS

**Definition 3.12**. If $\mathsf{C}$ is a category, $F : \mathsf{C} \to \mathsf{C}$ is an endofunctor on $\mathsf{C}$, and $f : F(x) \to x$ and $g : F(y) \to y$ are $F$-algebras, then a *$F$-homomorphism* between $f$ and $g$ is an arrow $h : x \to y$ in $\mathsf{C}$ such that the following diagram commutes:



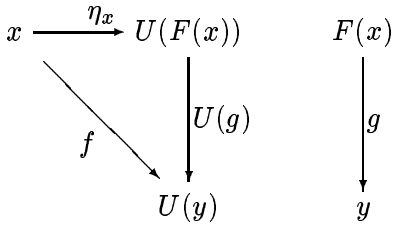

**Theorem 3.3**. $F$-algebras and $F$-homomorphisms form a category.

**Definition 3.13**. An *initial $F$-algebra* is an initial object of the category of Theorem 3.3.

**Theorem 3.4**. If $f : F(x) \to x$ is an initial $F$-algebra, then $f$ is an isomorphism.

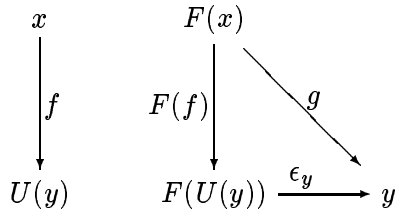
### 3.1.3 Adjunctions

**Definition 3.14**. If $F : \mathsf{C} \to \mathsf{D}$ and $U : \mathsf{D} \to \mathsf{C}$ are functors, then $F$ is *left adjoint* to $U$ and $U$ is *right adjoint* to $F$ if there is a natural transformation $\eta : id_{\mathsf{C}} \dot{\to} U \circ F$ such that for any objects $x$ in $\mathsf{C}$ and $y$ in $\mathsf{D}$ and any arrow $f : x \to U(y)$ in $\mathsf{C}$, there is a unique arrow $g : F(x) \to y$ such that the following diagram commutes.




Left and right adjoints are written as $F \dashv U$ and the triple $\langle F, U, \eta \rangle$ constitutes an *adjunction.* The transformation $\eta$ is called the *unit* of the adjunction.

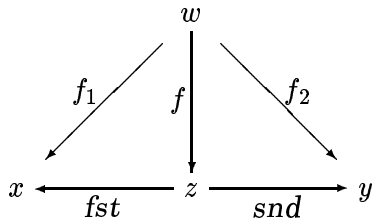
**Theorem 3.5**. If $F : \mathsf{C} \to \mathsf{D}$ and $U : \mathsf{D} \to \mathsf{C}$ are functors such that $F \dashv U$, then there is a natural transformation $\epsilon : F \circ U \dot{\to} id_{\mathsf{D}}$ such that for any objects $x$ in $\mathsf{C}$ and $y$ in $\mathsf{D}$ and any arrow $g : F(x) \to y$, there is a unique arrow $f : x \to U(y)$ in $\mathsf{C}$ such that the following diagram commutes.




The transformation $\epsilon$ is called the *counit* of the adjunction.

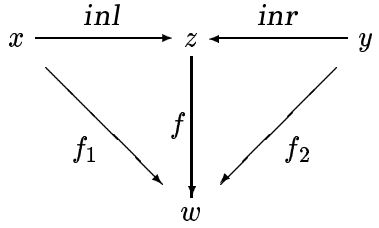
### 3.1.4 Products and sums

**Definition 3.15**. Let $x$ and $y$ be two objects in a category $\mathsf{C}$. Then, a *product* of $x$ and $y$ is an object $z \in \mathsf{C}$ together with two arrows $fst : z \to x$ and $snd : z \to y$ such that, for any object $w \in \mathsf{C}$ and arrows $f_1 : w \to x$ and $f_2 : w \to y$, there is a unique arrow $f : w \to z$ making the following diagram commute:




Such a product object is commonly written as $x \times y$ and the unique arrow $f$ is written as $\langle f_1, f_2 \rangle$.

**Definition 3.16**. Let $x$ and $y$ be two objects in a category $\mathsf{C}$. Then, a *sum* of $x$ and $y$ is an object $z \in \mathsf{C}$ together with two arrows $inl : x \to z$ and $inr : y \to z$ such that, for any object $w \in \mathsf{C}$ and arrows $f_1 : x \to w$ and $f_2 : y \to w$, there is a unique arrow $f : z \to w$ making the following diagram commute:




Such a sum object is commonly written as $x + y$ and the unique arrow $f$ is written as $[\, f_1, f_2 \,]$.

**Definition 3.17**. It is easy to generalize binary products and sums, as defined in Definition 3.15 and Definition 3.16, to *finite* products and sums of the form $x_1 \times x_2 \times \ldots \times x_n$ and $x_1 + x_2 + \ldots + x_n$ respectively. For $n = 0$ the product can be any terminal object and the sum can be any initial object of the category. Furthermore, for $n = 1$ both product and sum are isomorphic to $x_1$.

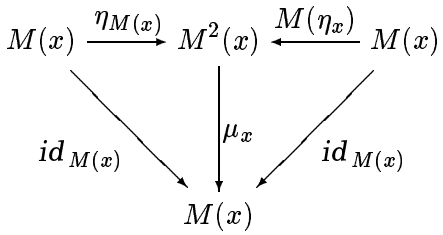
**Theorem 3.6**. Products and sums are unique up to isomorphism.

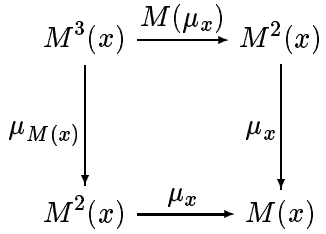
## 3.2 Monads and monad transformers

The notion of *monad*, also called triple, is not new in the context of category theory. In Computer Science, monads became very popular in the 1990s. The categorical properties of monads are discussed in most books on category theory, e.g. in [Barr96]. For a comprehensive introductions to monads and their use in denotational semantics the user is referred to [Mogg90]. A somehow different approach to the definition of monads is found in [Wadl92], which expresses the current practice of monads in functional programming. The two approaches are equivalent and they are both used in this thesis. The categorical approach is used for the definition of monads, since it is much more elegant, and the functional approach is used in the meta-language for describing semantics and in the rest of the thesis.

### 3.2.1 The categorical approach

**Definition 3.18**. A *monad* on a category $\mathsf{C}$ is a triple $\langle M, \eta, \mu \rangle$, where $M : \mathsf{C} \to \mathsf{C}$ is an endofunctor, $\eta : id_{\mathsf{C}} \dot{\to} M$ and $\mu : M^2 \dot{\to} M$ are natural transformations. For all objects $x$ in $\mathsf{C}$, the following diagrams must commute. MONADS




The transformation $\eta$ is called the *unit* of the monad, whereas the transformation $\mu$ is called the *multiplication* or *join.*

The commutativity of these two diagrams is equivalent to the following three equations, commonly called the three *monad laws*:

$\mu_x \circ \eta_{M(x)} = id_{M(x)}$ *(1st Monad Law)*

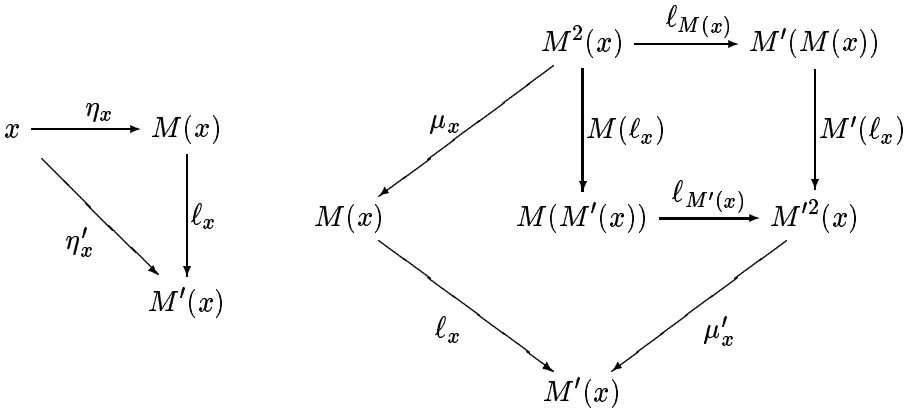
$\mu_x \circ M(\eta_x) = id_{M(x)}$ *(2nd Monad Law)*

$\mu_x \circ M(\mu_x) = \mu_x \circ \mu_{M(x)}$ *(3rd Monad Law)*

**Theorem 3.7**. If $F : \mathsf{C} \to \mathsf{D}$ and $U : \mathsf{D} \to \mathsf{C}$ are functors such that $F \dashv U$, with $\eta : id_{\mathsf{C}} \dot{\to} U \circ F$ and $\epsilon : F \circ U \dot{\to} id_{\mathsf{D}}$ being the unit and counit of the adjunction, then the triple $\langle U \circ F, \eta, U \circ \epsilon \circ F \rangle$ is a monad on $\mathsf{C}$. This monad is called the *induced* monad of the adjunction.

MONAD MORPHISMS

**Definition 3.19**. If $\langle M, \eta, \mu \rangle$ and $\langle M', \eta', \mu' \rangle$ are monads on a category $\mathsf{C}$, then a *monad morphism* is a natural transformation $\ell : M \dot{\to} M'$ such that for all objects $x$ in $\mathsf{C}$ the following two diagrams commute:




In the right diagram, the small square always commutes by the fact that $\ell$ is a natural transformation.

**Definition 3.20**. If $\mathsf{C}$ is a category, then it is possible to define a *monad category* $\mathsf{Mon}(\mathsf{C})$ by taking as objects the monads on $\mathsf{C}$ and as arrows the monad morphisms on $\mathsf{C}$.

MONAD TRANSFORMERS

**Definition 3.21**. If $\mathsf{C}$ is a category and $\mathsf{Mon}(\mathsf{C})$ the induced monad category, then a *monad transformer* on $\mathsf{C}$ is an endofunctor $R : \mathsf{Mon}(\mathsf{C}) \to \mathsf{Mon}(\mathsf{C})$.[2]

### 3.2.2 The functional approach

ALTERNATIVE DEFINITION

The alternative approach, which has become very popular in the functional programming community, defines a monad on a category $\mathsf{C}$ as a triple $\langle M, unit_{\mathsf{M}}, bind_{\mathsf{M}} \rangle$. In this triple, $M$ is a function between objects of $\mathsf{C}$, $unit_{\mathsf{M}} : x \to M(x)$ is a family of arrows in $\mathsf{C}$ and $bind_{\mathsf{M}}$ is a function between arrows on $\mathsf{C}$. If $x$ and $y$ are objects in $\mathsf{C}$ and $f : x \to M(y)$ is an arrow in $\mathsf{C}$, then $bind_{\mathsf{M}} f : M(x) \to M(y)$. Furthermore, the following properties must be satisfied:

$$bind_{\mathsf{M}}\ unit_{\mathsf{M}} = id \qquad \textit{(Alternative 1st Monad Law)}$$

$$(bind_{\mathsf{M}}\ f) \circ unit_{\mathsf{M}} = f \qquad \textit{(Alternative 2nd Monad Law)}$$

$$(bind_{\mathsf{M}}\ f) \circ (bind_{\mathsf{M}}\ g) = bind_{\mathsf{M}}\ ((bind_{\mathsf{M}}\ f) \circ g) \qquad \textit{(Alternative 3rd Monad Law)}$$

EQUIVALENCE

It can be shown that the two approaches are equivalent. The equations relating $unit_{\mathsf{M}}$ and $bind_{\mathsf{M}}$ in the alternative definition to $\eta$, $\mu$ and the arrow mapping of endofunctor $M$ in the original definition are given below, in both directions.[3] It can also be shown that the two sets of monad laws are equivalent.

[2] Many options for the definition of monad transformers have been suggested in literature. Apart from the definition used here as endofunctors on $\mathsf{Mon}(\mathsf{C})$, other possible definitions are as functions between objects in $\mathsf{Mon}(\mathsf{C})$, as premonads on $\mathsf{Mon}(\mathsf{C})$ (i.e. endofunctors with a unit), and as monads on $\mathsf{Mon}(\mathsf{C})$.

[3] These equations assume that a number of naturality properties, omitted from the alternative definition, are satisfied. Without this assumption, the two definitions are not equivalent.

$$unit_{\mathsf{M}} = \eta \qquad \eta = unit_{\mathsf{M}}$$
$$bind_{\mathsf{M}}\ f = \mu \circ M(f) \qquad \mu = bind_{\mathsf{M}}\ id$$
$$M(f) = bind_{\mathsf{M}}\ (unit_{\mathsf{M}} \circ f)$$

SET-ORIENTED BIND

In a category where objects are sets (or similar mathematical entities) and arrows are functions between sets, then $unit_{\mathsf{M}} : A \to M(A)$ and $bind_{\mathsf{M}} : (A \to M(B)) \to M(A) \to M(B)$ can be regarded as a polymorphic functions, where $A$ and $B$ are arbitrary sets. Then, by swapping the order of the arguments of $bind_{\mathsf{M}}$ and by making it an infix operator $\cdot *_{\mathsf{M}} \cdot : M(A) \to (A \to M(B)) \to M(B)$ the popular form of the *bind* operator as used in practice today is obtained. This is the notation for monads that is used in the following chapters. Furthermore, the notation $lift_{\mathsf{M} \to \mathsf{M}'} : M(A) \to M'(A)$ is sometimes used for the polymorphic functions representing monad morphisms.

## 3.3 Domain theory

The theory of domains was established by Scott and Strachey, in order to provide appropriate mathematical spaces on which to define the denotational semantics of programming languages. Introductions of various sizes and levels can be found in [Scot71, Scot82, Gunt90, Gunt92]. Various kinds of domains are commonly used in denotational semantics, the majority of them based on complete partial orders (cpo's). The variation used in this thesis is one of the possible options.

### 3.3.1 Preliminaries

**Definition 3.22**. A *partial order*, or *poset*, is a set $D$ together with a binary relation $\sqsubseteq$ that is reflexive, anti-symmetric and transitive.

**Definition 3.23**. A subset $P \subseteq D$ of a poset $D$ is *bounded* if there is a $x \in D$ such that $y \sqsubseteq x$ for all $y \in P$. In this case, $x$ is an *upper bound* of $P$.

**Definition 3.24**. The *least upper bound* of a subset $P \subseteq D$, written as $\bigsqcup P$, is an upper bound of $P$ such that, $\bigsqcup P \sqsubseteq x$ for all upper bounds $x$ of $P$.[4]

**Definition 3.25**. A subset $P \subseteq D$ of a poset $D$ is *directed* if every finite subset $F \subseteq P$ has an upper bound $x \in P$.

**Definition 3.26**. If $D$ is a poset and $x \in D$ is one of its elements, then the set $\downarrow x$ of elements *below* $x$ in $D$ is defined as $\downarrow x = \{\, y \in D \mid y \sqsubseteq x \,\}$.

**Definition 3.27**. A subset $P \subseteq D$ of a poset $D$ is *downward closed* if $\downarrow x \subseteq P$ for every element $x \in P$.

**Definition 3.28**. Let $D$ be a poset. An *ideal* over $D$ is a directed and downward closed subset of $D$.

**Definition 3.29**. A poset $D$ is *complete* if every directed subset $P \subseteq D$ has a least upper bound. A complete partial order is also called a *cpo*.

[4] The notation $a \sqcup b$ is used as an abbreviation of $\bigsqcup\{\, a, b \,\}$.

**Definition 3.30**. A non-empty cpo $D$ is *bounded complete* if every bounded subset $P \subseteq D$ has a least upper bound $\bigsqcup P \in D$.

**Definition 3.31**. An element $x \in D$ of a cpo $D$ is *compact* if for all directed subsets $P \subseteq D$ such that $x \sqsubseteq \bigsqcup P$ there is an element $y \in P$ such that $x \sqsubseteq y$. By $\mathsf{K}(D)$ we denote the set of compact elements of $D$.

**Definition 3.32**. A cpo $D$ is *algebraic* if for every $x \in D$ the set $P = \{\, a \in \mathsf{K}(D) \mid a \sqsubseteq x \,\}$ is directed and $\bigsqcup P = x$.

**Definition 3.33**. Let $D$ be a poset. A subset $N \subseteq D$ is *normal* in $D$, written as $N \triangleleft D$ if for every $x \in D$ the set $(\downarrow x) \cup N$ of elements below $x$ in $N$ is directed.

**Definition 3.34**. An algebraic cpo $D$ is *bifinite* if for any finite set $F \subseteq \mathsf{K}(D)$ there is a finite set $N$ such that $F \subseteq N \triangleleft \mathsf{K}(D)$.

**Theorem 3.8**. A bounded complete algebraic cpo is bifinite.

**Definition 3.35**. Let $D$ and $E$ be cpo's. A pair of continuous functions $\langle e, p\rangle$ such that $e : D \to E$ and $p : E \to D$ is called an *embedding-projection pair,* or *ep-pair,* if $p \circ e = id_D$ and $e \circ p \sqsubseteq id_E$.

**Definition 3.36**. Let $D$ and $E$ be cpo's. Let $\langle e_1, p_1\rangle$ and $\langle e_2, p_2\rangle$ be ep-pairs between $D$ and $E$. Then, the composition of $\langle e_1, p_1\rangle$ and $\langle e_2, p_2\rangle$ is the ep-pair $\langle e, p\rangle = \langle e_1 \circ e_2, p_2 \circ p_1\rangle$.

**Theorem 3.9**. Composition of ep-pairs is transitive and has as identity the ep-pair $\langle id, id\rangle$.

### 3.3.2 Domains

DEFINITION **Definition 3.37**. A *domain* is a bifinite cpo $D$ with a bottom element, written as $\bot$ and a top element, written as $\top$. For all elements $x \in D$, it must be $\bot \sqsubseteq x \sqsubseteq \top$.[5]

**Definition 3.38**. Every set $S$ defines a *flat domain* $S^{\mathrm{o}}$, whose underlying set is $S \cup \{\, \bot, \top \,\}$ and in which $x \sqsubseteq y$ iff $x = \bot$ or $y = \top$.

A number of useful domains can be defined at this point. The trivial domain $\mathbf{O}$ contains a single element $\bot$, which is both bottom and top. The flat domain that corresponds to the empty set is the domain $\mathbf{I}$ with elements $\bot$ and $\top$. A useful domain with a single ordinary element is $\mathbf{U} = \{\, u \,\}^{\mathrm{o}}$. The domain of truth values is defined as $\mathbf{T} = \{\, true, false \,\}^{\mathrm{o}}$. The domain $\mathbf{N}$ is defined as $\mathbf{N} = \mathsf{Z}^{\mathrm{o}}$, where $\mathsf{Z}$ is the set of integer numbers. The natural numbers also form a poset $\omega$ under their usual ordering $\leq$. It should be noted that $\omega$ is not a cpo nor a domain, since it does not have a top element.

**Definition 3.39**. If $D$ is a poset, an *$\omega$-chain* $(x_n)_{n\in\omega}$ in $D$ is a set of elements $x_n \in D$ such that $n \leq m$ implies $x_n \sqsubseteq x_m$.

**Theorem 3.10**. If $D$ is a poset with a bottom and a top element, the set of ideals over $D$ ordered by subset inclusion is a domain.

**Definition 3.40**. A function $f : D \to E$ between posets $D$ and $E$ is *monotone* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. FUNCTIONS

**Definition 3.41**. A function $f : D \to E$ between posets $D$ and $E$ is *continuous* if it is monotone and $f(\bigsqcup P) = \bigsqcup \{ f(x) \mid x \in P \}$ for all directed $P \subseteq D$.

**Definition 3.42**. A function $f : D \to E$ between domains $D$ and $E$ is *strict with respect to* $\bot$ if $f(\bot) = \bot$. It is *strict with respect to* $\top$ if $f(\top) = \top$. Finally, it is *strict* if it is strict with respect to both $\bot$ and $\top$. The notation $f : D \to_s E$ indicates that $f$ is strict.

**Definition 3.43**. A relation $\sqsubseteq$ can be defined for functions between domains $D$ and $E$ as follows. If $f, g : D \to E$, then $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in D$.

**Theorem 3.11**. The set of continuous functions between $D$ and $E$ under the relation defined in Definition 3.43 is a domain. This domain is denoted by $D \to E$.

**Definition 3.44**. An element $x \in D$ is a *fixed point* of a function $f : D \to D$ if $x = f(x)$.

**Theorem 3.12** (FIXED POINT). If $D$ is a domain and $f : D \to D$ is continuous, then $f$ has a least fixed point $\mathit{fix}(f) \in D$. That is $\mathit{fix}(f) = f(\mathit{fix}(f))$ and $\mathit{fix}(f) \sqsubseteq x$ for all $x$ such that $x = f(x)$. Furthermore:

$$\mathit{fix}(f) = \bigsqcup_{n=0}^{\infty} f^n(\bot)$$

**Definition 3.45**. Let $D$ be a domain, $x \in D$ be one of its elements and $f : D \to D$ be a continuous function. The *closure operator* can be defined in a way similar to the least fixed point operator, provided that $x \sqsubseteq f(x)$:

$$\mathit{clo}(x)(f) = \bigsqcup_{n=0}^{\infty} f^n(x)$$

It should be noted that $\mathit{clo}(x)(f)$ is also a fixed point of $f$, i.e. $f(\mathit{clo}(x)(f)) = \mathit{clo}(x)(f)$.

### 3.3.3 Domain constructions

**Definition 3.46**. If $D$ and $E$ are domains, then the *product* $D \times E$ is a domain. The elements of $D \times E$ are the pairs $\langle x, y \rangle$ with $x \in D$ and $y \in E$, and the ordering relation is defined as: PRODUCTS

$$\langle x_1, y_1 \rangle \sqsubseteq \langle x_2, y_2 \rangle \Leftrightarrow x_1 \sqsubseteq_D x_2 \wedge y_1 \sqsubseteq_E y_2$$

**Definition 3.47**. If $D \times E$ is a product domain, two continuous projection functions $\mathit{fst} : D \times E \to D$ and $\mathit{snd} : D \times E \to E$ can be defined as $\mathit{fst}\,\langle x, y \rangle = x$ and $\mathit{snd}\ \langle x, y \rangle = y$.

**Definition 3.48**. If $D$, $E$ and $F$ are domains and $f_1 : F \to D$ and $f_2 : F \to E$ are continuous functions, then a continuous function $\langle f_1, f_2 \rangle : F \to D \times E$ can be defined as $\langle f_1, f_2 \rangle\ x = \langle f_1\ x, f_2\ x \rangle$.

[5] Other approaches define domains as cpo's, or as *pointed* cpo's (i.e. cpo's with a bottom element).

**Theorem 3.13**. For any domains $D$, $E$ and $F$ and for any continuous functions $f_1 : F \to D$, $f_2 : F \to E$ and $g : F \to D \times E$ the following equations are satisfied:

- $fst \circ \langle f_1, f_2 \rangle = f_1$
- $snd \circ \langle f_1, f_2 \rangle = f_2$
- $\langle fst \circ g, snd \circ g \rangle = g$

**Definition 3.49**. A function $f : D \times E \to F$ is *bistrict with respect to* $\bot$ if $f\ \langle x, y \rangle = \bot$ whenever $x = \bot$ or $y = \bot$. It is *bistrict with respect to* $\top$ if $f\ \langle x, y \rangle = \top$ whenever $x = \top$ or $y = \top$. Finally, it is *bistrict* if it is first bistrict with respect to $\bot$ and then bistrict with respect to $\top$. That is, $f\ \langle \bot, \top \rangle = f\ \langle \top, \bot \rangle = \bot$.

SMASH PRODUCTS

**Definition 3.50**. If $D$ and $E$ are domains, then the *smash product* $D \otimes E$ is a domain. The elements of $D \otimes E$ are the pairs $\langle x, y \rangle$, with $x \in D - \{ \bot, \top \}$ and $y \in E - \{ \bot, \top \}$, plus two additional bottom and top elements $\bot_{D \otimes E}$ and $\top_{D \otimes E}$. The ordering relation is defined as in the case of Definition 3.46, with the additional relation that $\bot_{D \otimes E} \sqsubseteq \langle x, y \rangle \sqsubseteq \top_{D \otimes E}$ for all elements $\langle x, y \rangle$.

SEPARATED SUMS

**Definition 3.51**. If $D$ and $E$ are domains, then the *separated sum* $D + E$ is a domain. The set of elements of $D + E$ is:

$$\{ \langle x, 0 \rangle \mid x \in D \} \cup \{ \langle y, 1 \rangle \mid y \in E \} \cup \{ \bot_{D+E}, \top_{D+E} \}$$

The ordering relation is defined separately for each kind of pairs, i.e. $\langle x_1, 0 \rangle \sqsubseteq \langle x_2, 0 \rangle \Leftrightarrow x_1 \sqsubseteq_D x_2$ and $\langle y_1, 1 \rangle \sqsubseteq \langle y_2, 1 \rangle \Leftrightarrow y_1 \sqsubseteq_E y_2$. In addition, $\bot_{D+E} \sqsubseteq z \sqsubseteq \top_{D+E}$ for all $z \in D + E$.

**Definition 3.52**. If $D + E$ is a sum domain, two continuous injection functions $inl : D \to D + E$ and $inr : E \to D + E$ can be defined as $inl\ x = \langle x, 0 \rangle$ and $inr\ y = \langle y, 1 \rangle$.

**Definition 3.53**. If $D$, $E$ and $F$ are domains and $f_1 : D \to F$ and $f_2 : E \to F$ are continuous functions, then a continuous function $[\, f_1, f_2 \,] : D + E \to F$ can be defined as:

$$[\, f_1, f_2 \,]\ z = \begin{cases} \bot_F & , \ z = \bot_{D+E} \\ \top_F & , \ z = \top_{D+E} \\ f_1\ x & , \ z = \langle x, 0 \rangle \\ f_2\ y & , \ z = \langle y, 1 \rangle \end{cases}$$

**Theorem 3.14**. For any domains $D$, $E$ and $F$ and for any continuous functions $f_1 : F \to D$, $f_2 : F \to E$ and $g : F \to D \times E$ the following equations are satisfied:

- $[\, f_1, f_2 \,] \circ inl = f_1$
- $[\, f_1, f_2 \,] \circ inr = f_2$
- $[\, g \circ inl, g \circ inr \,] = g$

COALESCED SUMS

**Definition 3.54**. If $D$ and $E$ are domains, then the *coalesced sum* $D \oplus E$ is a domain. The set of elements of $D \oplus E$ is:

$$\{ \langle x, 0 \rangle \mid x \in D - \{ \bot, \top \} \} \cup \{ \langle y, 1 \rangle \mid y \in E - \{ \bot, \top \} \} \cup \{ \bot_{D \oplus E}, \top_{D \oplus E} \}$$

The ordering relation is defined as in Definition 3.46.

**Definition 3.55**. In the case of coalesced sums, the injection functions $inl : D \to D \oplus E$ and $inr : E \to D \oplus E$ are the strict versions of the functions used in separated sums.

LIFTED DOMAINS

**Definition 3.56**. If $D$ is a domain, the *lifted domain* $D_\perp$ is a domain whose elements are the pairs $\langle x, 0\rangle$ with $x \in D$, plus two additional bottom and top elements $\perp_{D_\perp}$ and $\top_{D_\perp}$. The ordering relation is defined as $\langle x, 0\rangle \sqsubseteq \langle y, 0\rangle \Leftrightarrow x \sqsubseteq_D y$ with the additional relation that $\perp_{D_\perp} \sqsubseteq \langle x, 0\rangle \sqsubseteq \top_{D_\perp}$ for all elements $\langle x, 0\rangle$.

**Definition 3.57**. If $D_\perp$ is a lifted domain, two continuous functions $up : D \to D_\perp$ and $down : D_\perp \to D$ can be defined as follows:

$$up\ x = \langle x, 0\rangle$$

$$down\ z = \begin{cases} \perp_D\ , & z = \perp_{D_\perp} \\ \top_D\ , & z = \top_{D_\perp} \\ x\quad , & z = \langle x, 0\rangle \end{cases}$$

**Theorem 3.15**. The following properties are satisfied for a lifted domain $D_\perp$:

- $down \circ up = id_D$
- $id_{D_\perp} \sqsubseteq up \circ down$

DEPENDENT FUNCTIONS AND PRODUCTS

Let $D$ be a domain and $F$ be a domain constructor such that $F(x)$ is a domain for all $x \in D$. Then, a *dependent function* is a function $f$ defined on elements of $D$, such that $f(x) \in F(x)$ for all elements $x \in D$. The domain of dependent functions is written as $x : D \twoheadrightarrow F(x)$. Also, a *dependent product* is a product $\langle x, y\rangle$ with $x \in D$ and $y \in F(x)$. The domain of dependent products is written as $x : D \ltimes F(x)$. For a formal definition of dependent functions and products and their properties, the reader is referred to [Gunt92].

### 3.3.4 Categorical properties of domains

**Definition 3.58**. Every poset $D$ induces a category. The objects of this category are the elements of $D$. Also, for every pair of elements $x, y \in$ such that $x \sqsubseteq y$ the induced category contains a unique arrow from $x$ to $y$. For every pair of elements that are not related by $\sqsubseteq$, there are no arrows in the induced category.

**Theorem 3.16**. Domains and continuous functions form a category $\mathsf{Dom}$.

**Theorem 3.17**. Domains and ep-pairs form a category $\mathsf{Dom}^{ep}$.

**Theorem 3.18**. The domain $\mathbf{O}$ is an initial object of the category $\mathsf{Dom}^{ep}$.

**Theorem 3.19**. Domain products and coalesced sums are products and sums in the categorical sense, in category $\mathsf{Dom}$.

**Definition 3.59**. A *continuous semi-lattice domain* is a domain $D$ together with a continuous binary function $* : D \times D \to D$ which is associative, commutative and idempotent.

**Definition 3.60**. A *homomorphism* between continuous semi-lattice domains $D$ and $E$ is a continuous function $f : D \to E$ such that $f(s *_D t) = f(s) *_E f(t)$ for all $s, t \in D$.

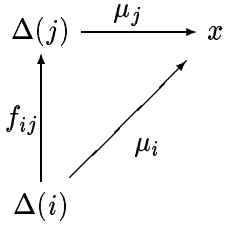
**Theorem 3.20**. Continuous semi-lattice domains and continuous homomorphisms form a category $\mathsf{Sld}$.

**Definition 3.61**. The *forgetful functor* $U : \mathsf{Sld} \to \mathsf{Dom}$ is defined by simply taking the underlying domain of a continuous semi-lattice domain. It has the identity action on arrows.

### 3.3.5 Diagrams, cones and colimits

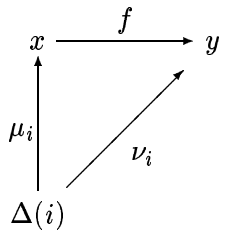
**Definition 3.62**. Let $I$ be a poset and $\mathsf{C}$ a category. A *diagram indexed by $I$ over* $\mathsf{C}$ is a functor $\Delta : I \to \mathsf{C}$.

**Definition 3.63**. A *cone* over a diagram $\Delta : I \to \mathsf{C}$ is an object $x$ in $\mathsf{C}$ together with a family of arrows $\mu_i : \Delta(i) \to x$ indexed by the elements of $I$, such that for all $i \sqsubseteq j$ in $I$ the following diagram commutes:




where $f_{ij}$ is the arrow in $\mathsf{C}$ which is the image mapped by $\Delta$ of the unique arrow in category $I$ corresponding to $i \sqsubseteq j$. The aforementioned cone is denoted by $\mu : \Delta \to x$.

**Definition 3.64**. If $\mu : \Delta \to x$ and $\nu : \Delta \to y$ are cones over a diagram $\Delta : I \to \mathsf{C}$, then arrow $f : x \to y$ is a *mediating arrow* if for all $i \in I$ the following diagram commutes:




The aforementioned mediating arrow is also denoted as $f : \mu \to \nu$.

**Definition 3.65**. A cone $\mu : \Delta \to x$ is *colimiting* if for any other cone $\nu : \Delta \to y$ there is a unique mediating arrow $f : \mu \to \nu$. In this case $x$ is a *colimit* of $\Delta$.

**Theorem 3.21**. The colimit of a diagram, if it exists, is unique up to isomorphism.

**Definition 3.66**. A functor $F : \mathsf{C} \to \mathsf{D}$ is *continuous* if for all directed diagrams $\Delta$ and for all colimiting cones over $\Delta$ the cone $F(\mu)$ is colimiting over $F(\Delta)$.[6]

The directed poset $\omega$ is commonly used for indexing diagrams. In this case, a diagram $\Delta : \omega \to \mathsf{C}$ can be viewed as the categorical analogue of an $\omega$-chain, that is, a family $\Delta = (\Delta_n, f_n)_{n \in \omega}$ where, for all $n \in \omega$, $\Delta_n$ is an object in $\mathsf{C}$ and $f_n : \Delta_n \to \Delta_{n+1}$ is an arrow in $\mathsf{C}$. The arrows corresponding to $i \leq g$ in $\omega$ can be then defined as the composition of $f_{j-1} \circ \ldots \circ f_i$. Given such a diagram $\Delta$, a diagram $\Delta^- = (\Delta_n^-, f_n^-)_{n \in \omega}$ can be defined by taking $\Delta_n^- = \Delta_{n+1}$ and $f_n^- = f_{n+1}$, for all $n \in \omega$. Also, given a diagram $\Delta : \omega \to \mathsf{C}$ and a cone $\mu : \Delta \to x$, the cone $\mu^- : \Delta^- \to \mathsf{C}$ can be defined by taking $\mu_n^- = \mu_{n+1}$, for all $n \in \omega$.

**Theorem 3.22**. Let $\mathsf{C}$ be a category with an initial object 0 and $F : \mathsf{C} \to \mathsf{C}$ be an endofunctor on $\mathsf{C}$. Let $i : 0 \to F(0)$ be the unique map from the initial object 0 to $F(0)$ and $\Delta : \omega \to C$ be the $\omega$-chain $(F^n(0), F^n(i))_{n \in \omega}$. If there is a colimiting cone $\mu : \Delta \to x$ and the cone $F(\mu) : F(\Delta) \to F(x)$ is also colimiting, then the mediating arrow $a : F(x) \to x$ between $F(\mu)$ and $\mu^-$ is an initial $F$-algebra.

[6] $F(\Delta)$ is defined as the composition $F \circ \Delta$ of functors $F$ and $\Delta$.

### 3.3.6 Powerdomains

**Definition 3.67**. If $S$ is a set, the set of finite non-empty subsets of $S$ is denoted by $\mathcal{P}_f^*(S)$. DEFINITION

**Definition 3.68**. If $D$ is a poset, a binary relation $\sqsubseteq^\natural$ can be defined on $\mathcal{P}_f^*(D)$ as follows:

$$u \sqsubseteq^\natural v \;\Leftrightarrow\; (\forall\, y \in v.\, \exists\, x \in u.\, x \sqsubseteq y) \;\wedge\; (\forall x \in u.\, \exists\, y \in v.\, x \sqsubseteq y)$$

**Theorem 3.23**. According to the ordering of Definition 3.68, the set $\mathcal{P}_f^*(D)$ is a poset. If $D$ has a bottom or top element, then also $\mathcal{P}_f^*(D)$ has a bottom or top element.

**Definition 3.69**. If $D$ is a domain, then the (convex) *powerdomain* of $D$, written as $D^\natural$, is the domain of ideals over the poset $\mathcal{P}_f^*(\mathsf{K}(D))$ ordered by $\sqsubseteq^\natural$.

**Theorem 3.24**. Let $D$ be a domain and $s, t \in D$. The following properties are satisfied:

- $s \sqsubseteq_{D^\natural} t \;\Leftrightarrow\; s \subseteq t$
- $\bot_{D^\natural} = \{\{\bot_D\}\}$
- $\top_{D^\natural} = \mathcal{P}_f^*(\mathsf{K}(D))$

**Definition 3.70**. Let $D$ and $E$ be domains. If $f : D \to E$ is a continuous function, then the function $f^\natural : D^\natural \to E^\natural$ defined below is also continuous:

$$f^\natural\, s \;=\; \{\, w \in \mathcal{P}_f^*(\mathsf{K}(D)) \;\mid\; \exists\, u \in s.\, w \sqsubseteq^\natural \{\, f\, x \;\mid\; x \in u \,\} \,\}$$

**Definition 3.71**. Let $D$ be a domain. Then, the *powerdomain singleton* is a continuous function $\{\!|\cdot|\!\} : D \to D^\natural$ defined as follows for all $x \in D$:

$$\{\!|x|\!\} \;=\; \{\, w \in \mathcal{P}_f^*(\mathsf{K}(D)) \;\mid\; \exists\, a \in \mathsf{K}(D).\, a \sqsubseteq x \wedge w \sqsubseteq^\natural \{\, a \,\} \,\}$$

**Theorem 3.25**. Let $D$ be a domain and $x, y \in D$. The following properties hold:

- $\{\!|\bot_D|\!\} \;=\; \bot_{D^\natural}$
- $\{\!|\top_D|\!\} \;=\; \top_{D^\natural}$
- $x = y \;\Leftrightarrow\; \{\!|x|\!\} = \{\!|y|\!\}$

**Definition 3.72**. Let $D$ be a domain. Then the *powerdomain union* is a continuous binary operation $\uplus^\natural : D^\natural \times D^\natural \to D^\natural$ defined as follows, for all $s, t \in D^\natural$:

$$s \uplus^\natural t \;=\; \{\, w \in \mathcal{P}_f^*(\mathsf{K}(D)) \;\mid\; \exists\, u \in s, v \in t.\, w \sqsubseteq^\natural u \cup v \,\}$$

**Theorem 3.26**. Let $D$ and $E$ be domains, $r, s, t \in D^\natural$ and $f : D \to E$ be a continuous function. The following properties hold:

- $s \uplus^\natural s \;=\; s$
- $s \uplus^\natural t \;=\; t \uplus^\natural s$
- $r \uplus^\natural (s \uplus^\natural t) \;=\; (r \uplus^\natural s) \uplus^\natural t$
- $f^\natural\, (s \uplus^\natural t) \;=\; (f^\natural\, s) \uplus^\natural (f^\natural\, t)$

**Theorem 3.27**. Let $D$ be a domain. Then, the domain $D^\natural$ together with the binary operation $\uplus^\natural : D^\natural \times D^\natural \to D^\natural$ form a continuous semi-lattice domain.

**Definition 3.73**. The *powerdomain functor* $P : \mathsf{Dom} \to \mathsf{Sld}$ is defined by taking $P(D)$ to be the continuous semi-lattice domain formed by $D^\natural$ and $\uplus^\natural$, for every object $D \in \mathsf{Dom}$, and $P(f) = f^\natural$ for every function $f : D \to E$ in $\mathsf{Dom}$.

**Theorem 3.28**. The powerdomain singleton is a natural transformation between the identity functor on $\mathsf{Dom}$ and the functor $U \circ P$, that is $\{\!|\ \cdot\ |\!\} : id_{\mathsf{Dom}} \dot{\to} U \circ P$.

**Theorem 3.29**. The powerdomain functor $P$ is left adjoint to the forgetful functor $U : \mathsf{Sld} \to \mathsf{Dom}$ with the powerdomain singleton as the unit of the adjunction.

**Definition 3.74**. Let $D$ be a domain and let $* : D \times D \to D$ be a binary operation on $D$ that is associative, commutative and idempotent. A continuous function $\epsilon_D : D^\natural \to D$ can be defined as follows, for all $s \in D^\natural$:

$$\epsilon_D\ s\ =\ \bigsqcup \{\, w^*\ \mid\ w \in s\,\}$$

where $w^*$ is defined as the result of the application of operation $*$ to all the elements of the finite non-empty set $w$, for all $w \in \mathcal{P}^*_f(\mathsf{K}(D))$.

**Theorem 3.30**. For all domains $D$, function $\epsilon_D : D^\natural \to D$ specified in Definition 3.74 is a continuous homomorphism between the continuous semi-lattice domains $\langle D, * \rangle$ and $\langle D^\natural,\ \uplus^\natural\ \rangle$.

**Theorem 3.31**. The family of continous functions $\epsilon_D$ specified in Definition 3.74 defines a natural transformation between the functor $P \circ U$ and the identity functor on $\mathsf{Sld}$, that is $\epsilon : P \circ U \dot{\to} id_{\mathsf{Sld}}$.

**Theorem 3.32**. The natural transformation $\epsilon : P \circ U \dot{\to} id_{\mathsf{Sld}}$ specified in Theorem 3.31 is the counit of the adjuction specified in Theorem 3.29.

POWERDOMAIN MONAD

As a result of Theorem 3.7, the adjunction $\langle P, U, \{\!|\ \cdot\ |\!\} \rangle$ induces a monad $\mathsf{P}$ on $\mathsf{Dom}$. The elements of this monad are:

- The functor $\mathsf{P} : \mathsf{Dom} \to \mathsf{Dom}$ defined by $\mathsf{P}(D) = D^\natural$, for all domains $D$, and $\mathsf{P}(f) = f^\natural$, for all continuous functions $f : D \to E$.
- The unit $\eta : id_{\mathsf{Dom}} \dot{\to} \mathsf{P}$ defined by $\eta_D\ x = \{\!|x|\!\}$ for all domains $D$ and all elements $x \in D$.
- The join $\mu : \mathsf{P}^2 \dot{\to} \mathsf{P}$ defined by $\mu_D\ X = \bigcup \{\, W\ ^{\uplus^\natural}\ \mid\ W \in X\,\}$ for all domains $D$ and all elements $X : D^{\natural\natural}$.

Two things should be noted about the monad's join. First, set union is used here as the least upper bound operator with respect to the subset inclusion relation $\sqsubseteq_{D^\natural}$. Second, the join is equivalent to the big union operator for the convex powerdomain, as defined in [Plot76].

The powerdomain monad can be also used in the spirit of Section 3.2.2. In this case, to facilitate the definition of the bind operator, the following definition is used.

**Definition 3.75**. Let $D$ and $E$ be domains and $f : D \to E^\natural$ be a continuous function. Then, function $ext^\natural\ f : D^\natural \to E^\natural$ is defined as follows for all $s \in D^\natural$:

$$ext^\natural\ f\ s\ =\ \mu_D\ (f^\natural\ s)$$

**Definition 3.76**. If $D$ and $E$ are domains, the bind operator $\cdot *_{\mathsf{P}} \cdot : \mathsf{P}(D) \to (D \to \mathsf{P}(E)) \to \mathsf{P}(E)$ is defined as follows, for all continuous functions $f : D \to \mathsf{P}(E)$ and for all $s \in \mathsf{P}(D)$:

$$s\ *_{\mathsf{P}}\ f\ =\ ext^\natural\ f\ s$$

## 3.4 The meta-language

In order to formulate specific elements of domains in denotational descriptions, a meta-language has to be employed. Although researchers have not agreed on a standard meta-language for this purpose and there is considerable variation in the notational conventions used by various authors, it seems that variations of the $\lambda$-calculus are very popular. Such a variation will be used in this thesis, following mostly the notation of [Moss90] and [Gunt90]. In this section, the meta-language is outlined, keeping in mind the definitions of Section 3.3. The core of the meta-language is first presented, followed by additional notational conventions that add nothing substantial to the language and can be regarded as syntactic sugar.

### 3.4.1 Core meta-language

Every well-formed phrase $e$ of the meta-language denotes an element $x$ of some domain $D$. The notation $e \triangleright x : D$ is used in the rest of this section as an abbreviation of the previous sentence. The well-formedness of a phrase depends generally on the well-formedness of its components and on a number of additional restrictions. In the rules to follow, letters $e$ and $I$ represent respectively phrases and identifiers of the meta-language, the letter $D$ represents domains and the letters $x, y, f, g$ represent elements of domains. In particular, the last two represent continuous functions.

**Basic notation:** Named elements of domains, (e.g. $\bot$, $\top$, integer numbers, etc.) are used to represent themselves. Parentheses can be used to group phrases of the meta-language.

**Equality:** The equality operator $= : D \times D \to \mathbf{T}$ defines a binary relation. Its continuity follows from its restriction to the elements $true$ and $false$ of $\mathbf{T}$. If $e_1 \triangleright x : D$ and $e_2 \triangleright y : D$, then:

$$\begin{array}{lll} e_1 = e_2 \ \triangleright\ true : \mathbf{T} & , & \text{if} \quad x \text{ and } y \text{ are the same element of } D \\ e_1 = e_2 \ \triangleright\ false : \mathbf{T} & , & \text{otherwise} \end{array}$$

**Conditional:** If $e_1 \triangleright x : D$ and $e_2 \triangleright y : D$, then:

$$\begin{array}{lll} e \to e_1,\ e_2 \ \triangleright\ x : D & , & \text{if} \quad e \triangleright true : \mathbf{T} \\ e \to e_1,\ e_2 \ \triangleright\ y : D & , & \text{if} \quad e \triangleright false : \mathbf{T} \\ e \to e_1,\ e_2 \ \triangleright\ \bot : D & , & \text{if} \quad e \triangleright \bot : \mathbf{T} \\ e \to e_1,\ e_2 \ \triangleright\ \top : D & , & \text{if} \quad e \triangleright \top : \mathbf{T} \end{array}$$

**Function abstraction:** The $\lambda$-notation $\lambda\, I.\ e$ is used to represent function abstractions. The phrase $e$ may contain instances of the identifier $I$. Let $f : D_1 \to D_2$ be a continuous function. If for all $x \in D_1$, by assuming that $I \triangleright x : D_1$ it is possible to deduce that $e \triangleright f(x) : D_2$, then:

$$\lambda\, I.\ e \ \triangleright\ f : D_1 \to D_2$$

**Function application:** If $e_1 \triangleright f : D_1 \to D_2$ and $e_2 \triangleright x : D_1$, then:

$$e_1\ e_2 \ \triangleright\ f(x) : D_2$$

**Function composition:** If $e_1 \triangleright f : D_1 \to D_2$ and $e_2 \triangleright g : D_2 \to D_3$ then:

$$e_2 \circ e_1 \ \triangleright\ g \circ f : D_1 \to D_3$$

**Least fixed point and closure operators:** If $e \triangleright f : D \to D$ and $e' \triangleright x : D$ then:

$$\mathit{fix}\ e \ \triangleright\ \mathit{fix}(f) : D$$
$$\mathit{clo}\ e'\ e \ \triangleright\ \mathit{clo}(x)(f) : D$$

**Domain constructions:** The notation used in Section 3.3.3 for representing the elements of various domain constructions is extended to phrases of the meta-language. In particular, special notation is used for products, sums, lifted domains and powerdomains.

**Monad notation:** The meta-language is further extended by using the notation defined in Section 3.2.2 for the representation of monads, monad morphisms, monad transformers and operations on them.

### 3.4.2 Syntactic sugar

**Unary and binary operators:** Functions may be written as prefix unary or infix binary operators, if this is considered appropriate.

**Abbreviations for products and sums:** Since the distinction between ordinary and smash products, as well as separated and coalesced sums, is usually clear from the context, it is possible to use the same notation for both. The injection functions $\mathit{inl}$ and $\mathit{inr}$ for separated and coalesced sums may be omitted whenever they can be deduced from the context. Also, $(e \in D)$ may be used for determining whether the phrase $e$ denotes an element of $D$ in a sum domain and, provided that this is the case, $(e \mid D)$ denotes the corresponding element of $D$. Both can be defined in terms of the core notation for sums.

**Indirect domain definition:** Flat domains may be defined by enumerating their ordinary elements. For example, the following definitions are equivalent:

$$\mathbf{T} = \mathit{true} \mid \mathit{false}$$
$$\mathbf{T} = \{\, \mathit{true}, \mathit{false} \,\}^{\mathrm{o}}$$

The same sort of abbreviation may be applied to coalesced sum domains. If it is assumed that $n \in \mathbf{N}$ and $t \in \mathbf{T}$, the following definitions are equivalent:

$$D = n \mid t$$
$$D = N \oplus T$$

Furthermore, the summands may be given descriptive names as illustrated in the following example:

$$D = \mathit{integer}[n] \mid \mathit{truth}[t]$$

**Let structure:** In its primary form, the *let* structure can be defined by means of function abstraction and application. The following equivalence holds:

$$\mathbf{let}\ I = e_1\ \mathbf{in}\ e_2 \equiv (\lambda\, I.\ e_2)\ e_1$$

It should be made clear that $I$ is only bound in $e_2$ and not in $e_1$. Therefore the *let* structure is not recursive. More than one binding clauses can be given in the same *let* structure, as stated by the following equivalence:

$$\left(\begin{array}{ll} \mathbf{let}\ I_1 & = e_1 \\ \quad I_2 & = e_2 \\ \quad \ldots & \\ \quad I_n & = e_n \\ \mathbf{in}\ e & \end{array}\right) \equiv \mathbf{let}\ I_1 = e_1\ \mathbf{in}\ (\mathbf{let}\ I_2 = e_2\ \mathbf{in}\ \ldots (\mathbf{let}\ I_n = e_n\ \mathbf{in}\ e)\ldots)$$

**Case structure:** The *case* structure in its primary form is used as syntactic sugar in order to distinguish between the summands of separated or coalesced sums. It can be defined by means of the core notation for sums. As a complex example, consider the following equivalent definitions:

$$D = \mathit{men}[n] \mid \mathit{women}[n] \mid \mathit{delay}[t]$$
$$D = \mathbf{N} \oplus (\mathbf{N} \oplus \mathbf{T})$$

Then, the following equivalence holds:

$$\left(\begin{array}{lll} \mathbf{case}\ e\ \mathbf{of} & & \\ \quad \mathit{men}[n] & \Rightarrow & e_1 \\ \quad \mathit{women}[n] & \Rightarrow & e_2 \\ \quad \mathit{delay}[t] & \Rightarrow & e_3 \end{array}\right) \equiv [\,\lambda\, n.\ e_1, [\,\lambda\, n.\ e_2, \lambda\, t.\ e_3\,]\,]\ e$$

Furthermore, a special *otherwise* clause can be used in order to uniformly treat all cases that have not been enumerated.

**Pattern matching:** The use of the *case* structure that was illustrated above introduces a way of binding identifiers in phrases of the meta-language that is similar to pattern matching. It applies to products, sums, bottom and top elements and is intuitively equivalent to the pattern matching supported by the functional programming language Haskell. On condition that the pattern can always be matched, the same notation can also be used in conjunction with the *let* construct. Pattern matching can be defined by means of the conditional construct and core notation for products and sums.

### 3.4.3 Auxiliary functions

Several polymorphic auxiliary functions are defined in order to facilitate the definition of the semantics. Although most of these correspond directly to categorical or domain theoretic mathematical objects, the following definitions use the defined meta-language. Also, various unary and binary operators on domains $\mathbf{T}$ (e.g. $\wedge$, $\vee$ and $\neg$) and $\mathbf{N}$ (e.g. +, -, $\cdot$) are used; their definitions are omitted. It should be noted that the same notation is used for separated and coalesced sums.

IDENTITY FUNCTION

▶ $id \;:\; A \to A$
$id \;=\; \lambda a.\, a$

FUNCTION UPDATE

▶ $\cdot\{\cdot \mapsto \cdot\} \;:\; (A \to B) \times A \times B \to A \to B$
$f\{a \mapsto b\} \;=\; \lambda x.\, (x = a) \to b\,,\; f\, x$

STRICTNESS

▶ $strict_{\bot} \;:\; (A \to B) \to A \to B$
$strict_{\bot} \;=\; \lambda f.\, \lambda a.\, (a = \bot_A) \to \bot_B\,,\; f\, a$

▶ $strict_{\top} \;:\; (A \to B) \to A \to B$
$strict_{\top} \;=\; \lambda f.\, \lambda a.\, (a = \top_A) \to \top_B\,,\; f\, a$

▶ $strict_{\mathrm{I}} \;:\; (A \to B) \to A \to B$
$strict_{\mathrm{I}} \;=\; strict_{\bot} \circ strict_{\top}$

BI-STRICTNESS

▶ $bi\text{-}strict_{\bot} \;:\; (A \times B \to C) \to A \times B \to C$
$bi\text{-}strict_{\bot} \;=\; \lambda f.\, \lambda \langle a, b\rangle.\, (a = \bot_A) \vee (b = \bot_B) \to \bot_C\,,\; f\, \langle a, b\rangle$

▶ $bi\text{-}strict_{\top} \;:\; (A \times B \to C) \to A \times B \to C$
$bi\text{-}strict_{\top} \;=\; \lambda f.\, \lambda \langle a, b\rangle.\, (a = \top_A) \vee (b = \top_B) \to \top_C\,,\; f\, \langle a, b\rangle$

▶ $bi\text{-}strict_{\mathrm{I}} \;:\; (A \times B \to C) \to A \times B \to C$
$bi\text{-}strict_{\mathrm{I}} \;=\; bi\text{-}strict_{\bot} \circ bi\text{-}strict_{\top}$

SUM OPERATIONS

▶ $isl \;:\; A + B \to \mathbf{T}$
$isl \;=\; [\lambda a.\, true, \lambda b.\, false\,]$

▶ $isr \;:\; A + B \to \mathbf{T}$
$isr \;=\; [\lambda a.\, true, \lambda b.\, false\,]$

▶ $outl \;:\; A + B \to A$
$outl \;=\; [id, \top\,]$

▶ $outr \;:\; A + B \to B$
$outr \;=\; [\top, id\,]$

MONAD FUNCTION COMPOSITION

▶ $\cdot\,;_{\mathsf{M}}\,\cdot \;:\; (A \to M(B)) \times (B \to M(C)) \to (A \to M(C))$
$f\,;_{\mathsf{M}}\, g \;=\; \lambda a.\, fa\; *_{\mathsf{M}}\; g$

MONAD FIXED POINTS

▶ $mfix_{\mathsf{M}} \;:\; (A \to M(A)) \to M(A)$
$mfix_{\mathsf{M}} \;=\; \lambda f.\, fix\, (\lambda x.\, x\; *_{\mathsf{M}}\; f)$

▶ $mclo_{\mathsf{M}} \;:\; M(A) \to (A \to M(A)) \to M(A)$
$mclo_{\mathsf{M}} \;=\; \lambda z.\, \lambda f.\, clo\, z\, (\lambda x.\, x\; *_{\mathsf{M}}\; f)$

# Part II

# Static semantics

# Chapter 4

# Static semantic domains

CHAPTER OVERVIEW

This chapter defines the domains which are used in order to describe the static semantics of C. Each domain is defined together with the basic operations that are allowed on its elements. Section 4.1 discusses the structure of static semantic domains. In Section 4.2 a small set of auxiliary domains is introduced. Section 4.3 defines the static semantic domains that represent C's types. In Section 4.4 a simple error monad is introduced, which is later used for simplifying the description of static semantics. Section 4.5 defines domains that represent environments, which most commonly are variations of mappings from identifiers to types. Finally, in Section 4.6 a large number of auxiliary functions is defined. These functions are not related to particular domains, however they are used to simplify the semantic equations that are given in the following chapters.

## 4.1 Domain ordering

It should be noted that for all domains used in specifying the static semantics of C, the domain ordering relation $\sqsubseteq$ is easily defined since these domains are either coalesced sums or given by domain equations, recursive or not. This ordering is crucial in the treatment of incomplete types, as is further discussed in Chapter 6. Whenever $x$ and $y$ are different elements of a static semantic domain, the relation $x \sqsubseteq y$ denotes that $y$ is a better approximation of the incomplete element $x$.

As a typical example, a structure that has been just declared is associated with type $\mathit{struct}\,[t, \bot]$, where $t$ is its tag. When the same structure is later completed, its type will become $\mathit{struct}\,[t, \pi]$, for some member environment $\pi$. Since $\bot \sqsubseteq \pi$, we deduce that:

$$\mathit{struct}\,[t, \bot] \sqsubseteq \mathit{struct}\,[t, \pi]$$

and therefore the completed type is indeed a better approximation of the previous incomplete type.

The top element of static domains is mainly used to signify an "over-completed" value, which typically denotes something unspecified, abnormal or erroneous. For many static domains, the relation $\sqsubseteq$ is more or less trivial, or even degenerates to the equality relation. Also, for many domains, bottom and top elements do not denote anything important and are not used at all.

## 4.2 Auxiliary domains

■ $I$ : **Ide** *(undefined)* IDENTIFIERS

The domain **Ide** is a flat domain whose elements represent all legal identifiers, as defined in the standard. It can be considered both as a semantic domain and a syntactic domain, since identifiers also participate in the language syntax. A complete definition is omitted at this point.

TAG TYPES ■ $\sigma$ : $\mathbf{TagType} = \mathit{tag\text{-}struct} \mid \mathit{tag\text{-}union} \mid \mathit{tag\text{-}enum}$

The flat domain $\mathbf{TagType}$ is used to distinguish between the three types of tags that are allowed by the C language. Its three ordinary elements correspond respectively to structure tags, union tags and enumeration tags.

TAGS ■ $t$ : $\mathbf{Tag} = \mathit{tagged}\,[I, n] \mid \mathit{untagged}\,[n]$

The domain $\mathbf{Tag}$ is used to distinguish between different structure, union and enumeration types. Each such type is characterized by an element of this domain. Ordinary elements of this domain belong to two categories. Elements of the form $\mathit{tagged}\,[I, n]$ characterize types that have been given a distinguishing tag in the program, namely $I$, whereas elements of the form $\mathit{untagged}\,[n]$ characterize tagless structure, union or enumeration types. The integer number $n$ in both cases is used to uniquely determine different types.

## 4.3 Types

The domains defined in this section represent various kinds of types that are used, explicitly or implicitly, by the type system of C. Some of these kinds correspond directly to the type classification, as defined in §6.1.2.5 of the standard. Others are only introduced to facilitate the type system's formalization.

DATA TYPES ■ $\tau$ : $\mathbf{Type}_{dat} = \mathit{void} \mid \mathit{char} \mid \mathit{signed\text{-}char} \mid \mathit{unsigned\text{-}char}$
$\mid \mathit{short\text{-}int} \mid \mathit{unsigned\text{-}short\text{-}int} \mid \mathit{int} \mid \mathit{unsigned\text{-}int}$
$\mid \mathit{long\text{-}int} \mid \mathit{unsigned\text{-}long\text{-}int} \mid \mathit{float} \mid \mathit{double} \mid \mathit{long\text{-}double}$
$\mid \mathit{ptr}\,[\phi] \mid \mathit{enum}\,[\epsilon] \mid \mathit{struct}\,[t, \pi] \mid \mathit{union}\,[t, \pi]$

Data types, represented by the domain $\mathbf{Type}_{dat}$, provide the basis of the type system. They correspond to the different primary types of data that a C program can manipulate as first class elements, i.e. data that can be used in expressions, assigned to variables, passed as function parameters and returned as function results. Data types include all scalar types, structures, unions and the *void* type. The latter is only included to reduce the type system's complexity; the standard explicitly states that there can be no data elements of this type.

PREDEFINED DATA TYPES

Apart from the basic types, the standard defines the following type synonyms $\mathit{size\_t}$, $\mathit{wchar\_t}$ and $\mathit{ptrdiff\_t}$. These stand for specific data types and are implementation-defined. They are treated as ordinary elements of $\mathbf{Type}_{dat}$, satisfying the following requirements:

- $\mathit{size\_t}$ is an unsigned integral type,
- $\mathit{wchar\_t}$ is an integral type, and
- $\mathit{ptrdiff\_t}$ is an signed integral type.

QUALIFIERS ■ $q$ : $\mathbf{Qual} = \mathit{noqual} \mid \mathit{const} \mid \mathit{volatile} \mid \mathit{const\text{-}volatile}$

Qualifiers are represented by the flat domain $\mathbf{Qual}$. The standard supports two qualifiers, *const* and *volatile*, resulting in four different combinations. It should be noted here that, for the sake of generality, an unqualified type is considered as qualified with $\mathit{noqual}$.

■ $\alpha \;:\; \mathbf{Type}_{obj} \;=\; obj\,[\tau, q] \mid array\,[\alpha, n]$ OBJECT TYPES

Object types, represented by the domain $\mathbf{Type}_{obj}$, are associated with *objects* and comprise of all qualified versions of data types and array types. Array types are not qualified; their elements are.

■ $f \;:\; \mathbf{Type}_{fun} \;=\; func\,[\tau, p]$ FUNCTION TYPES

Function types are represented by the domain $\mathbf{Type}_{fun}$. They are characterized by the data type of the result and the types of their parameters, as specified in a function prototype. It should be noted that function types cannot be qualified, nor can the types of their results. It is probably a controversial issue whether the standard allows any of these and, if it does, what this means.

■ $\phi \;:\; \mathbf{Type}_{den} \;=\; \alpha \mid f$ DENOTABLE TYPES

Denotable types, represented by the domain $\mathbf{Type}_{den}$, are object types and function types. These are the types that can be associated with identifiers in environments.

■ $m \;:\; \mathbf{Type}_{mem} \;=\; \alpha \mid bitfield\,[\beta, q, n]$ MEMBER TYPES

Member types, represented by the domain $\mathbf{Type}_{mem}$, are the ones that can be associated with identifiers in member environments, i.e. with members of structures or unions. Member types include object types and bit-fields. The latter can be qualified and are characterized by their bit-field type and their length in bits.

■ $\beta \;:\; \mathbf{Type}_{bit} \;=\; int \mid signed\text{-}int \mid unsigned\text{-}int$ BIT-FIELD TYPES

Bit-field types are the integer types that can be associated with bitfields. They are represented by the flat domain $\mathbf{Type}_{bit}$. It should be noted that, in this context, the three types $int$, $signed\text{-}int$ and $unsigned\text{-}int$ are all different from each other.

■ $v \;:\; \mathbf{Type}_{val} \;=\; \tau \mid f$ VALUE TYPES

Value types, represented by the domain $\mathbf{Type}_{val}$, are associated with values that participate in expressions. Such values are often called r-values in literature. Value types consist of data types and function types. Qualifiers are not allowed in values.

■ $\delta \;:\; \mathbf{Type}_{ide} \;=\; normal\,[\phi] \mid typedef\,[\phi] \mid enum\text{-}const\,[n]$ IDENTIFIER TYPES

Identifier types are represented by the domain $\mathbf{Type}_{ide}$. They are associated with identifiers in environments, taking into consideration whether an identifier denotes an object of some particular type, a type synonym defined by *typedef*, or an enumeration constant.

■ $\theta \;:\; \mathbf{Type}_{phr} \;=\; exp\,[v] \mid lvalue\,[m] \mid val\,[\tau] \mid arg\,[p] \mid stmt\,[\tau]$
$\quad\quad \mid\; tunit \mid xdecl \mid decl \mid prot\,[p] \mid par\,[\tau] \mid idtor$
$\quad\quad \mid\; dtor\,[\phi] \mid init\,[m] \mid init\text{-}a\,[\alpha] \mid init\text{-}s\,[\pi] \mid init\text{-}u\,[\pi]$ PHRASE TYPES

Phrase types are represented by the domain $\mathbf{Type}_{phr}$. These types are associated with phrases, i.e. entire program segments, and are described in more detail in Table 4.1. The first four are associated with expressions, the fifth with statements and the rest with declarations and initializations. They are not used until Part III.

**Table 4.1:** Description of phrase types.

| | |
|---|---|
| $exp\,[v]$ | expression, whose result is a non-constant r-value of type $v$ |
| $lvalue\,[m]$ | expression, whose result is an l-value of type $m$ |
| $val\,[\tau]$ | expression, whose result is a constant r-value of type $\tau$ |
| $arg\,[p]$ | actual arguments of a function with prototype $p$ |
| $stmt\,[\tau]$ | statement in a function returning a result of type $\tau$ |
| $tunit$ | translation unit |
| $xdecl$ | external declaration |
| $decl$ | declaration |
| $prot\,[p]$ | description of a function prototype specified by $p$ |
| $par\,[\tau]$ | description of parameter of type $\tau$ |
| $idtor$ | declarator with initializer |
| $dtor\,[\phi]$ | declarator for identifier of type $\phi$ |
| $init\,[m]$ | initializer for a member of type $m$ |
| $init\text{-}a\,[\alpha]$ | initializer for an array object with elements of type $\alpha$ |
| $init\text{-}s\,[\pi]$ | initializer for a structure object with members specified in $\pi$ |
| $init\text{-}u\,[\pi]$ | initializer for a union object with members specified in $\pi$ |

## 4.4 The error monad

MOTIVATION The use of a simple error monad in the definition of the static semantics provides a way of generating and propagating errors. Within the individual static semantic domains, top elements are often used to represent abnormal situations. However, in order to propagage these error messages, it would be necessary to enforce that all functions be strict with respect to top elements, something which would create a number of undesirable side effects. The definition of $\mathsf{E}$ is considered a better and more modular approach.

DEFINITION ■ $\mathsf{E}(D) \quad = \quad D \oplus \mathbf{U}$

For each domain $D$, the domain $\mathsf{E}(D)$ is defined as the coalesced sum of $D$ with $\mathbf{U}$, which contains a single ordinary element. This element, namely $u$, is used to represent errors. Equality and domain ordering are trivially defined for $\mathsf{E}(D)$, in terms of the corresponding relations in $D$. The monad's unit is defined as follows:

▶ $unit_{\mathsf{E}} \;:\; D \to \mathsf{E}(D)$

$unit_{\mathsf{E}} \;=\; inl$

and it is easy to verify that the error monad preserves bottom and top elements:

$unit_{\mathsf{E}}\ \bot_D = \bot_{\mathsf{E}(D)}$
$unit_{\mathsf{E}}\ \top_D = \top_{\mathsf{E}(D)}$

The bind operator for monad $\mathsf{E}$ is used in order to propagate errors. It is defined as follows:

▶ $\cdot\ *_{\mathsf{E}}\ \cdot \;:\; \mathsf{E}(A) \times (A \to \mathsf{E}(B)) \to \mathsf{E}(B)$

$m\ *_{\mathsf{E}}\ f \;=\; [f, inr\,]\,m$

It is easy to prove that $(\mathsf{E}, unit_{\mathsf{E}}, *_{\mathsf{E}})$ satisfies the three monad laws and is indeed a monad:

$$unit_{\mathsf{E}}\ a\ *_{\mathsf{E}}\ f = f\ a$$
$$m\ *_{\mathsf{E}}\ unit_{\mathsf{E}} = m$$
$$(m\ *_{\mathsf{E}}\ f)\ *_{\mathsf{E}}\ g = m\ *_{\mathsf{E}}\ (\lambda a.\ f\ a\ *_{\mathsf{E}}\ g)$$

In order to generate errors, function $error_{\mathsf{E}}$ is defined as follows: ERRORS

▶ $error_{\mathsf{E}} : \mathsf{E}(D)$
$error_{\mathsf{E}} = inr\ u$

and, again, it is easy to verify that errors are correctly propagated, as shown by the following properties:

$$error_{\mathsf{E}}\ *_{\mathsf{E}}\ f = error_{\mathsf{E}}$$
$$m\ *_{\mathsf{E}}\ (\lambda a.\ error_{\mathsf{E}}) = error_{\mathsf{E}}$$

As a matter of convention and since $\mathsf{E}$ is the only monad used in the description of static semantics, subscripts are be omitted from all monad related operations in the rest of Part II and whenever they can easily be deduced from the context. SUBSCRIPTS

## 4.5 Environments

The domains defined in this section represent various kinds of environments. Environments typically associate types with identifiers and are variations of functions from identifiers to static type domains. For each environment domain, the domain definition is followed by several functions that are used for the manipulation of its elements.

### 4.5.1 Type environments

Type environments, represented by the domain **Ent**, contain information about identifiers that are declared in C programs. They should contain at least the name spaces for ordinary identifiers and tags. Ordinary identifiers should be associated with identifier types, whereas tags should be associated with the subset of data types containing structure, union and enumeration types. Type environments should be structured in the same way that C program scopes are, that is, they should follow a tree-like structure of nested scopes. MOTIVATION

■ $e : \mathbf{Ent} = (\mathbf{Ide} \to \mathbf{Type}_{ide}) \times (\mathbf{Ide} \to \mathbf{Type}_{dat}) \times \mathbf{Ent}$ DEFINITION

The name space of ordinary identifiers is represented by the first part of the product, i.e. a function from identifiers to identifier types. The name space of tags is represented by the second part of the product, i.e. a function from identifiers (tags) to data types. The third part of the product represents the parent (enclosing) environment and is equal to $\top$ in the case of the outermost scope. Thus, domain **Ent** is a recursively defined domain. Its definition is slightly changed in Chapter 6. In the present section the domain ordering for types is completely ignored. This is corrected in Chapter 6.

EMPTY ENVIRONMENT

$$\blacktriangleright \quad e_\circ \ : \ \mathbf{Ent}$$
$$e_\circ \ = \ \top$$

The empty environment associates all identifiers with the top type. Thus, since the top type denotes an unspecified or erroneous value, in the empty environment all identifiers are not associated with types and their use leads to errors.

ORDINARY LOOKUP

$$\blacktriangleright \quad \cdot[\cdot\ ide] \ : \ \mathbf{Ent} \times \mathbf{Ide} \to \mathsf{E}(\mathbf{Type}_{ide})$$
$$e[I\ ide] \ =$$
$$\quad \mathbf{let}\ \delta = e[I\ raw\ ide]$$
$$\quad \mathbf{in}\ \ (\delta \neq \top) \to unit\ \delta\,,\ error$$

This function returns the identifier type that environment $e$ associates with the ordinary identifier $I$. An error occurs if the identifier is not included in $e$ or any of its ancestors.

ORDINARY RAW LOOKUP

$$\blacktriangleright \quad \cdot[\cdot\ raw\ ide] \ : \ \mathbf{Ent} \times \mathbf{Ide} \to \mathbf{Type}_{ide}$$
$$e[I\ raw\ ide] \ =$$
$$\quad \mathbf{let}\ \langle m_i, m_t, e_p\rangle = e$$
$$\quad \mathbf{in}\ \ (m_i\ I \neq \top) \to m_i\ I\,,$$
$$\qquad (e_p \neq \top) \to e_p[I\ raw\ ide]\,,\ \top$$

This auxiliary function performs the actual lookup of ordinary identifiers. If the identifier is not local in this environment, the lookup continues in the parent enviroment. A top element is returned if the identifier is not found.

TAG LOOKUP

$$\blacktriangleright \quad \cdot[\cdot\ tag\ \cdot] \ : \ \mathbf{Ent} \times \mathbf{Ide} \times \mathbf{TagType} \to \mathsf{E}(\mathbf{Ent} \times \mathbf{Type}_{dat})$$
$$e[I\ tag\ \sigma] \ =$$
$$\quad \mathbf{let}\ \tau = e[I\ raw\ tag]$$
$$\quad \mathbf{in}\ \ \mathbf{case}\ \tau\ \mathbf{of}$$
$$\qquad \top \ \Rightarrow \ \mathbf{case}\ \sigma\ \mathbf{of}$$
$$\qquad\qquad tag\text{-}struct \ \Rightarrow \ \mathbf{let}\ \tau' = struct\,[fresh\text{-}tagged\ I, \bot]$$
$$\qquad\qquad\qquad\qquad \mathbf{in}\ \ e[I \mapsto tag\ \tau'] \ * \ (\lambda\, e'.\ unit\ \langle e', \tau'\rangle)$$
$$\qquad\qquad tag\text{-}union \ \Rightarrow \ \mathbf{let}\ \tau' = union\,[fresh\text{-}tagged\ I, \bot]$$
$$\qquad\qquad\qquad\qquad \mathbf{in}\ \ e[I \mapsto tag\ \tau'] \ * \ (\lambda\, e'.\ unit\ \langle e', \tau'\rangle)$$
$$\qquad\qquad \mathbf{otherwise} \ \Rightarrow \ error$$
$$\qquad struct\,[t, \pi] \ \Rightarrow \ (\sigma = tag\text{-}struct) \to unit\ \langle e, \tau\rangle\,,\ error$$
$$\qquad union\,[t, \pi] \ \Rightarrow \ (\sigma = tag\text{-}union) \to unit\ \langle e, \tau\rangle\,,\ error$$
$$\qquad enum\,[\epsilon] \ \Rightarrow \ (\sigma = tag\text{-}enum) \to unit\ \langle e, \tau\rangle\,,\ error$$
$$\qquad \mathbf{otherwise} \ \Rightarrow \ error$$

This is the lookup function for tags. It is similar to the lookup function for ordinary identifiers, with two exceptions. First, the given type of the tag is compared with the actual data type that is associated with it in the environment $e$ and an error occurs if the two do not match. Second, if a structure or union tag is not found in $e$ or any of its ancestors, it is created as an incomplete structure or union.

TAG RAW LOOKUP

$$\blacktriangleright \quad \cdot[\cdot\ raw\ tag] \ : \ \mathbf{Ent} \times \mathbf{Ide} \to \mathbf{Type}_{den}$$
$$e[I\ raw\ tag] \ =$$
$$\quad \mathbf{let}\ \langle m_i, m_t, e_p\rangle = e$$
$$\quad \mathbf{in}\ \ (m_t\ I \neq \top) \to m_t\ I\,,$$
$$\qquad (e_p \neq \top) \to e_p[I\ raw\ tag]\,,\ \top$$

This auxiliary function performs the actual lookup of tags. Similar to the one for ordinary identifiers.

▶ $\cdot[\cdot\ tagID\ \cdot] : \mathbf{Ent} \times \mathbf{Ide} \times \mathbf{TagType} \to \mathsf{E}(\mathbf{Tag})$ GET TAG

$$
\begin{array}{l}
e[I\ tagID\ \sigma] = \\
\quad \textbf{let}\ \tau = e[I\ raw\ tag] \\
\quad \textbf{in}\ \ \textbf{case}\ \tau\ \textbf{of} \\
\qquad\quad struct\,[t,\pi] \Rightarrow (\sigma = tag\text{-}struct) \to unit\ t\,,\ error \\
\qquad\quad union\,[t,\pi] \Rightarrow (\sigma = tag\text{-}union) \to unit\ t\,,\ error \\
\qquad\quad \textbf{otherwise} \Rightarrow error
\end{array}
$$

This function looks up a tag in an environment and returns the corresponding element of **Tag**, i.e. a unique tag element. It also checks that the tag is of the right kind.

▶ $\cdot[\cdot \mapsto ide\ \cdot] : \mathbf{Ent} \times \mathbf{Ide} \times \mathbf{Type}_{ide} \to \mathsf{E}(\mathbf{Ent})$ ORDINARY UPDATE

$$
\begin{array}{l}
e[I \mapsto ide\ \delta] = \\
\quad \textbf{let}\ \langle m_i, m_t, e_p\rangle = e \\
\quad \textbf{in}\ \ (m_i\ I = \top) \to unit\ \langle m_i\{I \mapsto \delta\}, m_t, e_p\rangle\,,\ error
\end{array}
$$

This function returns an environment that is identical to $e$, with the exception that identifier $I$ is associated with type $\delta$. In case $I$ is already a local in $e$, an error occurs.

▶ $\cdot[\cdot \mapsto tag\ \cdot] : \mathbf{Ent} \times \mathbf{Ide} \times \mathbf{Type}_{dat} \to \mathsf{E}(\mathbf{Ent})$ TAG UPDATE

$$
\begin{array}{l}
e[I \mapsto tag\ \tau] = \\
\quad \textbf{let}\ \langle m_i, m_t, e_p\rangle = e \\
\quad \textbf{in}\ \ (m_t\ I = \top) \vee isDeclaredTag(m_t\ I) \to unit\ \langle m_i, m_t\{I \mapsto \tau\}, e_p\rangle\,,\ error
\end{array}
$$

Similar to the ordinary update function for tags. However, a tag can be updated even if it is local in $e$, provided it is a structure or union tag that has only been declared.

▶ $\cdot[\cdot \mapsto fresh\ tag\ \cdot] : \mathbf{Ent} \times \mathbf{Ide} \times \mathbf{TagType} \to \mathsf{E}(\mathbf{Ent})$ FRESH TAG

$$
\begin{array}{l}
e[I \mapsto fresh\ tag\ \sigma] = \\
\quad \textbf{let}\ \langle m_i, m_t, e_p\rangle = e \\
\qquad \tau = m_t\ I \\
\quad \textbf{in}\ \ \textbf{case}\ \tau\ \textbf{of} \\
\qquad\quad \top \Rightarrow \textbf{case}\ \sigma\ \textbf{of} \\
\qquad\qquad\qquad\quad tag\text{-}struct \Rightarrow e[I \mapsto tag\ struct\,[fresh\text{-}tagged\ I, \bot]] \\
\qquad\qquad\qquad\quad tag\text{-}union \Rightarrow e[I \mapsto tag\ union\,[fresh\text{-}tagged\ I, \bot]] \\
\qquad\qquad\qquad\quad \textbf{otherwise} \Rightarrow error \\
\qquad\quad struct\,[t,\pi] \Rightarrow (\sigma = tag\text{-}struct) \to unit\ e\,,\ error \\
\qquad\quad union\,[t,\pi] \Rightarrow (\sigma = tag\text{-}union) \to unit\ e\,,\ error \\
\qquad\quad enum\,[\epsilon] \Rightarrow (\sigma = tag\text{-}enum) \to unit\ e\,,\ error \\
\qquad\quad \textbf{otherwise} \Rightarrow error
\end{array}
$$

This function creates an incomplete structure or union tag in $e$, if it is not already local in $e$. Otherwise, it checks that the tag is of the right kind and generates an error if it is not.

▶ $\uparrow\cdot : \mathbf{Ent} \to \mathbf{Ent}$ OPEN SCOPE

$$\uparrow e = \langle \top, \top, e\rangle$$

This function returns an environment that corresponds to a newly opened scope, using $e$ as the parent scope. The new environment has no locally defined ordinary identifiers or tags.

CLOSE SCOPE ▶ $\downarrow \cdot \;:\; \mathbf{Ent} \to \mathbf{Ent}$

$\downarrow e \;=\; \mathbf{let}\ \langle m_i, m_t, e_p \rangle = e\ \mathbf{in}\ e_p$

This function returns the parent scope of $e$.

LOCAL ORDINARY ▶ $isLocal(\cdot, \cdot\ ide) \;:\; \mathbf{Ent} \to \mathbf{Ide} \to \mathbf{T}$

$isLocal(e, I\ ide) \;=\; \mathbf{let}\ \langle m_i, m_t, e_p \rangle = e\ \mathbf{in}\ m_i\ I \neq \top$

This function checks whether an $I$ is a local ordinary identifier in environment $e$.

LOCAL TAG ▶ $isLocal(\cdot, \cdot\ tag) \;:\; \mathbf{Ent} \to \mathbf{Ide} \to \mathbf{T}$

$isLocal(e, I\ tag) \;=\; \mathbf{let}\ \langle m_i, m_t, e_p \rangle = e\ \mathbf{in}\ m_t\ I \neq \top$

This function checks whether an $I$ is a local tag in environment $e$.

### 4.5.2 Enumeration environments

MOTIVATION Enumeration environments are represented by the domain $\mathbf{Enum}$. They are used to associate the named constants in an enumeration to the numeric values that they represent. Since the standard does not specify the size of an enumeration, but leaves it implementation-defined, it is reasonable to allow the size of a particular enumeration to depend on the numeric values of its enumeration constants. For tis reason, these values are necessary in the static environment.

DEFINITION ■ $\epsilon \;:\; \mathbf{Enum} \;=\; \mathbf{Ide} \to \mathbf{N}$

An enumeration environment is simply represented by a function from identifiers to integer values.

EMPTY ▶ $\epsilon_\circ \;:\; \mathbf{Enum}$

$\epsilon_\circ \;=\; \top$

The empty enumeration environment does not contain any identifier.

UPDATE ▶ $\cdot[\cdot \mapsto \cdot] \;:\; \mathbf{Enum} \times \mathbf{Ide} \times \mathbf{N} \to \mathsf{E}(\mathbf{Enum})$

$\epsilon[I \mapsto n] \;=\; (\epsilon\ I = \top) \to unit\ \epsilon\{I \mapsto n\}\,,\ error$

This function updates an enumeration environment by associating identifier $I$ with the constant integer value $n$. If identifier $I$ is already contained in the enumeration, an error occurs.

### 4.5.3 Member environments

MOTIVATION The domain $\mathbf{Memb}$ represents environments containing the members of structures or unions. In such environments, each identifier is associated with a member type. However, in contrast to enumeration environments, the order of individual members in a structure or union is important and must be stored in the environment for two reasons:

- Initialization of a structure or union heavily depends on the order of its members.
- Members of a structure object must be allocated increasing addresses in memory, in the order in which they are defined.

Thus, in order to store the order of members it is necessary to define $\mathbf{Memb}$ in a more complicated way than just as a function from identifiers to member types.

■ $\pi \;:\; \mathbf{Memb} \;=\; \mathbf{N} \times (\mathbf{N} \to \mathbf{Ide}) \times (\mathbf{Ide} \to \mathbf{Type}_{mem})$ DEFINITION

The first part of the product is an integer number that denotes the number of members that a member environment contains. Valid values of this part are all the non-negative integer numbers. A bottom value is only used in the case of incomplete member environments, as is discussed in Chapter 6. The second part of the product represents the order in which the members have been defined. It is a function from integers to identifiers, which must map all integers between 1 and the actual number of members to the corresponding identifiers, and all other elements of $\mathbf{N}$ to the top value. Finally, the third and last part of the product is the actual mapping from identifiers to member types. Identifiers that participate in the second part must be mapped to the corresponding member types in the third part, and all other elements of $\mathbf{Ide}$ must be mapped to the top value. It is easy to see that all operations on member environments that are defined below preserve the validity restrictions that were just stated.

▶ $\pi_\circ \;:\; \mathbf{Memb}$ EMPTY

$\pi_\circ \;=\; \langle 0, \top, \top \rangle$

An empty member environment does not contain any members. However, it should be noted that an empty member environment is not incomplete.

▶ $\cdot[\cdot] \;:\; \mathbf{Memb} \times \mathbf{Ide} \to \mathsf{E}(\mathbf{Type}_{mem})$ LOOKUP

$\pi[I] \;=$
$\quad \mathbf{let}\ \langle n, m_n, m_i \rangle = \pi$
$\quad \mathbf{in}\ \ (m_i\ I \neq \top) \to unit\ (m_i\ I)\,,\ error$

This function looks up an identifier $I$ in a member environment $\pi$. If $I$ is contained in $\pi$ its member type is returned, otherwise an error occurs.

▶ $\cdot[\cdot \mapsto \cdot] \;:\; \mathbf{Memb} \times \mathbf{Ide} \times \mathbf{Type}_{mem} \to \mathsf{E}(\mathbf{Memb})$ APPEND

$\pi[I \mapsto m] \;=$
$\quad \mathbf{let}\ \langle n, m_n, m_i \rangle = \pi$
$\quad \mathbf{in}\ \ (m_i\ I = \top) \to unit\ \langle n+1, m_n\{n+1 \mapsto I\}, m_i\{I \mapsto m\} \rangle\,,\ error$

This function appends identifier $I$ in a member environment $\pi$ as a member of type $m$. The resulting environment is returned. An error occurs if a member named $I$ is already contained in $\pi$.

▶ $\Downarrow \cdot \;:\; \mathbf{Memb} \to \mathsf{E}(\mathbf{Ide} \times \mathbf{Type}_{mem} \times \mathbf{Memb})$ DECOMPOSE

$\Downarrow \pi \;=$
$\quad \mathbf{let}\ \langle n, m_n, m_i \rangle = \pi$
$\quad \mathbf{in}\ \ (n = \bot) \vee (n = \top) \vee (n \leq 0) \to error\,,$
$\qquad \mathbf{let}\ I \quad = m_n\ 1$
$\qquad\quad m'_n = \lambda k.\ (k \geq 1) \wedge (k < n) \to m_n\ (k+1)\,,\ \top$
$\qquad\quad m'_i = m_i\{I \mapsto \top\}$
$\qquad\quad \pi' \;= \langle n-1, m'_n, m'_i \rangle$
$\qquad \mathbf{in}\ \ unit\ \langle I, m_i\ I, e' \rangle$

This function decomposes a member environment by extracting its first member. It returns the name and type of the first member and the environment that results from removing this member. An error occurs if the member environment is incomplete (bottom) or empty. This function can be thought of as the opposite of an imaginary "prepend" function.

### 4.5.4 Function prototypes

MOTIVATION Function prototypes are represented by the domain $\mathbf{Prot}$. The main piece of information that must be contained in a function prototype is the number of parameters and their types. The names of the parameters are not important, but their order obviously is. In addition, the C standard allows "incomplete" function prototypes by using the ellipsis notation after the last parameter. Storage specifiers and qualifiers are ignored in function parameters, as suggested by the standard in §6.5.4.3.

DEFINITION ■ $p \;:\; \mathbf{Prot} \;=\; \mathbf{N} \times (\mathbf{N} \to \mathbf{Type}_{dat}) \times \mathbf{T}$

The first part of the product is an integer number that denotes the number of parameters in the function prototype. Its valid values are all non-negative integer numbers. The second part of the product is a function from integer number to data types, which maps all integers between 1 and the actual number of parameters to the corresponding types, and all other elements of $\mathbf{N}$ to the top value. Finally, the third part of the product is a truth value; it contains the value *true* if the prototype ends with an ellipsis, *false* otherwise. If an ellipsis is used, the number of parameters in the function prototype must not be zero, according to the standard. However, this test is not performed here, as stated by deviation D-10 in Section 2.3. It can be enforced in the abstract syntax.

EMPTY ▶ $p_\circ \;:\; \mathbf{Prot}$

$p_\circ \;=\; \langle 0, \top, \mathit{false} \rangle$

An empty function prototype simply contains no parameters and no ellipsis.

APPEND ▶ $\cdot \ll \cdot \;:\; \mathbf{Prot} \times \mathbf{Type}_{dat} \to \mathsf{E}(\mathbf{Prot})$

$p \ll \tau \;=\; \mathbf{let}\ \langle n, m_p, b_{ell} \rangle = p\ \mathbf{in}\ \neg b_{ell} \to \mathit{unit}\ \langle n+1, m_p\{n+1 \mapsto \tau\}, b_{ell} \rangle\,,\ \mathit{error}$

This function appends a parameter of type $\tau$ to prototype $p$. If an ellipsis in already present in $p$, an error occurs.

PREPEND ▶ $\cdot \lessdot \cdot \;:\; \mathbf{Type}_{dat} \times \mathbf{Prot} \to \mathsf{E}(\mathbf{Prot})$

$\tau \lessdot p \;=$
$\quad \mathbf{let}\ \langle n, m_p, b_{ell} \rangle = p$
$\quad\quad m'_p \quad\quad = \lambda\, k.\ (k = 1) \to \tau\,,\ (k \le n+1) \to m_p\ (k-1)\,,\ \top$
$\quad \mathbf{in}\ \ \mathit{unit}\ \langle n+1, m'_p, b_{ell} \rangle$

This function prepends a parameter of type $\tau$ to prototype $p$, that is, the type of the first parameter becomes $\tau$ and all other parameters are "pushed" one place to the right.

APPEND ELLIPSIS ▶ $\mathit{ellipsis} \;:\; \mathbf{Prot} \to \mathsf{E}(\mathbf{Prot})$

$\mathit{ellipsis}\ p \;=\; \mathbf{let}\ \langle n, m_p, b_{ell} \rangle = p\ \mathbf{in}\ \neg b_{ell} \to \mathit{unit}\ \langle n, m_p, \mathit{true} \rangle\,,\ \mathit{error}$

This function adds an ellipsis at the end of a function prototype. If an ellipsis is already present, an error occurs.

## 4.6 Auxiliary functions

A number of auxiliary functions is defined in this section. These functions are used in order to simplify the semantic equations. Most of them correspond directly to notions expressed in the standard. Some functions apply to various domains and are subscripted accordingly, e.g. $isInteger_{dat}$ is the function that identifies integer data types and $isInteger_{obj}$ does the same for object types. When the domain can easily be deduced from context, the subscripts are omitted.

### 4.6.1 Predicates related to types

A set of predicates is defined in an attempt to classify types. These predicates take a type as argument and return a truth value. They are all strict with respect to bottom and top elements.

▶ $isInteger_{dat} : \mathbf{Type}_{dat} \to \mathbf{T}$ IS INTEGER

$$\begin{aligned} isInteger_{dat} \;=\; & strict_{\mathrm{I}}\,(\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ & \quad short\text{-}int \mid unsigned\text{-}short\text{-}int \mid int \mid unsigned\text{-}int \mid long\text{-}int \mid unsigned\text{-}long\text{-}int \Rightarrow true \\ & \quad \mathbf{otherwise} \Rightarrow false) \end{aligned}$$

▶ $isInteger_{obj} : \mathbf{Type}_{obj} \to \mathbf{T}$

$$\begin{aligned} isInteger_{obj} \;=\; & strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of} \\ & \quad obj\,[\tau, q] \Rightarrow isInteger_{dat}(\tau) \\ & \quad array\,[\alpha, n] \Rightarrow false) \end{aligned}$$

▶ $isInteger_{den} : \mathbf{Type}_{den} \to \mathbf{T}$

$$\begin{aligned} isInteger_{den} \;=\; & strict_{\mathrm{I}}\,(\lambda\phi.\ \mathbf{case}\ \phi\ \mathbf{of} \\ & \quad \alpha \Rightarrow isInteger_{obj}(\alpha) \\ & \quad f \Rightarrow false) \end{aligned}$$

▶ $isInteger_{mem} : \mathbf{Type}_{mem} \to \mathbf{T}$

$$\begin{aligned} isInteger_{mem} \;=\; & strict_{\mathrm{I}}\,(\lambda m.\ \mathbf{case}\ m\ \mathbf{of} \\ & \quad \alpha \Rightarrow isInteger_{obj}(\alpha) \\ & \quad bitfield\,[\beta, q, n] \Rightarrow true) \end{aligned}$$

The family of $isInteger$ functions is used to identify integer types. They return $true$ if the argument is an integer type, $false$ if it is not.

▶ $isIntegral_{dat} : \mathbf{Type}_{dat} \to \mathbf{T}$ IS INTEGRAL

$$\begin{aligned} isIntegral_{dat} \;=\; & strict_{\mathrm{I}}\,(\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ & \quad char \mid signed\text{-}char \mid unsigned\text{-}char \mid enum\,[\epsilon] \Rightarrow true \\ & \quad \mathbf{otherwise} \Rightarrow isInteger_{dat}(\tau)) \end{aligned}$$

▶ $isIntegral_{obj} : \mathbf{Type}_{obj} \to \mathbf{T}$

$$\begin{aligned} isIntegral_{obj} \;=\; & strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of} \\ & \quad obj\,[\tau, q] \Rightarrow isIntegral_{dat}(\tau) \\ & \quad array\,[\alpha, n] \Rightarrow false) \end{aligned}$$

▶ $isIntegral_{den} : \mathbf{Type}_{den} \to \mathbf{T}$

$isIntegral_{den} = strict_{\mathrm{I}}\,(\lambda\phi.\ \mathbf{case}\ \phi\ \mathbf{of}$
$\quad \alpha \Rightarrow isIntegral_{obj}(\alpha)$
$\quad f \Rightarrow false)$

▶ $isIntegral_{mem} : \mathbf{Type}_{mem} \to \mathbf{T}$

$isIntegral_{mem} = strict_{\mathrm{I}}\,(\lambda m.\ \mathbf{case}\ m\ \mathbf{of}$
$\quad \alpha \Rightarrow isIntegral_{obj}(\alpha)$
$\quad bitfield\,[\beta, q, n] \Rightarrow true)$

The family of *isIntegral* functions is used to identify integral types, i.e. character types, integer types and enumeration types. They return *true* if the argument is an integral type, *false* if it is not.

IS ARITHMETIC

▶ $isArithmetic_{dat} : \mathbf{Type}_{dat} \to \mathbf{T}$

$isArithmetic_{dat} = strict_{\mathrm{I}}\,(\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of}$
$\quad float \mid double \mid long\text{-}double \Rightarrow true$
$\quad \mathbf{otherwise} \Rightarrow isIntegral_{dat}(\tau))$

▶ $isArithmetic_{obj} : \mathbf{Type}_{obj} \to \mathbf{T}$

$isArithmetic_{obj} = strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of}$
$\quad obj\,[\tau, q] \Rightarrow isArithmetic_{dat}(\tau)$
$\quad array\,[\alpha, n] \Rightarrow false)$

▶ $isArithmetic_{den} : \mathbf{Type}_{den} \to \mathbf{T}$

$isArithmetic_{den} = strict_{\mathrm{I}}\,(\lambda\phi.\ \mathbf{case}\ \phi\ \mathbf{of}$
$\quad \alpha \Rightarrow isArithmetic_{obj}(\alpha)$
$\quad f \Rightarrow false)$

▶ $isArithmetic_{mem} : \mathbf{Type}_{mem} \to \mathbf{T}$

$isArithmetic_{mem} = strict_{\mathrm{I}}\,(\lambda m.\ \mathbf{case}\ m\ \mathbf{of}$
$\quad \alpha \Rightarrow isArithmetic_{obj}(\alpha)$
$\quad bitfield\,[\beta, q, n] \Rightarrow true)$

The family of *isArithmetic* functions is used to identify arithmetic types, i.e. integral and floating types. They return *true* if the argument is an arithmetic type, *false* if it is not.

IS SCALAR

▶ $isScalar_{dat} : \mathbf{Type}_{dat} \to \mathbf{T}$

$isScalar_{dat} = strict_{\mathrm{I}}\,(\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of}$
$\quad ptr\,[\phi] \Rightarrow true$
$\quad \mathbf{otherwise} \Rightarrow isArithmetic_{dat}(\tau))$

▶ $isScalar_{obj} : \mathbf{Type}_{obj} \to \mathbf{T}$

$isScalar_{obj} = strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of}$
$\quad obj\,[\tau, q] \Rightarrow isScalar_{dat}(\tau)$
$\quad array\,[\alpha, n] \Rightarrow false)$

▶ $isScalar_{den} : \mathbf{Type}_{den} \to \mathbf{T}$

$$\begin{aligned} isScalar_{den} \;=\; & strict_{\mathrm{I}}\,(\lambda\phi.\ \mathbf{case}\ \phi\ \mathbf{of} \\ & \alpha \Rightarrow isScalar_{obj}(\alpha) \\ & f \Rightarrow false) \end{aligned}$$

▶ $isScalar_{mem} : \mathbf{Type}_{mem} \to \mathbf{T}$

$$\begin{aligned} isScalar_{mem} \;=\; & strict_{\mathrm{I}}\,(\lambda m.\ \mathbf{case}\ m\ \mathbf{of} \\ & \alpha \Rightarrow isScalar_{obj}(\alpha) \\ & bitfield\,[\beta, q, n] \Rightarrow true) \end{aligned}$$

The family of *isScalar* functions is used to identify scalar types, i.e. arithmetic and pointer types. They return *true* if the argument is a scalar type, *false* if it is not.

IS STRUCTURE OR UNION

▶ $isStructUnion_{dat} : \mathbf{Type}_{dat} \to \mathbf{T}$

$$\begin{aligned} isStructUnion_{dat} \;=\; & strict_{\mathrm{I}}\,(\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ & struct\,[t, \pi] \mid union\,[t, \pi] \Rightarrow true \\ & \mathbf{otherwise} \Rightarrow false) \end{aligned}$$

▶ $isStructUnion_{obj} : \mathbf{Type}_{obj} \to \mathbf{T}$

$$\begin{aligned} isStructUnion_{obj} \;=\; & strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of} \\ & obj\,[\tau, q] \Rightarrow isStructUnion_{dat}(\tau) \\ & array\,[\alpha, n] \Rightarrow false) \end{aligned}$$

▶ $isStructUnion_{den} : \mathbf{Type}_{den} \to \mathbf{T}$

$$\begin{aligned} isStructUnion_{den} \;=\; & strict_{\mathrm{I}}\,(\lambda\phi.\ \mathbf{case}\ \phi\ \mathbf{of} \\ & \alpha \Rightarrow isStructUnion_{obj}(\alpha) \\ & f \Rightarrow false) \end{aligned}$$

▶ $isStructUnion_{mem} : \mathbf{Type}_{mem} \to \mathbf{T}$

$$\begin{aligned} isStructUnion_{mem}(\alpha) \;=\; & strict_{\mathrm{I}}\,(\lambda m.\ \mathbf{case}\ m\ \mathbf{of} \\ & \alpha \Rightarrow isStructUnion_{obj}(\alpha) \\ & bitfield\,[\beta, q, n] \Rightarrow false) \end{aligned}$$

The family of *isStructUnion* functions is used to identify structure and union types. They return *true* if the argument is a structure or union type, *false* if it is not.

IS DECLARED TAG

▶ $isDeclaredTag_{dat} : \mathbf{Type}_{dat} \to \mathbf{T}$

$$\begin{aligned} isDeclaredTag_{dat} \;=\; & strict_{\mathrm{I}}\,(\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ & struct\,[t, \bot] \mid union\,[t, \bot] \Rightarrow true \\ & \mathbf{otherwise} \Rightarrow false) \end{aligned}$$

▶ $isDeclaredTag_{obj} : \mathbf{Type}_{obj} \to \mathbf{T}$

$$\begin{aligned} isDeclaredTag_{obj} \;=\; & strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of} \\ & obj\,[\tau, q] \Rightarrow isDeclaredTag_{dat}(\tau) \\ & array\,[\alpha, n] \Rightarrow false) \end{aligned}$$

▶ $isDeclaredTag_{den} : \mathbf{Type}_{den} \to \mathbf{T}$

$$\begin{aligned} isDeclaredTag_{den} = {} & strict_{\mathrm{I}}\,(\lambda\phi.\ \mathbf{case}\ \phi\ \mathbf{of} \\ & \alpha \Rightarrow isDeclaredTag_{obj}(\alpha) \\ & f \Rightarrow false) \end{aligned}$$

▶ $isDeclaredTag_{mem} : \mathbf{Type}_{mem} \to \mathbf{T}$

$$\begin{aligned} isDeclaredTag_{mem} = {} & strict_{\mathrm{I}}\,(\lambda m.\ \mathbf{case}\ m\ \mathbf{of} \\ & \alpha \Rightarrow isDeclaredTag_{obj}(\alpha) \\ & bitfield\,[\beta, q, n] \Rightarrow false) \end{aligned}$$

The family of *isDeclaredTag* functions is used to identify incomplete structures and unions whose tags have only been declared. They return *true* if the argument is such a structure or union, *false* if it is not.

IS COMPLETE ▶ $isComplete_{dat} : \mathbf{Type}_{dat} \to \mathbf{T}$

$$\begin{aligned} isComplete_{dat} = {} & strict_{\mathrm{I}}\,(\lambda\tau.\ \mathbf{case}\ \tau\ \mathbf{of} \\ & void \Rightarrow false \\ & \mathbf{otherwise} \Rightarrow \neg isDeclaredTag_{dat}(\tau)) \end{aligned}$$

▶ $isComplete_{obj} : \mathbf{Type}_{obj} \to \mathbf{T}$

$$\begin{aligned} isComplete_{obj} = {} & strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of} \\ & obj\,[\tau, q] \Rightarrow isComplete_{dat}(\tau) \\ & array\,[\alpha, n] \Rightarrow (n = \bot) \to false\,,\ isComplete_{obj}(\alpha)) \end{aligned}$$

▶ $isComplete_{den} : \mathbf{Type}_{den} \to \mathbf{T}$

$$\begin{aligned} isComplete_{den} = {} & strict_{\mathrm{I}}\,(\lambda\phi.\ \mathbf{case}\ \phi\ \mathbf{of} \\ & \alpha \Rightarrow isComplete_{obj}(\alpha) \\ & f \Rightarrow true) \end{aligned}$$

▶ $isComplete_{mem} : \mathbf{Type}_{mem} \to \mathbf{T}$

$$\begin{aligned} isComplete_{mem} = {} & strict_{\mathrm{I}}\,(\lambda m.\ \mathbf{case}\ m\ \mathbf{of} \\ & \alpha \Rightarrow isComplete_{obj}(\alpha) \\ & bitfield\,[\beta, q, n] \Rightarrow true) \end{aligned}$$

▶ $isComplete_{val} : \mathbf{Type}_{val} \to \mathbf{T}$

$$\begin{aligned} isComplete_{val} = {} & strict_{\mathrm{I}}\,(\lambda v.\ \mathbf{case}\ v\ \mathbf{of} \\ & \tau \Rightarrow isComplete_{dat}(\tau) \\ & f \Rightarrow true) \end{aligned}$$

The family of *isComplete* functions is used to distinguish between complete and incomplete types. They return *true* if the argument is a complete type, *false* if it is not. According to the standard, incomplete types are *void*, structures and unions whose tags have only been declared, arrays of unknown size and aggregate types that contain elements of incomplete types. The latter is not even allowed in declarations and this is the reason why it is not necessary to require that all members of structures or arrays belong to complete types.

▶ $isBitfield_{mem} : \mathbf{Type}_{mem} \to \mathbf{T}$ IS BIT-FIELD

$$isBitfield_{mem} = strict_{\mathrm{I}}\,(\lambda\, m.\ \mathbf{case}\ m\ \mathbf{of}\ \alpha \Rightarrow false\ \ bitfield\,[\beta, q, n] \Rightarrow true)$$

▶ $isBitfield_{phr} : \mathbf{Type}_{phr} \to \mathbf{T}$

$$isBitfield_{phr} = strict_{\mathrm{I}}\,(\lambda\, \theta.\ \mathbf{case}\ \theta\ \mathbf{of}\ lvalue\,[m] \Rightarrow isBitfield_{mem}(m)\ \ \mathbf{otherwise} \Rightarrow false)$$

The family of *isBitfield* functions is used to distinguish between normal and bit-field member types. They return *true* if the argument is a bit-field type, *false* if it is not.

▶ $isModifiable_{dat} : \mathbf{Type}_{dat} \to \mathbf{T}$ IS MODIFIABLE

$$isModifiable_{dat} = strict_{\mathrm{I}}\,(\lambda\, \tau.\ \mathbf{case}\ \tau\ \mathbf{of}\ struct\,[t, \pi]\ |\ union\,[t, \pi] \Rightarrow (\pi \neq \bot) \wedge isModifiable_{Memb}(\pi)\ \ \mathbf{otherwise} \Rightarrow isComplete_{dat}(\tau))$$

▶ $isModifiable_{obj} : \mathbf{Type}_{obj} \to \mathbf{T}$

$$isModifiable_{obj} = strict_{\mathrm{I}}\,(\lambda\, \alpha.\ \mathbf{case}\ \alpha\ \mathbf{of}\ obj\,[\tau, q] \Rightarrow \neg(const \subseteq q) \wedge isModifiable_{dat}(\tau)\ \ array\,[\alpha, n] \Rightarrow false)$$

▶ $isModifiable_{mem} : \mathbf{Type}_{mem} \to \mathbf{T}$

$$isModifiable_{mem} = strict_{\mathrm{I}}\,(\lambda\, m.\ \mathbf{case}\ m\ \mathbf{of}\ \alpha \Rightarrow isModifiable_{obj}(\alpha)\ \ bitfield\,[\beta, q, n] \Rightarrow \neg(const \subseteq q))$$

▶ $isModifiable_{Memb} : \mathbf{Memb} \to \mathbf{T}$

$$isModifiable_{Memb} = strict_{\mathrm{I}}\,(\lambda\, \pi.\ \forall\, I \in dom(\pi).\ isModifiable_{mem}(\pi[I]))$$

The family of *isModifiable* functions is used to identify types corresponding to modifiable l-values. They return *true* if the argument is such a type, *false* if it is not. According to the standard, the type of a modifiable l-value must not be an array type, must not be incomplete, must not be *const*-qualified and, if it is an aggregate type, must not contain non-modifiable members.

### 4.6.2 Functions related to types

Two relations and five functions are defined in this section. All of them are bi-strict with respect to bottom and top elements. The two relations are specified as implementation-defined in the standard. For this reason, they are not fully defined here. Instead, a number of properties that they must satisfy is given.

TYPE INCLUSION

▶ $\cdot \subseteq \cdot \;:\; \mathbf{Type}_{dat} \times \mathbf{Type}_{dat} \to \mathbf{T}$

$\mathit{signed\text{-}char} \subseteq \mathit{short\text{-}int} \subseteq \mathit{int} \subseteq \mathit{long\text{-}int}$

$\mathit{enum}\,[\epsilon] \subseteq \mathit{int}$

$\mathit{unsigned\text{-}char} \subseteq \mathit{unsigned\text{-}short\text{-}int} \subseteq \mathit{unsigned\text{-}int} \subseteq \mathit{unsigned\text{-}long\text{-}int}$

$\mathit{float} \subseteq \mathit{double} \subseteq \mathit{long\text{-}double}$

$\forall\, \phi.\, \exists\, \tau.\, \mathit{isIntegral}(\tau) \;\wedge\; (\mathit{ptr}\,[\phi] \subseteq \tau)$

$\forall\, \tau.\, \tau \subseteq \tau$

$\forall\, \tau_1, \tau_2, \tau_3.\, (\tau_1 \subseteq \tau_2) \;\wedge\; (\tau_2 \subseteq \tau_3) \;\Rightarrow\; (\tau_1 \subseteq \tau_3)$

The type inclusion relation between data types has the intuitive meaning that all values of the first type are also valid values for the second, i.e. the set of values of the first type is a subset of that of the second type. The standard specifies the aforementioned inclusions for character, integer, floating and enumeration types, and that all pointers can be represented by some implementation-defined integral type. The last two properties state that type inclusion is a reflexive and transitive relation.

REPRESENTATION OF CONSTANTS

▶ $\cdot \in \cdot \;:\; \mathit{constant} \times \mathbf{Type}_{dat} \to \mathbf{T}$

$\forall\, c, \tau_1, \tau_2.\, (c \in \tau_1) \;\wedge\; (\tau_1 \subseteq \tau_2) \;\Rightarrow\; (c \in \tau_2)$

This is also an implementation-defined relation between constants and data types. The intuitive meaning of the statement $c \in \tau$ is that the value of the constant $c$ can be stored in a variable of type $\tau$ without any loss of information. The aforementioned property, that this relation must satisfy, only connects this relation with type inclusion.

FIRST TO REPRESENT

▶ $\mathit{firstToRepresent} \;:\; \mathit{integer\text{-}constant} \times \mathbf{Type}_{dat}\ \mathbf{list} \to \mathsf{E}(\mathbf{Type}_{dat})$

$\mathit{firstToRepresent}(n, [\,]) \;=\; \mathit{error}$

$\mathit{firstToRepresent}(n, \mathit{cons}(\tau, \ell)) \;=\; (n \in \tau) \to \mathit{unit}\ \tau\,,\ \mathit{firstToRepresent}(n, \ell)$

This function returns the first type in a list of possible data types, in which a given integer constant can be represented. An error occurs if the constant cannot be represented in any of the given types.

INTEGRAL PROMOTIONS

▶ $\mathit{intPromote} \;:\; \mathbf{Type}_{dat} \to \mathsf{E}(\mathbf{Type}_{dat})$

$\mathit{intPromote} \;=\; \mathit{strict}_{\mathrm{I}}\ (\lambda\, \tau.\, \mathbf{case}\ \tau\ \mathbf{of}$

| | | |
|---|---|---|
| $\mathit{char}$ | $\Rightarrow$ | $\mathit{unit}\ ((\mathit{char} \subseteq \mathit{int}) \to \mathit{int}\,,\ \mathit{unsigned\text{-}int})$ |
| $\mathit{signed\text{-}char}$ | $\Rightarrow$ | $\mathit{unit}\ \mathit{int}$ |
| $\mathit{unsigned\text{-}char}$ | $\Rightarrow$ | $\mathit{unit}\ ((\mathit{unsigned\text{-}char} \subseteq \mathit{int}) \to \mathit{int}\,,\ \mathit{unsigned\text{-}int})$ |
| $\mathit{short\text{-}int}$ | $\Rightarrow$ | $\mathit{unit}\ \mathit{int}$ |
| $\mathit{unsigned\text{-}short\text{-}int}$ | $\Rightarrow$ | $\mathit{unit}\ ((\mathit{unsigned\text{-}short\text{-}int} \subseteq \mathit{int}) \to \mathit{int}\,,\ \mathit{unsigned\text{-}int})$ |
| $\mathit{enum}\,[\epsilon]$ | $\Rightarrow$ | $\mathit{unit}\ \mathit{int}$ |
| **otherwise** | $\Rightarrow$ | $\mathit{unit}\ \tau)$ |

This function performs the integral promotions to its argument, as specified in §6.2.1.1 of the standard. The argument is left unchanged if the integral promotions cannot be applied.

▶ $arithConv \;:\; \mathbf{Type}_{dat} \times \mathbf{Type}_{dat} \to \mathsf{E}(\mathbf{Type}_{dat})$ — USUAL ARITHMETIC CONVERSIONS

$$
\begin{array}{l}
arithConv \;=\; bi\text{-}strict_{\mathrm{I}}\,(\lambda\,\langle \tau_1, \tau_2\rangle.\\
\quad (\tau_1 = \mathit{long\text{-}double}) \vee (\tau_2 = \mathit{long\text{-}double}) \to \mathit{unit}\ \mathit{long\text{-}double}\,,\\
\quad (\tau_1 = \mathit{double}) \vee (\tau_2 = \mathit{double}) \to \mathit{unit}\ \mathit{double}\,,\\
\quad (\tau_1 = \mathit{float}) \vee (\tau_2 = \mathit{float}) \to \mathit{unit}\ \mathit{float}\,,\\
\quad intPromote\ \tau_1\ *\ (\lambda\,\tau_1'.\\
\quad intPromote\ \tau_2\ *\ (\lambda\,\tau_2'.\\
\quad (\tau_1' = \mathit{unsigned\text{-}long\text{-}int}) \vee (\tau_2' = \mathit{unsigned\text{-}long\text{-}int}) \to \mathit{unit}\ \mathit{unsigned\text{-}long\text{-}int}\,,\\
\quad ((\tau_1' = \mathit{long\text{-}int}) \wedge (\tau_2' = \mathit{unsigned\text{-}int})) \vee ((\tau_1' = \mathit{unsigned\text{-}int}) \wedge (\tau_2' = \mathit{long\text{-}int})) \to\\
\qquad \mathit{unit}\ ((\mathit{unsigned\text{-}int} \subseteq \mathit{long\text{-}int}) \to \mathit{long\text{-}int}\,,\ \mathit{unsigned\text{-}long\text{-}int})\,,\\
\quad (\tau_1' = \mathit{long\text{-}int}) \vee (\tau_2' = \mathit{long\text{-}int}) \to \mathit{unit}\ \mathit{long\text{-}int}\,,\\
\quad (\tau_1' = \mathit{unsigned\text{-}int}) \vee (\tau_2' = \mathit{unsigned\text{-}int}) \to \mathit{unit}\ \mathit{unsigned\text{-}int}\,,\\
\quad (\tau_1' = \mathit{int}) \wedge (\tau_2' = \mathit{int}) \to \mathit{unit}\ \mathit{int}\,,\ \mathit{error})))
\end{array}
$$

This function performs the usual arithmetic conversions to its arguments, as specified in §6.2.1.5 of the standard. An error occurs if the usual arithmetic conversions cannot be applied.

▶ $argPromote \;:\; \mathbf{Type}_{dat} \to \mathsf{E}(\mathbf{Type}_{dat})$ — DEFAULT ARGUMENT PROMOTIONS

$$
\begin{array}{l}
argPromote \;=\; strict_{\mathrm{I}}\,(\lambda\,\tau.\ \mathbf{case}\ \tau\ \mathbf{of}\\
\quad \mathit{float} \;\Rightarrow\; \mathit{unit}\ \mathit{double}\\
\quad \mathbf{otherwise} \;\Rightarrow\; intPromote\ \tau)
\end{array}
$$

This function performs the default argument promotions, as specified in §6.3.2.2 of the standard.

▶ $datify_{obj} \;:\; \mathbf{Type}_{obj} \to \mathsf{E}(\mathbf{Type}_{dat})$ — DATIFY

$$
\begin{array}{l}
datify_{obj} \;=\; strict_{\mathrm{I}}\,(\lambda\,\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of}\\
\quad \mathit{obj}\,[\tau, q] \;\Rightarrow\; \mathit{unit}\ \tau\\
\quad \mathit{array}\,[\alpha, n] \;\Rightarrow\; \mathit{error})
\end{array}
$$

▶ $datify_{den} \;:\; \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Type}_{dat})$

$$
\begin{array}{l}
datify_{den} \;=\; strict_{\mathrm{I}}\,(\lambda\,\phi.\ \mathbf{case}\ \phi\ \mathbf{of}\\
\quad \alpha \;\Rightarrow\; datify_{obj}\ \alpha\\
\quad f \;\Rightarrow\; \mathit{error})
\end{array}
$$

▶ $datify_{mem} \;:\; \mathbf{Type}_{mem} \to \mathsf{E}(\mathbf{Type}_{dat})$

$$
\begin{array}{l}
datify_{mem} \;=\; strict_{\mathrm{I}}\,(\lambda\,m.\ \mathbf{case}\ m\ \mathbf{of}\\
\quad \alpha \;\Rightarrow\; datify_{obj}\ \alpha\\
\quad \mathit{bitfield}\,[\beta, q, n] \;\Rightarrow\; \mathit{unit}\ (datify_{bit}\ \beta))
\end{array}
$$

▶ $datify_{bit} \;:\; \mathbf{Type}_{bit} \to \mathbf{Type}_{dat}$

$$
\begin{array}{l}
datify_{bit} \;=\; strict_{\mathrm{I}}\,(\lambda\,\beta.\ \mathbf{case}\ \beta\ \mathbf{of}\\
\quad \mathit{int} \;\Rightarrow\; \mathit{int}\\
\quad \mathit{signed\text{-}int} \;\Rightarrow\; \mathit{int}\\
\quad \mathit{unsigned\text{-}int} \;\Rightarrow\; \mathit{unsigned\text{-}int})
\end{array}
$$

This function extracts a data type from other kinds of types. An error occurs if a data type cannot be extracted, e.g. if the argument is an array or function type. The part of the function that converts from bit-field types succeeds for all possible arguments. For this reason, it returns a data type, instead of a monadic element.

### 4.6.3 Compatible and composite types

Compatible types, as defined in §6.1.2.6 of the standard and elsewhere, define a binary relation between types. This relation is represented by function *isCompatible*. Function *composite*, also defined in §6.1.2.6 and elsewhere, maps a pair of compatible types to a third type, compatible with both, which shares the characteristics of both types. The two functions *isCompatibleQual* and *compositeQual* are similar, with the exception that they effectively ignore qualifiers when checking for compatibility. All functions are bi-strict with respect to bottom and top elements.

IS COMPATIBLE

The relation *isCompatible* for data types is partly implementation-defined and for this reason it is not fully specified here. Some properties that must be satisfied are given instead.

▶ $isCompatible_{dat} : \mathbf{Type}_{dat} \times \mathbf{Type}_{dat} \to \mathbf{T}$

$isCompatible_{dat}(\tau, \tau) = true$
$isCompatible_{dat}(\tau_1, \tau_2) = isCompatible_{dat}(\tau_2, \tau_1)$
$isCompatible_{dat}(ptr\,[\phi_1], ptr\,[\phi_2]) = isCompatible_{den}(\phi_1, \phi_2)$
$\forall\, \epsilon.\ \exists\, \tau.\ isInteger(\tau) \wedge isCompatible_{dat}(enum\,[\epsilon], \tau)$

The relation is reflexive and commutative, as stated by the first two properties. Compatibility of pointer types is defined in terms of the types that are pointed. For each enumeration type, there is a compatible integer type. This treatment of compatible types omits the case of structure, union or enumerated types that are defined in separate translation units, as stated by deviation D-11 in Section 2.3. The same relation for other kinds of types is defined as follows:

▶ $isCompatible_{obj} : \mathbf{Type}_{obj} \times \mathbf{Type}_{obj} \to \mathbf{T}$

$$\begin{aligned} isCompatible_{obj} = {} & bi\text{-}strict_{\mathrm{I}}\ (\lambda\, \langle \alpha_1, \alpha_2 \rangle.\ \mathbf{case}\ \langle \alpha_1, \alpha_2 \rangle\ \mathbf{of} \\ & \langle obj\,[\tau_1, q_1], obj\,[\tau_2, q_2] \rangle \Rightarrow isCompatible_{dat}(\tau_1, \tau_2) \wedge (q_1 = q_2) \\ & \langle array\,[\alpha_1, n_1], array\,[\alpha_2, n_2] \rangle \Rightarrow isCompatible_{obj}(\alpha_1, \alpha_2) \wedge (n_1 \sqcup n_2 \neq \top) \\ & \mathbf{otherwise} \Rightarrow false) \end{aligned}$$

▶ $isCompatible_{fun} : \mathbf{Type}_{fun} \times \mathbf{Type}_{fun} \to \mathbf{T}$

$$\begin{aligned} isCompatible_{fun} = {} & bi\text{-}strict_{\mathrm{I}}\ (\lambda\, \langle f_1, f_2 \rangle.\ \mathbf{case}\ \langle f_1, f_2 \rangle\ \mathbf{of} \\ & \langle func\,[\tau_1, p_1], func\,[\tau_2, p_2] \rangle \Rightarrow isCompatible_{dat}(\tau_1, \tau_2) \wedge isCompatible_{Prot}(p_1, p_2)) \end{aligned}$$

▶ $isCompatible_{den} : \mathbf{Type}_{den} \times \mathbf{Type}_{den} \to \mathbf{T}$

$$\begin{aligned} isCompatible_{den} = {} & bi\text{-}strict_{\mathrm{I}}\ (\lambda\, \langle \phi_1, \phi_2 \rangle.\ \mathbf{case}\ \langle \phi_1, \phi_2 \rangle\ \mathbf{of} \\ & \langle \alpha_1, \alpha_2 \rangle \Rightarrow isCompatible_{obj}(\alpha_1, \alpha_2) \\ & \langle f_1, f_2 \rangle \Rightarrow isCompatible_{fun}(f_1, f_2) \\ & \mathbf{otherwise} \Rightarrow false) \end{aligned}$$

▶ $isCompatible_{mem} : \mathbf{Type}_{mem} \times \mathbf{Type}_{mem} \to \mathbf{T}$

$$\begin{aligned} isCompatible_{mem} = {} & bi\text{-}strict_{\mathrm{I}}\ (\lambda\, \langle m_1, m_2 \rangle.\ \mathbf{case}\ \langle m_1, m_2 \rangle\ \mathbf{of} \\ & \langle \alpha_1, \alpha_2 \rangle \Rightarrow isCompatible_{obj}(\alpha_1, \alpha_2) \\ & \langle bitfield\,[\beta_1, q_1, n_1], bitfield\,[\beta_2, q_2, n_2] \rangle \Rightarrow (\beta_1 = \beta_2) \wedge (q_1 = q_2) \wedge (n_1 = n_2) \\ & \mathbf{otherwise} \Rightarrow false) \end{aligned}$$

▶ $isCompatible_{Prot} \;:\; \mathbf{Prot} \times \mathbf{Prot} \to \mathbf{T}$

$$\begin{array}{l} isCompatible_{Prot} \;=\; bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle p_1, p_2\rangle. \\ \quad \mathbf{let}\ \langle n_1, m_1, b_1\rangle = p_1 \\ \quad\quad \langle n_2, m_2, b_2\rangle = p_2 \\ \quad \mathbf{in}\ (n_1 = n_2)\ \wedge\ (b_1 = b_2)\ \wedge\ (\forall i : 1 \le i \le n_1.\ isCompatible_{dat}(m_1\, i, m_2\, i))) \end{array}$$

When comparing arrays for compatibility, the least upper bound of array sizes is top when both sizes have been specified and they are different. Notice here that the presence of qualifiers destroys compatibility.

IS COMPATIBLE IGNORING QUALIFIERS

▶ $isCompatibleQual_{dat} \;:\; \mathbf{Type}_{dat} \times \mathbf{Type}_{dat} \to \mathbf{T}$

$$isCompatibleQual_{dat}(\tau, \tau) \;=\; isCompatible_{dat}(\tau, \tau)$$

▶ $isCompatibleQual_{obj} \;:\; \mathbf{Type}_{obj} \times \mathbf{Type}_{obj} \to \mathbf{T}$

$$\begin{array}{l} isCompatibleQual_{obj} \;=\; bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle m_1, m_2\rangle.\ \mathbf{case}\ \langle m_1, m_2\rangle\ \mathbf{of} \\ \quad \langle obj\,[\tau_1, q_1], obj\,[\tau_2, q_2]\rangle \;\Rightarrow\; isCompatible_{dat}(\tau_1, \tau_2) \\ \quad \langle array\,[\alpha_1, n_1], array\,[\alpha_2, n_2]\rangle \;\Rightarrow\; isCompatibleQual_{obj}(\alpha_1, \alpha_2)\ \wedge\ (n_1 \sqcup n_2 \neq \top) \\ \quad \mathbf{otherwise} \;\Rightarrow\; false) \end{array}$$

▶ $isCompatibleQual_{fun} \;:\; \mathbf{Type}_{fun} \times \mathbf{Type}_{fun} \to \mathbf{T}$

$$isCompatibleQual_{fun}(f_1, f_2) \;=\; isCompatible_{fun}(f_1, f_2)$$

▶ $isCompatibleQual_{den} \;:\; \mathbf{Type}_{den} \times \mathbf{Type}_{den} \to \mathbf{T}$

$$\begin{array}{l} isCompatibleQual_{den} \;=\; bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle \phi_1, \phi_2\rangle.\ \mathbf{case}\ \langle \phi_1, \phi_2\rangle\ \mathbf{of} \\ \quad \langle \alpha_1, \alpha_2\rangle \;\Rightarrow\; isCompatibleQual_{obj}(\alpha_1, \alpha_2) \\ \quad \langle f_1, f_2\rangle \;\Rightarrow\; isCompatibleQual_{fun}(f_1, f_2) \\ \quad \mathbf{otherwise} \;\Rightarrow\; false) \end{array}$$

▶ $isCompatibleQual_{mem} \;:\; \mathbf{Type}_{mem} \times \mathbf{Type}_{mem} \to \mathbf{T}$

$$\begin{array}{l} isCompatibleQual_{mem} \;=\; bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle m_1, m_2\rangle.\ \mathbf{case}\ \langle m_1, m_2\rangle\ \mathbf{of} \\ \quad \langle \alpha_1, \alpha_2\rangle \;\Rightarrow\; isCompatibleQual_{obj}(\alpha_1, \alpha_2) \\ \quad \langle bitfield\,[\beta_1, q_1, n_1], bitfield\,[\beta_2, q_2, n_2]\rangle \;\Rightarrow\; (\beta_1 = \beta_2)\ \wedge\ (n_1 = n_2) \\ \quad \mathbf{otherwise} \;\Rightarrow\; false) \end{array}$$

This relation is similar to *isCompatible*, only the presence of qualifiers in the compared types does not destroy compatibility. It does however when there are members of structures or unions that are qualified with different qualifiers.

COMPOSITE TYPES

▶ $composite_{dat} \;:\; \mathbf{Type}_{dat} \times \mathbf{Type}_{dat} \to \mathsf{E}(\mathbf{Type}_{dat})$

$$\begin{array}{l} composite_{dat} \;=\; bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle \tau_1, \tau_2\rangle.\ \mathbf{case}\ \langle \tau_1, \tau_2\rangle\ \mathbf{of} \\ \quad \langle ptr\,[\phi_1], ptr\,[\phi_2]\rangle \;\Rightarrow\; composite_{den}(\phi_1, \phi_2)\ *\ (\lambda\phi.\ unit\ ptr\,[\phi]) \\ \quad \langle enum\,[\epsilon], \tau\rangle \mid \langle \tau, enum\,[\epsilon]\rangle \mid \;\Rightarrow\; isCompatible_{dat}(enum\,[\epsilon], \tau) \to unit\ \tau\,,\ error \\ \quad \mathbf{otherwise} \;\Rightarrow\; (\tau_1 = \tau_2) \to unit\ \tau_1\,,\ error) \end{array}$$

▶ $composite_{obj} \;:\; \mathbf{Type}_{obj} \times \mathbf{Type}_{obj} \to \mathsf{E}(\mathbf{Type}_{obj})$

$$\begin{array}{l} composite_{obj} \;=\; bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle \alpha_1, \alpha_2\rangle.\ \mathbf{case}\ \langle \alpha_1, \alpha_2\rangle\ \mathbf{of} \\ \quad \langle obj\,[\tau_1, q_1], obj\,[\tau_2, q_2]\rangle \;\Rightarrow\; (q_1 = q_2) \to \\ \qquad\qquad composite_{dat}(\tau_1, \tau_2)\ *\ (\lambda\tau.\ unit\ obj\,[\tau, q_1])\,,\ error \\ \quad \langle array\,[\alpha_1, n_1], array\,[\alpha_2, n_2]\rangle \;\Rightarrow\; (n_1 \sqcup n_2 \neq \top) \to \\ \qquad\qquad composite_{obj}(\alpha_1, \alpha_2)\ *\ (\lambda\alpha.\ unit\ array\,[\alpha, n_1 \sqcup n_2])\,,\ error \\ \quad \mathbf{otherwise} \;\Rightarrow\; error) \end{array}$$

▶ $composite_{fun} \;:\; \mathbf{Type}_{fun} \times \mathbf{Type}_{fun} \to \mathsf{E}(\mathbf{Type}_{fun})$

$$\begin{aligned}
composite_{fun} \;=\; & bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle f_1, f_2\rangle.\ \mathbf{case}\ \langle f_1, f_2\rangle\ \mathbf{of}\\
& \langle func\,[\tau_1, p_1], func\,[\tau_2, p_2]\rangle \Rightarrow composite_{dat}(\tau_1, \tau_2)\ *\ (\lambda\tau.\\
& \qquad composite_{Prot}(p_1, p_2)\ *\ (\lambda p.\ unit\ func\,[\tau, p])))
\end{aligned}$$

▶ $composite_{den} \;:\; \mathbf{Type}_{den} \times \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Type}_{den})$

$$\begin{aligned}
composite_{den} \;=\; & bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle \phi_1, \phi_2\rangle.\ \mathbf{case}\ \langle \phi_1, \phi_2\rangle\ \mathbf{of}\\
& \langle \alpha_1, \alpha_2\rangle \Rightarrow composite_{obj}(\alpha_1, \alpha_2)\\
& \langle f_1, f_2\rangle \Rightarrow composite_{fun}(f_1, f_2)\\
& \mathbf{otherwise} \Rightarrow error)
\end{aligned}$$

▶ $composite_{mem} \;:\; \mathbf{Type}_{mem} \times \mathbf{Type}_{mem} \to \mathsf{E}(\mathbf{Type}_{mem})$

$$\begin{aligned}
composite_{mem} \;=\; & bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle m_1, m_2\rangle.\ \mathbf{case}\ \langle m_1, m_2\rangle\ \mathbf{of}\\
& \langle \alpha_1, \alpha_2\rangle \Rightarrow composite_{obj}(\alpha_1, \alpha_2)\\
& \langle bitfield\,[\beta_1, q_1, n_1], bitfield\,[\beta_2, q_2, n_2]\rangle \Rightarrow (\beta_1 = \beta_2) \wedge (q_1 = q_2) \wedge (n_1 = n_2) \to\\
& \qquad unit\ bitfield\,[\beta_1, q_1, n_1],\ error\\
& \mathbf{otherwise} \Rightarrow error)
\end{aligned}$$

▶ $composite_{Prot} \;:\; \mathbf{Prot} \times \mathbf{Prot} \to \mathsf{E}(\mathbf{Prot})$

$$\begin{aligned}
composite_{Prot} \;=\; & bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle p_1, p_2\rangle.\\
& \mathbf{let}\ \langle n_1, m_1, b_1\rangle = p_1\\
& \quad \langle n_2, m_2, b_2\rangle = p_2\\
& \mathbf{in}\ (n_1 = n_2) \wedge (b_1 = b_2) \to\\
& \qquad traverse\,[1, n_1]\,(\lambda i.\ composite_{dat}(m_1\ i, m_2\ i))\ \top\ *\ (\lambda m.\ unit\ \langle n_1, m, b_1\rangle),\ error)
\end{aligned}$$

This function is used to determine composite types. According to §6.1.2.6 of the standard, it just combines array sizes and function parameter lists.

COMPOSITE TYPES IGNORING QUALIFIERS

▶ $compositeQual_{dat} \;:\; \mathbf{Type}_{dat} \times \mathbf{Type}_{dat} \to \mathsf{E}(\mathbf{Type}_{dat})$

$$compositeQual_{dat}(\tau, \tau) \;=\; composite_{dat}(\tau, \tau)$$

▶ $compositeQual_{obj} \;:\; \mathbf{Type}_{obj} \times \mathbf{Type}_{obj} \to \mathsf{E}(\mathbf{Type}_{obj})$

$$\begin{aligned}
compositeQual_{obj} \;=\; & bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle \alpha_1, \alpha_2\rangle.\ \mathbf{case}\ \langle \alpha_1, \alpha_2\rangle\ \mathbf{of}\\
& \langle obj\,[\tau_1, q_1], obj\,[\tau_2, q_2]\rangle \Rightarrow composite_{dat}(\tau_1, \tau_2)\ *\ (\lambda\tau.\ unit\ obj\,[\tau, q_1\ \&\ q_2])\\
& \langle array\,[\alpha_1, n_1], array\,[\alpha_2, n_2]\rangle \Rightarrow (n_1 \sqcup n_2 \neq \top) \to\\
& \qquad compositeQual_{obj}(\alpha_1, \alpha_2)\ *\ (\lambda\alpha.\\
& \qquad unit\ array\,[\alpha, n_1 \sqcup n_2]),\ error\\
& \mathbf{otherwise} \Rightarrow error)
\end{aligned}$$

▶ $compositeQual_{fun} \;:\; \mathbf{Type}_{fun} \times \mathbf{Type}_{fun} \to \mathsf{E}(\mathbf{Type}_{fun})$

$$compositeQual_{fun}(f_1, f_2) \;=\; composite_{fun}(f_1, f_2)$$

▶ $compositeQual_{den} \;:\; \mathbf{Type}_{den} \times \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Type}_{den})$

$$\begin{aligned}
compositeQual_{den} \;=\; & bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle \phi_1, \phi_2\rangle.\ \mathbf{case}\ \langle \phi_1, \phi_2\rangle\ \mathbf{of}\\
& \langle \alpha_1, \alpha_2\rangle \Rightarrow compositeQual_{obj}(\alpha_1, \alpha_2)\\
& \langle f_1, f_2\rangle \Rightarrow compositeQual_{fun}(f_1, f_2)\\
& \mathbf{otherwise} \Rightarrow error)
\end{aligned}$$

▶ $compositeQual_{mem} \;:\; \mathbf{Type}_{mem} \times \mathbf{Type}_{mem} \to \mathsf{E}(\mathbf{Type}_{mem})$

$$\begin{array}{l} compositeQual_{mem} \;=\; bi\text{-}strict_{\mathrm{I}}\,(\lambda\langle m_1, m_2\rangle.\ \mathbf{case}\ \langle m_1, m_2\rangle\ \mathbf{of} \\ \quad \langle \alpha_1, \alpha_2\rangle \qquad\qquad\qquad\qquad \Rightarrow\ compositeQual_{obj}\,(\alpha_1, \alpha_2) \\ \quad \langle bitfield\,[\beta_1, q_1, n_1], bitfield\,[\beta_2, q_2, n_2]\rangle \ \Rightarrow\ (\beta_1 = \beta_2) \wedge (n_1 = n_2) \to \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad unit\ bitfield\,[\beta_1, q_1\ \&\ q_2, n_1]\,,\ error \\ \quad \mathbf{otherwise} \qquad\qquad\qquad\qquad \Rightarrow\ error) \end{array}$$

This function is similar to $compositeQual$, with the exception that it also combines qualifiers.

### 4.6.4 Functions related to qualifiers

▶ $\cdot\ \&\ \cdot \;:\; \mathbf{Qual} \times \mathbf{Qual} \to \mathbf{Qual}$ COMBINE QUALIFIERS

| $\cdot\ \&\ \cdot$ | $\bot$ | *noqual* | *const* | *volatile* | *const-volatile* | $\top$ |
|---|---|---|---|---|---|---|
| $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\top$ |
| *noqual* | $\bot$ | *noqual* | *const* | *volatile* | *const-volatile* | $\top$ |
| *const* | $\bot$ | *const* | *const* | *const-volatile* | *const-volatile* | $\top$ |
| *volatile* | $\bot$ | *volatile* | *const-volatile* | *volatile* | *const-volatile* | $\top$ |
| *const-volatile* | $\bot$ | *const-volatile* | *const-volatile* | *const-volatile* | *const-volatile* | $\top$ |
| $\top$ | $\top$ | $\top$ | $\top$ | $\top$ | $\top$ | $\top$ |

This function is bi-strict with respect to bottom and top elements and combines two qualifiers.

▶ $\cdot \subseteq \cdot \;:\; \mathbf{Qual} \times \mathbf{Qual} \to \mathbf{T}$ QUALIFIER INCLUSION

$$q_1 \subseteq q_2 \;=\; (q_1\ \&\ q_2 = q_2)$$

This binary relation between qualifiers checks whether its first argument is included in its second argument.

▶ $getQualifier_{obj} \;:\; \mathbf{Type}_{obj} \to \mathbf{Qual}$ GET QUALIFIER

$$\begin{array}{l} getQualifier_{obj} \;=\; strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of} \\ \quad obj\,[\tau, q] \;\Rightarrow\ q \\ \quad array\,[\alpha, n] \;\Rightarrow\ getQualifier_{obj}\ \alpha) \end{array}$$

▶ $getQualifier_{den} \;:\; \mathbf{Type}_{den} \to \mathbf{Qual}$

$$\begin{array}{l} getQualifier_{den} \;=\; strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of} \\ \quad \alpha \;\Rightarrow\ getQualifier_{obj}\ \alpha \\ \quad f \;\Rightarrow\ noqual) \end{array}$$

▶ $getQualifier_{mem} \;:\; \mathbf{Type}_{mem} \to \mathbf{Qual}$

$$\begin{array}{l} getQualifier_{mem} \;=\; strict_{\mathrm{I}}\,(\lambda m.\ \mathbf{case}\ m\ \mathbf{of} \\ \quad \alpha \qquad\qquad\quad \Rightarrow\ getQualifier_{obj}\ \alpha \\ \quad bitfield\,[\beta, q, n] \;\Rightarrow\ q) \end{array}$$

This function extracts the qualifier of a type. Function types are not qualified.

▶ $qualify_{obj} \;:\; \mathbf{Qual} \to \mathbf{Type}_{obj} \to \mathsf{E}(\mathbf{Type}_{obj})$ QUALIFY

$$\begin{array}{l} qualify_{obj} \;=\; strict_{\mathrm{I}}\,(\lambda q.\ strict_{\mathrm{I}}\,(\lambda\alpha.\ \mathbf{case}\ \alpha\ \mathbf{of} \\ \quad obj\,[\tau, q'] \;\Rightarrow\ unit\ obj\,[\tau, q\ \&\ q'] \\ \quad array\,[\alpha, n] \;\Rightarrow\ qualify_{obj}\ q\ \alpha\ *\ (\lambda\alpha'.\ unit\ array\,[\alpha', n]))) \end{array}$$

▶ $qualify_{den} : \mathbf{Qual} \to \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Type}_{den})$

$$\begin{aligned} qualify_{den} \;=\; & strict_{\mathrm{I}}\,(\lambda q.\; strict_{\mathrm{I}}\,(\lambda \phi.\; \mathbf{case}\ \phi\ \mathbf{of} \\ & \quad \alpha \;\Rightarrow\; qualify_{obj}\ q\ \alpha \\ & \quad f \;\Rightarrow\; error)) \end{aligned}$$

▶ $qualify_{mem} : \mathbf{Qual} \to \mathbf{Type}_{mem} \to \mathsf{E}(\mathbf{Type}_{mem})$

$$\begin{aligned} qualify_{mem} \;=\; & strict_{\mathrm{I}}\,(\lambda q.\; strict_{\mathrm{I}}\,(\lambda \phi.\; \mathbf{case}\ \phi\ \mathbf{of} \\ & \quad \alpha \;\Rightarrow\; qualify_{obj}\ q\ \alpha \\ & \quad bitfield\,[\beta, q', n] \;\Rightarrow\; unit\ bitfield\,[\beta, q\ \&\ q', n])) \end{aligned}$$

This function returns the $q$-qualified version of its second argument. An error occurs if a function type is used.

### 4.6.5 Miscellaneous functions

FIX PARAMETER

▶ $fix\text{-}parameter : \mathbf{Type}_{den} \to \mathbf{Type}_{dat} \times \mathbf{Qual}$

$$\begin{aligned} fix\text{-}parameter \;=\; & strict_{\mathrm{I}}\,(\lambda \phi.\; \mathbf{case}\ \phi\ \mathbf{of} \\ & \quad obj\,[\tau, q] \;\Rightarrow\; \langle \tau, q \rangle \\ & \quad array\,[\alpha, n] \;\Rightarrow\; \langle ptr\,[\alpha], noqual \rangle \\ & \quad func\,[\tau, p] \;\Rightarrow\; \langle ptr\,[func\,[\tau, p]], noqual \rangle) \end{aligned}$$

This function fixes the type of a formal parameter by converting parameters of array or function types to pointers.

FRESH TAGS

▶ $fresh\text{-}tagged : \mathbf{Ide} \to \mathbf{Tag}$

▶ $fresh\text{-}untagged : \mathbf{Tag}$

These two functions return fresh tags, named and nameless respectively. A fresh tag is meant to be distinct from all other existing tags. The implementation of these functions is omitted.

SIZES OF TYPES

▶ $sizeof_{dat} : \mathbf{Type}_{dat} \to \mathbf{N}$

▶ $sizeof_{obj} : \mathbf{Type}_{obj} \to \mathbf{N}$

▶ $sizeof_{mem} : \mathbf{Type}_{mem} \to \mathbf{N}$

These are implementation-defined functions, returning the size in bytes of a given data, object and member type. They are strict with respect to bottom and top elements.

SIZE OF BYTE

▶ $bits\text{-}in\text{-}byte : \mathbf{N}$

This is an implementation-defined constant element, representing the number of bits in a byte. Its value must satisfy a number of requirements stated in the standard.

IS VALID BIT-FIELD SIZE

▶ $isValidBitfieldSize : \mathbf{N} \to \mathbf{T}$

$$isValidBitfieldSize \;=\; strict_{\mathrm{I}}\,(\lambda n.\; (n \geq 0) \wedge (n \leq sizeof(int) \cdot bits\text{-}in\text{-}byte))$$

This predicate takes an integer number and returns *true* if this can be used as the size of a bit-field, *false* otherwise. Bit-field sizes must be non-negative and not larger than the number of bits in an integer. The returned values must satisfy a number of requirements stated in the standard.

SYNTAX OF CONSTANTS

- $suffix$ : *constant* $\rightarrow$ *character* **set**
- $prefix$ : *constant* $\rightarrow$ *character* **set**
- $isDecimal$ : *integer-constant* $\rightarrow \mathbf{T}$
- $isStringLit$ : *expression* $\rightarrow \mathbf{T}$
- $isWideStringLit$ : *expression* $\rightarrow \mathbf{T}$
- $lengthOf$ : *string-literal* $\rightarrow \mathbf{N}$

These functions are related to the syntax of C constants. Function $suffix$ returns the set of characters that are contained in the suffix of a constant; valid suffix characters are “F”, “L” and “U”. Function $prefix$ acts similarly for constant prefixes; the only allowed prefix is “L”. The next three predicates return $true$ if their argument is a decimal integer constant, a string literal or a wide string literal, respectively. Finally, function $lengthOf$ returns the length of a string (or wide string) literal in characters (or wide characters).

TRAVERSE

- $traverse\,[\cdot,\cdot]$ : $\mathbf{N} \times \mathbf{N} \rightarrow (\mathbf{N} \rightarrow \mathsf{E}(A)) \rightarrow (\mathbf{N} \rightarrow A) \rightarrow \mathsf{E}(\mathbf{N} \rightarrow A)$

  $traverse\,[a,b] \;=\; \lambda f.\, \lambda m.\, (a \leq b) \rightarrow f\ a\ *\ (\lambda \tau.\ traverse\,[a+1,b]\ f\ m\{a \mapsto \tau\})\,,\ unit\ m$

This recursively defined function traverses function $m : \mathbf{N} \rightarrow A$ over the range $[a..b]$ and applies function $f : \mathbf{N} \rightarrow \mathsf{E}(A)$ to each element in this range. If no errors occur from any of these applications, $traverse$ returns a new function that contains the results of the applications in the range $[a..b]$ and agrees with $m$ outside this range. Otherwise, if any errors occur, $traverse$ generates an error.

# Chapter 5

# Static semantics of declarations

CHAPTER OVERVIEW

This chapter defines the static semantics of C declarations, including external declarations and translation units. In Section 5.1 the notions of semantic functions and equations are presented and the adopted notation is defined. The static semantics of external declarations and normal declarations is defined in Section 5.2 and Section 5.3 respectively.

## 5.1 Static semantic functions and equations

SEMANTIC FUNCTIONS

The static semantics of the C programming language result in the construction of *static environments*, which mainly contain type information about program identifiers. These environments are used in the following stages of the semantic description, namely typing and dynamic semantics. Following the denotational approach, the static semantics of the C programming language is defined by a number of *static semantic functions.*

In general, a static semantic function maps well formed phrases, as these are specified by the abstract syntax of C, to static semantic domains. For the construction of static environments, it is possible to process only the parts of a C program that contain *declarations*, since these contain all the necessary type information about program identifiers.[1] It is reasonable, therefore, to restrict the definition of static semantics to declarations. For each non-terminal symbol used in the grammar that defines the abstract syntax of declarations (see Section 2.2.1), a static semantic function is defined which maps well formed phrases generated by this non-terminal to an appropriately chosen static semantic domain. This function is denoted by $\{\!|\ \cdot\ |\!\}$.

SEMANTIC EQUATIONS

The definition of a static semantic function is given by a number of *semantic equations.* In the abstract syntax, non-terminal symbols are defined in rules containing one or more alternative clauses. For each such clause, the definition of the static semantic function requires the presence of a semantic equation. Therefore, there will be as many semantic equations as there are alternative clauses for the non-terminal in the abstract syntax. The right-hand side of each semantic equation may use the static semantic meanings of subphrases, as expressed by the abstract syntax, respecting thus the compositional character of denotational semantics. The definition of semantic equations may also be inductive, based on the length of syntactic productions.

NOTATION EXAMPLE

This small example clarifies the notation that will be used for the definition of static semantic functions using equations. Consider the hypothetical non-terminal symbol *non-term*, defined in the abstract syntax by the following rule:

[1] This approach depends on deviation D-4 in Section 2.3, which guarantees that no program identifiers, other than labels, can be defined in expressions or statements and, consequently, all type information can be gathered from declarations.

♦ *non-term* ::= `term` *non-term* | `(` *other-non-term* `)` | $\epsilon$

where `term` and the parentheses are terminal symbols and *other-non-term* is another non-terminal symbol. Notice that the definition of *non-term* is recursive, as specified by the first alternative clause. Suppose that an appropriate static semantic domain for the range of *non-term*'s is $D$. Therefore, the static semantic function for *non-term* maps well formed phrases generated by *non-term* to elements of $D$. Similarly, suppose that $D'$ is the range of the static semantic function for *other-non-term.* The definition of the static semantic function for *non-term* can given as follows:

▶ $\{\!|$*non-term*$|\!\}$ : $D$

$\{\!|$`term` *non-term*$|\!\}$ = *(expression of type D, which may use expression* $\{\!|$*non-term*$|\!\}$ *of type D)*

$\{\!|$`(` *other-non-term* `)`$|\!\}$ = *(expression of type D, which may use expression* $\{\!|$*other-non-term*$|\!\}$ *of type D′)*

$\{\!|\epsilon|\!\}$ = *(constant expression of type D)*

The first line states that the static meaning of *non-term* is represented by elements of domain $D$. The next three lines are the semantic equations defining $\{\!|$*non-term*$|\!\}$. They correspond directly to the alternative clauses in the abstract syntax of *non-term.* A similar definition must be given for the non-terminal symbol *other-non-term.*

MULTIPLE STATIC MEANINGS

There are some cases where the static meaning of a phrase must provide more than one pieces of information. This can be achieved by using a product domain as the range of the semantic function; this approach is followed in many cases in the semantic description that follows. However, in some cases there is clearly one prominent static meaning, which is used most of the time, and one or more other meanings which are rarely used. In this cases, it is convenient to split the semantic function in two or more functions, each one returning a different aspect of the whole static meaning. The function that corresponds to the prominent meaning can be still denoted by $\{\!|\cdot|\!\}$ for brevity. Functions corresponding to alternative meanings are denoted by prepending caligraphic letters, as in $\mathcal{X}\{\!|\cdot|\!\}$. Each function is defined separately, using a separate set of semantic equations.

## 5.2 External declarations

TRANSLATION UNITS

▶ $\{\!|$*translation-unit*$|\!\}$ : $\mathsf{E}(\mathbf{Ent})$

$\{\!|$*external-declaration-list*$|\!\}$ = $rec$ $\{\!|$*external-declaration-list*$|\!\}$ $e_\circ$

The static meaning of a translation unit is defined as the type environment that reflects all the external declarations that the translation unit contains. The environment that is used as a basis, i.e. before any external declarations have taken place, is the empty type environment. The presence of function $rec$ is explained in Chapter 6. For the time being, let it be said that $rec$ allows the type environment to contain recursively defined types. The reader may want to treat it as the identity function, if that helps his or her understanding.

EXTERNAL DECLARATION LISTS

▶ $\{\!|$*external-declaration-list*$|\!\}$ : $\mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$

$\{\!|$*external-declaration*$|\!\}$ = $\{\!|$*external-declaration*$|\!\}$

$\{\!|$*external-declaration external-declaration-list*$|\!\}$ =
$\{\!|$*external-declaration*$|\!\}$ ; $\{\!|$*external-declaration-list*$|\!\}$

The static meaning of an external declaration list is a function from environments to environments.[2] The result of this function is the type environment that results from applying the list of external declarations to an initial environment, which is given as the function's argument. Notice the monadic inverse composition operator ";" in the second semantic equation, which passes the result of its first operand as an argument to its second operand.

EXTERNAL DECLARATIONS

▶ $\{\!|\textit{external-declaration}|\!\} \;:\; \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$

$\{\!|\textit{declaration}|\!\} \;=\; \{\!|\textit{declaration}|\!\}$

$\{\!|\textit{declaration-specifiers declarator } \{ \textit{ declaration-list statement-list } \}|\!\} \;=\; \lambda e.$
$\quad \{\!|\textit{declaration-specifiers}|\!\}\ e \;*\; (\lambda \langle \phi, u_s, e_u, e_n \rangle.$
$\quad \{\!|\textit{declarator}|\!\}\ e_u\ \phi \;*\; (\lambda \langle I, \phi' \rangle.\ \mathbf{case}\ \phi'\ \mathbf{of}$
$\qquad \mathit{func}\,[\tau, p] \;\Rightarrow\; u_s\ e_u\ \langle I, \phi' \rangle$
$\qquad \mathbf{otherwise} \;\Rightarrow\; \mathit{error}))$

The static meaning of an external declaration is again a function from environments to environments, with the same behaviour as before. If the external declaration is simply a declaration, its meaning is simply used. Otherwise, if it is a function definition, the function's identifier and type are calculated in $I$ and $\phi'$ respectively, and if it is really a function, the environment is updated.[3]

## 5.3 Declarations

DECLARATION LISTS

▶ $\{\!|\textit{declaration-list}|\!\} \;:\; \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$

$\{\!|\epsilon|\!\} \;=\; \mathit{unit}$

$\{\!|\textit{declaration declaration-list}|\!\} \;=\; \{\!|\textit{declaration}|\!\}\ ;\ \{\!|\textit{declaration-list}|\!\}$

The static meaning of a declaration list and the corresponding semantic function are very similar to those of external declaration lists.

DECLARATIONS

▶ $\{\!|\textit{declaration}|\!\} \;:\; \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$

$\{\!|\textit{declaration-specifiers init-declarator-list ;}|\!\} \;=\; \{\!|\textit{declaration-specifiers}|\!\}\ ;\ \{\!|\textit{init-declarator-list}|\!\}$

Still the same static meaning for declarations. Notice that the semantic functions for *declaration-specifiers* and *init-declarator-list*, which are defined below, match in such a way as to allow such a simple semantic equation for the declaration.

DECLARATION SPECIFIERS

▶ $\{\!|\textit{declaration-specifiers}|\!\} \;:$
$\quad \mathbf{Ent} \to \mathsf{E}(\mathbf{Type}_{den} \times (\mathbf{Ent} \to \mathbf{Ide} \times \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Ent})) \times \mathbf{Ent} \times \mathbf{Ent})$

$\{\!|\textit{storage-class-specifier type-qualifier type-specifier}|\!\} \;=\; \lambda e.$
$\quad \{\!|\textit{type-specifier}|\!\}\ e \;*\; (\lambda \langle \phi, e_u, e_n \rangle.$
$\quad \{\!|\textit{type-qualifier}|\!\}\ \phi \;*\; (\lambda \phi'.\ \mathit{unit}\ \langle \phi', \{\!|\textit{storage-class-specifier}|\!\}, e_u, e_n \rangle))$

[2] The presence of the error monad $\mathsf{E}$ in the semantic functions simply indicates that an error may occur when the static semantics is determined. For brevity, the error monad is overlooked by the textual descriptions in the sequel. Without this simplifying convention, the phrase "function from environments to environments" would have been "function from environments to environments, allowing for errors".

[3] This approach allows the use of the *typedef* storage specifier in function definitions, as stated by deviation D-9 in Section 2.3.

The static meaning of declaration specifiers is a function that is somewhat more complicated than the previous ones. The function's argument is the initial type environment. The result is a product which contains: the type that is specified; the meaning of the storage specifier that is applied; and two new type environments which encorporate the effect of the declaration specifiers.

The reason why there are two such environments, instead of one, is the behaviour of the declaration "`struct tag;`" which, as stated in §6.5.2.3 of the standard, defines a new structure tag for an incomplete structure type and overrides any definitions of the same tag in enclosing scopes. On the contrary, "`struct tag x;`" does not have the same effect: if the tag has already been defined in an enclosing scope, the previous definition is used and no new structure tag is defined. The meaning of the declaration specifier, in this case "`struct tag`", should clearly incorporate both effects. Thus, the first element in the aforementioned pair is the environment that will result if the subsequent *init-declarator-list* is not empty, whereas the second element is the result if it is empty. In most declaration specifiers the two environments are equal.

STORAGE SPECIFIERS

▶ $\{\!|\textit{storage-class-specifier}|\!\} \;:\; \mathbf{Ent} \to \mathbf{Ide} \times \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Ent})$

$\{\!|\epsilon|\!\} \;=\; \lambda e.\, \lambda \langle I, \phi \rangle.\, isComplete(\phi) \to e[I \mapsto ide\ normal\ [\phi]],\ error$

$\{\!|\texttt{typedef}|\!\} \;=\; \lambda e.\, \lambda \langle I, \phi \rangle.\, e[I \mapsto ide\ typedef\ [\phi]]$

The static meaning of a storage specifier is a function whose first argument is the initial type environment and its second argument is a pair, containing the identifier $I$ that is defined and its type $\phi$.[4] The result is the type environment after $I$ is appropriately associated with $\phi$.[5] Only complete types can be used with the empty storage specifier.[6]

### 5.3.1 Basic types and qualifiers

TYPE QUALIFIERS

▶ $\{\!|\textit{type-qualifier}|\!\} \;:\; \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Type}_{den})$

$\{\!|\epsilon|\!\} \;=\; qualify\ noqual$

$\{\!|\texttt{const}|\!\} \;=\; qualify\ const$

$\{\!|\texttt{volatile}|\!\} \;=\; qualify\ volatile$

$\{\!|\texttt{const volatile}|\!\} \;=\; qualify\ const\text{-}volatile$

The static meaning of qualifiers is a function from denotable types to denotable types. The result is simply the qualified version of the function's argument.

[4] Again, a simplifying convention is used in the textual descriptions of higher-order functions, such as the meaning of storage specifiers. A function $f : D_1 \to D_2 \to \ldots \to D_n \to D$ is considered in its informal description to have $n$ arguments of types $D_i$. The reader should be warned that, for brevity, a function $f : D_1 \times D_2 \times \ldots \times D_n \to D$ is also considered as a function of $n$ arguments of types $D_i$.

[5] This approach is reasonable for the empty storage specifier and *typedef*, which are the only ones considered according to deviation D-5 in Section 2.3.

[6] This results from deviation D-2 in Section 2.3.

▶ $\{\!|\textit{type-specifier}|\!\} : \mathbf{Ent} \to \mathsf{E}(\mathbf{Type}_{den} \times \mathbf{Ent} \times \mathbf{Ent})$ — TYPE SPECIFIERS

$\{\!|\texttt{void}|\!\} = \lambda e.\ unit\ \langle obj\ [void, noqual], e, e\rangle$
$\{\!|\texttt{char}|\!\} = \lambda e.\ unit\ \langle obj\ [char, noqual], e, e\rangle$
$\{\!|\texttt{signed char}|\!\} = \lambda e.\ unit\ \langle obj\ [signed\text{-}char, noqual], e, e\rangle$
$\{\!|\texttt{unsigned char}|\!\} = \lambda e.\ unit\ \langle obj\ [unsigned\text{-}char, noqual], e, e\rangle$
$\{\!|\texttt{short int}|\!\} = \lambda e.\ unit\ \langle obj\ [short\text{-}int, noqual], e, e\rangle$
$\{\!|\texttt{unsigned short int}|\!\} = \lambda e.\ unit\ \langle obj\ [unsigned\text{-}short\text{-}int, noqual], e, e\rangle$
$\{\!|\texttt{int}|\!\} = \lambda e.\ unit\ \langle obj\ [int, noqual], e, e\rangle$
$\{\!|\texttt{unsigned int}|\!\} = \lambda e.\ unit\ \langle obj\ [unsigned\text{-}int, noqual], e, e\rangle$
$\{\!|\texttt{long int}|\!\} = \lambda e.\ unit\ \langle obj\ [long\text{-}int, noqual], e, e\rangle$
$\{\!|\texttt{unsigned long int}|\!\} = \lambda e.\ unit\ \langle obj\ [unsigned\text{-}long\text{-}int, noqual], e, e\rangle$
$\{\!|\texttt{float}|\!\} = \lambda e.\ unit\ \langle obj\ [float, noqual], e, e\rangle$
$\{\!|\texttt{double}|\!\} = \lambda e.\ unit\ \langle obj\ [double, noqual], e, e\rangle$
$\{\!|\texttt{long double}|\!\} = \lambda e.\ unit\ \langle obj\ [long\text{-}double, noqual], e, e\rangle$
$\{\!|\textit{struct-specifier}|\!\} = \{\!|\textit{struct-specifier}|\!\}$
$\{\!|\textit{union-specifier}|\!\} = \{\!|\textit{union-specifier}|\!\}$
$\{\!|\textit{enum-specifier}|\!\} = \{\!|\textit{enum-specifier}|\!\}$
$\{\!|\textit{typedef-name}|\!\} = \lambda e.\ \{\!|\textit{typedef-name}|\!\}\ e\ *\ (\lambda \phi.\ unit\ \langle \phi, e, e\rangle)$

The static meaning of a type specifier is a function that takes the initial environment as an argument and returns a triple. The first returned element is the specified type, whereas the second and third elements are the updated type environments (see the discussion in Section 5.3 where the meaning of declaration specifiers was defined). The semantic equations are straighforward, with the exception of structure, union and enumeration specifiers which are treated separately in following sections.

### 5.3.2 Initializations

▶ $\{\!|\textit{init-declarator-list}|\!\} : \mathbf{Type}_{den} \times (\mathbf{Ent} \to \mathbf{Ide} \times \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Ent})) \times \mathbf{Ent} \times \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$ — DECLARATOR LISTS WITH INITIALIZERS

$\{\!|\epsilon|\!\} = \lambda \langle \phi, u_s, e_u, e_n\rangle.\ unit\ e_n$
$\{\!|\textit{init-declarator init-declarator-list}|\!\} = \lambda \langle \phi, u_s, e_u, e_n\rangle.$
$\quad \{\!|\textit{init-declarator}|\!\}\ e_u\ \phi\ *\ (\lambda \langle I, \phi'\rangle.$
$\quad u_s\ e_u\ \langle I, \phi'\rangle\ *\ (\lambda e'.\ \{\!|\textit{init-declarator-list}|\!\}\ \langle \phi, u_s, e', e'\rangle))$

The static meaning of a declarator list with initializers is used in connection with that of declaration specifiers, as discussed in Section 5.3. Its arguments are: $\phi$, the specified type; $u_s$, the method for updating the environment; $e_u$, the initial environment if at least one identifier is contained in the list; and $e_n$, the initial environment if the list is empty. The result is the updated environment. Notice that when the list is empty $e_n$ is simply returned, otherwise it is not used at all.

▶ $\{\!|\textit{init-declarator}|\!\} : \mathbf{Ent} \times \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Ide} \times \mathbf{Type}_{den})$ — DECLARATORS WITH INITIALIZERS

$\{\!|\textit{declarator}|\!\} = \{\!|\textit{declarator}|\!\}$
$\{\!|\textit{declarator = initializer}|\!\} = \lambda \langle e, \phi\rangle.$
$\quad \{\!|\textit{declarator}|\!\}\ \langle e, \phi\rangle\ *\ (\lambda \langle I, \phi'\rangle.$
$\quad \{\!|\textit{initializer}|\!\}\ e\ \phi'\ *\ (\lambda \phi''.\ unit\ \langle I, \phi''\rangle))$

The static meaning of a declarator with initializer is a function mapping the current environment $e$ and the specified type $\phi$ to a pair, which contains the declared identifier and a (possibly) updated type. The

only case when the updated type is different from $\phi$ is when $\phi$ is an incomplete array type and the initializer specifies the number of its elements. In this case, the updated type is a complete array type.

INITIALIZERS ▶ $\{\!|\mathit{initializer}|\!\} : \mathbf{Ent} \to \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Type}_{den})$

$$\begin{aligned}
&\{\!|\mathit{expression}|\!\} = \lambda e.\, \lambda \phi.\, \mathbf{case}\ \phi\ \mathbf{of} \\
&\quad \mathit{array}\,[\phi', \bot] \Rightarrow (\mathit{isCompatible}(\phi', \mathit{obj}\,[\mathit{char}, \mathit{noqual}]) \wedge \mathit{isStringLit}(\mathit{expression})) \vee \\
&\qquad\qquad (\mathit{isCompatible}(\phi', \mathit{obj}\,[\mathit{wchar_t}, \mathit{noqual}]) \wedge \mathit{isWideStringLit}(\mathit{expression})) \\
&\qquad\qquad \to \mathit{unit}\ \mathit{array}\,[\phi', \mathit{lengthOf}(\mathit{expression})],\ \mathit{unit}\ \phi \\
&\quad \mathbf{otherwise} \Rightarrow \mathit{unit}\ \phi \\
&\{\!|\{\ \mathit{initializer\text{-}list}\ \}|\!\} = \lambda e.\, \lambda \phi.\, \mathbf{case}\ \phi\ \mathbf{of} \\
&\quad \mathit{array}\,[\phi', \bot] \Rightarrow \{\!|\mathit{initializer\text{-}list}|\!\}\ e\ \phi'\ *\ (\lambda n.\ \mathit{unit}\ \mathit{array}\,[\phi', n]) \\
&\quad \mathbf{otherwise} \Rightarrow \mathit{unit}\ \phi
\end{aligned}$$

The static meaning of an initializer is a function taking the environment and the specified type and returning a (possibly) updated type, as discussed before. Notice that arrays can be initialized by string or wide string literals, or by an explicit list of initializers.

INITIALIZER LISTS ▶ $\{\!|\mathit{initializer\text{-}list}|\!\} : \mathbf{Ent} \to \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{N})$

$$\begin{aligned}
&\{\!|\mathit{initializer}|\!\} = \lambda e.\, \lambda \phi.\, \mathit{unit}\ 1 \\
&\{\!|\mathit{initializer}\ ,\ \mathit{initializer\text{-}list}|\!\} = \lambda e.\, \lambda \phi.\, \{\!|\mathit{initializer\text{-}list}|\!\}\ e\ \phi\ *\ (\lambda n.\ \mathit{unit}\ (1 + n))
\end{aligned}$$

The static meaning of an initializer list is a function returning the number of elements in the list.[7]

### 5.3.3 Structures and unions

Structure and union specifiers are probably the most complicated aspect to be defined in the semantics of the type system of C. This is due to the presence of incomplete structure and union types and the related intricacies in the standard.

STRUCTURE SPECIFIERS ▶ $\{\!|\mathit{struct\text{-}specifier}|\!\} : \mathbf{Ent} \to \mathsf{E}(\mathbf{Type}_{den} \times \mathbf{Ent} \times \mathbf{Ent})$

$$\begin{aligned}
&\{\!|\texttt{struct}\ I\ \{\ \mathit{struct\text{-}declaration\text{-}list}\ \}|\!\} = \lambda e. \\
&\quad e[I \mapsto \mathit{fresh\ tag\ tag\text{-}struct}]\ *\ (\lambda e_c. \\
&\quad e_c[I\ \mathit{tagID\ tag\text{-}struct}]\ *\ (\lambda t. \\
&\quad \{\!|\mathit{struct\text{-}declaration\text{-}list}|\!\}\ \langle e_c, \pi_\circ\rangle\ *\ (\lambda \langle e', \pi\rangle. \\
&\quad \mathbf{let}\ \tau = \mathit{struct}\,[t, \pi] \\
&\quad \mathbf{in}\ e'[I \mapsto \mathit{tag}\ \tau]\ *\ (\lambda e_f.\ \mathit{unit}\ \langle \mathit{obj}\,[\tau, \mathit{noqual}], e_f, e_f\rangle)))) \\
&\{\!|\texttt{struct}\ \{\ \mathit{struct\text{-}declaration\text{-}list}\ \}|\!\} = \lambda e. \\
&\quad \{\!|\mathit{struct\text{-}declaration\text{-}list}|\!\}\ \langle e, \pi_\circ\rangle\ *\ (\lambda \langle e', \pi\rangle. \\
&\quad \mathit{unit}\ \langle \mathit{obj}\,[\mathit{struct}\,[\mathit{fresh\text{-}untagged}, \pi], \mathit{noqual}], e', e'\rangle) \\
&\{\!|\texttt{struct}\ I|\!\} = \lambda e. \\
&\quad e[I\ \mathit{tag\ tag\text{-}struct}]\ *\ (\lambda \langle e', \tau\rangle.\ e[I \mapsto \mathit{fresh\ tag\ tag\text{-}struct}]\ *\ (\lambda e_c.\ \mathit{unit}\ \langle \mathit{obj}\,[\tau, \mathit{noqual}], e', e_c\rangle))
\end{aligned}$$

The static meaning of a structure specifier is a function with the same behaviour as the one for normal type specifiers. The definition of this function is rather complicated and, for this reason, it is described in detail. We distinguish between three cases of structure specifiers, as suggested by the abstract syntax.

[7] The arguments of this function are ignored in this implementation, because of deviation D-4 in Section 2.3. Also, notice that deviation D-6 disallows the use of partially bracketted initializer lists.

- *Definition of a tagged structure.* This is the most complicated case. Starting with the initial environment $e$, a fresh structure tag is generated, resulting in $e_c$. The fresh tag, which contains an incomplete type, is necessary in order to hide definitions of the same tag in enclosing scopes.[8] Let $t$ be this fresh tag. Starting with the empty member environment, the structure declaration list is processed and results in the correct member environment $\pi$ for the structure. The (possibly) updated type environment that results from this process is denoted by $e'$. Finally, the complete type updates the tag in $e'$, resulting in $e_f$, and the correct final result is returned.

- *Definition of an untagged structure.* In this case, it is not necessary to define a fresh tag, since untagged structure definitions cannot be recursive. The structure declaration list is again processed, starting from the empty member environment. The result is the correct member environment $\pi$ and the (possibly) updated type environment $e'$. Finally, the structure type is put in $e'$ and is associated with a fresh unnamed tag.

- *Declaration or use of a tagged structure.* This is the only actual case where the two returned environments are different, as discussed in Section 5.3. The first step is to lookup for the named tag in the initial environment, with the side effect that a fresh tag is created, if it does not exist locally or in any enclosed scope. Let $\tau$ be its type and $e'$ the (possibly) updated type environment. Obviously, $e'$ is the resulting environment in the case "`struct tag x;`", i.e. when the *init-declarator-list* is not empty. On the other hand, a fresh tag must be generated in the case "`struct tag;`", i.e. when the *init-declarator-list* is empty. This results in $e_c$, which is the second type environment to be returned.

▶ $\{\!|$*union-specifier*$|\!\}$ : $\mathbf{Ent} \to \mathsf{E}(\mathbf{Type}_{den} \times \mathbf{Ent} \times \mathbf{Ent})$ — UNION SPECIFIERS

$\{\!|$`union` $I$ `{` *struct-declaration-list* `}`$|\!\}$ $=$ $\lambda e.$
$e[I \mapsto \mathit{fresh\ tag\ tag\text{-}union}] * (\lambda e_c.$
$e_c[I\ \mathit{tagID\ tag\text{-}union}] * (\lambda t.$
$\{\!|$*struct-declaration-list*$|\!\}\ \langle e_c, \pi_\circ \rangle * (\lambda \langle e', \pi \rangle.$
**let** $\tau = \mathit{union}\,[t, \pi]$
**in** $e'[I \mapsto \mathit{tag}\ \tau] * (\lambda e_f.\ \mathit{unit}\ \langle \mathit{obj}\,[\tau, \mathit{noqual}], e_f, e_f \rangle)))))$

$\{\!|$`union {` *struct-declaration-list* `}`$|\!\}$ $=$ $\lambda e.$
$\{\!|$*struct-declaration-list*$|\!\}\ \langle e, \pi_\circ \rangle * (\lambda \langle e', \pi \rangle.$
$\mathit{unit}\ \langle \mathit{obj}\,[\mathit{union}\,[\mathit{fresh\text{-}untagged}, \pi], \mathit{noqual}], e', e' \rangle)$

$\{\!|$`union` $I$$|\!\}$ $=$ $\lambda e.$
$e[I\ \mathit{tag\ tag\text{-}union}] * (\lambda \langle e', \tau \rangle.\ e[I \mapsto \mathit{fresh\ tag\ tag\text{-}union}] * (\lambda e_c.\ \mathit{unit}\ \langle \mathit{obj}\,[\tau, \mathit{noqual}], e', e_c \rangle))$

The static meaning for union specifiers, as well as its implementation, is very similar to that of structure specifiers.

▶ $\{\!|$*struct-declaration-list*$|\!\}$ : $\mathbf{Ent} \times \mathbf{Memb} \to \mathsf{E}(\mathbf{Ent} \times \mathbf{Memb})$ — STRUCTURE DECLARATION LISTS

$\{\!|$*struct-declaration*$|\!\}$ $=$ $\{\!|$*struct-declaration*$|\!\}$
$\{\!|$*struct-declaration struct-declaration-list*$|\!\}$ $=$ $\{\!|$*struct-declaration*$|\!\}$ ; $\{\!|$*struct-declaration-list*$|\!\}$

The static meaning of structure declaration lists is simply a function updating the member environment and the type environment. Its definition is straightforward.

STRUCTURE DECLARATIONS

▶ $\{\!|\textit{struct-declaration}|\!\} \;:\; \mathbf{Ent} \times \mathbf{Memb} \to \mathsf{E}(\mathbf{Ent} \times \mathbf{Memb})$

$\{\!|\textit{struct-specifiers struct-declarator-list}|\!\} \;=\; \lambda\langle e, \pi\rangle.$
$\quad \{\!|\textit{struct-specifiers}|\!\}\; e \;*\; (\lambda\langle \phi, e'\rangle.\; \{\!|\textit{struct-declarator-list}|\!\}\; e'\; \phi\; \pi \;*\; (\lambda\, \pi'.\; \mathit{unit}\; \langle e', \pi'\rangle)))$

The static meaning of structure declaration lists is again a function updating the member environment and the type environment. The latter is updated by processing the *struct-specifiers*, where as the former by processing the *struct-declarator-list*.

STRUCTURE SPECIFIERS

▶ $\{\!|\textit{struct-specifiers}|\!\} \;:\; \mathbf{Ent} \to \mathsf{E}(\mathbf{Type}_{den} \times \mathbf{Ent})$

$\{\!|\textit{type-qualifier type-specifier}|\!\} \;=\; \lambda\, e.$
$\quad \{\!|\textit{type-specifier}|\!\}\; e \;*\; (\lambda\langle \phi, e_u, e_n\rangle.\; \{\!|\textit{type-qualifier}|\!\}\; \phi \;*\; (\lambda\, \phi'.\; \mathit{unit}\; \langle \phi', e_u\rangle))$

Structure specifiers can be viewed as a special case of declaration specifiers, with two exceptions which affect the semantic function. First, no storage specifiers are allowed and the returning product is greatly simplified. Second, only one resulting environment is necessary, because there can be no tag declarations with empty declarator lists.

STRUCTURE DECLARATOR LISTS

▶ $\{\!|\textit{struct-declarator-list}|\!\} \;:\; \mathbf{Ent} \to \mathbf{Type}_{den} \to \mathbf{Memb} \to \mathsf{E}(\mathbf{Memb})$

$\{\!|\textit{struct-declarator}|\!\} \;=\; \lambda\, e.\; \lambda\, \phi.\; \lambda\, \pi.$
$\quad \{\!|\textit{struct-declarator}|\!\}\; e\; \phi \;*\; (\lambda\langle I, \phi'\rangle.\; \pi[I \mapsto \phi'])$
$\{\!|\textit{struct-declarator struct-declarator-list}|\!\} \;=\; \lambda\, e.\; \lambda\, \phi.\; \lambda\, \pi.$
$\quad \{\!|\textit{struct-declarator}|\!\}\; e\; \phi \;*\; (\lambda\langle I, \phi'\rangle.\; \pi[I \mapsto \phi'] \;*\; (\lambda\, \pi'.\; \{\!|\textit{struct-declarator-list}|\!\}\; e\; \phi\; \pi'))$

The static meaning of a structure declarator list is a function mapping the current type environment, the specified type and the initial member environment to an updated member environment. Its definition is straightforward.

STRUCTURE DECLARATORS

▶ $\{\!|\textit{struct-declarator}|\!\} \;:\; \mathbf{Ent} \to \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Ide} \times \mathbf{Type}_{mem})$

$\{\!|\textit{declarator}|\!\} \;=\; \lambda\, e.\; \lambda\, \phi.$
$\quad \{\!|\textit{declarator}|\!\}\; e\; \phi \;*\; (\lambda\langle I, \phi'\rangle.$
$\quad \neg \mathit{isComplete}(\phi') \to \mathit{error}\;,\; \mathbf{case}\; \phi'\; \mathbf{of}$
$\qquad \alpha \qquad\qquad \Rightarrow\; \mathit{unit}\; \langle I, \alpha\rangle$
$\qquad \mathbf{otherwise}\; \Rightarrow\; \mathit{error})$
$\{\!|\textit{declarator} : \textit{constant-expression}|\!\} \;=\; \lambda\, e.\; \lambda\, \phi.$
$\quad \{\!|\textit{declarator}|\!\}\; e\; \phi \;*\; (\lambda\langle I, \phi'\rangle.$
$\quad \mathcal{IC}[\![\textit{constant-expression}]\!]\; e \;*\; (\lambda\, n.$
$\quad \neg \mathit{isValidBitfieldSize}(n) \to \mathit{error}\;,\; \mathbf{case}\; \phi'\; \mathbf{of}$
$\qquad \mathit{obj}\,[\mathit{int}, q] \qquad\qquad \Rightarrow\; \mathit{unit}\; \langle I, \mathit{bitfield}\,[\mathit{int}, q, n]\rangle$
$\qquad \mathit{obj}\,[\mathit{unsigned\text{-}int}, q] \;\Rightarrow\; \mathit{unit}\; \langle I, \mathit{bitfield}\,[\mathit{unsigned\text{-}int}, q, n]\rangle$
$\qquad \mathbf{otherwise} \qquad\qquad \Rightarrow\; \mathit{error}))$

The static meaning of a structure declarator is a function taking as arguments the current type environment and the specified type. It returns the declared identifier and the associated member type. Notice that in the case of ordinary fields only complete object types are allowed. Furthermore, in the case of bit-fields only *int*, and *unsigned int* are allowed.[9] The dynamic meaning function $\mathcal{IC}[\![\ \cdot\ ]\!]$ simply returns the value of a constant expression and will be defined in Part IV.

[8] It is also necessary that a fresh tag is visible in the structure declaration list, to allow recursively defined structures.

[9] Note that *signed int* bit-fields cannot be defined using this approach. The problem is due to the fact that *signed int* and *int* are identified as the same type in $\mathbf{Type}_{obj}$. See deviation D-7 in Section 2.3.

### 5.3.4 Enumerations

ENUMERATION SPECIFIERS

▶ $\{\!|\textit{enum-specifier}|\!\} : \mathbf{Ent} \to \mathsf{E}(\mathbf{Type}_{den} \times \mathbf{Ent} \times \mathbf{Ent})$

$\{\!|\mathsf{enum}\ I\ \{\ \textit{enumerator-list}\ \}|\!\} = \lambda e.$
$\quad \{\!|\textit{enumerator-list}|\!\}\ \langle e, \epsilon_\circ, 0\rangle\ *\ (\lambda\langle e', \epsilon, n\rangle.$
$\quad \mathbf{let}\ \tau = enum\ [\epsilon]$
$\quad \mathbf{in}\ \ e'[I \mapsto tag\ \tau]\ *\ (\lambda e_f.\ unit\ \langle obj\ [\tau, noqual], e_f, e_f\rangle)))$

$\{\!|\mathsf{enum}\ \{\ \textit{enumerator-list}\ \}|\!\} = \lambda e.$
$\quad \{\!|\textit{enumerator-list}|\!\}\ \langle e, \epsilon_\circ, 0\rangle\ *\ (\lambda\langle e', \epsilon, n\rangle.$
$\quad unit\ \langle obj\ [enum\ [\epsilon], noqual], e', e'\rangle)$

$\{\!|\mathsf{enum}\ I|\!\} = \lambda e.\ e[I\ tag\ tag\text{-}enum]\ *\ (\lambda\langle e', \tau\rangle.\ unit\ \langle obj\ [\tau, noqual], e, e\rangle)$

The static meaning of an enumeration specifier is again a function with the same behaviour as the one for normal type specifiers. The standard forbids the declaration of incomplete enumerations and, therefore, the two returned environments are equal. There are again three cases of enumeration specifiers but the definition of their meanings is relatively straightforward. Notice that the values of enumerators start with the number zero.

ENUMERATOR LISTS

▶ $\{\!|\textit{enumerator-list}|\!\} : \mathbf{Ent} \times \mathbf{Enum} \times \mathbf{N} \to \mathsf{E}(\mathbf{Ent} \times \mathbf{Enum} \times \mathbf{N})$

$\{\!|\textit{enumerator}|\!\} = \{\!|\textit{enumerator}|\!\}$

$\{\!|\textit{enumerator enumerator-list}|\!\} = \{\!|\textit{enumerator}|\!\}\ ;\ \{\!|\textit{enumerator-list}|\!\}$

The static meaning of an enumerator list is a function updating the type environment, the enumeration environment and the constant value that will be given to the next enumeration constant. Its definition is straghtforward.

ENUMERATORS

▶ $\{\!|\textit{enumerator}|\!\} : \mathbf{Ent} \times \mathbf{Enum} \times \mathbf{N} \to \mathsf{E}(\mathbf{Ent} \times \mathbf{Enum} \times \mathbf{N})$

$\{\!|I|\!\} = \lambda\langle e, \epsilon, n\rangle.$
$\quad e[I \mapsto ide\ enum\text{-}const\ [n]]\ *\ (\lambda e'.\ \epsilon[I \mapsto n]\ *\ (\lambda\epsilon'.\ unit\ \langle e', \epsilon', n+1\rangle))$

$\{\!|I = \textit{constant-expression}|\!\} = \lambda\langle e, \epsilon, n\rangle.$
$\quad \mathcal{IC}[\![\textit{constant-expression}]\!]\ e\ *\ (\lambda m.$
$\quad e[I \mapsto ide\ enum\text{-}const\ [m]]\ *\ (\lambda e'.\ \epsilon[I \mapsto m]\ *\ (\lambda\epsilon'.\ unit\ \langle e', \epsilon', m+1\rangle))))$

The static meaning of enumerators is the same as that of enumerator lists. Just notice that if a constant expression is specified as the value for the enumeration constant, its value is used instead of the parameter and numbering continues from the successor of this value.

### 5.3.5 Declarators

DECLARATORS ▶ $\{\!|\textit{declarator}|\!\} : \mathbf{Ent} \to \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Ide} \times \mathbf{Type}_{den})$

$\{\!|I|\!\} = \lambda e.\, \lambda \phi.\, unit\ \langle I, \phi \rangle$

$\{\!|\textit{declarator [ constant-expression ]}|\!\} = \lambda e.\, \lambda \phi.\, \mathbf{case}\ \phi\ \mathbf{of}$
$\quad \alpha \Rightarrow \mathcal{IC}[\![\textit{constant-expression}]\!]\ e\ *\ (\lambda n.$
$\quad\quad (n > 0) \to \{\!|\textit{declarator}|\!\}\ e\ array\,[\alpha, n],\ error)$
$\quad \mathbf{otherwise} \Rightarrow error$

$\{\!|\textit{declarator [ ]}|\!\} = \lambda e.\, \lambda \phi.\, \mathbf{case}\ \phi\ \mathbf{of}$
$\quad \alpha \Rightarrow \{\!|\textit{declarator}|\!\}\ e\ array\,[\alpha, \bot]$
$\quad \mathbf{otherwise} \Rightarrow error$

$\{\!|\textit{* type-qualifier declarator}|\!\} = \lambda e.\, \lambda \phi.$
$\quad \{\!|\textit{type-qualifier}|\!\}\ obj\,[ptr\,[\phi], noqual]\ *\ (\lambda \phi'.\, \{\!|\textit{declarator}|\!\}\ e\ \phi')$

$\{\!|\textit{declarator ( parameter-type-list )}|\!\} = \lambda e.\, \lambda \phi.$
$\quad rec\ (\mathcal{F}\{\!|\textit{parameter-type-list}|\!\})\ (\uparrow e)\ *\ (\lambda e_p.$
$\quad \{\!|\textit{parameter-type-list}|\!\}\ e_p\ p_\circ\ *\ (\lambda p.\, \mathbf{case}\ \phi\ \mathbf{of}$
$\quad\quad obj\,[\tau, q] \Rightarrow \{\!|\textit{declarator}|\!\}\ e\ func\,[\tau, p]$
$\quad\quad \mathbf{otherwise} \Rightarrow error))$

The primary static meaning of a declarator is a function taking as arguments the current type environment and the specified type. It returns the declared identifier and its type. Its definition is relatively straightforward. In array declarators, if a constant value is specified within the brackets, it is evaluated using the dynamic semantic function $\mathcal{IC}[\![\ \cdot\ ]\!]$. In function declarators, a new scope is opened for the function prototype and the type environment $e_p$ that corresponds to it is calculated. The parameter type list is then processed.

DECLARATORS $\mathcal{T}$-MEANING ▶ $\mathcal{T}\{\!|\textit{declarator}|\!\} : \mathbf{T}$

$\mathcal{T}\{\!|I|\!\} = true$
$\mathcal{T}\{\!|\textit{declarator [ constant-expression ]}|\!\} = false$
$\mathcal{T}\{\!|\textit{declarator [ ]}|\!\} = false$
$\mathcal{T}\{\!|\textit{* type-qualifier declarator}|\!\} = false$
$\mathcal{T}\{\!|\textit{declarator ( parameter-type-list )}|\!\} = false$

This alternative is an auxiliary static meaning for declarators, represented by a truth value. It is *true* if the declarator consists of a single identifier, *false* otherwise. This meaning is only used to determine whether a declarator is a leave at the abstract syntax tree.

DECLARATORS $\mathcal{F}$-MEANING ▶ $\mathcal{F}\{\!|\textit{declarator}|\!\} : \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$

$\mathcal{F}\{\!|I|\!\} = \lambda e.\, error$
$\mathcal{F}\{\!|\textit{declarator [ constant-expression ]}|\!\} = \mathcal{F}\{\!|\textit{declarator}|\!\}$
$\mathcal{F}\{\!|\textit{declarator [ ]}|\!\} = \mathcal{F}\{\!|\textit{declarator}|\!\}$
$\mathcal{F}\{\!|\textit{* type-qualifier declarator}|\!\} = \mathcal{F}\{\!|\textit{declarator}|\!\}$
$\mathcal{F}\{\!|\textit{declarator ( parameter-type-list )}|\!\} = (\mathcal{T}\{\!|\textit{declarator}|\!\}) \to \mathcal{F}\{\!|\textit{parameter-type-list}|\!\},\ \mathcal{F}\{\!|\textit{declarator}|\!\}$

This alternative static meaning for declarators is used for determining the type environment that is associated with function prototype scopes. It is a function from type environments to type environments. Assuming that the declarator is used for the declaration of a function, the argument is the initial type environment just before the function's parameters are declared. In other words, it is assumed that a new scope must already be open for the function definition and must be passed here. The result is the

initial type environment for the function's body, after the function's parameters have been declared. An error occurs if the declarator is not used for the definition of a function.

▶ $\{\!|\textit{parameter-type-list}|\!\}\ :\ \mathbf{Ent} \to \mathbf{Prot} \to \mathsf{E}(\mathbf{Prot})$ PARAMETER TYPE LISTS

$\{\!|\epsilon|\!\}\ =\ \lambda\, e.\ unit$
$\{\!|\ldots|\!\}\ =\ \lambda\, e.\ ellipsis$
$\{\!|\textit{parameter-declaration parameter-type-list}|\!\}\ =\ \lambda\, e.\ (\{\!|\textit{parameter-declaration}|\!\}\ e)\ ;\ (\{\!|\textit{parameter-type-list}|\!\}\ e)$

The static meaning of parameter type lists is a function taking the type environment for a function prototype scope and returning a function which updates the function prototype. Its definition is straightforward.

▶ $\mathcal{F}\{\!|\textit{parameter-type-list}|\!\}\ :\ \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$ PARAMETER TYPE LISTS $\mathcal{F}$-MEANING

$\mathcal{F}\{\!|\epsilon|\!\}\ =\ unit$
$\mathcal{F}\{\!|\ldots|\!\}\ =\ unit$
$\mathcal{F}\{\!|\textit{parameter-declaration parameter-type-list}|\!\}\ =\ \mathcal{F}\{\!|\textit{parameter-declaration}|\!\}\ ;\ \mathcal{F}\{\!|\textit{parameter-type-list}|\!\}$

This alternative static meaning for type parameter lists is used for determining the type environment that is associated with function prototype scopes. It is a function from type environments to type environments. The updated type environment contains the declaration of parameters contained in the parameter type list.

▶ $\{\!|\textit{parameter-declaration}|\!\}\ :\ \mathbf{Ent} \to \mathbf{Prot} \to \mathsf{E}(\mathbf{Prot})$ PARAMETER DECLARATIONS

$\{\!|\textit{declaration-specifiers declarator}|\!\}\ =\ \lambda\, e.\ \lambda\, p.$
$\quad \{\!|\textit{declaration-specifiers}|\!\}\ e\ *\ (\lambda\, \langle \phi, u_s, e_u, e_n\rangle.$
$\quad \{\!|\textit{declarator}|\!\}\ e\ \phi\ *\ (\lambda\, \langle I, \phi'\rangle.$
$\quad \mathbf{let}\ \langle \tau, q\rangle = \textit{fix-parameter}\ \phi'$
$\quad \mathbf{in}\ \ p \ll \tau))$
$\{\!|\textit{declaration-specifiers abstract-declarator}|\!\}\ =\ \lambda\, e.\ \lambda\, p.$
$\quad \{\!|\textit{declaration-specifiers}|\!\}\ e\ *\ (\lambda\, \langle \phi, u_s, e_u, e_n\rangle.$
$\quad \{\!|\textit{abstract-declarator}|\!\}\ e\ \phi\ *\ (\lambda\, \phi'.$
$\quad \mathbf{let}\ \langle \tau, q\rangle = \textit{fix-parameter}\ \phi'$
$\quad \mathbf{in}\ \ p \ll \tau))$

The static meaning of parameter declarations is similar to that of parameter declaration lists. Just notice two things. First, since the type environment $e$ has already been fixed, it already contains whatever identifiers may be defined because of *declaration-specifiers* and the results $e_u$ and $e_n$ need not be used . Second, *fix-parameter* is used, in order to translate array and function types to pointer types.

▶ $\mathcal{F}\{\!|\textit{parameter-declaration}|\!\}\ :\ \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$ PARAMETER DECLARATIONS $\mathcal{F}$-MEANING

$\mathcal{F}\{\!|\textit{declaration-specifiers declarator}|\!\}\ =\ \lambda\, e.$
$\quad \{\!|\textit{declaration-specifiers}|\!\}\ e\ *\ (\lambda\, \langle \phi, u_s, e_u, e_n\rangle.$
$\quad \{\!|\textit{declarator}|\!\}\ e_u\ \phi\ *\ (\lambda\, \langle I, \phi'\rangle.$
$\quad \mathbf{let}\ \langle \tau, q\rangle = \textit{fix-parameter}\ \phi'$
$\quad \mathbf{in}\ \ e_u[I \mapsto ide\ normal\ [obj\ [\tau, q]]]))$
$\mathcal{F}\{\!|\textit{declaration-specifiers abstract-declarator}|\!\}\ =\ \lambda\, e.$
$\quad \{\!|\textit{declaration-specifiers}|\!\}\ e\ *\ (\lambda\, \langle \phi, u_s, e_u, e_n\rangle.\ unit\ e_u)$

This alternative static meaning for parameter declarations is again used in for determining the type environment that is associated with function prototype scopes. It is similar to the corresponding meaning for parameter type lists.

ABSTRACT DECLARATORS

▶ $\{\!|\textit{abstract-declarator}|\!\} : \mathbf{Ent} \to \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Type}_{den})$

$\{\!|\epsilon|\!\} = \lambda e.\ unit$

$\{\!|\textit{abstract-declarator}\ [\ \textit{constant-expression}\ ]|\!\} = \lambda e.\ \lambda \phi.\ \mathbf{case}\ \phi\ \mathbf{of}$
$\quad \alpha \Rightarrow \mathcal{IC}[\![\textit{constant-expression}]\!]\ e\ *\ (\lambda n.$
$\quad\quad (n > 0) \to \{\!|\textit{abstract-declarator}|\!}\ e\ array\ [\alpha, n],\ error)$
$\quad \mathbf{otherwise} \Rightarrow error$

$\{\!|\textit{abstract-declarator}\ [\ ]|\!\} = \lambda e.\ \lambda \phi.\ \mathbf{case}\ \phi\ \mathbf{of}$
$\quad \alpha \Rightarrow \{\!|\textit{abstract-declarator}|\!}\ e\ array\ [\alpha, \bot]$
$\quad \mathbf{otherwise} \Rightarrow error$

$\{\!|*\ \textit{type-qualifier abstract-declarator}|\!\} = \lambda e.\ \lambda \phi.$
$\quad \{\!|\textit{type-qualifier}|\!}\ obj\ [ptr\ [\phi], noqual]\ *\ (\lambda \phi'.\ \{\!|\textit{abstract-declarator}|\!}\ e\ \phi')$

$\{\!|\textit{abstract-declarator}\ (\ \textit{parameter-type-list}\ )|\!\} = \lambda e.\ \lambda \phi.$
$\quad rec\ (\mathcal{F}\{\!|\textit{parameter-type-list}|\!\})\ (\uparrow e)\ *\ (\lambda e_p.$
$\quad \{\!|\textit{parameter-type-list}|\!}\ e_p\ p_\circ\ *\ (\lambda p.\ \mathbf{case}\ \phi\ \mathbf{of}$
$\quad\quad obj\ [\tau, q] \Rightarrow \{\!|\textit{abstract-declarator}|\!}\ e\ func\ [\tau, p]$
$\quad\quad \mathbf{otherwise} \Rightarrow error))$

The static meaning of abstract declarators is similar to that of declarators, with the exception that no identifier is returned. Its definition is similar also.

### 5.3.6 Type names

TYPE SYNONYMS

▶ $\{\!|\textit{typedef-name}|\!\} : \mathbf{Ent} \to \mathsf{E}(\mathbf{Type}_{den})$

$\{\!|I|\!\} = \lambda e.\ e[I\ ide]\ *\ (\lambda \delta.\ \mathbf{case}\ \delta\ \mathbf{of}$
$\quad typedef\ [\phi] \Rightarrow unit\ \phi$
$\quad \mathbf{otherwise} \Rightarrow error)$

The static meaning of a type synonym, i.e. an identifier declared with the *typedef* storage specifier, is simply a function from the current type environment to the type denoted by the synonym. An error occurs if the identifier is not a type synonym.

TYPE NAMES

▶ $\{\!|\textit{type-name}|\!\} : \mathbf{Ent} \to \mathsf{E}(\mathbf{Type}_{den})$

$\{\!|\textit{type-qualifier type-specifier abstract-declarator}|\!\} = \lambda e.$
$\quad \{\!|\textit{type-specifier}|\!}\ e\ *\ (\lambda \langle \phi, e_u, e_n \rangle.\ \{\!|\textit{type-qualifier}|\!}\ \phi\ *\ (\lambda \phi'.\ \{\!|\textit{abstract-declarator}|\!}\ e_u\ \phi'))$

The static meaning of a type name is a function from the current type environment to the type denoted by the type name. Notice that, while processing the type name, the type environment is possibly updated.[10] However, this updated environment is discarded, in accordance to deviation D-4 in Section 2.3.

[10] To be precise, the only modification in the type environment that is allowed by the standard at this point is the definition of a structure, union or enumeration tag.

# Chapter 6

# Static semantics of recursively defined types

CHAPTER OVERVIEW

This chapter deals with the static semantics of recursively defined types. Section 6.1 identifies a number of problems in the present static semantic by studying examples of recursively defined types. It also motivates the changes that take place in the following sections. In Section 6.2, the domain **Ent** that represents type environments is revisited and replaced by a modified version, suitable for recursively defined types. Finally, in Section 6.3 the process of fixing type environments is formally defined.

## 6.1 Some examples

SIMPLE RECURSION

The simplest example of a recursively defined structure type is given in the following declaration:

```
struct tag { struct tag * p; };
```

that is, a structure type containing a pointer to a similar structure. The static semantics of C, as defined in the previous chapters, can only associate *"tag"* with type $\tau$, where:

$$\begin{aligned} \tau &= \mathit{struct}\,[\mathit{tagged}\,[\text{“}\mathit{tag}\text{”}, 1], \{\,\text{“}\mathit{p}\text{”} \mapsto \mathit{obj}\,[\mathit{ptr}\,[\mathit{obj}\,[\tau_0, \mathit{noqual}]], \mathit{noqual}]\,\}] \\ \tau_0 &= \mathit{struct}\,[\mathit{tagged}\,[\text{“}\mathit{tag}\text{”}, 1], \bot] \end{aligned}$$

where 1 is the unique number associated with this tag. Note that $\tau_0$ is the incomplete type associated with the fresh tag that is declared upon entering the structure declaration list. Clearly this is not the intended static meaning of this declaration. The structure's member *"p"* should not be a pointer to an incomplete structure, but a pointer to the complete structure type containing it. It is easy to see that the present static semantics fails to correctly represent recursively defined types.

LEAST FIXED POINTS

A correct static meaning for this declaration must associate *"tag"* with type $\tau$, where $\tau$ is the least solution of the equation:

$$\tau = \mathit{struct}\,[\mathit{tagged}\,[\text{“}\mathit{tag}\text{”}, 1], \{\,\text{“}\mathit{p}\text{”} \mapsto \mathit{obj}\,[\mathit{ptr}\,[\mathit{obj}\,[\tau, \mathit{noqual}]], \mathit{noqual}]\,\}]$$

Since $\tau$ is an element of domain $\mathbf{Type}_{dat}$, it possible to express it by means of the least fixed point operator, applied to a function from $\mathbf{Type}_{dat}$ to $\mathbf{Type}_{dat}$:

$$\mathit{fix}\,(\lambda\,\tau.\ \mathit{struct}\,[\mathit{tagged}\,[\text{“}\mathit{tag}\text{”}, 1], \{\,\text{“}\mathit{p}\text{”} \mapsto \mathit{obj}\,[\mathit{ptr}\,[\mathit{obj}\,[\tau, \mathit{noqual}]], \mathit{noqual}]\,\}]$$

MUTUAL RECURSION

In a similar way it is possible to correctly define the static semantics of mutually recursive type definitions. Consider the following declaration list, defining two structure tags *"tagx"* and *"tagy"*:

```
struct tagx { struct tagy * py; };
struct tagy { struct tagx * px; };
```

The types that a correct static meaning of this declaration list must associate with the two tags are again those given by the least solution of the following system of equations:

$$\begin{aligned} \tau_x &= \mathit{struct}\,[\mathit{tagged}\,[\text{“}\mathit{tagx}\text{”}, 1], \{\,\text{“}\mathit{py}\text{”} \mapsto \mathit{obj}\,[\mathit{ptr}\,[\mathit{obj}\,[\tau_y, \mathit{noqual}]], \mathit{noqual}]\,\}] \\ \tau_y &= \mathit{struct}\,[\mathit{tagged}\,[\text{“}\mathit{tagy}\text{”}, 2], \{\,\text{“}\mathit{px}\text{”} \mapsto \mathit{obj}\,[\mathit{ptr}\,[\mathit{obj}\,[\tau_x, \mathit{noqual}]], \mathit{noqual}]\,\}] \end{aligned}$$

In order to express this solution as a closed formula, it is necessary to group the two types in a pair and apply the least fixed point operator to a function over pairs of types. The result of the expression:

$$\begin{aligned} &\mathit{fix}\ (\lambda\,\langle\tau_x, \tau_y\rangle. \\ &\quad\langle \mathit{struct}\,[\mathit{tagged}\,[\text{“}\mathit{tagx}\text{”}, 1], \{\,\text{“}\mathit{py}\text{”} \mapsto \mathit{obj}\,[\mathit{ptr}\,[\mathit{obj}\,[\tau_y, \mathit{noqual}]], \mathit{noqual}]\,\}] \\ &\quad\ \mathit{struct}\,[\mathit{tagged}\,[\text{“}\mathit{tagy}\text{”}, 2], \{\,\text{“}\mathit{px}\text{”} \mapsto \mathit{obj}\,[\mathit{ptr}\,[\mathit{obj}\,[\tau_x, \mathit{noqual}]], \mathit{noqual}]\,\}]\rangle) \end{aligned}$$

is the pair that corresponds to the least solution of the system, and it is easy to extract its two elements.

GENERALIZATION

This approach can be generalized for any finite number of types defined by mutual recursion. As shown above, a tuple must be used to group the types before the least fixed point operator is applied. However, this may be a problem in the definition of static semantics, since the number of types that participate in the multiple recursion is not fixed. Even worse, this number depends on the syntax of C programs. In order to avoid introducing products with a variable number of elements, a different solution is adopted, a variation of which was first proposed in [Seth80]. Instead of applying the least fixed point operator to a function that updates a tuple of types, the proposed solution applies it to a function updating the whole type environment.

Consider again the first given example of a single recursively defined data type. In order to obtain the correct data type for *"tag"*, the following algorithm is used, which will be called *type environment fixing.* The first few steps and the limit are shown in Figure 6.1. The static meaning of the declaration is first applied to the initial type environment and the result is a type environment containing the aforementioned incorrect type for *"tag"*. This can be used as a first approximation. (In Figure 6.1 the empty environment is used as the initial environment in "Pass #0" and the first approximation is labeled "Pass #1".) Subsequently, let us consider applying the static meaning of the declaration again, this time using the first approximation as the initial type environment. Ignoring for the moment the error that would occur from the redefinition of *"tag"*, the new type environment would be a better approximation. As shown in Figure 6.1 under the label "Pass #2", the type associated with *"tag"* in this approximation would be a more complete structure type, containing a pointer to the structure type that was given by the previous approximation. By repeating this, the type environment that would result in each step would be a better approximation of the previous one. The least upper bound of this infinite sequence of type environments would contain the correct type information for *"tag"*. In this way it is possible to obtain the correct type information for any number of types given by mutually recursive definitions.

**Figure 6.1:** Example of type environment fixing (simple).

| | | | |
|---|---|---|---|
| *(Pass #0)* | *"tag"* | $\mapsto$ | $\top$ |
| *(Pass #1)* | *"tag"* | $\mapsto$ | $struct\,[tagged\,[\text{"}tag\text{"}, 1], \pi^1]$ |
| | $\pi^1$ | $=$ | $\{\,\text{"}p\text{"} \mapsto obj\,[ptr\,[obj\,[struct\,[tagged\,[\text{"}tag\text{"}, 1], \bot], noqual]], noqual]\,\}$ |
| *(Pass #2)* | *"tag"* | $\mapsto$ | $struct\,[tagged\,[\text{"}tag\text{"}, 1], \pi^2]$ |
| | $\pi^2$ | $=$ | $\{\,\text{"}p\text{"} \mapsto obj\,[ptr\,[obj\,[struct\,[tagged\,[\text{"}tag\text{"}, 1], \pi^1], noqual]], noqual]\,\}$ |
| *(Pass #3)* | *"tag"* | $\mapsto$ | $struct\,[tagged\,[\text{"}tag\text{"}, 1], \pi^3]$ |
| | $\pi^3$ | $=$ | $\{\,\text{"}p\text{"} \mapsto obj\,[ptr\,[obj\,[struct\,[tagged\,[\text{"}tag\text{"}, 1], \pi^2], noqual]], noqual]\,\}$ |
| | | ... | |
| *(LUB)* | *"tag"* | $\mapsto$ | $struct\,[tagged\,[\text{"}tag\text{"}, 1], \pi^\infty]$ |
| | $\pi^\infty$ | $=$ | $\{\,\text{"}p\text{"} \mapsto obj\,[ptr\,[obj\,[struct\,[tagged\,[\text{"}tag\text{"}, 1], \pi^\infty], noqual]], noqual]\,\}$ |

PROBLEMS IN UPDATING

However, there is still a minor problem with the proposed solution. The functions for updating type environments, defined in Section 4.5.1, generate an error if an identifier or a tag that has already been defined is redefined in the same scope. Obviously, type environments do not know of "better approximations", except for the simple case of structure or union tags that have just been declared. This must be corrected before the proposed solution for recursively defined types can be adopted.

The use of the least fixed point operator to take the least upper bound of the infinite sequence of generated type environments in the algorithm described before, leads to the adoption of the domain ordering relation for expressing "better approximations" of types and type environments. The relation $\tau_1 \sqsubseteq \tau_2$ is taken to mean that "data type $\tau_2$ is at least as good an approximation as $\tau_1$". This is generalized for other kinds of types. Also, for the case of type environments, the domain ordering specifies that $e_1 \sqsubseteq e_2$ if and only if, for every identifier $I$, the type that $e_2$ associates with $I$ is at least as good an approximation as the one that $e_1$ associates with $I$. The domain ordering that has been defined in Chapter 4 needs not be modified, since it expresses correctly this notion of "better approximation".

A TRICKY EXAMPLE

One might consider that, as a conclusion from the previous discussion, the problems in updating type environments could be solved by redefining the update operations, in such a way as to allow updating any identifier with a "better approximation" of its associated type. However, this is not true, as will be shown in the following example. Let us consider the following situation, in which the declaration list that is of primary interest is located in an enclosed scope. In addition to that, notice that the recursively defined structure type already exists in the parent scope and that identifier *"a"* should be associated with the structure containing *"x"*, not *"p"*.

```
struct tag { int x; };
{
   struct tag a;
   struct tag { struct tag * p; };
}
```

**Figure 6.2:** Example of type environment fixing (tricky).

| | | | |
|---|---|---|---|
| *(Outer, #0)* | $\text{“}tag\text{”}$ | $\mapsto$ | $struct\,[tagged\,[\text{“}tag\text{”}, 1], \pi_x]$ |
| | $\pi_x$ | $=$ | $\{\,\text{“}x\text{”} \mapsto obj\,[int, noqual]\,\}$ |
| *(Inner, #1)* | $\text{“}a\text{”}$ | $\mapsto$ | $obj\,[struct\,[tagged\,[\text{“}tag\text{”}, 1], \pi_x], noqual]$ |
| | $\text{“}tag\text{”}$ | $\mapsto$ | $struct\,[tagged\,[\text{“}tag\text{”}, 2], \pi_p^1]$ |
| | $\pi_p^1$ | $=$ | $\{\,\text{“}p\text{”} \mapsto obj\,[ptr\,[obj\,[struct\,[tagged\,[\text{“}tag\text{”}, 2], \bot], noqual]], noqual]\,\}$ |
| *(Inner, #2)* | $\text{“}a\text{”}$ | $\mapsto$ | $obj\,[struct\,[tagged\,[\text{“}tag\text{”}, 2], \pi_p^1], noqual]$ |
| | $\text{“}tag\text{”}$ | $\mapsto$ | $struct\,[tagged\,[\text{“}tag\text{”}, 2], \pi_p^2]$ |
| | $\pi_p^2$ | $=$ | $\{\,\text{“}p\text{”} \mapsto obj\,[ptr\,[obj\,[struct\,[tagged\,[\text{“}tag\text{”}, 2], \pi_p^1], noqual]], noqual]\,\}$ |
| | | | ... |
| *(Inner, LUB)* | $\text{“}a\text{”}$ | $\mapsto$ | $\top$ *(LUB of unrelated types, see footnote 1)* |
| | $\text{“}tag\text{”}$ | $\mapsto$ | $struct\,[tagged\,[\text{“}tag\text{”}, 2], \pi_p^\infty]$ |
| | $\pi_p^\infty$ | $=$ | $\{\,\text{“}p\text{”} \mapsto obj\,[ptr\,[obj\,[struct\,[tagged\,[\text{“}tag\text{”}, 2], \pi_p^\infty], noqual]], noqual]\,\}$ |
| *(Inner, correct)* | $\text{“}a\text{”}$ | $\mapsto$ | $obj\,[struct\,[tagged\,[\text{“}tag\text{”}, 1], \pi_x], noqual]$ |
| | $\text{“}tag\text{”}$ | $\mapsto$ | $struct\,[tagged\,[\text{“}tag\text{”}, 2], \pi_p^\infty]$ |
| | $\pi_p^\infty$ | $=$ | $\{\,\text{“}p\text{”} \mapsto obj\,[ptr\,[obj\,[struct\,[tagged\,[\text{“}tag\text{”}, 2], \pi_p^\infty], noqual]], noqual]\,\}$ |

By applying the proposed solution to the inner declaration list, one ends up with the infinite sequence of type environments that is shown in Figure 6.2. It should be mentioned that, for the time being, the update operation for type environments unconditionally replaces type information if an identifier has already been defined. The initial type environment for the inner declaration list corresponds to a newly opened scope, based on the environment resulting from the outer declaration. It is labelled “Outer, #0” in the figure. The first step is labelled “Inner, #1”, and so on for subsequent steps.

The first thing to notice is that, in the first step the type associated with *“a”* is correct and the type associated with *“tag”* is indeed a first approximation. However, the second step incorrectly associates *“a”* with the newly defined structure type, and this association is repeated in all subsequent steps. On the other hand, the types associated with *“tag”* are correct. The result of this inconvenient situation is that in the final type environment, taken as the least upper bound of the infinite sequence, the type associated with *“tag”* would be correct but the type associated with *“a”* would be the least upper bound of the two unrelated structure types, which is clearly wrong.[1]

UPDATING MECHANISM

After all this, it is evident that the problem lies on the behaviour of the update operation on type environments, in the case of identifier redefinition. Specifically, there is a conflict among several desired cases of redefinition, as summarized in the following requirements:

(R1) Identifier redefinition should not normally be allowed, i.e. the static meaning of “`int x; double x;`” should clearly be an error. This requirement is satisfied by the present static semantics.

[1] The final environment in Figure 6.2 is not entirely correct. The least upper bound for *“a”* is not the top element, since the two types are not completely unrelated. However, it is clearly not a valid denotable type.

**Figure 6.3:** Intended behaviour of type environment updating.

**Ordinary update**

| $\cdot[\cdot \mapsto ide\ \cdot]$ | normally | | while fixing | |
|---|---|---|---|---|
| | existing $= \top$ | existing $\neq \top$ | existing $= \top$ | existing $\neq \top$ |
| existing $\sqsubseteq$ new | impossible | error | impossible | update |
| existing $\not\sqsubseteq$ new | update | error | impossible | ignore |

**Tag update**

| $\cdot[\cdot \mapsto tag\ \cdot]$ | normally | | while fixing | |
|---|---|---|---|---|
| | existing $= \top$ | existing $\neq \top$ | existing $= \top$ | existing $\neq \top$ |
| existing $\sqsubseteq$ new $\wedge$ jd(existing) | impossible | update | impossible | update |
| existing $\sqsubseteq$ new $\wedge \neg$jd(existing) | impossible | error | impossible | update |
| existing $\not\sqsubseteq$ new | update | error | impossible | ignore |

(R2) It should be allowed to redefine structure or union tags, on condition that they were previously just declared, as in the case of "`struct tag; struct tag { int x; };`". This is also satisfied by the present static semantics.

(R3) During the fixing process, redefinition of an identifier using a "better approximation" should be allowed and performed, as in the case of *"tag"* in Figure 6.2, pass #2. This requirement is not satisfied by the present static semantics.

(R4) During the fixing process, redefinition of an identifier using a type that is not a "better approximation" should be allowed but ignored, as in the case of *"a"* in Figure 6.2, pass #1. This requirement is not satisfied by the present static semantics.

It is not hard to see that requirements R1 and R2 are in conflict with R3 and R4. There seems to be no plausible solution without taking special measures for the fixing process. One approach to fix this problem is to include information about the fixing process in type environments. As shown in the next section, a truth value is included to determine whether the fixing process has started. The update operations should also be revised to use this information in such a way as to satisfy the aforementioned requirements. Figure 6.3 shows the desired behaviour for the update operations on type environments. By "jd(existing)" it is meant that the existing value is a structure or union tag that has just been declared; as will be seen in the following section, the function $isDeclaredTag$ is used to determine this. Possible behaviours are: *update*, i.e. redefine the identifier; *ignore*, i.e. keep the existing definition; *error*, i.e. disallow the redefinition and generate an error; and *impossible*, i.e. a situation that can never happen or, if it does, the result of the operation is of no importance.

## 6.2 Type environments revised

SUMMARY OF CHANGES

In the light of the conclusions drawn in the previous section, the domain $\mathbf{Ent}$ that represents type environments is subject to revision. The main modification that is required is the addition of information about the fixing process. Additional operations must be defined in relation with this information.

Some of the existing operations, such as ordinary and tag update, must change in order to use the newly available information. Others, such as the raw lookup operations, must only change slightly, because of the modification in the domain's definition. Finally, some operations remain unchanged and will not be repeated in this section. The update operations are examples of the last category.

DEFINITION ■ $e \;:\; \mathbf{Ent} \;=\; (\mathbf{Ide} \to \mathbf{Type}_{ide}) \times (\mathbf{Ide} \to \mathbf{Type}_{dat}) \times \mathbf{T} \times \mathbf{Ent}$

A truth value is added as an element of the product. It will be *true* if the fixing progress is under way, *false* otherwise.

EMPTY ENVIRONMENT ▶ $e_\circ \;:\; \mathbf{Ent}$

$e_\circ \;=\; \langle \top, \top, \mathit{false}, \top \rangle$

The empty environment is not being fixed, when it is constructed.

ORDINARY RAW LOOKUP ▶ $\cdot[\cdot \; \mathit{raw\ ide}] \;:\; \mathbf{Ent} \times \mathbf{Ide} \to \mathbf{Type}_{ide}$

$e[I \; \mathit{raw\ ide}] \;=$
$\quad \mathbf{let}\ \langle m_i, m_t, b_{fix}, e_p \rangle = e$
$\quad \mathbf{in}\ \ (m_i\ I \neq \top) \to m_i\ I\ ,\ (e_p \neq \top) \to e_p[I \; \mathit{raw\ ide}]\ ,\ \mathit{error}$

This is just a minor modification, in order to allow for the extra element $b_{fix}$ of the type environment.

TAG RAW LOOKUP ▶ $\cdot[\cdot \; \mathit{raw\ tag}] \;:\; \mathbf{Ent} \times \mathbf{Ide} \to \mathbf{Type}_{dat}$

$e[I \; \mathit{raw\ tag}] \;=$
$\quad \mathbf{let}\ \langle m_i, m_t, b_{fix}, e_p \rangle = e$
$\quad \mathbf{in}\ \ (m_t\ I \neq \top) \to m_t\ I\ ,\ (e_p \neq \top) \to e_p[I \; \mathit{raw\ tag}]\ ,\ \mathit{error}$

A minor modification again.

ORDINARY UPDATE ▶ $\cdot[\cdot \mapsto \mathit{ide}\ \cdot] \;:\; \mathbf{Ent} \times \mathbf{Ide} \times \mathbf{Type}_{ide} \to \mathsf{E}(\mathbf{Ent})$

$e[I \mapsto \mathit{ide}\ \delta] \;=$
$\quad \mathbf{let}\ \langle m_i, m_t, b_{fix}, e_p \rangle = e$
$\quad \mathbf{in}\ \ b_{fix} \wedge (m_i\ I \not\sqsubseteq \delta) \to \mathit{unit}\ e\ ,$
$\qquad b_{fix} \vee (m_i\ I = \top) \to \mathit{unit}\ \langle m_i\{I \mapsto \delta\}, m_t, b_{fix}, e_p \rangle\ ,\ \mathit{error}$

This is one of the operations that changed the most. It reflects the requirements that were summarized in the upper part of Figure 6.3. In order to simplify the operation's definition, the two *impossible* cases on the upper row are treated as if they were *update*, and the one on the lower row as if it was *ignore*. It is easy to see that, for the case $b_{fix} = \mathit{false}$, the result is the same as in the old definition.

TAG UPDATE ▶ $\cdot[\cdot \mapsto \mathit{tag}\ \cdot] \;:\; \mathbf{Ent} \times \mathbf{Ide} \times \mathbf{Type}_{dat} \to \mathsf{E}(\mathbf{Ent})$

$e[I \mapsto \mathit{tag}\ \tau] \;=$
$\quad \mathbf{let}\ \langle m_i, m_t, b_{fix}, e_p \rangle = e$
$\quad \mathbf{in}\ \ b_{fix} \wedge (m_t\ I \not\sqsubseteq \tau) \to \mathit{unit}\ e\ ,$
$\qquad b_{fix} \vee (m_t\ I = \top) \vee \mathit{isDeclaredTag}(m_t\ I) \to \mathit{unit}\ \langle m_i, m_t\{I \mapsto \tau\}, b_{fix}, e_p \rangle\ ,\ \mathit{error}$

This operation also required a lot of changes. It reflects the requirements summarized in the lower part of Figure 6.3. In order to simplify the operation's definition, the two *impossible* cases on the two upper rows are treated as if they were *update*, and the one on the lower row as if it was *ignore*. It is easy to see again that, for the case $b_{fix} = \mathit{false}$, the result is the same as in the old definition.

▶ $\cdot[\cdot \mapsto \mathit{fresh\ tag}\ \cdot] \ :\ \mathbf{Ent} \times \mathbf{Ide} \times \mathbf{TagType} \to \mathsf{E}(\mathbf{Ent})$ FRESH TAG

$$
\begin{array}{l}
e[I \mapsto \mathit{fresh\ tag}\ \sigma] \ = \\
\quad \mathbf{let}\ \langle m_i, m_t, b_{fix}, e_p \rangle = e \\
\qquad \tau \qquad\qquad\quad = m_t\ I \\
\quad \mathbf{in}\ \ \mathbf{case}\ \tau\ \mathbf{of} \\
\qquad\quad \top \qquad\quad \Rightarrow\ \ \mathbf{case}\ \sigma\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\quad \mathit{tag\text{-}struct}\ \Rightarrow\ e[I \mapsto \mathit{tag\ struct}\ [\mathit{fresh\text{-}tagged}\ I, \bot]] \\
\qquad\qquad\qquad\qquad\quad \mathit{tag\text{-}union}\ \Rightarrow\ e[I \mapsto \mathit{tag\ union}\ [\mathit{fresh\text{-}tagged}\ I, \bot]] \\
\qquad\qquad\qquad\qquad\quad \mathbf{otherwise}\ \Rightarrow\ \mathit{error} \\
\qquad\quad \mathit{struct}\ [t, \pi] \Rightarrow\ (\sigma = \mathit{tag\text{-}struct}) \to \mathit{unit}\ e\ ,\ \mathit{error} \\
\qquad\quad \mathit{union}\ [t, \pi] \Rightarrow\ (\sigma = \mathit{tag\text{-}union}) \to \mathit{unit}\ e\ ,\ \mathit{error} \\
\qquad\quad \mathit{enum}\ [\epsilon] \quad \Rightarrow\ (\sigma = \mathit{tag\text{-}enum}) \to \mathit{unit}\ e\ ,\ \mathit{error} \\
\qquad\quad \mathbf{otherwise}\ \Rightarrow\ \mathit{error}
\end{array}
$$

A minor modification, in order to allow for the extra element $b_{fix}$ of the type environment.

▶ $\uparrow \cdot \ :\ \mathbf{Ent} \to \mathbf{Ent}$ OPEN SCOPE

$$\uparrow e \ =\ \langle \top, \top, \mathit{false}, e \rangle$$

Newly opened scopes are not being fixed.

▶ $\downarrow \cdot \ :\ \mathbf{Ent} \to \mathbf{Ent}$ CLOSE SCOPE

$$\downarrow e \ =\ \mathbf{let}\ \langle m_i, m_t, b_{fix}, e_p \rangle = e\ \mathbf{in}\ e_p$$

A minor modification again.

▶ $\mathit{isLocal}(\cdot, \cdot\ \mathit{ide}) \ :\ \mathbf{Ent} \to \mathbf{Ide} \to \mathbf{T}$ LOCAL ORDINARY

$$\mathit{isLocal}(e, I\ \mathit{ide}) \ =\ \mathbf{let}\ \langle m_i, m_t, b_{fix}, e_p \rangle = e\ \mathbf{in}\ \neg(m_i\ I = \top)$$

▶ $\mathit{isLocal}(\cdot, \cdot\ \mathit{tag}) \ :\ \mathbf{Ent} \to \mathbf{Ide} \to \mathbf{T}$ LOCAL TAG

$$\mathit{isLocal}(e, I\ \mathit{tag}) \ =\ \mathbf{let}\ \langle m_i, m_t, b_{fix}, e_p \rangle = e\ \mathbf{in}\ \neg(m_t\ I = \top)$$

More minor modifications.

▶ $\mathit{init\text{-}fix} \ :\ \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$ INITIALIZE FIXING

$$\mathit{init\text{-}fix}\ e \ =\ \mathbf{let}\ \langle m_i, m_t, b_{fix}, e_p \rangle = e\ \mathbf{in}\ \neg b_{fix} \to \mathit{unit}\ \langle m_i, m_t, \mathit{true}, e_p \rangle\ ,\ \mathit{error}$$

This function sets the truth value for fixing to $\mathit{true}$, declaring the start of the fixing process. An error occurs if the environment is already being fixed.

## 6.3 The fixing process

The algorithm for fixing type environments has been described informally in Section 6.1. In this section it is formally defined. In addition, it is specified where and when this fixing process takes place.

DEFINITION ▶
$$rec \;:\; (\mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})) \to \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$$
$$rec \;=\; \lambda f.\, \lambda e.\, \mathbf{let}\; z = f\; e \;*\; \mathit{init\text{-}fix}\; \mathbf{in}\; mclo\; z\; f$$

Function *rec* performs the fixing of type environments for recursively defined types. It takes two arguments: the static semantic meaning of a declaration list, i.e. a function $f$ from type environments to type environments; and an initial type environment $e$. It returns the type environment that results from first applying $f$ to $e$ and then fixing all recursively defined types. The monadic closure operator *mclo* is used in this function's definition. The starting environment for the closure is $z$, i.e. the type environment which results from the first pass in Figure 6.1. Notice also how $z$ is initialized for fixing. In order to verify that the result of the monadic closure operation is indeed the least upper bound of the infinite sequence of gradually fixed environments, recall the definition of *mclo* and *clo*:

$$mclo\; z\; f \;=\; clo\; z\; (\lambda x.\, x \;*\; f) \;=\; \bigsqcup_{n=0}^{\infty} (\lambda x.\, x \;*\; f)^n\; z$$

The definition is somewhat perplexed by the fact that errors are allowed; however, it is easy to see that the least upper bound of the infinite sequence is taken.

WHERE AND WHEN

The equation for the static meaning of *translation-unit*, as shown in Section 5.2, involves a call to *rec*. The same is true for *declarator* and *abstract-declarator* in Section 5.3.5, when function declarations are involved. Its presence is obviously necessary at these points, since recursive definitions may occur at file scope or function prototype scope. In general, *rec* must be called whenever a new scope that may contain recursive type definitions is opened. However, as was evident in Chapter 5, the proposed static semantics only opens new scopes when necessary information can only be obtained from them. Therefore, only file scope and function prototype scopes need to be fixed. For all other scopes including function scope and block scopes, *rec* is called in the typing inference rules, presented in Part III and in the equations for the dynamic semantics, presented in Part IV.

# Part III

# Typing semantics

# Chapter 7

# Typing judgements

CHAPTER OVERVIEW

This chapter introduces the notions of typing semantics, typing judgements and typing derivations, that will be the primary issues of interest in Part III. In Section 7.1 a brief introduction is attempted. Section 7.2 describes the typing judgements that are used throughout Part III in the definition of C's typing semantics. Finally, in Section 7.3 a number of issues is discussed, related to the uniqueness of typing results and typing derivations for given program phrases.

## 7.1 Introduction to typing

Typing semantics is used in the formal definition of programming languages in order to define aspects that cannot be expressed by a context-free grammar. The primary aim of a language's typing semantics is the association of *phrases*, participating in syntactically well-formed programs, with *phrase types*. In this way, valuable information is given concerning the meaning of program phrases. In simple programming languages, typing semantics can often replace the language's abstract syntax completely. That is, the syntactic productions can be defined at the same time when phrase types are associated with program phrases. In this case, the language's static semantics is also defined as part of the typing semantics. However, in the case of a complex programming language such as C, the three phases of abstract syntax, static semantics and typing need be separate.

TYPING JUDGEMENTS

Since the primary interest in typing is to define a relation between program phrases and phrase types, a formal way is needed to describe this relation. This is achieved by means of *typing judgements*, also called in literature *typing assertions*. The most common form of a typing judgement is the following:

$$\Gamma \vdash M : \theta$$

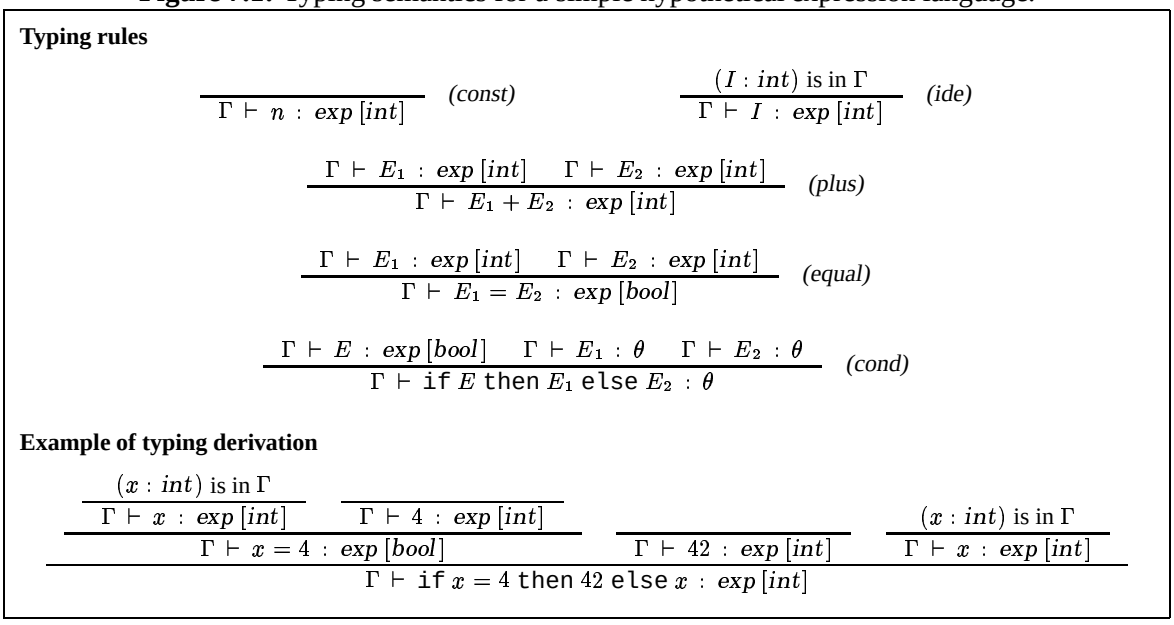
where $\Gamma$ is a *type assignment*,[1] i.e. a given association of identifiers with types, $M$ is a program phrase and $\theta$ is a phrase type. The intuitive meaning of this judgement is that $M$ is a well-typed phrase of type $\theta$, under the assumptions stated in $\Gamma$. In specifying the typing semantics of a complex programming language, additional forms of typing judgements may be necessary. Typing judgements may be viewed as truth value predicates, that can be $true$ or $false$. A typing judgement is $true$ if it can be proved to hold using the language's typing semantics. Otherwise, it is $false$.

TYPING RULES

The typing semantics of a programming language is most commonly defined as a consistent set of *axioms* and *inference rules*, which are commonly called *typing rules*. Axioms are typically used to define the typing semantics of simple phrases, whereas inference rules are used in the case of

[1] Type assignments are also called *typing contexts* in literature.

**Figure 7.1:** Typing semantics for a simple hypothetical expression language.




compound phrases that consist of smaller sub-phrases. Since axioms can be considered as a special case of inference rules, the latter are obviously the most important issue in the formal definition of typing semantics. The general form of an inference rule is the following:

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{C} \quad \textit{(rule)}$$

where "rule" is an identifier specifying the rule's unique name, $n$ is a natural number and may be zero and all other components are typing judgements. In particular, the typing judgements $P_i$ are the rule's *premises* whereas the typing judgement $C$ is the rule's *conclusion.* The intuitive meaning of this rule is that, on condition that the premises are $true$, the conclusion is also $true$. In the case where no premises are present ($n = 0$), the inference rule represents an axiom.

As an example, a small number of typing rules for a hypothetical simple expression language is given in the upper part of Figure 7.1. The rule "ide" is the only one that makes use of the type assignment and its premise is given here in an informal way.

TYPING DERIVATIONS

The proof of a typing judgement using a set of typing rules is called *typing derivation.* A typing derivation typically combines a number of typing rules in a tree-like structure, in such a way that the conclusion of one rule becomes a premise of some other rule. In this way, typing derivations may be considered as complex inference rules. As mentioned previously, a typing judgement is $true$ if and only if it can be proved in the language's semantics, i.e. if there exists a valid typing derivation. A small example of a typing derivation for the same hypothetical expression language is shown in the lower part of Figure 7.1.

TYPING PROBLEMS

A number of *typing problems* that play an important role in the semantics of programming languages can be identified. Typing problems are families of questions that must be solved. Each particular question in such a family is called an *instance* of the problem and is generally related to the validity of typing judgements. A proof of the solution must be given in terms of typing derivations. A

typing problem is *solvable* if a proof of the solution can be given for every instance of this problem. Of course, the typing semantics of a programming language is more useful if solvable typing problems can be defined. For a more thorough discussion of typing and typing problems, the reader is referred to [Mitc90, Mitc96].

It is possible to ask many different kinds of questions about a typing judgement. Probably the most important typing problems in programming language semantics are the following, which are the only ones studied in the rest of this thesis.[2]

- *Decision problem:* Given a typing judgement $\Gamma \vdash M : \theta$, determine whether it is $true$, i.e. if there exists a valid typing derivation with this typing judgement as its conclusion. In case such a derivation exists, an important question to be asked is whether this derivation is unique. This issue is further discussed in Section 7.3.
- *Type inference:* Given a syntactically well-formed program phrase $M$ and a type assignment $\Gamma$, find a phrase type $\theta$ such that the typing judgement $\Gamma \vdash M : \theta$ is $true$. If such a type can be inferred, an important issue is whether it is unique. The issue of uniqueness of the underlying typing derivation is again important.

The type inference problem is "harder" than the decision problem, in the sense that the latter can be reduced to the former. That is, if the type inference problem can be solved, then the decision problem can also be solved.

## 7.2 Typing judgements

The typing semantics of C, as specified informally in §6 of the standard, is largely complicated. Apart from the main typing relation, an example of which was presented in the previous section, a number of other forms of typing judgements are necessary un order to simplify the typing rules. A summary of all forms of typing judgements that are used in C's typing semantics is given in Table 7.1.

TYPE ASSIGNMENTS

As was previously mentioned, in simple programming languages it is possible to define type assignments at the same time with typing. However, in the case of real programming languages, such as C, this cannot be done in an easy and elegant way. Instead, the various kinds of environments that are produced from the static semantics are used as type assignments. Type environments contain all necessary information about the name spaces of ordinary and tag identifiers and are used in most typing judgements. Another kind of static environment that is used as a type assignment is member environments, containing information about the members of structures or unions.

MAIN TYPING RELATION

The main typing relation is expressed by typing judgements of the form:

$$e \vdash \mathit{phrase} : \theta$$

where $e$ is a type environment, *phrase* is a syntactically well-formed program phrase and $\theta$ is a phrase type. As mentioned in the preliminary discussion, the intuitive meaning of such a judgement is that the phrase is well typed and has type $\theta$, given the identifier declarations that are included in $e$.

**Table 7.1:** Summary of typing judgements.

| | |
|---|---|
| **Main typing relation** | |
| $e \vdash \mathit{phrase} : \theta$ | The given *phrase* can be attributed phrase type $\theta$ in type environment $e$. |
| **Predicates as judgements** | |
| $P$ | Predicate $P$ is $\mathit{true}$, where $P$ can be any valid predicate over truth values $\mathbf{T}$. |
| $x = y$ | Elements $x, y \in D$ are equal, w.r.t. domain equality in $D$. |
| $v := z$ | The static semantic valuation $z \in \mathsf{E}(D)$ produces the (non-error) value $v \in D$. |
| **Judgements related to environments** | |
| $e \vdash I \lhd \delta$ | Identifier $I$ is associated with identifier type $\delta$ in type environment $e$. |
| $\pi \vdash I \lhd m$ | Identifier $I$ is associated with member type $m$ in member environment $\pi$. |
| **Judgements related to expressions** | |
| $e \vdash E \gg \tau$ | Expression $E$ can be assigned to an object of data type $\tau$. |
| $e \vdash E = \mathtt{NULL}$ | Expression $E$ is a null pointer constant. |
| $e \vdash T \equiv \phi$ | Type name $T$ denotes type $\phi$ in type environment $e$. |

PREDICATES

Ordinary predicates can be used in the place of typing judgements. Predicates are meta-language expressions resulting in truth values, i.e. elements of domain $\mathbf{T}$. A predicate whose value is $\mathit{true}$ is considered as a typing axiom, i.e. as a $\mathit{true}$ typing judgement. On the other hand, a predicate whose value is $\mathit{false}$, $\bot$ or $\top$, simply cannot be used in a typing derivation. Special cases of predicates used as typing judgements are equality tests of the form $x = y$, between elements $x$ and $y$ of domain $D$.

STATIC VALUATIONS

Another form of typing judgement, which is really a special notation for a simple predicate, is used to allow the use of static valuations in typing rules. Static valuations often result in elements of domain $\mathsf{E}(D)$, where $D$ is any domain and $\mathsf{E}$ is the error monad defined in Section 4.4. A typing judgement of the form:

$$v := z$$

where $v \in D$ and $z \in \mathsf{E}(D)$, has the intuitive meaning that the static valuation $z$ results in a normal, i.e. non-error, value $v$ of the underlying domain $D$. This typing judgement is equivalent to the predicate:

$$z = \mathit{unit}_{\mathsf{E}}\, v$$

but the special form is used for brevity.

ENVIRONMENT RELATED

The next two forms of typing judgements are strongly related to static environments and are used to extract typing information from them. The difference between the two lies in the kind of static environment that contains the type information. In the case of typing judgement:

$$e \vdash I \lhd \delta$$

[2] Other typing problems are *type inhabitation*, and *context derivation*. The first is about finding a program phrase which inhabits a given phrase type under a given type assignment. The second is concerned with determining type assignments.

the information is contained in type environment $e$ and identifiers are associated with identifier types. On the other hand, in the case of typing judgement:

$$\pi \vdash I \triangleleft m$$

the information is contained in the member environment of a structure or union and identifiers are associated with member types.

ASSIGNABILITY

The last three forms of typing judgements are used in the typing semantics of expressions. A typing judgement of the form:

$$e \vdash E \gg \tau$$

where $e$ is a type environment, $E$ is an expression phrase and $\tau$ is a data type, has the intuitive meaning that the value of expression $E$ can be assigned to an object of type $\tau$, given the type environment $e$.

NULL POINTER CONSTANTS

A special form of typing judgement is used to identify null pointer constants in expressions:

$$e \vdash E = \texttt{NULL}$$

where $e$ is a type environment and $E$ is an expression phrase. The intuitive meaning of this typing judgement is that $E$ is a null pointer constant, given the type environment $e$.

TYPE NAMES

The last form of typing judgement is used to simplify the association of type names with denotable types. A judgement of the form:

$$e \vdash T \equiv \phi$$

where $e$ is a type environment, $T$ is a phrase generated by the non-terminal symbol *type-name*, and $\phi$ is a denotable type, has the intuitive meaning that $T$ is a synonym for the type $\phi$, given the type environment $e$.

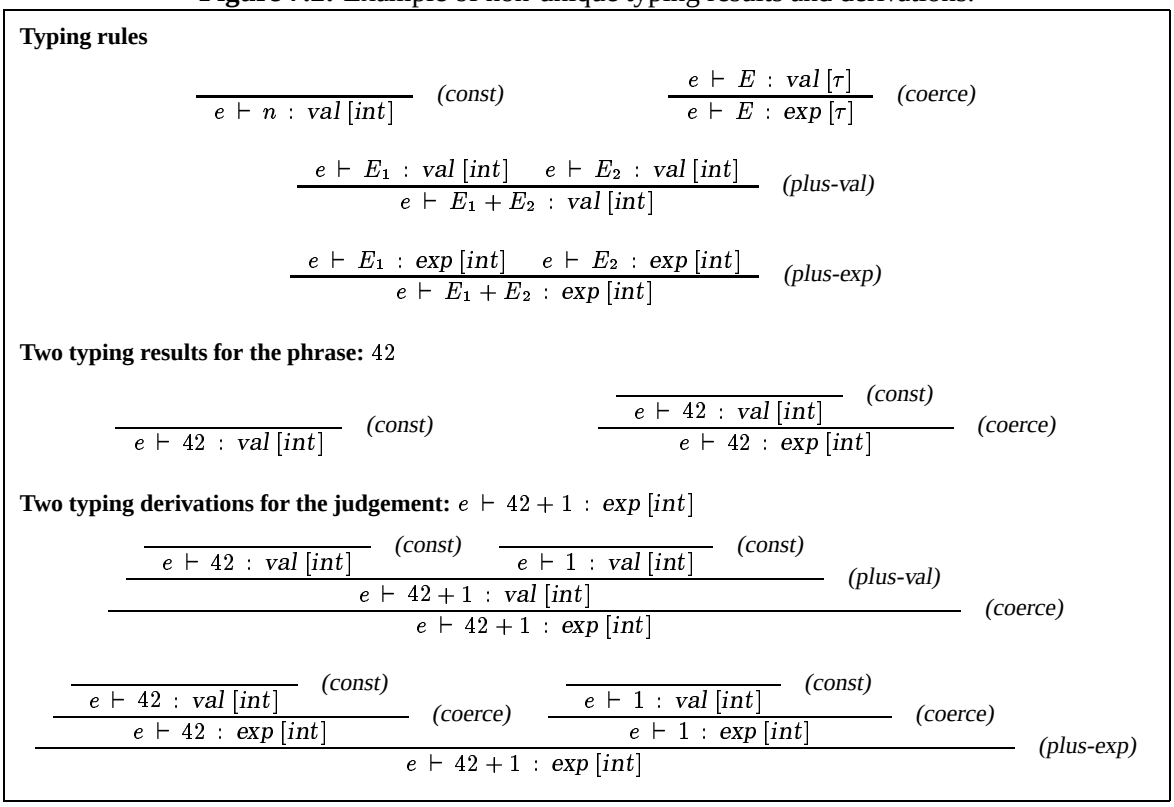
NEGATIVE TYPING JUDGEMENTS

Typing judgements may also appear in a negative sense. For the main type relation, a negative judgement has the form:

$$e \nvdash \mathit{phrase} : \theta$$

and the intuitive meaning is that the positive judgement $e \vdash \mathit{phrase} : \theta$ is not $\mathit{true}$, that is, there does not exist a typing derivation with it as the conclusion. Negative judgements are only useful when the related typing problems are solvable, since otherwise it cannot be determined whether such a derivation exists.

**Figure 7.2:** Example of non-unique typing results and derivations.




## 7.3 Discussion of uniqueness in typing

INTRODUCTION AND EXAMPLE

The typing semantics of a real programming language, expressed in the form of inference rules, often leads to ambiguity problems. There are two forms of such problems that appear in the study of C's typing semantics, as defined in the next chapters. Both problems are presented in this section and briefly discussed. They are introduced in a small example that uses a very small subset of the typing rules for C, slightly distorted for the sake of simplicity. The complete example is illustrated in Figure 7.2. The upper part of the figure contains four typing rules:

- Rule *const*, stating that integer numbers are constant values of type $int$.
- Rule *coerce*, stating that a constant value can be coerced to a non-constant value, if the fact that it is constant is not important. This rule is essential, in order to avoid duplicating all rules that do not care for constant values.
- Rules *plus-val* and *plus-exp* are two similar rules for the typing semantics of the addition operator. This operator distinguishes between constant and non-constant values. The sum of two constant values is a constant value, in the sense that it can be computed at compile-time. On the other hand, the sum of two non-constant values is a non-constant value and must be computed at run-time. Notice how the presence of rule *coerce* makes it unnecessary to include all possible combinations of constant and non-constant values.

The lower part of the same figure identifies two kinds of potential problems that are related to the typing of expressions. The first has to do with the uniqueness of the typing results, and the second with the uniqueness of typing derivations. Both issues are discussed in the following paragraphs. All typing derivations in the figure are tagged with the names of the typing rules that are applied, for the sake of clarity.

UNIQUENESS OF RESULTS

Consider the simple expression "`42`", consisting of just an integer number. The middle section of Figure 7.2 shows two valid typing derivations that conclude in two different phrase types for this phrase: $val\,[int]$ and $exp\,[int]$. This proves that the typing results, as far as the main typing relation for this example is concerned, are not unique. In fact, the phrase types that are associated with program phrases need not be unique. In some cases, it is useful or even necessary to consider an integer number as a constant integer value, e.g. in the case of C's constant expressions, defined in §6.4 of the standard. In other cases, however, this information is not necessary and the more general type of integer expression can be used. In conclusion, non-uniqueness of typing results *does not* pose a problem, since different phrase types represent different aspects in the semantics of program phrases.

UNIQUENESS OF DERIVATIONS

The lower section of Figure 7.2 shows two different valid derivations, both concluding with the same typing result for the simple expression "`42+1`". This expression is finally associated with phrase type $exp\,[int]$, that is, an integer expression. Both derivations start from the fact that the two operands are constant integer values. The upper derivation first adds the two constant values and then coerces the result to an integer expression. On the other hand, the lower derivation first coerces the two operands and then adds the two resulting integer expressions.

Of course both derivations are valid and, since the typing result is the same, they both describe the same semantic aspect of the given expression. However, as will be illustrated in Part IV, the dynamic semantics for a well-typed program phrase largely depends on the typing derivation that is used. Consequently, the issue of non-uniqueness of typing derivations *does* pose a problem for the typing semantics of C. In order to overcome this problem, it must be guaranteed that the dynamic semantics that correspond to all possible typing derivations for a program phrase are all equal. In the example that was discussed above, the result of first adding and then coercing must be guaranteed to be equal to the result of first coercing and then adding.

# Chapter 8

# Typing semantics of expressions

CHAPTER OVERVIEW

This chapter contains the typing rules for expressions, i.e. inference rules aiming primarily at associating expressions with phrase types by means of formal typing derivations. Section 8.1 defines the main typing relation, corresponding to typing judgements of the form $e \vdash E : \theta$. Its structure corresponds to the structure of §6.3 of the standard, with small deviations. In Section 8.2 typing rules corresponding to other forms of typing judgements are defined.

## 8.1 Main typing relation

INTUITIVE MEANING

The main typing relation for expressions corresponds to typing judgements of the form:

$$e \vdash E : \theta$$

where $E$ is an expression, $e$ is the type environment for the scope which contains the expression, and $\theta$ is a type that is *attributed* to $E$. The intended meaning of typing derivations for the aforementioned typing judgement is double. First, if a typing derivation can be found having this judgement as its conclusion, then $E$ is a well typed expression of type $\theta$, i.e. it complies to the typing rules according to the standard and has the appropriate type. Second, it must not be possible to produce typing derivations for not well typed expressions, i.e. incorrect expressions containing one or more violations of the typing rules stated in the standard. Valid values for $\theta$ are:

| | |
|---|---|
| $val\,[\tau]$ | Attributed to constant expressions which can only be treated as r-values. Constant expressions need not access the program state, i.e. the values of stored objects, in order to be evaluated. |
| $exp\,[v]$ | Attributed to non-constant expressions which can only be treated as r-values. Evaluation of non-constant expressions generally requires access to the program state. |
| $lvalue\,[m]$ | Attributed to non-constant expressions that designate objects and can be treated as l-values. |

UNIQUENESS ISSUES

As has been discussed in Section 7.3, the types that can be attributed to expressions are not unique. This is primarily the effect of the implicit coercion rules that are defined in Section 8.1.15. Furthermore, for a given typing judgement it may be possible to produce different typing derivations. This is a result of the distiction between constant and non-constant expressions, that is introduced in §6.4 of the standard. In all arithmetic operations the typing rules must distinguish between constant and non-constant values, i.e. expressions of types $val\,[\tau]$ and $exp\,[\tau]$, as was illustrated in the example given

in Section 7.3. As has already been mentioned, the dynamic semantics that correspond to different possible derivations must be equal.

### 8.1.1 Primary expressions

The typing semantics of primary expressions is described in §6.3.1 of the standard. As the reader would expect, types of the form $val\ [\tau]$ are attributed to all kinds of constants, whereas string literals and identifiers designating objects have types of the form $lvalue\ [m]$.

FLOATING CONSTANTS

$$\frac{suffix(f) = \varnothing}{e \vdash f\ :\ val\ [double]} \quad (E1)$$

$$\frac{suffix(f) = \{\text{'F'}\}}{e \vdash f\ :\ val\ [float]} \quad (E2)$$

$$\frac{suffix(f) = \{\text{'L'}\}}{e \vdash f\ :\ val\ [long\text{-}double]} \quad (E3)$$

The type of a floating constant is determined by its suffix, as stated in §6.1.3.1 of the standard. Absence of a suffix makes it a *double* constant.

INTEGER CONSTANTS

$$\frac{suffix(n) = \varnothing \qquad isDecimal(n) \qquad \tau := firstToRepresent(n, [\,int, long\text{-}int, unsigned\text{-}long\text{-}int\,])}{e \vdash n\ :\ val\ [\tau]} \quad (E4)$$

$$\frac{suffix(n) = \varnothing \qquad \neg isDecimal(n) \qquad \tau := firstToRepresent(n, [\,int, unsigned\text{-}int, long\text{-}int, unsigned\text{-}long\text{-}int\,])}{e \vdash n\ :\ val\ [\tau]} \quad (E5)$$

$$\frac{suffix(n) = \{\text{'U'}\} \qquad \tau := firstToRepresent(n, [\,unsigned\text{-}int, unsigned\text{-}long\text{-}int\,])}{e \vdash n\ :\ val\ [\tau]} \quad (E6)$$

$$\frac{suffix(n) = \{\text{'L'}\} \qquad \tau := firstToRepresent(n, [\,long\text{-}int, unsigned\text{-}long\text{-}int\,])}{e \vdash n\ :\ val\ [\tau]} \quad (E7)$$

$$\frac{suffix(n) = \{\text{'U'}, \text{'L'}\} \qquad \tau := firstToRepresent(n, [\,unsigned\text{-}long\text{-}int\,])}{e \vdash n\ :\ val\ [\tau]} \quad (E8)$$

The types of integer constants are somewhat more complicated, as stated in §6.1.3.2 of the standard. They are determined not only by the suffix, but also by the base of the system in which the constant is expressed.

CHARACTER CONSTANTS

$$\frac{prefix(c) = \varnothing}{e \vdash c\ :\ val\ [int]} \quad (E9)$$

$$\frac{prefix(c) = \{\text{'L'}\}}{e \vdash c\ :\ val\ [wchar\_t]} \quad (E10)$$

Character constants, as stated in §6.1.3.4 of the standard, are distinguished in *normal* and *wide*.

STRING LITERALS

$$\frac{prefix(s) = \varnothing \qquad n = lengthOf(s) + 1}{e \vdash s\ :\ lvalue\ [array\ [obj\ [char, noqual], n]]} \quad (E11)$$

$$\frac{prefix(s) = \{\text{`L'}\} \qquad n = lengthOf(s) + 1}{e \vdash s : lvalue\,[array\,[obj\,[wchar\_t, noqual], n]]} \quad (E12)$$

String literals are also distinguished in *normal* and *wide*, as stated in §6.1.4 of the standard. They are l-values of appropriate array types.

IDENTIFIERS

$$\frac{e \vdash I \triangleleft normal\,[\alpha]}{e \vdash I : lvalue\,[\alpha]} \quad (E13)$$

$$\frac{e \vdash I \triangleleft normal\,[f]}{e \vdash I : exp\,[f]} \quad (E14)$$

$$\frac{e \vdash I \triangleleft enum\text{-}const\,[n]}{e \vdash I : val\,[int]} \quad (E15)$$

Identifiers that are present in the environment are attributed the types that the environment associates with them. Object designators are l-values, function designators are r-values and enumeration constants are constant values of type *int*.

### 8.1.2 Postfix operators

The typing semantics of postfix expressions is described in §6.3.2 of the standard. Postfix increment and decrement operators are defined in Section 8.1.3.

ARRAY SUBSCRIPTS

$$\frac{e \vdash *(E_1 + E_2) : lvalue\,[\alpha]}{e \vdash E_1[E_2] : lvalue\,[\alpha]} \quad (E16)$$

As specified in §6.3.2.1 of the standard, the connection between arrays and pointers is obvious in the definition of the array subscripting operator. Its type relies on the correct typing for indirection and the addition operator.

FUNCTION CALLS

$$\frac{e \vdash E : exp\,[ptr\,[func\,[\tau, p]]] \qquad e \vdash arguments : arg\,[p]}{e \vdash E(arguments) : exp\,[\tau]} \quad (E17)$$

The typing rule for function calls is straightforward. The types of actual arguments must match the types of formal arguments, in the function prototype. The following rules define the typing semantics for arguments.

$$\frac{}{e \vdash \epsilon : arg\,[p_\circ]} \quad (R1)$$

$$\frac{p := ellipsis\ p_\circ}{e \vdash \epsilon : arg\,[p]} \quad (R2)$$

$$\frac{e \vdash E \gg \tau \qquad e \vdash arguments : arg\,[p] \qquad p' := \tau \lessdot p}{e \vdash E\ ,\ arguments : arg\,[p']} \quad (R3)$$

$$\frac{e \vdash E : exp\,[\tau] \qquad e \vdash arguments : arg\,[p] \qquad p := ellipsis\ p_\circ \qquad \tau' := argPromote\ \tau}{e \vdash E\ ,\ arguments : arg\,[p]} \quad (R4)$$

A function's arguments are distinguished in two categories: those passed as parameters that are specifically declared in the function's prototype and those passed in the ellipsis part. Rules R1 and R3 treat arguments that correspond to the first category, whereas rules R2 and R4 treat arguments corresponding to the second category. In the first category, arguments are converted as if by assignment to the appropriate parameter type. In the second category, the default argument promotions are applied.

MEMBER OPERATORS

$$\frac{e \vdash E : lvalue\,[obj\,[struct\,[t,\pi],q]] \quad \pi \vdash I \triangleleft m \quad m' := qualify\ q\ m}{e \vdash E.I : lvalue\,[m']} \quad (E18)$$

$$\frac{e \vdash E : exp\,[struct\,[t,\pi]] \quad \pi \vdash I \triangleleft m \quad \tau := datify\ m}{e \vdash E.I : exp\,[\tau]} \quad (E19)$$

$$\frac{e \vdash E : lvalue\,[obj\,[union\,[t,\pi],q]] \quad \pi \vdash I \triangleleft m \quad m' := qualify\ q\ m}{e \vdash E.I : lvalue\,[m']} \quad (E20)$$

$$\frac{e \vdash E : exp\,[union\,[t,\pi]] \quad \pi \vdash I \triangleleft m \quad \tau := datify\ m}{e \vdash E.I : exp\,[\tau]} \quad (E21)$$

The typing rules for the dot operator depend on two independent factors. The first is whether the left operand is a structure or a union; this factor does not affect the type of the whole expression. The second is whether the left operand is an l-value. If it is, the result of the expression is also an l-value, otherwise it is not. Thus, four typing rules for the dot operator are necessary.

$$\frac{e \vdash E : exp\,[ptr\,[obj\,[struct\,[t,\pi],q]]] \quad \pi \vdash I \triangleleft m \quad m' := qualify\ q\ m}{e \vdash E\text{->}I : lvalue\,[m']} \quad (E22)$$

$$\frac{e \vdash E : exp\,[ptr\,[obj\,[union\,[t,\pi],q]]] \quad \pi \vdash I \triangleleft m \quad m' := qualify\ q\ m}{e \vdash E\text{->}I : lvalue\,[m']} \quad (E23)$$

In the case of the arrow operator, on the other hand, the result is always an l-value. There is only a distinction between structures and unions.

### 8.1.3 Unary operators

The typing semantics of unary operators is described in §6.3.3 of the standard.

POSTFIX INCREMENT

$$\frac{e \vdash E : lvalue\,[m] \quad isScalar(m) \quad isModifiable(m) \quad \tau := datify\ m}{e \vdash E\text{++} : exp\,[\tau]} \quad (E24)$$

POSTFIX DECREMENT

$$\frac{e \vdash E : lvalue\,[m] \quad isScalar(m) \quad isModifiable(m) \quad \tau := datify\ m}{e \vdash E\text{--} : exp\,[\tau]} \quad (E25)$$

PREFIX INCREMENT

$$\frac{e \vdash E : lvalue\,[m] \quad isScalar(m) \quad isModifiable(m) \quad \tau := datify\ m}{e \vdash \text{++}E : exp\,[\tau]} \quad (E26)$$

PREFIX DECREMENT

$$\frac{e \vdash E : lvalue\,[m] \quad isScalar(m) \quad isModifiable(m) \quad \tau := datify\ m}{e \vdash \text{--}E : exp\,[\tau]} \quad (E27)$$

Increment and decrement operators can be applied to modifiable l-values of scalar type. The result is not an l-value.

ADDRESS OPERATOR

$$\frac{e \vdash E : lvalue\,[\alpha]}{e \vdash \&E : exp\,[ptr\,[\alpha]]} \quad (E28)$$

$$\frac{e \vdash E : exp\,[f]}{e \vdash \&E : exp\,[ptr\,[f]]} \quad (E29)$$

The address of an object or function designator is attributed an appropriate pointer type. Notice that an object designator has type $lvalue\,[\alpha]$, i.e. bit-fields are not allowed, and that a function designator has type $exp\,[f]$.

$$\frac{e \vdash E : exp\,[ptr\,[\alpha]]}{e \vdash *E : lvalue\,[\alpha]} \quad (E30)$$ INDIRECTION OPERATOR

$$\frac{e \vdash E : exp\,[ptr\,[f]]}{e \vdash *E : exp\,[f]} \quad (E31)$$

The indirection operator is the inverse of the address operator. Pointers to objects and functions are taken back to object and function designators respectively.

$$\frac{e \vdash E : val\,[\tau] \quad isArithmetic(\tau) \quad \tau' := intPromote\,\tau}{e \vdash +E : val\,[\tau']} \quad (E32)$$ UNARY PLUS

$$\frac{e \vdash E : exp\,[\tau] \quad isArithmetic(\tau) \quad \tau' := intPromote\,\tau}{e \vdash +E : exp\,[\tau']} \quad (E33)$$

$$\frac{e \vdash E : val\,[\tau] \quad isArithmetic(\tau) \quad \tau' := intPromote\,\tau}{e \vdash -E : val\,[\tau']} \quad (E34)$$ UNARY MINUS

$$\frac{e \vdash E : exp\,[\tau] \quad isArithmetic(\tau) \quad \tau' := intPromote\,\tau}{e \vdash -E : exp\,[\tau']} \quad (E35)$$

Unary sign operators take an arithmetic operand and apply the integral promotions. They distinguish between constant and non-constant operands.

$$\frac{e \vdash E : val\,[\tau] \quad isIntegral(\tau) \quad \tau' := intPromote\,\tau}{e \vdash \sim E : val\,[\tau']} \quad (E36)$$ BITWISE NEGATION

$$\frac{e \vdash E : exp\,[\tau] \quad isIntegral(\tau) \quad \tau' := intPromote\,\tau}{e \vdash \sim E : exp\,[\tau']} \quad (E37)$$

The bitwise negation operator takes an integral operand and apply the integral promotions. It distinguishes between constant and non-constant operands.

$$\frac{e \vdash E{==}0 : \theta}{e \vdash !E : \theta} \quad (E38)$$ LOGICAL NEGATION

The logical negation operator is defined in terms of the equality operator; a zero value is taken to be the *false* truth value in C.

$$\frac{e \vdash E : lvalue\,[m] \quad \neg isBitfield\,(m)}{e \vdash \texttt{sizeof}\ E : val\,[size\_t]} \quad (E39)$$ SIZEOF OPERATOR

$$\frac{e \vdash E : exp\,[f]}{e \vdash \texttt{sizeof}\ E : \top} \quad (E40)$$

$$\frac{e \vdash E : exp\,[\tau] \quad isComplete(\tau)}{e \vdash \texttt{sizeof}\ E : val\,[size\_t]} \quad (E41)$$

$$\frac{e \vdash T \equiv \alpha \quad isComplete(\alpha)}{e \vdash \texttt{sizeof(}T\texttt{)} : val\,[size\_t]} \quad (E42)$$

The *sizeof* operator comes in two flavours. The first flavour takes an expression as its operand. The expression must not be a function designator. It must also be of a complete, non bit-field type. Three typing rules are needed to define these requirements; rule E40 is needed to invalidate function designators because of their implicit coercion C4 to function pointers. The second flavour of the *sizeof* operator takes a type name as its operand. Again, the type that is denoted by the name must be complete.

### 8.1.4 Cast operators

The typing semantics of cast operators is described in §6.3.4 of the standard.

$$\frac{e \vdash E : exp\,[v] \quad e \vdash T \equiv obj\,[void, q]}{e \vdash (T)\,E : exp\,[void]} \quad \textit{(E43)}$$

Expressions of any type can be cast to a qualified or unqualified version of the type *void*.

$$\frac{e \vdash E : val\,[\tau] \quad isScalar(\tau) \quad e \vdash T \equiv \alpha' \quad isScalar(\alpha') \quad \tau' := datify\ \alpha'}{e \vdash (T)\,E : val\,[\tau']} \quad \textit{(E44)}$$

$$\frac{e \vdash E : exp\,[\tau] \quad isScalar(\tau) \quad e \vdash T \equiv \alpha' \quad isScalar(\alpha') \quad \tau' := datify\ \alpha'}{e \vdash (T)\,E : exp\,[\tau']} \quad \textit{(E45)}$$

Otherwise, the type $\tau$ of the operand and the specified target type $\tau'$ must be both scalar. A distinction is made between constant and non-constant operands.

### 8.1.5 Multiplicative operators

The typing semantics of multiplicative operators is described in §6.3.5 of the standard.

MULTIPLICATION

$$\frac{\begin{array}{c}e \vdash E_1 : val\,[\tau_1] \qquad e \vdash E_2 : val\,[\tau_2] \\ isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle\end{array}}{e \vdash E_1 * E_2 : val\,[\tau']} \quad \textit{(E46)}$$

$$\frac{\begin{array}{c}e \vdash E_1 : exp\,[\tau_1] \qquad e \vdash E_2 : exp\,[\tau_2] \\ isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle\end{array}}{e \vdash E_1 * E_2 : exp\,[\tau']} \quad \textit{(E47)}$$

DIVISION

$$\frac{\begin{array}{c}e \vdash E_1 : val\,[\tau_1] \qquad e \vdash E_2 : val\,[\tau_2] \\ isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle\end{array}}{e \vdash E_1 / E_2 : val\,[\tau']} \quad \textit{(E48)}$$

$$\frac{\begin{array}{c}e \vdash E_1 : exp\,[\tau_1] \qquad e \vdash E_2 : exp\,[\tau_2] \\ isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle\end{array}}{e \vdash E_1 / E_2 : exp\,[\tau']} \quad \textit{(E49)}$$

The multiplication and division operators are very similar. They take two arithmetic operands and apply the usual arithmetic conversions. They distinguish between constant and non-constant values.

MODULO OPERATOR

$$\frac{\begin{array}{c}e \vdash E_1 : val\,[\tau_1] \qquad e \vdash E_2 : val\,[\tau_2] \\ isIntegral(\tau_1) \quad isIntegral(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle\end{array}}{e \vdash E_1 \% E_2 : val\,[\tau']} \quad \textit{(E50)}$$

$$\frac{\begin{array}{c}e \vdash E_1 : exp\,[\tau_1] \qquad e \vdash E_2 : exp\,[\tau_2] \\ isIntegral(\tau_1) \quad isIntegral(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle\end{array}}{e \vdash E_1 \% E_2 : exp\,[\tau']} \quad \textit{(E51)}$$

The modulo operator is similar to the multiplication and division operators, with the exception that its operands must be integral.

### 8.1.6 Additive operators

The typing semantics of additive operators is described in §6.3.6 of the standard.

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad isArithmetic(\tau_1) \quad e \vdash E_2 : val\,[\tau_2] \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle}{e \vdash E_1 + E_2 : val\,[\tau']} \quad (E52)$$ ADDITION

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad isArithmetic(\tau_1) \quad e \vdash E_2 : exp\,[\tau_2] \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle}{e \vdash E_1 + E_2 : exp\,[\tau']} \quad (E53)$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad e \vdash E_2 : exp\,[\tau_2] \quad isComplete(\alpha_1) \quad isIntegral\,(\tau_2)}{e \vdash E_1 + E_2 : exp\,[ptr\,[\alpha_1]]} \quad (E54)$$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[ptr\,[\alpha_2]] \quad isIntegral(\tau_1) \quad isComplete(\alpha_2)}{e \vdash E_1 + E_2 : exp\,[ptr\,[\alpha_2]]} \quad (E55)$$

When two arithmetic operands are added, the usual arithmetic conversions are applied and a distinction is made between constant and non-constant values. The last two rules are symmetric, with respect to the order of the two operands and define the typing of pointer arithmetic. The one operand must be a pointer to a complete object type, whereas the second must be of an integral type.

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad isArithmetic(\tau_1) \quad e \vdash E_2 : val\,[\tau_2] \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle}{e \vdash E_1 - E_2 : val\,[\tau']} \quad (E56)$$ SUBTRACTION

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad isArithmetic(\tau_1) \quad e \vdash E_2 : exp\,[\tau_2] \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle}{e \vdash E_1 - E_2 : exp\,[\tau']} \quad (E57)$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad e \vdash E_2 : exp\,[\tau_2] \quad isComplete(\alpha_1) \quad isIntegral\,(\tau_2)}{e \vdash E_1 - E_2 : exp\,[ptr\,[\alpha_1]]} \quad (E58)$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad isComplete(\alpha_1) \quad e \vdash E_2 : exp\,[ptr\,[\alpha_2]] \quad isComplete(\alpha_2) \quad isCompatibleQual(\alpha_1, \alpha_2)}{e \vdash E_1 - E_2 : exp\,[ptrdiff\_t]} \quad (E59)$$

The typing rules for subtraction are similar to the ones for addition. The difference lies on the last rule. Pointer subtraction is not symmetric with respect to the order of the two operands. Operands of integral types may be subtracted from pointers to complete types, but not vice versa. However, as specified by the last rule, two pointers to complete types may be subtracted, on condition that the types are compatible ignoring qualifiers. The result is of type $ptrdiff\_t$.

### 8.1.7 Bitwise shift operators

The typing semantics of bitwise shift operators is described in §6.3.7 of the standard.

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad isArithmetic(\tau_1) \quad e \vdash E_2 : val\,[\tau_2] \quad isArithmetic(\tau_2) \quad \tau_1' := intPromote\,\tau_1 \quad \tau_2' := intPromote\,\tau_2}{e \vdash E_1 << E_2 : val\,[\tau_1']} \quad (E60)$$ LEFT SHIFT

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau_1' := intPromote\ \tau_1 \quad e \vdash E_2 : exp\,[\tau_2] \quad \tau_2' := intPromote\ \tau_2}{e \vdash E_1 << E_2 : exp\,[\tau_1']} \quad (E61)$$

RIGHT SHIFT

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau_1' := intPromote\ \tau_1 \quad e \vdash E_2 : val\,[\tau_2] \quad \tau_2' := intPromote\ \tau_2}{e \vdash E_1 >> E_2 : val\,[\tau_1']} \quad (E62)$$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau_1' := intPromote\ \tau_1 \quad e \vdash E_2 : exp\,[\tau_2] \quad \tau_2' := intPromote\ \tau_2}{e \vdash E_1 >> E_2 : exp\,[\tau_1']} \quad (E63)$$

Typing rules for left and right bitwise shift operators are very similar. They take arithmetic operands and apply the integral promotions to both operands. They distinguish between constant and non-constant values.

### 8.1.8 Relational operators

The typing semantics of relational operators is described in §6.3.8 of the standard.

LESS THAN

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \quad e \vdash E_2 : val\,[\tau_2]}{e \vdash E_1 < E_2 : val\,[int]} \quad (E64)$$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \quad e \vdash E_2 : exp\,[\tau_2]}{e \vdash E_1 < E_2 : exp\,[int]} \quad (E65)$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad e \vdash E_2 : exp\,[ptr\,[\alpha_2]] \quad isCompatibleQual(\alpha_1, \alpha_2)}{e \vdash E_1 < E_2 : exp\,[int]} \quad (E66)$$

GREATER THAN

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \quad e \vdash E_2 : val\,[\tau_2]}{e \vdash E_1 > E_2 : val\,[int]} \quad (E67)$$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \quad e \vdash E_2 : exp\,[\tau_2]}{e \vdash E_1 > E_2 : exp\,[int]} \quad (E68)$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad e \vdash E_2 : exp\,[ptr\,[\alpha_2]] \quad isCompatibleQual(\alpha_1, \alpha_2)}{e \vdash E_1 > E_2 : exp\,[int]} \quad (E69)$$

LESS OR EQUAL

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \quad e \vdash E_2 : val\,[\tau_2]}{e \vdash E_1 <= E_2 : val\,[int]} \quad (E70)$$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \quad e \vdash E_2 : exp\,[\tau_2]}{e \vdash E_1 <= E_2 : exp\,[int]} \quad (E71)$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad e \vdash E_2 : exp\,[ptr\,[\alpha_2]] \quad isCompatibleQual(\alpha_1, \alpha_2)}{e \vdash E_1 <= E_2 : exp\,[int]} \quad (E72)$$

GREATER OR EQUAL

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \quad e \vdash E_2 : val\,[\tau_2]}{e \vdash E_1 >= E_2 : val\,[int]} \quad (E73)$$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle}{e \vdash E_1 \texttt{>=} E_2 : exp\,[int]} \quad \textit{(E74)}$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad e \vdash E_2 : exp\,[ptr\,[\alpha_2]] \quad isCompatibleQual(\alpha_1, \alpha_2)}{e \vdash E_1 \texttt{>=} E_2 : exp\,[int]} \quad \textit{(E75)}$$

Typing rules for all relational operators are very similar. The operands may either be arithmetic or pointers to object types, which must be compatible ignoring qualifiers.

### 8.1.9 Equality operators

The typing semantics of equality operators is described in §6.3.9 of the standard.

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad e \vdash E_2 : val\,[\tau_2] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle}{e \vdash E_1 \texttt{==} E_2 : val\,[int]} \quad \textit{(E76)}$$

EQUALITY

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle}{e \vdash E_1 \texttt{==} E_2 : exp\,[int]} \quad \textit{(E77)}$$

If the operands are of arithmetic type, the equality operator simply dinstiguishes between constant and non-constant values.

$$\frac{e \vdash E_1 : exp\,[ptr\,[\phi_1]] \quad e \vdash E_2 : exp\,[ptr\,[\phi_2]] \quad isCompatibleQual(\phi_1, \phi_2)}{e \vdash E_1 \texttt{==} E_2 : exp\,[int]} \quad \textit{(E78)}$$

Pointers to object or function types may be compared for equality. The pointed types must be compatible ignoring qualifiers.

$$\frac{e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad e \vdash E_2 : exp\,[ptr\,[obj\,[void, q]]]}{e \vdash E_1 \texttt{==} E_2 : exp\,[int]} \quad \textit{(E79)}$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[obj\,[void, q]]] \quad e \vdash E_2 : exp\,[ptr\,[\alpha_2]]}{e \vdash E_1 \texttt{==} E_2 : exp\,[int]} \quad \textit{(E80)}$$

A pointer to any object type may be compared for equality to a pointer to a qualified or unqualified version of *void*.

$$\frac{e \vdash E_1 : exp\,[ptr\,[\phi_1]] \quad e \vdash E_2 = \texttt{NULL}}{e \vdash E_1 \texttt{==} E_2 : exp\,[int]} \quad \textit{(E81)}$$

$$\frac{e \vdash E_1 = \texttt{NULL} \quad e \vdash E_2 : exp\,[ptr\,[\phi_2]]}{e \vdash E_1 \texttt{==} E_2 : exp\,[int]} \quad \textit{(E82)}$$

A pointer to any object or function type may be compared for equality to a null pointer constant.

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad e \vdash E_2 : val\,[\tau_2] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle}{e \vdash E_1 \texttt{!=} E_2 : val\,[int]} \quad \textit{(E83)}$$

INEQUALITY

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad isArithmetic(\tau_1) \quad isArithmetic(\tau_2) \quad \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle}{e \vdash E_1 \texttt{!=} E_2 : exp\,[int]} \quad \textit{(E84)}$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\phi_1]] \quad e \vdash E_2 : exp\,[ptr\,[\phi_2]] \quad isCompatibleQual\,(\phi_1, \phi_2)}{e \vdash E_1\,!{=}E_2 : exp\,[int]} \quad \textit{(E85)}$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad e \vdash E_2 : exp\,[ptr\,[obj\,[void, q]]]}{e \vdash E_1\,!{=}E_2 : exp\,[int]} \quad \textit{(E86)}$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[obj\,[void, q]]] \quad e \vdash E_2 : exp\,[ptr\,[\alpha_2]]}{e \vdash E_1\,!{=}E_2 : exp\,[int]} \quad \textit{(E87)}$$

$$\frac{e \vdash E_1 : exp\,[ptr\,[\phi_1]] \quad e \vdash E_2 = \texttt{NULL}}{e \vdash E_1\,!{=}E_2 : exp\,[int]} \quad \textit{(E88)}$$

$$\frac{e \vdash E_1 = \texttt{NULL} \quad e \vdash E_2 : exp\,[ptr\,[\phi_2]]}{e \vdash E_1\,!{=}E_2 : exp\,[int]} \quad \textit{(E89)}$$

Typing rules for the inequality operator are very similar to those for the equality operator.

### 8.1.10 Bitwise logical operators

The typing semantics of bitwise logical operators is described in §§6.3.10, 6.3.11 and 6.3.12 of the standard.

BITWISE CONJUNCTION

$$\frac{\begin{array}{cc} e \vdash E_1 : val\,[\tau_1] & e \vdash E_2 : val\,[\tau_2] \\ isIntegral\,(\tau_1) \quad isIntegral\,(\tau_2) & \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \end{array}}{e \vdash E_1 \& E_2 : val\,[\tau']} \quad \textit{(E90)}$$

$$\frac{\begin{array}{cc} e \vdash E_1 : exp\,[\tau_1] & e \vdash E_2 : exp\,[\tau_2] \\ isIntegral\,(\tau_1) \quad isIntegral\,(\tau_2) & \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \end{array}}{e \vdash E_1 \& E_2 : exp\,[\tau']} \quad \textit{(E91)}$$

BITWISE DISJUNCTION

$$\frac{\begin{array}{cc} e \vdash E_1 : val\,[\tau_1] & e \vdash E_2 : val\,[\tau_2] \\ isIntegral\,(\tau_1) \quad isIntegral\,(\tau_2) & \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \end{array}}{e \vdash E_1 \,|\, E_2 : val\,[\tau']} \quad \textit{(E92)}$$

$$\frac{\begin{array}{cc} e \vdash E_1 : exp\,[\tau_1] & e \vdash E_2 : exp\,[\tau_2] \\ isIntegral\,(\tau_1) \quad isIntegral\,(\tau_2) & \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \end{array}}{e \vdash E_1 \,|\, E_2 : exp\,[\tau']} \quad \textit{(E93)}$$

BITWISE EXCLUSIVE DISJUNCTION

$$\frac{\begin{array}{cc} e \vdash E_1 : val\,[\tau_1] & e \vdash E_2 : val\,[\tau_2] \\ isIntegral\,(\tau_1) \quad isIntegral\,(\tau_2) & \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \end{array}}{e \vdash E_1 \,\hat{}\, E_2 : val\,[\tau']} \quad \textit{(E94)}$$

$$\frac{\begin{array}{cc} e \vdash E_1 : exp\,[\tau_1] & e \vdash E_2 : exp\,[\tau_2] \\ isIntegral\,(\tau_1) \quad isIntegral\,(\tau_2) & \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle \end{array}}{e \vdash E_1 \,\hat{}\, E_2 : exp\,[\tau']} \quad \textit{(E95)}$$

The typing rules for bitwise logical operators are very similar. Both operands must be of integral types and the usual arithmetic conversions are applied. A distinction between constant and non-constant values is made.

### 8.1.11 Logical operators

The typing semantics of logical operators is described in §§6.3.13 and 6.3.14 of the standard. The typing rules are somewhat perplexed by the short-circuit semantics of the two operators. All operands must be of scalar type.

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad e \vdash E_2 : val\,[\tau_2] \quad isScalar(\tau_1) \quad isScalar(\tau_2)}{e \vdash E_1 \&\& E_2 : val\,[int]} \quad (E96)$$

LOGICAL CONJUNCTION

If both operands to the logical conjunction operator are constant values, the result is also a constant value.

$$\frac{\begin{array}{cc} e \vdash E_1 : val\,[\tau_1] & e \vdash E_2 : exp\,[\tau_2] \\ isScalar(\tau_1) \quad isScalar(\tau_2) & \neg checkBoolean_{\tau_1}([\![E_1]\!]\, e) \end{array}}{e \vdash E_1 \&\& E_2 : val\,[int]} \quad (E97)$$

If the left operand is a constant value that evaluates to zero, the result is a constant value. It will also be zero, as is specified by the dynamic semantics for the logical conjunction operator. This rule reflects the short-circuit semantics.

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad isScalar(\tau_1) \quad isScalar(\tau_2)}{e \vdash E_1 \&\& E_2 : exp\,[int]} \quad (E98)$$

Finally, if both operands are non-constant values, the result is also a non-constant value.

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad e \vdash E_2 : val\,[\tau_2] \quad isScalar(\tau_1) \quad isScalar(\tau_2)}{e \vdash E_1 \,||\, E_2 : val\,[int]} \quad (E99)$$

LOGICAL DISJUNCTION

$$\frac{\begin{array}{cc} e \vdash E_1 : val\,[\tau_1] & e \vdash E_2 : exp\,[\tau_2] \\ isScalar(\tau_1) \quad isScalar(\tau_2) & checkBoolean_{\tau_1}([\![E_1]\!]\, e) \end{array}}{e \vdash E_1 \,||\, E_2 : val\,[int]} \quad (E100)$$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad isScalar(\tau_1) \quad isScalar(\tau_2)}{e \vdash E_1 \,||\, E_2 : exp\,[int]} \quad (E101)$$

The typing rules for the logical disjunction operator are similar to the ones for the logical conjunction. The only difference is in the short-circuit semantics: the result value is known to be a constant if the value of the left operand is constant and non-zero.

### 8.1.12 Conditional operator

The typing semantics of the conditional operator is described in §6.3.15 of the standard. The typing rules are again somewhat perplexed by the short-circuit semantics, in the case of arithmetic operands. The first operand must always be of scalar type.

$$\frac{\begin{array}{cccc} e \vdash E : val\,[\tau] & isScalar(\tau) & e \vdash E_1 : val\,[\tau_1] & e \vdash E_2 : exp\,[\tau_2] \\ isArithmetic(\tau_1) & isArithmetic(\tau_2) & \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle & checkBoolean_{\tau}([\![E]\!]\, e) \end{array}}{e \vdash E ? E_1 : E_2 : val\,[\tau']} \quad (E102)$$

$$\frac{\begin{array}{cccc} e \vdash E : val\,[\tau] & isScalar(\tau) & e \vdash E_1 : exp\,[\tau_1] & e \vdash E_2 : val\,[\tau_2] \\ isArithmetic(\tau_1) & isArithmetic(\tau_2) & \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle & \neg checkBoolean_{\tau}([\![E]\!]\, e) \end{array}}{e \vdash E ? E_1 : E_2 : val\,[\tau']} \quad (E103)$$

$$\frac{\begin{array}{cccc} e \vdash E : exp\,[\tau] & isScalar(\tau) & e \vdash E_1 : exp\,[\tau_1] & e \vdash E_2 : exp\,[\tau_2] \\ isArithmetic(\tau_1) & isArithmetic(\tau_2) & \tau' := arithConv\,\langle \tau_1, \tau_2 \rangle & \end{array}}{e \vdash E ? E_1 : E_2 : exp\,[\tau']} \quad (E104)$$

When the second and third operands are arithmetic, the usual arithmetic conversions are applied. The result may only be a constant value in two cases: (i) if the first operand is a constant zero value and the third operand is also constant, as specified by the first rule; or (ii) if the first operand is a constant non-zero value and the second operand is also constant, as specified by the second rule. The third rule covers all other cases.

$$\frac{e \vdash E : exp\,[\tau] \quad isScalar(\tau) \quad e \vdash E_1 : exp\,[struct\,[t,\pi]] \quad e \vdash E_2 : exp\,[struct\,[t,\pi]]}{e \vdash E?E_1 : E_2 : exp\,[struct\,[t,\pi]]} \quad \textit{(E105)}$$

$$\frac{e \vdash E : exp\,[\tau] \quad isScalar(\tau) \quad e \vdash E_1 : exp\,[union\,[t,\pi]] \quad e \vdash E_2 : exp\,[union\,[t,\pi]]}{e \vdash E?E_1 : E_2 : exp\,[union\,[t,\pi]]} \quad \textit{(E106)}$$

$$\frac{e \vdash E : exp\,[\tau] \quad isScalar(\tau) \quad e \vdash E_1 : exp\,[void] \quad e \vdash E_2 : exp\,[void]}{e \vdash E?E_1 : E_2 : exp\,[void]} \quad \textit{(E107)}$$

If the second and third operands are of the same structure or union type[1] or of type *void*, the result is of the same type.

$$\frac{\begin{array}{c}e \vdash E : exp\,[\tau] \quad isScalar(\tau) \quad e \vdash E_1 : exp\,[ptr\,[\phi_1]] \quad e \vdash E_2 : exp\,[ptr\,[\phi_2]] \\ isCompatibleQual(\phi_1,\phi_2) \qquad\qquad \phi' := compositeQual\ \langle \phi_1,\phi_2 \rangle\end{array}}{e \vdash E?E_1 : E_2 : exp\,[ptr\,[\phi']]} \quad \textit{(E108)}$$

The second and third operands may be pointers to compatible object or function types, ignoring qualifiers. In this case, the result has the composite type.

$$\frac{e \vdash E : exp\,[\tau] \quad isScalar(\tau) \quad e \vdash E_1 : exp\,[ptr\,[\phi_1]] \quad e \vdash E_2 = \texttt{NULL}}{e \vdash E?E_1 : E_2 : exp\,[ptr\,[\phi_1]]} \quad \textit{(E109)}$$

$$\frac{e \vdash E : exp\,[\tau] \quad isScalar(\tau) \quad e \vdash E_1 = \texttt{NULL} \quad e \vdash E_2 : exp\,[ptr\,[\phi_2]]}{e \vdash E?E_1 : E_2 : exp\,[ptr\,[\phi_2]]} \quad \textit{(E110)}$$

The second operand may be a null pointer constant; in this case the third operand may be a pointer to an object or function type and the result is of the same type. The same applies symmetrically.

$$\frac{\begin{array}{c}e \vdash E : exp\,[\tau] \quad isScalar(\tau) \quad e \vdash E_1 : exp\,[ptr\,[\alpha_1]] \quad e \vdash E_2 : exp\,[ptr\,[obj\,[void,q_2]]] \\ q_1 = getQualifier\ \alpha_1 \qquad\qquad e \not\vdash E_2 = \texttt{NULL}\end{array}}{e \vdash E?E_1 : E_2 : exp\,[ptr\,[obj\,[void,q_1\ \&\ q_2]]]} \quad \textit{(E111)}$$

$$\frac{\begin{array}{c}e \vdash E : exp\,[\tau] \quad isScalar(\tau) \quad e \vdash E_1 : exp\,[ptr\,[obj\,[void,q_1]]] \quad e \vdash E_2 : exp\,[ptr\,[\alpha_2]] \\ e \not\vdash E_1 = \texttt{NULL} \qquad\qquad q_2 = getQualifier\ \alpha_2\end{array}}{e \vdash E?E_1 : E_2 : exp\,[ptr\,[obj\,[void,q_1\ \&\ q_2]]]} \quad \textit{(E112)}$$

Finally, the second operand may be a pointer to a qualified or unqualified version of *void*; in this case the third operand may be a pointer to an object type and the result is a pointer to *void*, qualified with all the qualifiers of the two operands. The same applies symmetrically. The *void* pointer operand must not be a null pointer constant, in order to disambiguate these two rules from the previous two.

### 8.1.13 Assignment operators

The typing semantics of assignment operators is described in §6.3.16 of the standard.

SIMPLE ASSIGNMENT

$$\frac{e \vdash E_1 : lvalue\,[m] \quad isModifiable(m) \quad \tau := datify\ m \quad e \vdash E_2 \gg \tau}{e \vdash E_1 = E_2 : exp\,[\tau]} \quad \textit{(E113)}$$

The left operand in a simple assignment must be a modifiable l-value and the right operand must be assignable to the type of the left operand.

$$\frac{e \vdash E_1 = E_1 \; op \; E_2 \; : \; exp\,[\tau]}{e \vdash E_1 \; op{=} \; E_2 \; : \; exp\,[\tau]} \quad \textit{(E114)}$$

COMPOSITE ASSIGNMENT

The typing of composite assignments is defined in terms of the typing of simple assignments. The typing rules for the corresponding binary operators guarantee that the operands have arithmetic types, appropriate for each operator.

### 8.1.14 Comma operator

The typing semantics of the comma operator is described in §6.3.17 of the standard.

$$\frac{e \vdash E_1 \; : \; exp\,[\tau_1] \quad e \vdash E_2 \; : \; exp\,[\tau_2]}{e \vdash E_1 , E_2 \; : \; exp\,[\tau_2]} \quad \textit{(E115)}$$

The type of left operand does not affect the type of the result. However, both operands must be expressions and the result is not an l-value, a function designator or a constant expression. The last is specified in §6.4 of the standard.

### 8.1.15 Implicit coercions

A number of implicit coercions are specified in §6.2 of the standard. These coercions are expressed by means of inference rules, which attribute "additional" types for expressions, e.g. in the presence of the declaration "`int x;`", expression "`x`" can have type $lvalue\,[obj\,[int, noqual]]$ but also $exp\,[int]$, as specified by inference rule C1. The first would be used if "`x`" was the left operand of an assignment, whereas the second would would be used in most other places.

$$\frac{e \vdash E \; : \; lvalue\,[obj\,[\tau, q]] \quad isComplete(\tau)}{e \vdash E \; : \; exp\,[\tau]} \quad \textit{(C1)}$$

As specified in §6.2.2.1 of the standard, an l-value that does not have array type is converted to the value stored in the designated object, except when an l-value is explicitly required by the typing semantics of a particular operator. The assumption that objects types are complete is implicit in the standard.

$$\frac{e \vdash E \; : \; lvalue\,[array\,[\alpha, n]]}{e \vdash E \; : \; exp\,[ptr\,[\alpha]]} \quad \textit{(C2)}$$

Similarly, l-values of array types are converted to pointers to array elements, as specified again in §6.2.2.1 of the standard.

$$\frac{e \vdash E \; : \; lvalue\,[bitfield\,[\beta, q, n]] \quad \tau \; = \; datify\;\beta}{e \vdash E \; : \; exp\,[\tau]} \quad \textit{(C3)}$$

This implicit coercion is specified in §6.2.1.1 of the standard. According to it, bit-fields may be used in expressions whenever the corresponding integer types may be used.

$$\frac{e \vdash E \; : \; exp\,[f]}{e \vdash E \; : \; exp\,[ptr\,[f]]} \quad \textit{(C4)}$$

In §6.2.2.1 of the standard it is specified that function designators are converted to pointers to functions.[2]

$$\frac{e \vdash E : val\,[\tau]}{e \vdash E : exp\,[\tau]} \quad \textit{(C5)}$$

Finally, an implicit coercion that is not specified in the standard but is a necessary one for the proposed type systems. Expressions with constant values can be considered as expressions with non-constant values. Thus, expression "`1`" can be attributed type $val\,[int]$, but also $exp\,[int]$.

## 8.2 Auxiliary rules

A number of auxiliary rules is used to define the typing semantics of auxiliary typing judgements, which are used in order to simplify the main typing rules. Most of these rules are very simply defined in terms of predicates, static domain functions or other typing judgements.

### 8.2.1 Typing from declarations

$$\frac{\delta := e[I\ ide]}{e \vdash I \lhd \delta} \quad \textit{(T1)}$$

An identifier $I$ declared in an environment $e$ is simply attributed the identifier type $\delta$ that $e$ associates with it.

$$\frac{m := \pi[I]}{\pi \vdash I \lhd m} \quad \textit{(T2)}$$

The case of identifiers declared in member environments is similar. They are attributed the member types associated with them by the member environments.

### 8.2.2 Type names

$$\frac{\phi := \{\!|T|\!\}\,e}{e \vdash T \equiv \phi} \quad \textit{(T3)}$$

The type that is denoted by a type name $T$ in a type environment $e$ is simply the denotable type that results from applying the static meaning of $T$ to $e$.

### 8.2.3 Assignment rules

This set of rules defines when expression $E$ is "assignable" to a data type $\tau$. The definition is taken from §6.3.16.1 of the standard. One should keep in mind that $E$ is the right operand in a simple assignment, whereas $\tau$ is the data type of an object that will store the value of $E$. Whether the left operand of the assignment is indeed a modifiable l-value is not the issue at this point.

$$\frac{e \vdash E : exp\,[\tau] \quad isArithmetic(\tau) \quad isArithmetic(\tau')}{e \vdash E \gg \tau'} \quad \textit{(A1)}$$

An expression of arithmentic type $\tau$ can be assigned to an object of any arithmetic type $\tau'$.

$$\frac{e \vdash E : exp\,[\tau] \quad isComplete(\tau)}{e \vdash E \gg \tau} \quad \textit{(A2)}$$

An expression of type $\tau$ can be assigned to an object of the same type, provided that it is a complete type.

$$\frac{\begin{array}{c} e \vdash E : exp\,[ptr\,[\phi]] \quad isCompatibleQual(\phi, \phi') \\ q = getQualifier\ \phi \quad q' = getQualifier\ \phi' \quad q \subseteq q' \end{array}}{e \vdash E \gg ptr\,[\phi']} \quad \textit{(A3)}$$

A pointer to a type $\phi$ can be assigned to an object of type pointer to $\phi'$ on condition that $\phi$ and $\phi'$ are compatible, ignoring qualifiers, and that all qualifiers in $\phi$ are present in $\phi'$.

$$\frac{e \vdash E : exp\,[ptr\,[\alpha]] \quad q = getQualifier\ \alpha \quad q \subseteq q'}{e \vdash E \gg ptr\,[obj\,[void, q']]} \quad \textit{(A4)}$$

$$\frac{e \vdash E : exp\,[ptr\,[obj\,[void, q]]] \quad q' = getQualifier\ \alpha' \quad q \subseteq q'}{e \vdash E \gg ptr\,[\alpha']} \quad \textit{(A5)}$$

A pointer to a type $\phi$ can be assigned to an object of type pointer to (possibly qualified) void, and vice versa. The condition again that must be satisfied is that the type pointed by the left operand of the assignment should include all qualifiers of the type pointed by the right operand.

$$\frac{e \vdash E = \texttt{NULL}}{e \vdash E \gg ptr\,[\phi']} \quad \textit{(A6)}$$

A null pointer constant can be assigned to any pointer type.

### 8.2.4 Null pointer constants

$$\frac{e \vdash E : val\,[\tau] \quad isIntegral(\tau) \quad \neg checkBoolean_{\tau}([\![E]\!]\,e)}{e \vdash E = \texttt{NULL}} \quad \textit{(N1)}$$

$$\frac{e \vdash E : val\,[\tau] \quad isIntegral(\tau) \quad \neg checkBoolean_{\tau}([\![E]\!]\,e) \quad e \vdash T \equiv obj\,[void, noqual]}{e \vdash \texttt{(}T\texttt{)}\ E = \texttt{NULL}} \quad \textit{(N2)}$$

As specified in §6.2.2.3 of the standard, an integral constant expression that evaluates to zero, or such an expression cast to type "`void *`", is a null pointer constant.

[1] See deviation D-11 in Section 2.3.

[2] The only exception here is the *sizeof* operator, whose typing semantics in rule E40 prevents this implicit coercion.

# Chapter 9

# Typing semantics of declarations

CHAPTER OVERVIEW

This chapter contains the typing rules for declarations, i.e. inference rules aiming primarily at associating declarations and related phrases with phrase types by means of formal typing derivations. The typing rules for external declarations are defined in Section 9.1, whereas those for other declarations are defined in Section 9.2.

## 9.1 External declarations

The typing semantics of external declarations is specified in §6.7 of the standard. Its definition is straightforward.

$$\frac{e \vdash \textit{external-declaration-list} : \mathit{xdecl}}{e \vdash \textit{external-declaration-list} : \mathit{tunit}} \quad \textit{(X1)}$$

TRANSLATION UNITS

Any valid list of external declarations forms a translation unit.

$$\frac{e \vdash \textit{external-declaration} : \mathit{xdecl} \qquad e \vdash \textit{external-declaration-list} : \mathit{xdecl}}{e \vdash \textit{external-declaration external-declaration-list} : \mathit{xdecl}} \quad \textit{(X2)}$$

EXTERNAL DECLARATION LISTS

This typing rule allows forming sequences of external declarations, provided that the parts themselves are valid external declarations.

$$\frac{e \vdash \textit{declaration} : \mathit{decl}}{e \vdash \textit{declaration} : \mathit{xdecl}} \quad \textit{(X3)}$$

EXTERNAL DECLARATIONS

Any valid declaration also forms an external declaration.

$$\frac{\begin{array}{c} e \vdash \textit{declarator} : \mathit{dtor}\,[\mathit{func}\,[\tau, p]] \qquad e' := \mathit{rec}\,(\mathcal{F}\{\!|\textit{declarator}|\!\};\ \{\!|\textit{declaration-list}|\!\})\ (\uparrow e) \\ e' \vdash \textit{declaration-list} : \mathit{decl} \qquad\qquad e' \vdash \textit{statement-list} : \mathit{stmt}\,[\tau] \end{array}}{e \vdash \textit{declaration-specifiers declarator}\ \{\ \textit{declaration-list statement-list}\ \} : \mathit{xdecl}} \quad \textit{(X4)}$$

This typing rule reflects the typing semantics of function definitions, as specified in §6.7.1 of the standard, and is probably the most complicated one in the typing semantics of declarations. The first condition is that the declarator must indeed be a function declarator. Then, the type environment for the function's body is constructed and fixed, by first adding the formal parameters and then adding all declarations from the function's declaration list. In addition, the declaration and statement lists must be valid and the returned type from the statement list must indeed be the one specified in the function's declarator.

## 9.2 Declarations

The typing semantics of declarations are specified in §6.5 of the standard. It should be mentioned that not all non-terminal symbols from the abstract syntax grammar of C need be attributed types. In some cases, all the necessary information is already present in the type environment, e.g. in rules D8 and D9 about the typing of declarators, and can be extracted from there. The typing rules are trivial in many cases.

DECLARATION LISTS

$$\frac{}{e \vdash \epsilon : \mathit{decl}} \quad \textit{(D1)}$$

$$\frac{e \vdash \textit{declaration} : \mathit{decl} \qquad e \vdash \textit{declaration-list} : \mathit{decl}}{e \vdash \textit{declaration declaration-list} : \mathit{decl}} \quad \textit{(D2)}$$

Declaration lists may be empty or may consist of one or more declarations.

DECLARATIONS

$$\frac{e \vdash \textit{init-declarator-list} : \mathit{idtor}}{e \vdash \textit{declaration-specifiers init-declarator-list} \; ; : \mathit{decl}} \quad \textit{(D3)}$$

The typing rule for declarations only requires that the initializer declarator list is valid.

### 9.2.1 Declarators

DECLARATOR LISTS WITH INITIALIZERS

$$\frac{}{e \vdash \epsilon : \mathit{idtor}} \quad \textit{(D4)}$$

$$\frac{e \vdash \textit{init-declarator} : \mathit{idtor} \qquad e \vdash \textit{init-declarator-list} : \mathit{idtor}}{e \vdash \textit{init-declarator init-declarator-list} : \mathit{idtor}} \quad \textit{(D5)}$$

A list of declarators with initializers may be either empty or may consist of several declarators with initializers.

DECLARATORS WITH INITIALIZERS

$$\frac{e \vdash \textit{declarator} : \mathit{dtor}\,[\phi]}{e \vdash \textit{declarator} : \mathit{idtor}} \quad \textit{(D6)}$$

$$\frac{e \vdash \textit{declarator} : \mathit{dtor}\,[\alpha] \qquad e \vdash \textit{initializer} : \mathit{init}\,[\alpha]}{e \vdash \textit{declarator} = \textit{initializer} : \mathit{idtor}} \quad \textit{(D7)}$$

Declarators may or may not have initializers. If no initializer is present, any denotable type may be specified by the declarator. If an initializer is present, the type specified by the declarator must be an object type and the initializer must be valid for this object type.

SIMPLE DECLARATORS

$$\frac{e \vdash I \triangleleft \mathit{normal}\,[\phi]}{e \vdash I : \mathit{dtor}\,[\phi]} \quad \textit{(D8)}$$

$$\frac{e \vdash I \triangleleft \mathit{typedef}\,[\phi]}{e \vdash I : \mathit{dtor}\,[\phi]} \quad \textit{(D9)}$$

Simple declarators correspond to ordinary identifiers that may be object designators or type synonyms. The corresponding denotable types are extracted from the type environment.

ARRAY DECLARATORS

$$\frac{e \vdash \textit{declarator} : \mathit{dtor}\,[\phi] \qquad e \vdash \textit{constant-expression} : \mathit{val}\,[\tau] \qquad \mathit{isIntegral}\,(\tau)}{e \vdash \textit{declarator} \; [\; \textit{constant-expression} \; ] : \mathit{dtor}\,[\phi]} \quad \textit{(D10)}$$

$$\frac{e \vdash \textit{declarator} : \textit{dtor}[\phi]}{e \vdash \textit{declarator}\ [\ ] : \textit{dtor}[\phi]} \quad \textit{(D11)}$$

In the case of array declarators, the necessary type information is extracted from the type environment.

$$\frac{e \vdash \textit{declarator} : \textit{dtor}[\phi]}{e \vdash *\ \textit{type-qualifier declarator} : \textit{dtor}[\phi]} \quad \textit{(D12)}$$

Again the type information is extracted from the type environment.

$$\frac{\neg(\mathcal{T}\{\!|\textit{declarator}|\!\}) \qquad e \vdash \textit{declarator} : \textit{dtor}[\phi]}{e \vdash \textit{declarator}\ (\ \textit{parameter-type-list}\ ) : \textit{dtor}[\phi]} \quad \textit{(D13)}$$

FUNCTION DECLARATORS

$$\frac{\mathcal{T}\{\!|\textit{declarator}|\!\} \qquad e \vdash \textit{declarator} : \textit{dtor}[\textit{func}[\tau, p]] \qquad e' := \textit{rec}\ (\mathcal{F}\{\!|\textit{parameter-type-list}|\!\})\ (\uparrow e) \qquad e' \vdash \textit{parameter-type-list} : \textit{prot}[p]}{e \vdash \textit{declarator}\ (\textit{parameter-type-list}) : \textit{dtor}[\textit{func}[\tau, p]]} \quad \textit{(D14)}$$

In the case of function declarators, two subcases must be distinguished. If the function declarator is not "terminal", in the sence that the declarator it contains is not a simple identifier, then the necessary type information is simply extracted from the type environment. On the other hand, if the function declarator is "terminal", the type information is again extracted from the type environment and the parameter type list must be valid and must correspond to the function's type.

### 9.2.2 Function prototypes and parameters

$$\frac{}{e \vdash \epsilon : \textit{prot}[p_\circ]} \quad \textit{(D15)}$$

PARAMETER TYPE LISTS

$$\frac{p := \textit{ellipsis}\ p_\circ}{e \vdash \ldots : \textit{prot}[p]} \quad \textit{(D16)}$$

$$\frac{e \vdash \textit{parameter-declaration} : \textit{par}[\tau] \qquad e \vdash \textit{parameter-type-list} : \textit{prot}[p] \qquad p' := \tau \lessdot p}{e \vdash \textit{parameter-declaration parameter-type-list} : \textit{prot}[p']} \quad \textit{(D17)}$$

The cases of empty and ellipsis parameter type lists are straightforward. In the case of a parameter type list consisting of a series of parameter declarations, all components must be valid and the type of the first parameter is prepended to the function prototype that corresponds to all other parameters.

$$\frac{e \vdash \textit{declarator} : \textit{dtor}[\textit{obj}[\tau, q]]}{e \vdash \textit{declaration-specifiers declarator} : \textit{par}[\tau]} \quad \textit{(D18)}$$

PARAMETER DECLARATIONS

In the case of parameter declarations, the necessary type information is extracted by the type environment.

### 9.2.3 Initializations

The typing semantics of initializations is specified in §6.5.7 of the standard. Deviation D-6 stated in Section 2.3 should be kept in mind.

$$\frac{e \vdash E \gg \tau \qquad \textit{isScalar}(\tau) \vee \textit{isStructUnion}(\tau)}{e \vdash E : \textit{init}[\textit{obj}[\tau, q]]} \quad \textit{(I1)}$$

SIMPLE INITIALIZATION

$$\frac{e \vdash E \gg \tau \qquad \tau = datify\ \beta}{e \vdash E : init\,[bitfield\,[\beta, q, n]]} \quad (I2)$$

An expression that can be assigned to a scalar, structure or union type is a valid initializer for such an object. Initializers for bit-fields must be valid initializers for the corresponding data types.

STRING INITIALIZATION

$$\frac{isStringLit(E) \qquad isCompatibleQual(\alpha, obj\,[char, q])}{e \vdash E : init\,[array\,[a, n]]} \quad (I3)$$

$$\frac{isWideStringLit(E) \qquad isCompatibleQual(\alpha, obj\,[wchar\_t, q])}{e \vdash E : init\,[array\,[a, n]]} \quad (I4)$$

String and wide string literals are valid initializers for arrays of the appropriate character type, or any type compatible to that. The length of the arrays is of no importance here.

AGGREGATE INITIALIZATION

$$\frac{e \vdash \textit{initializer-list} : init\text{-}a\,[\alpha]}{e \vdash \{\ \textit{initializer-list}\ \} : init\,[array\,[\alpha, n]]} \quad (I5)$$

$$\frac{e \vdash \textit{initializer-list} : init\text{-}s\,[\pi]}{e \vdash \{\ \textit{initializer-list}\ \} : init\,[obj\,[struct\,[t, \pi], q]]} \quad (I6)$$

$$\frac{e \vdash \textit{initializer-list} : init\text{-}u\,[\pi]}{e \vdash \{\ \textit{initializer-list}\ \} : init\,[obj\,[union\,[t, \pi], q]]} \quad (I7)$$

Except for the case of rules I3 and I4, initializers for aggregate types must be bracketed lists of initializers. Three cases are distinguished: array, structure and union initializers. The initializer list in each case must be of the appropriate phrase type.

ARRAY INITIALIZATION

$$\frac{e \vdash \textit{initializer} : init\,[\alpha]}{e \vdash \textit{initializer} : init\text{-}a\,[\alpha]} \quad (I8)$$

$$\frac{e \vdash \textit{initializer} : init\,[\alpha] \qquad e \vdash \textit{initializer-list} : init\text{-}a\,[\alpha]}{e \vdash \textit{initializer initializer-list} : init\text{-}a\,[\alpha]} \quad (I9)$$

An initializer list for an array type must consist of a sequence of one or more initializers for the type of the array's elements.

STRUCTURE INITIALIZATION

$$\frac{e \vdash \textit{initializer} : init\,[m] \qquad \langle I, m, \pi' \rangle := \Downarrow \pi}{e \vdash \textit{initializer} : init\text{-}s\,[\pi]} \quad (I10)$$

$$\frac{e \vdash \textit{initializer} : init\,[m] \qquad \langle I, m, \pi' \rangle := \Downarrow \pi \qquad e \vdash \textit{initializer-list} : init\text{-}s\,[\pi']}{e \vdash \textit{initializer initializer-list} : init\text{-}s\,[\pi]} \quad (I11)$$

An initializer for a structure must consist of a sequence of one or more initializers for the structure's members, in the order in which the members were declared. The number of initializers in the sequence needs not be equal to the number of members in the structure.

UNION INITIALIZATION

$$\frac{e \vdash \textit{initializer} : init\,[m] \qquad \langle m, \pi' \rangle := \Downarrow \pi}{e \vdash \textit{initializer} : init\text{-}u\,[\pi]} \quad (I12)$$

An initializer for a union must consist of a single initializer for the union's first declared member.

# Chapter 10

# Typing semantics of statements

CHAPTER OVERVIEW

This chapter contains the typing rules for statements, i.e. inference rules aiming primarily at associating statements and statement lists with phrase types of the form $stmt\ [\tau]$ by means of formal typing derivations. Section 10.1 defines the main typing relation for statement lists, whereas Section 10.2 does the same for statements. The structure of the latter corresponds roughly to the structure of §6.6 of the standard. In Section 10.3 a final typing rule that corresponds to missing optional expressions is defined.

## 10.1 Statement lists

The typing semantics of statement lists is only indirectly defined in the standard.

$$\frac{}{e \vdash \epsilon : stmt\ [\tau]} \quad (S1)$$

EMPTY STATEMENT LIST

An empty statement list can be attributed any type of the form $stmt\ [\tau]$.

$$\frac{e \vdash \textit{statement} : stmt\ [\tau] \qquad e \vdash \textit{statement-list} : stmt\ [\tau]}{e \vdash \textit{statement statement-list} : stmt\ [\tau]} \quad (S2)$$

NON-EMPTY STATEMENT LIST

A non-empty statement list can be attributed type $stmt\ [\tau]$ if all its components can be attributed the same type.

## 10.2 Statements

The typing semantics of statements is defined in §6.6 of the standard. Further distinction is made between different kinds of statements.

### 10.2.1 Empty and expression statements

The typing semantics of the empty and expression statements is defined in §6.6.3 of the standard.

$$\frac{}{e \vdash \texttt{;} : stmt\ [\tau]} \quad (S3)$$

EMPTY STATEMENT

An empty statement can be attributed any phrase type of the form $stmt\ [\tau]$.

EXPRESSION STATEMENT
$$\frac{e \vdash \textit{expression} : \textit{exp}\,[\tau']}{e \vdash \textit{expression}\ \texttt{;} : \textit{stmt}\,[\tau]} \quad \textit{(S4)}$$

An expression statement can also attributed any phrase type of the form $\textit{stmt}\,[\tau]$, on condition that its expression is attributed a valid expression type.

### 10.2.2 Compound statement

The typing semantics of compound statements is defined in §6.6.2 of the standard.

BLOCKS
$$\frac{e' := \textit{rec}\,\{|\textit{declaration-list}|\}\,(\uparrow e) \quad e' \vdash \textit{declaration-list} : \textit{decl} \quad e' \vdash \textit{statement-list} : \textit{stmt}\,[\tau]}{e \vdash \{\ \textit{id declaration-list statement-list}\ \} : \textit{stmt}\,[\tau]} \quad \textit{(S5)}$$

Compound statements or blocks can be attributed type $\textit{stmt}\,[\tau]$ on two conditions: (i) its declaration list can be attributed type $\textit{decl}$; and (ii) its statement list can be attributed type $\textit{stmt}\,[\tau]$.

### 10.2.3 Selection statements

The typing semantics of selection statements is defined in §6.6.4 of the standard.

IF-THEN STATEMENT
$$\frac{e \vdash \textit{expression} : \textit{exp}\,[\tau'] \quad \textit{isScalar}(\tau') \quad e \vdash \textit{statement} : \textit{stmt}\,[\tau]}{e \vdash \texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement} : \textit{stmt}\,[\tau]} \quad \textit{(S6)}$$

IF-THEN-ELSE STATEMENT
$$\frac{\begin{array}{cc} e \vdash \textit{expression} : \textit{exp}\,[\tau'] & \textit{isScalar}(\tau') \\ e \vdash \textit{statement}_1 : \textit{stmt}\,[\tau] & e \vdash \textit{statement}_2 : \textit{stmt}\,[\tau] \end{array}}{e \vdash \texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement}_1\ \texttt{else}\ \textit{statement}_2 : \textit{stmt}\,[\tau]} \quad \textit{(S7)}$$

In both forms of the *if* statement, the condition must be attributed a valid scalar expression type. The types of the two clauses determine the type of the whole statement. If the *if* statement has an *else* clause, then both clauses must be attributed the common result type.

SWITCH STATEMENT
$$\frac{e \vdash \textit{expression} : \textit{exp}\,[\tau'] \quad \textit{isIntegral}(\tau') \quad \tau'' := \textit{intPromote}(\tau') \quad e \vdash \textit{statement} : \textit{stmt}\,[\tau]}{e \vdash \texttt{switch (}\ \textit{expression}\ \texttt{)}\ \textit{statement} : \textit{stmt}\,[\tau]} \quad \textit{(S8)}$$

The controlling expression of the *switch* statement must be attributed a valid integral expression type. The statement forming its body determines the type of the whole statement.

### 10.2.4 Labeled statements

The typing semantics of labeled statements is defined in §6.6.1 of the standard. In general, the type of a labeled statement is determined by the type of the underlying statement, without the label.

CASE LABEL
$$\frac{e \vdash \textit{constant-expression} : \textit{val}\,[\tau'] \quad \textit{isIntegral}(\tau') \quad e \vdash \textit{statement} : \textit{stmt}\,[\tau]}{e \vdash \texttt{case}\ \textit{constant-expression}\ \texttt{:}\ \textit{statement} : \textit{stmt}\,[\tau]} \quad \textit{(S9)}$$

As specified in §6.6.4.2 of the standard, the expression present in a *case*-labeled statement must be a constant integral expression.

DEFAULT LABEL
$$\frac{e \vdash \textit{statement} : \textit{stmt}\,[\tau]}{e \vdash \texttt{default :}\ \textit{statement} : \textit{stmt}\,[\tau]} \quad \textit{(S10)}$$

$$\frac{e \vdash \textit{statement} : stmt\,[\tau]}{e \vdash I : \textit{statement} : stmt\,[\tau]} \quad \textit{(S11)}$$ IDENTIFIER LABELS

The typing semantics of these two cases is straightforward.

### 10.2.5 Iteration statements

The typing semantics of iteration statements is defined in §6.6.5 of the standard. In general, the type of an iteration statement is determined by the type of its body.

$$\frac{e \vdash \textit{expression} : exp\,[\tau'] \quad isScalar(\tau') \quad e \vdash \textit{statement} : stmt\,[\tau]}{e \vdash \texttt{while (}\ \textit{expression}\ \texttt{)}\ \textit{statement} : stmt\,[\tau]} \quad \textit{(S12)}$$ WHILE STATEMENT

$$\frac{e \vdash \textit{statement} : stmt\,[\tau] \quad e \vdash \textit{expression} : exp\,[\tau'] \quad isScalar(\tau')}{e \vdash \texttt{do}\ \textit{statement}\ \texttt{while (}\ \textit{expression}\ \texttt{) ;} : stmt\,[\tau]} \quad \textit{(S13)}$$ DO-WHILE STATEMENT

The typing semantics of the *while* and *do* statements is similar. The controlling expression must be of a valid scalar expression type.

$$\frac{\begin{array}{ccc} e \vdash \textit{expression-optional}_1 : exp\,[\tau_1] & & e \vdash \textit{expression-optional}_3 : exp\,[\tau_3] \\ e \vdash \textit{expression-optional}_2 : exp\,[\tau_2] & isScalar(\tau_2) & e \vdash \textit{statement} : stmt\,[\tau] \end{array}}{e \vdash \texttt{for (}\ \textit{expression-optional}_1\ \texttt{;}\ \textit{expression-optional}_2\ \texttt{;}\ \textit{expression-optional}_3\ \texttt{)}\ \textit{statement} : stmt\,[\tau]} \quad \textit{(S14)}$$ FOR STATEMENT

The typing of the *for* statement is slightly more complicated. All expressions must be attributed valid expression types. In addition, the second expression, which controls the statement, must be of scalar type.

### 10.2.6 Jump statements

The typing semantics of jump statements is defined in §6.6.6 of the standard.

$$\frac{}{e \vdash \texttt{continue ;} : stmt\,[\tau]} \quad \textit{(S15)}$$ CONTINUE STATEMENT

$$\frac{}{e \vdash \texttt{break ;} : stmt\,[\tau]} \quad \textit{(S16)}$$ BREAK STATEMENT

$$\frac{}{e \vdash \texttt{goto}\ I\ \texttt{;} : stmt\,[\tau]} \quad \textit{(S17)}$$ GOTO STATEMENT

These three statements can be attributed any type of the form $stmt\,[\tau]$.

$$\frac{}{e \vdash \texttt{return ;} : stmt\,[\tau]} \quad \textit{(S18)}$$ RETURN STATEMENT

According to the standard, a *return* statement that does not specify the returned value can be used in any function, provided that the returned value is not used. Thus, this form of the return statement can be attributed any type of the form $stmt\,[\tau]$.

$$\frac{e \vdash \textit{expression} \gg \tau}{e \vdash \texttt{return}\ \textit{expression}\ \texttt{;} : stmt\,[\tau]} \quad \textit{(S19)}$$

This form of the *return* statement is the one that determines the types of all statements. It can be attributed type $stmt\,[\tau]$, on condition that the specified expression is assignable to the data type $\tau$.

## 10.3 Optional expressions

MISSING EXPRESSION

$$\frac{}{e \vdash \epsilon : val\,[int]} \quad \textit{(S20)}$$

In *for* statements, it is possible to omit any of the three controlling expressions. In order to preserve the simple typing semantics of the *for* statement, such missing expressions are attributed type $val\,[int]$, which is a valid scalar type and can be used in the desired way. This type is preferred over $exp\,[int]$, since missing optional expressions could be replaced by the constant expression "`1`".

# Part IV

# Dynamic semantics

# Chapter 11

# Dynamic semantic domains

CHAPTER OVERVIEW

This chapter defines the domains that are used for the description of the dynamic semantics of C. Domains are defined together with operations that are allowed on their elements. Section 11.1 discusses the structure of dynamic semantic domains. In Section 11.2 a few auxiliary domains are defined, whereas Section 11.3 defines the domains that are used for the representation of C's types. The dynamic semantics of C is defined by means of a number of monads. The value monad is defined in Section 11.4 and the powerdomain monad in Section 11.5. Section 11.6 presents an abstract definition of the program state, and the definition of monads continues in Section 11.7 with the continuation monad, Section 11.8 with the resumption monad transformer and Section 11.9 with the monad used for expressions. In Section 11.10 and Section 11.11 the dynamic domains for environments and scopes are defined respectively, whereas Section 11.12 presents monads that are responsible for statement semantics. Finally, in Section 11.14 a set of auxiliary functions is defined.

## 11.1 Domain ordering

The domain ordering relation is easily defined for most dynamic semantic domains, since these are typically defined by using standard domain constructors. This ordering is again very important. It represents execution properties of C programs. Bottom elements model non-termination, as is usually the case with dynamic semantics and top elements model the occurence of errors, usually run-time errors. Intermediate values represent results of computations which produce at least some non-erroneous results. Non-termination and errors are propagated when necessary by various operations of the dynamic domains and by monad operations.

## 11.2 Auxiliary domains

OBJECT IDENTIFIERS

■ $h$ : $\mathbf{Obj}$ *(undefined)*

The elements of domain $\mathbf{Obj}$ are used to uniquely identify objects in memory. A complete definition of this domain is not given here. However, it is expected that information about object types be present in elements of $\mathbf{Obj}$. It should be noted that objects contained in other objects, e.g. array elements or structure members, are not assigned separate identifiers: elements of $\mathbf{Obj}$ correspond to the largest possible objects.

▶ $newObject_\alpha$ : $\mathbf{Obj}$

Function $newObject_\alpha$ returns a fresh identifier for a new object of type $\alpha$.

FUNCTION IDENTIFIERS ■ $f$ : **Fun** *(undefined)*

In the same way, domain **Fun** represents unique identifiers for functions. The types of functions are kept separately and need not be present in elements of **Fun**.

▶ $newFunction$ : **Fun**

Function $newFunction$ returns a fresh identifier for a new function.

ADDRESSES ■ $a$ : **Addr** = **Obj** × **Offset**

The addresses of objects in memory are represented by elements of domain **Addr**. Such addresses contain two pieces of information: the (possibly) larger object that completely contains the addressed object and the offset of the addressed object in the larger object.

■ $j$ : **Offset** = **N**
■ $b$ : **BitOfs** = **N**

The domain of offsets, as well as that of offsets to specific bits in a byte, is simply a synonym for the domain of integer numbers.

▶ $addrOffset_\alpha$ : $\mathbf{N} \to \mathbf{Addr} \to \mathbf{Addr}$
$addrOffset_\alpha = \lambda n.\, \lambda a.\, \mathbf{let}\ \langle h, j\rangle = a\ \mathbf{in}\ \langle h, j + n \cdot sizeof(\alpha)\rangle$

Function $addrOffset_\alpha$ returns the address of an object of type $\alpha$ that is displaced by $n$ positions from a similar object with given address $a$. This function is used for calculating the addresses of array elements.

## 11.3 Types

The domains defined in this section represent dynamic elements of various kinds of static types, with the exception of phrase types that are treated in a different way in the next chapters. Table 11.1 presents the definitions of dynamic semantic domains for types. It should be noted that domains such as **VC** are flat domains whose elements represent the values of the corresponding types. All integral domains, e.g. **VSI** and **VUL**, are subdomains of **N**. Pointers are represented as addresses or a special value denoting the null pointer. The dynamic meaning of structures and unions, treated as data types, is a mapping returning their members' values. The domain constructor $[\![\pi \Downarrow I]\!]_{dat}$ is defined in Section 11.10.2.

The dynamic meaning of object types is straightforward. Single objects are represented by their addresses and arrays by functions returning the meanings of their elements. Function types are represented by functions from the dynamic meaning of their arguments to the dynamic meaning of an expression computation, resulting in the returned type; $[\![p]\!]_{Prot}$ is defined in Section 11.10.3 and monad $\mathsf{G}$ in Section 11.9. The definition of dynamic semantic domains for denotable, member and value types is also straightforward. It should also be mentioned that if two types $x$ and $y$ are compatible, i.e. $isCompatible(x, y)$, then the corresponding dynamic semantic domains are also compatible, i.e. $[\![x]\!] = [\![y]\!]$.

**Table 11.1:** Dynamic semantic domains for various kinds of static types.

**Data types**

$$
\begin{array}{lcl}
[\![ \mathit{void} ]\!]_{dat} & = & \mathbf{U} \\
[\![ \mathit{char} ]\!]_{dat} & = & \mathbf{VC} \\
[\![ \mathit{signed\text{-}char} ]\!]_{dat} & = & \mathbf{VSC} \\
[\![ \mathit{unsigned\text{-}char} ]\!]_{dat} & = & \mathbf{VUC} \\
[\![ \mathit{short\text{-}int} ]\!]_{dat} & = & \mathbf{VSS} \\
[\![ \mathit{unsigned\text{-}short\text{-}int} ]\!]_{dat} & = & \mathbf{VUC} \\
[\![ \mathit{int} ]\!]_{dat} & = & \mathbf{VSI} \\
[\![ \mathit{unsigned\text{-}int} ]\!]_{dat} & = & \mathbf{VUI} \\
[\![ \mathit{long\text{-}int} ]\!]_{dat} & = & \mathbf{VSL} \\
[\![ \mathit{unsigned\text{-}long\text{-}int} ]\!]_{dat} & = & \mathbf{VUL} \\
[\![ \mathit{float} ]\!]_{dat} & = & \mathbf{VF} \\
[\![ \mathit{double} ]\!]_{dat} & = & \mathbf{VD} \\
[\![ \mathit{long\text{-}double} ]\!]_{dat} & = & \mathbf{VLD} \\
[\![ \mathit{ptr}\,[\alpha] ]\!]_{dat} & = & \mathbf{Addr} \oplus \mathbf{U} \\
[\![ \mathit{ptr}\,[f] ]\!]_{dat} & = & \mathbf{Fun} \oplus \mathbf{U} \\
[\![ \mathit{enum}\,[\epsilon] ]\!]_{dat} & = & \mathbf{VE}_{\epsilon} \\
[\![ \mathit{struct}\,[t,\pi] ]\!]_{dat} & = & I : \mathbf{Ide} \twoheadrightarrow [\![ \pi \Downarrow I ]\!]_{dat} \\
[\![ \mathit{union}\,[t,\pi] ]\!]_{dat} & = & I : \mathbf{Ide} \twoheadrightarrow [\![ \pi \Downarrow I ]\!]_{dat}
\end{array}
$$

**Object types**

$$
\begin{array}{lcl}
[\![ \mathit{obj}\,[\tau,q] ]\!]_{obj} & = & \mathbf{Addr} \\
[\![ \mathit{array}\,[\alpha,n] ]\!]_{obj} & = & \mathbf{N} \to [\![ \alpha ]\!]_{obj}
\end{array}
$$

**Function types**

$$
[\![ \mathit{func}\,[\tau,p] ]\!]_{fun} = [\![ p ]\!]_{Prot} \to \mathsf{G}([\![ \tau ]\!]_{dat})
$$

**Denotable types**

$$
\begin{array}{lcl}
[\![ \alpha ]\!]_{den} & = & [\![ \alpha ]\!]_{obj} \\
[\![ f ]\!]_{den} & = & \mathbf{Fun}
\end{array}
$$

**Member types**

$$
\begin{array}{lcl}
[\![ \alpha ]\!]_{mem} & = & [\![ \alpha ]\!]_{obj} \\
[\![ \mathit{bitfield}\,[\tau,q,n] ]\!]_{mem} & = & \mathbf{Addr} \times \mathbf{BitOfs}
\end{array}
$$

**Value types**

$$
\begin{array}{lcl}
[\![ \tau ]\!]_{val} & = & [\![ \tau ]\!]_{dat} \\
[\![ f ]\!]_{val} & = & \mathbf{Fun}
\end{array}
$$

OPERATIONS ON ADDRESSES

- $\mathit{toAddr}_{\alpha} : [\![ \alpha ]\!]_{obj} \to \mathbf{Addr}$

  $\mathit{toAddr}_{obj\,[\tau,q]} = \mathit{id}$

  $\mathit{toAddr}_{array\,[\alpha,n]} = \lambda d_f .\ \mathit{toAddr}_{\alpha}\ (d_f\ 0)$

- $\mathit{fromAddr}_{\alpha} : \mathbf{Addr} \to [\![ \alpha ]\!]_{obj}$

  $\mathit{fromAddr}_{obj\,[\tau,q]} = \mathit{id}$

  $\mathit{fromAddr}_{array\,[\alpha,n]} = \lambda a .\ \lambda k .\ (k \geq 0) \wedge (k < n) \to \mathit{fromAddr}_{\alpha}\ (\mathit{addrOffset}_{\alpha}\ k\ a)\,,\ \top$

These two functions convert between addresses and dynamic meanings of objects. In the case of single objects, their addresses are identical with their dynamic meanings. The address of an array is the address of its first element.

TYPE CASTING

- $\mathit{cast}_{\tau \mapsto \tau'} : [\![ \tau ]\!]_{dat} \to [\![ \tau' ]\!]_{dat}$

This function models the dynamic semantics of type casting. It converts a dynamic value of type $\tau$ to one of type $\tau'$. Its definition is omitted here.

ZERO VALUES

- $\mathit{zeroValue}_{\tau} : [\![ \tau ]\!]_{dat}$
- $\mathit{checkBoolean}_{\tau} : [\![ \tau ]\!]_{dat} \to \mathbf{T}$

These two functions create and detect zero dynamic values of a given data type $\tau$. Zero values are specially treated by C in initializations and pseudo-boolean conditions. Function $\mathit{zeroValue}_{\tau}$ returns a zero value of type $\tau$, while function $\mathit{checkBoolean}_{\tau}$ returns $\mathit{true}$ if its parameter is non-zero, $\mathit{false}$ otherwise. Their definitions are omitted here.

## 11.4 Value monad

The trivial identity monad is used for representing the computation of constant values, that is, computations that do not affect the program state or any other part of the abstract interpreter's environment. The definition of monad $\mathsf{V}$ is given below.

DEFINITION ■ $v \;:\; \mathsf{V}(D) \;=\; D$

Equality and domain ordering are trivially defined, and the definitions of the monad's operations are also trivial. The monad's unit is defined by:

▶ $unit_{\mathsf{V}} \;:\; D \to \mathsf{V}(D)$
$unit_{\mathsf{V}} \;=\; id$

and the bind operator by:

▶ $\cdot *_{\mathsf{V}} \cdot \;:\; \mathsf{V}(A) \times (A \to \mathsf{V}(B)) \to \mathsf{V}(B)$
$v *_{\mathsf{V}} f \;=\; f\, v$

It is easy to see that monad $\mathsf{V}$ satisfies the three monad laws and preserves bottom and top elements.

ERRORS An operation for generating errors can also be defined, using the top element as a representation for errors. However, it should be noted that errors are only propagated by using the bind operator on strict functions.

▶ $error_{\mathsf{V}} \;:\; \mathsf{V}(D)$
$error_{\mathsf{V}} \;=\; \top$

LIFTING Finally, a polymorphic operation for converting elements of type $\mathsf{E}(A)$ to elements of type $\mathsf{V}(A)$ is required in the sequel. This operation needs only distinguish the case of errors.

▶ $lift_{\mathsf{E}\to\mathsf{V}} \;:\; \mathsf{E}(A) \to \mathsf{V}(A)$
$lift_{\mathsf{E}\to\mathsf{V}} \;=\; [\,id, \lambda u.\, \top\,]$

The inverse operation is also needed. It is defined as:

▶ $lift_{\mathsf{V}\to\mathsf{E}} \;:\; \mathsf{V}(A) \to \mathsf{E}(A)$
$lift_{\mathsf{V}\to\mathsf{E}} \;=\; \lambda v.\, (v = \top) \to unit\ v\,,\ error_{\mathsf{E}}$

## 11.5 Powerdomain monad

The convex powerdomain monad $\mathsf{P}$ has been defined in Section 3.3.6 and many of its properties have already been discussed there. A summary of its definition is repeated here for completeness. Monad $\mathsf{P}$ is used in the rest of the thesis to represent non-deterministic computations. It is defined as:

DEFINITION ■ $p \;:\; \mathsf{P}(D) \;=\; D^{\natural}$

The unit is simply the singleton operation for the convex powerdomain:

▶ $unit_{\mathsf{P}} \;:\; D \to \mathsf{P}(D)$
$unit_{\mathsf{P}} \;=\; \lambda v.\, \{v\}$

and the bind operator is defined using the $ext^{\natural}$ function.

▶ $\cdot *_{\mathsf{P}} \cdot \;:\; \mathsf{P}(A) \times (A \to \mathsf{P}(B)) \to \mathsf{P}(B)$
$p *_{\mathsf{P}} f \;=\; ext^{\natural}\, f\, p$

The resulting monad satisfies the three monad laws and preserves bottom and top elements. Moreover, the powerdomain operator $\sqcup^{\natural}$ can be used on domains of type $\mathsf{P}(D)$:

▶ $\cdot \sqcup^{\natural} \cdot \;:\; \mathsf{P}(D) \times \mathsf{P}(D) \to \mathsf{P}(D)$

This operator implements the combination of multiple values in a single element of $\mathsf{P}(D)$ and is used in expressing the semantics of non-determinism.

## 11.6 Program state

The domain of program states $\mathbf{S}$ is one of the most delicate dynamic domains in the specification of the semantics of C. In this thesis, domain $\mathbf{S}$ is defined only indirectly as an abstract data type, by fear that a complete definition would be overly complicated and would impose unnecessary restrictions on the semantics.

■ $s \;:\; \mathbf{S}$ *(undefined)*

Elements of $\mathbf{S}$ are program states, i.e. abstract representations of the contents of the computer's memory when the C program is executed. Program states should reflect the memory model suggested in the standard, where each memory location contains a single byte and the values of objects are stored in a series of consecutive memory locations. In brief, a program state should know:

- which memory locations are occupied;
- what the contents of the occupied locations are; and
- what side effects have been generated since the last sequence point.

A set of operations allows the manipulation of program states. The requirements from these operations are briefly stated below.

▶ $stateAllocate_m \;:\; [\![m]\!]_{mem} \to \mathbf{S} \to \mathbf{S}$ OBJECT ALLOCATION

This function allocates an object in memory. The address of the object is given in the first parameter, while the initial program state is given in the second. Note that addresses are not absolute and this is the reason why they are passed as parameters; it not the responsibility of function $stateAllocate_m$ to find a free address for a new object. The result is the new program state. An erroneous state $\top_{\mathbf{S}}$ is returned in case of an error, e.g. if the given address has already been allocated.

▶ $stateDestroy_m \;:\; [\![m]\!]_{mem} \to \mathbf{S} \to \mathbf{S}$ OBJECT DEALLOCATION

This function deallocates an object from memory. The address of the object is given in the first parameter, while the initial program state is given in the second. The result is the new program state. An erroneous state $\top_{\mathbf{S}}$ is returned in case of an error, e.g. if the given address has not been allocated.

READ ACCESS ▶ $stateRead_{m \mapsto \tau} \; : \; [\![m]\!]_{mem} \to \mathbf{S} \to [\![\tau]\!]_{dat}$

This function reads the value of an object from memory. The address of the object is given in the first parameter, and the program state in the second. The result is the contents of that address, regarded as a value of data type $\tau$. It is required that $m$ and $\tau$ satisfy the condition $\tau := datify(m)$. It should be noted that an error must occur if type $m$ is not compatible with the type of value stored in the given address, as stated in §6.3 of the standard. An error should also occur if a write side effect is pending for the same memory location. Moreover, if the contents of a volatile address are read, a read side effect should be generated.

WRITE ACCESS ▶ $stateWrite_{\tau \mapsto m} \; : \; [\![m]\!]_{mem} \to [\![\tau]\!]_{dat} \to \mathbf{S} \to \mathbf{S}$

This function writes a value to an object stored in memory. The address of the object is given in the first parameter, the value to be written is given in the second parameter and the program state in the third. The result is the new program state. It is again required that $m$ and $\tau$ satisfy the condition $\tau := datify(m)$. Also, an error must occur if type $m$ is not compatible with the type of value stored in the given address, as stated in §6.3 of the standard, or if a write side effect is pending for the same memory location. In addition, a new write side effect should be generated.

COMMIT CHANGES ▶ $stateCommit \; : \; \mathbf{S} \to \mathbf{S}$

This function implements sequence points. It takes as a parameter the current program states and performs all pending side effects. The result is the new program state.

## 11.7 Continuation monad

CONTINUATIONS Continuations have been suggested a long time ago for specifying the semantics of programming languages with complex control structures. Their presence in the developed semantics is mainly dictated by C's jump statements.

- $\mathbf{C} \;=\; \mathbf{S} \to \mathsf{P}(\mathbf{A})$
- $\mathbf{A}$ *(undefined)*

The elements of domain $\mathbf{C}$ are continuations, i.e. functions that take as parameter the current program state and return the final result of the program's execution. It should be mentioned that the final result is in general non-deterministic[1] and this is the reason why the powerdomain monad is used. The final result of the program's execution is represented by an element of the domain $\mathbf{A}$ of answers. This domain needs not be defined in this thesis. Useful options for its elements are:

- The *int* result of function *main*;
- The final program state; or
- A function from inputs to outputs, assuming that library I/O functions have been modelled.

DEFINITION The continuation monad has been one of the first applications of monads in the semantics of programming languages. Its definition follows:

■ $c \;:\; \mathsf{C}(D) \;=\; (D \to \mathbf{C}) \to \mathbf{C}$

Intuitively, elements of $\mathsf{C}(D)$ represent computations that result in a value of type $D$. The meaning of such a computation is a function which, given the continuation that corresponds to what follows the computation, parameterized by the computation's result, returns a continuation for the whole program including the computation. The unit of monad $\mathsf{C}$ is defined as:

▶ $unit_{\mathsf{C}} \;:\; D \to \mathsf{C}(D)$

$unit_{\mathsf{C}} \;=\; \lambda v.\, \lambda \kappa.\, \kappa\, v$

whereas the bind operator is defined as:

▶ $\cdot *_{\mathsf{C}} \cdot \;:\; \mathsf{C}(A) \times (A \to \mathsf{C}(B)) \to \mathsf{C}(B)$

$c *_{\mathsf{C}} f \;=\; \lambda \kappa.\, c(\lambda a.\, f\, a\, \kappa)$

It is easy to show that the three monad laws are satisfied. However, the continuation monad does not preserve bottom and top elements, i.e. $unit_{\mathsf{C}}\, \bot_D \neq \bot_{\mathsf{C}(D)}$.

Errors in computations are again represented by top elements of domains constructed with $\mathsf{C}$. ERRORS

▶ $error_{\mathsf{C}} \;:\; \mathsf{C}(D)$

$error_{\mathsf{C}} \;=\; \top_{\mathsf{C}(D)}$

It is easy to verify that errors are correctly propagated, that is:

$error_{\mathsf{C}} *_{\mathsf{C}} f = error_{\mathsf{C}}$

$m *_{\mathsf{C}} (\lambda a.\, error_{\mathsf{C}}) = error_{\mathsf{C}}$

The same propagating properties can be verified for bottom elements $\bot_{\mathsf{C}(D)}$, which represent non-termination.

Two polymorphic function are required for the lifting of computations expressed by previously defined monads. The primary lifting function is $lift_{\mathsf{V}\to\mathsf{C}}$ while $lift_{\mathsf{E}\to\mathsf{C}}$ is defined in terms of other already defined lifting functions. LIFTING

▶ $lift_{\mathsf{V}\to\mathsf{C}} \;:\; \mathsf{V}(A) \to \mathsf{C}(A)$

$lift_{\mathsf{V}\to\mathsf{C}} \;=\; unit_{\mathsf{C}}$

▶ $lift_{\mathsf{E}\to\mathsf{C}} \;:\; \mathsf{E}(A) \to \mathsf{C}(A)$

$lift_{\mathsf{E}\to\mathsf{C}} \;=\; lift_{\mathsf{V}\to\mathsf{C}} \circ lift_{\mathsf{E}\to\mathsf{V}}$

Three special functions for the continuation monad are defined next, together with a non-deterministic option operator. They are useful in the developed semantics and will be used in the following sections.

▶ $escape \;:\; \mathbf{C} \to \mathsf{C}(A)$ ESCAPE

$escape \;=\; \lambda c.\, \lambda \kappa.\, c$

This function takes a continuation $c$ as a parameter. It returns a computation which ignores its normal continuation and uses $c$ instead. It is useful in specifying the semantics of jump statements.

[1] According to the author's interpretation of the standard, C allows the development of programs that may produce different answers depending on implementation-defined or unspecified matters, such as evaluation order. An example of such a program is discussed in Section 17.3.

ACCESS THE STATE

▶ $getState_{\mathsf{C}} \;:\; \mathsf{C}(\mathbf{S})$

$getState_{\mathsf{C}} \;=\; \lambda\kappa.\,\lambda s.\,\kappa\, s\, s$

▶ $setState_{\mathsf{C}} \;:\; (\mathbf{S} \to \mathbf{S}) \to \mathsf{C}(\mathbf{S})$

$setState_{\mathsf{C}} \;=\; \lambda f.\,\lambda\kappa.\,\lambda s.\,\kappa\, s\,(f\, s)$

These two functions are used for accessing the program state, which is hidden inside continuations. The result of $getState_{\mathsf{C}}$ is a computation which results in the current program state, leaving it intact. Function $setState_{\mathsf{C}}$ takes as parameter a function $f$ from states to states. It returns a computation in which the current program state is modified by applying $f$ and whose result is the program state prior to the modification.[2] It should be noted that $getState_{\mathsf{C}}$ could have been defined in terms of $setState_{\mathsf{C}}$ as:

$$getState_{\mathsf{C}} = setState_{\mathsf{C}}\; id$$

OPTION OPERATOR

▶ $\cdot \parallel_{\mathsf{C}} \cdot \;:\; \mathsf{C}(A) \times \mathsf{C}(A) \to \mathsf{C}(A)$

$c_1 \parallel_{\mathsf{C}} c_2 \;=\; \lambda\kappa.\,\lambda s.\,(c_1\,\kappa\, s) \uplus^{\natural} (c_2\,\kappa\, s)$

The polymorphic option operator executes one of its operands in a non-deterministic way. The same continuation and program state is used in both cases and the final program answers are combined.

## 11.8 Resumption monad transformer

MOTIVATION In order to express the dynamic semantics of interleaved evaluation of C expressions, a special type of recursive domain is needed. This domain represents the notion of *interleaved computations.* An interleaved computation can be viewed as a sequence of atomic steps. In isolation, these atomic steps are performed one after another, until the expression's evaluation is complete. However, in the presence of other computations, it is allowed that the sequences of atomic steps are *interleaved.* The atomic steps of any given computation must be executed in order, but this process can be interrupted by the execution of atomic steps belonging to different computations.

In this section we attempt to define a domain capable of modelling generic interleaved computations of type $M(D)$. One possible solution is the domain which contains as elements the *resumptions* of computations defined by $M$. This domain is denoted as $\mathsf{R}(M)(D)$ and satisfies the following isomorphism:

■ $\mathsf{R}(M)(D) \;\simeq\; D \oplus M(\mathsf{R}(M)(D))$

In this domain, atomic steps are arbitrary computations defined by $M$. The left part of the coalesced sum represents an already evaluated result, i.e. a computation that consists of zero atomic steps. The right part represents a computation that requires at least one atomic step. The result of this atomic step is a new element of the resumption domain.

Resumptions have been long suggested as a model of interleaved execution in programming languages. For an extensive treatment, the reader is referred to [dBak96], where domains like the one

[2] The result of $setState_{\mathsf{C}}$ is useful in the case of postfix unary assignment operators.

defined above are called *branching domains* and many variations for specific instances of $M$ are explored. In the present thesis, a structured generalization of this technique is attempted. The atomic steps are allowed to represent any type of computation and are defined by an arbitrary monad $M$. In this way, a *monad transformer* is defined, which transforms monad $M$ to a new monad $\mathsf{R}(M)$ of interleaved computations. The definition of this monad, its properties and implementation are given in the next subsections.

### 11.8.1 Definition

GOAL

Consider a locally continuous arbitrary monad $M$. In order to define the monad transformer $\mathsf{R}$ it is necessary to define a monad $\mathsf{R}(M)$. Therefore, it is necessary to define:

- A *domain constructor*, i.e. a domain $\mathsf{R}(M)(D)$ for each domain $D$;
- A *function mapping*, i.e. a continuous function $\mathsf{R}(M)(f) : \mathsf{R}(M)(A) \to \mathsf{R}(M)(B)$ for each continuous function $f : A \to B$; and
- The *unit* and *bind* operators, which must satisfy the three monad laws.

Moreover, the domain constructor should satisfy the aforementioned isomorphism.

PRELIMINARIES

Consider an arbitrary monad $M$ and an arbitrary domain $D$. Then it is possible to define an endofunctor $\mathbf{R}_{M,D} : \mathsf{Dom} \to \mathsf{Dom}$ using the following domain constructor:

$$\mathbf{R}_{M,D}(X) = D \oplus M(X)$$

For an arbitrary continuous function $f : A \to B$, function $\mathbf{R}_{M,D}(f) : \mathbf{R}_{M,D}(A) \to \mathbf{R}_{M,D}(B)$ is defined as follows:

$$\mathbf{R}_{M,D}(f) = [\, inl, inr \circ M(f) \,]$$

**Lemma 11.1**. $\mathbf{R}_{M,D}(f) \circ inr = inr \circ M(f)$

**Proof:** According to the previous definition:

$$\mathbf{R}_{M,D}(f)\ (inr\ m) \ = \ [\, inl, inr \circ M(f) \,]\ (inr\ m) \ = \ inr \circ M(f) \qquad \square$$

Also, let $\breve{\mathbf{R}}_{M,D} : \mathsf{Dom}^{ep} \to \mathsf{Dom}^{ep}$ be the endofunctor induced by $\mathbf{R}_{M,D}$ on the category of ep-pairs:

$$\breve{\mathbf{R}}_{M,D}(X) = \mathbf{R}_{M,D}(X)$$
$$\breve{\mathbf{R}}_{M,D}(\breve{f}) = \langle \mathbf{R}_{M,D}(\breve{f}^e), \mathbf{R}_{M,D}(\breve{f}^p) \rangle$$

According to Theorem 3.18, domain $\mathbf{O}$ is an initial object in category $\mathsf{Dom}^{ep}$ and there is a unique ep-pair $\breve{\iota} : \mathbf{O} \to \mathbf{R}_{M,D}(\mathbf{O})$ with $\breve{\iota} = \langle \bot, \bot \rangle$.

Let us now define a diagram $\Delta : \omega \to \mathsf{Dom}^{ep}$ as:

$$\Delta = \langle \breve{\mathbf{R}}^n_{M,D}(\mathbf{O}), \breve{\mathbf{R}}^n_{M,D}(\breve{\iota}) \rangle_{n \in \omega}$$

Following the process described in [Gunt92, p. 325] let us define a domain $\mathbf{D}_M$ as follows:

$$\mathbf{D}_M = \{\, (x_n)_{n \in \omega} \mid \forall n \in \omega.\ x_n \in \mathbf{R}^n_{M,D}(\mathbf{O}) \ \wedge \ x_n = \mathbf{R}^n_{M,D}(\breve{\iota}^p)(x_{n+1}) \,\}$$

and a point-wise ordering of its elements:

$$(x_n)_{n\in\omega} \sqsubseteq (y_n)_{n\in\omega} \;\Leftrightarrow\; \forall\, n \in \omega.\; x_n \sqsubseteq y_n$$

Let us also define a cone $\breve{\mu} : \Delta \to \mathbf{D}_M$ with ep-pairs $\breve{\mu}_n : \mathbf{R}^n_{M,D}(\mathbf{O}) \to \mathbf{D}_M$ given by:

$$\breve{\mu}^e_n(z) = (x_m)_{m\in\omega} \quad , \quad \text{where} \quad x_m = \begin{cases} (\mathbf{R}^m_{M,D}(\breve{\imath}^p) \circ \ldots \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))(z) \;\; , \; m \le n \\ (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \mathbf{R}^n_{M,D}(\breve{\imath}^e))(z) \;\; , \; m > n \end{cases}$$
$$\breve{\mu}^p_n(x) = x_n$$

In [Gunt92, p. 325] it is proved that the cone $\breve{\mu}$ defined in this way is colimiting. Also, by continuity of $\breve{\mathbf{R}}_{M,D}$ if follows that the cone $\breve{\mathbf{R}}_{M,D}(\breve{\mu}) : \breve{\mathbf{R}}_{M,D}(\Delta) \to \breve{\mathbf{R}}_{M,D}(\mathbf{D}_M)$ is also colimiting and, due to Theorem 3.22, the unique mediating ep-pair $\breve{h}_{M,D} : \mathbf{R}_{M,D}(\mathbf{D}_M) \to \mathbf{D}_M$ between cones $\breve{\mathbf{R}}_{M,D}(\breve{\mu})$ and $\breve{\mu}^-$ is an initial $\breve{\mathbf{R}}_{M,D}$-algebra. The mediating ep-pair $\breve{h}_{M,D}$ satisfies the following properties for all $n \in \omega$:

$$\breve{h}^e_{M,D} \circ \mathbf{R}_{M,D}(\breve{\mu}^e_n) = \breve{\mu}^e_{n+1}$$
$$\mathbf{R}_{M,D}(\breve{\mu}^p_n) \circ \breve{h}^p_{M,D} = \breve{\mu}^p_{n+1}$$

ISOMORPHISM ESTABLISHED

Taking into account that the domain $\mathbf{R}_{M,D}(\mathbf{D}_M)$ has been defined as $D \oplus M(\mathbf{D}_M)$, the above proof that $\breve{h}_{M,D}$ is a unique $\breve{\mathbf{R}}_{M,D}$-algebra establishes an isomorphism between domains $\mathbf{D}_M$ and $D \oplus M(\mathbf{D}_M)$. The components of $\breve{h}_{M,D}$ are continuous functions of the following types:

$$\breve{h}^e_{M,D} : D \oplus M(\mathbf{D}_M) \to \mathbf{D}_M$$
$$\breve{h}^p_{M,D} : \mathbf{D}_M \to D \oplus M(\mathbf{D}_M)$$

Notice however that the existence of the ep-pair $\breve{h}_{M,D}$ has been proved but the ep-pair has not been constructed.

DOMAIN CONSTRUCTOR

With all this in mind, it is a reasonable choice to define the domain constructor of the monad transformer $\mathsf{R}$ as follows:

$$\mathsf{R}(M)(D) = \mathbf{D}_M$$

FUNCTION MAPPING

Consider a continuous function $f : A \to B$. A continuous function $f_M : \mathbf{A}_M \to \mathbf{B}_M$ can be defined as follows:

$$f_M\ (x_n)_{n\in\omega} = (\zeta^{A,B}_n\ f\ x_n)_{n\in\omega}$$

where function $\zeta^{A,B}_n : (A \to B) \to \mathbf{R}^n_{M,A}(\mathbf{O}) \to \mathbf{R}^n_{M,B}(\mathbf{O})$ is defined as:

$$\zeta^{A,B}_0\ f = \bot$$
$$\zeta^{A,B}_{n+1}\ f = [\, inl \circ f, inr \circ M(\zeta^{A,B}_n\ f)\,]$$

In this way, it is reasonable to define the function mapping required for the monad transformer $\mathsf{R}$ as:

$$\mathsf{R}(M)(f) = f_M$$

UNIT AND BIND

The unit function $unit_{\mathsf{R(M)}} : D \to \mathsf{R}(M)(D)$ can be defined as follows:

$$unit_{\mathsf{R(M)}}\ t = (x_n)_{x\in\omega} \quad , \quad \text{where} \quad \begin{array}{l} x_0 = \bot_{\mathbf{o}} \\ x_n = inl\ t \;\; , \; n > 0 \end{array}$$

Also, the bind operator $\cdot\ *_{\mathsf{R(M)}}\ \cdot : \mathsf{R}(M)(A) \times (A \to \mathsf{R}(M)(B)) \to \mathsf{R}(M)(B)$ can be defined as follows:

$$(x_n)_{n\in\omega} \;*_{\mathsf{R(M)}}\; f = (\xi_n^{A,B}\; f\; x_n)_{n\in\omega}$$

where function $\xi_n^{A,B} : (A \to \mathbf{B}_M) \to \mathbf{R}^n_{M,A}(\mathbf{O}) \to \mathbf{R}^n_{M,B}(\mathbf{O})$ is defined as:

$$\xi_0^{A,B}\; f = \bot$$
$$\xi_{n+1}^{A,B}\; f = [\,\lambda t.\,(f\; t)_{n+1}, inr \circ M(\xi_n^{A,B}\; f)\,]$$

**Theorem 11.2** (1ST MONAD LAW). $(unit_{\mathsf{R(M)}}\; t)\; *_{\mathsf{R(M)}}\; f = f\; t$ MONAD LAWS

**Proof:** The left hand side is equal to $(\xi_n^{A,B}\; f\; (unit_{\mathsf{R(M)}}\; t)_n)_{n\in\omega}$
If $n = 0$ then $\xi_0^{A,B}\; f\; (unit_{\mathsf{R(M)}}\; t)_n \;=\; \xi_0^{A,B}\; f\; \bot \;=\; \bot \;=\; (f\; t)_0$
If $n > 0$ then $\xi_0^{A,B}\; f\; (unit_{\mathsf{R(M)}}\; t)_n \;=\; \xi_0^{A,B}\; f\; (inl\; t) \;=\; (f\; t)_n$
Therefore, for all $n$ it is $\xi_n^{A,B}\; f\; (unit_{\mathsf{R(M)}}\; t)_n \;=\; (f\; t)_n$ □

**Lemma 11.3.** $\xi_n^{A,A}\; unit_{\mathsf{R(M)}} = id$

**Proof:** By induction on $n$. If $n = 0$ then $\xi_0^{A,A}\; unit_{\mathsf{R(M)}} \;=\; \bot \;=\; id_{\mathbf{O}\to\mathbf{O}}$
If $n > 0$ let us assume that it holds for $n-1$. Then:

$$\begin{aligned}
& \xi_n^{A,A}\; unit_{\mathsf{R(M)}} \\
= \;& [\,\lambda a.\,(unit_{\mathsf{R(M)}}\; a)_n, inr \circ M(\xi_{n-1}^{A,A}\; unit_{\mathsf{R(M)}})\,] \\
= \;& [\,\lambda a.\, inl\; a, inr \circ M(id)\,] \\
= \;& [\,inl, inr \circ id\,] \\
= \;& [\,inl, inr\,] \\
= \;& id
\end{aligned}$$

□

**Theorem 11.4** (2ND MONAD LAW). $x\; *_{\mathsf{R(M)}}\; unit_{\mathsf{R(M)}} = x$

**Proof:** Let $x = (x_n)_{n\in\omega}$. Starting from the left hand side and using Lemma 11.3 we have:

$$x\; *_{\mathsf{R(M)}}\; unit_{\mathsf{R(M)}} \;=\; (\xi_n^{A,A}\; unit_{\mathsf{R(M)}}\; x_n)_{n\in\omega} \;=\; (id\; x_n)_{n\in\omega} \;=\; (x_n)_{n\in\omega} \;=\; x$$

□

**Lemma 11.5.** $\xi_n^{A,B}\; f \circ inr = inr \circ M(\xi_{n-1}^{A,B}\; f)$

**Proof:** If $n = 0$ both sides are equal to $\bot$. If $n > 0$, from the definition of $\xi_n^{A,B}\; f$ we have:

$$\xi_n^{A,B}\; f\; (inr\; m) \;=\; (inr \circ M(\xi_{n-1}^{A,B}\; f))\; m$$

□

**Lemma 11.6.** $\xi_n^{A,C}\; (\lambda a.\,(\xi_m^{B,C}\; g\; (f\; a)_m)_{m\in\omega}) = (\xi_n^{B,C}\; g) \circ (\xi_n^{A,B}\; f)$

**Proof:** By induction on $n$. If $n = 0$ then both sides are equal to $\top$. If $n > 0$, assume that it holds for $n-1$. Then:

$$
\begin{array}{ll}
 & \xi_N^{A,C}\ (\lambda\, a.\ (\xi_m^{B,C}\ g\ (f\ a)_m)_{m\in\omega}) \\
= & [\,\lambda\, a.\ \xi_n^{B,C}\ g\ (f\ a)_n, inr \circ M(\xi_{n-1}^{A,C}\ (\lambda\, a.\ (\xi_m^{B,C}\ g\ (f\ a)_m)_{m\in\omega}))\,] \\
= & \langle\ \textit{Definition of } \xi_n^{A,B}\ f\ (inl\ a) \textit{ and inductive hypothesis}\ \rangle \\
 & [\,\lambda\, a.\ \xi_n^{B,C}\ g\ (\xi_n^{A,B}\ f\ (inl\ a)), inr \circ M(\xi_{n-1}^{B,C}\ g \circ \xi_{n-1}^{A,B}\ f)\,] \\
= & [\,\xi_n^{B,C}\ g \circ \xi_n^{A,B}\ f \circ inl, inr \circ M(\xi_{n-1}^{B,C}\ g) \circ M(\xi_{n-1}^{A,B}\ f)\,] \\
= & \langle\ \textit{Lemma 11.5 twice}\ \rangle \\
 & [\,\xi_n^{B,C}\ g \circ \xi_n^{A,B}\ f \circ inl, \xi_n^{B,C}\ g \circ inr \circ M(\xi_{n-1}^{A,B}\ f)\,] \\
= & [\,\xi_n^{B,C}\ g \circ \xi_n^{A,B}\ f \circ inl, \xi_n^{B,C}\ g \circ \xi_n^{A,B}\ f \circ inr\,] \\
= & \xi_n^{B,C}\ g \circ \xi_n^{A,B}\ f \circ [\,inl, inr\,] \\
= & \xi_n^{B,C}\ g \circ \xi_n^{A,B}\ f \circ id \\
= & \xi_n^{B,C}\ g \circ \xi_n^{A,B}\ f
\end{array}
$$

□

**Theorem 11.7** (3RD MONAD LAW). $x\ *_{\mathsf{R(M)}}\ (\lambda\, a.\ f\ a\ *_{\mathsf{R(M)}}\ g) = x\ *_{\mathsf{R(M)}}\ f\ *_{\mathsf{R(M)}}\ g$

**Proof:** Let $x = (x_n)_{n\in\omega}$. Starting from the left hand side and using Lemma 11.6 we have:

$$
\begin{array}{ll}
 & x\ *_{\mathsf{R(M)}}\ (\lambda\, a.\ f\ a\ *_{\mathsf{R(M)}}\ g) \\
= & (\xi_n^{A,C}\ (\lambda\, a.\ f\ a\ *_{\mathsf{R(M)}}\ g)\ x_n)_{n\in\omega} \\
= & (\xi_n^{A,C}\ (\lambda\, a.\ (\xi_m^{B,C}\ g\ (f\ a)_m)_{m\in\omega})\ x_n)_{n\in\omega} \\
= & ((\xi_n^{B,C}\ g \circ \xi_n^{A,B}\ f)\ x_n)_{n\in\omega} \\
= & (\xi_n^{B,C}\ g\ (\xi_n^{A,B}\ f\ x_n))_{n\in\omega} \\
= & (\xi_n^{A,B}\ f\ x_n)_{n\in\omega}\ *_{\mathsf{R(M)}}\ g \\
= & (x_n)_{n\in\omega}\ *_{\mathsf{R(M)}}\ f\ *_{\mathsf{R(M)}}\ g \\
= & x\ *_{\mathsf{R(M)}}\ f\ *_{\mathsf{R(M)}}\ g
\end{array}
$$

□

### 11.8.2 Definition of the isomorphism

BACK TO THE ISOMORPHISM

Some operations involving domains of the type $\mathsf{R}(M)(D)$ can be defined in a much more natural way in the isomorphic domain $D \oplus M(\mathsf{R}(M)(D))$. For this reason, it is useful to define the ep-pair $\breve{h}_{M,D}$ which establishes the isomorphism. Then, it is possible to define operations in any of the two domains and induce the corresponding operations on the other domain by applying $\breve{h}_{M,D}$ appropriately.

Consider an arbitrary element $(x_n)_{n\in\omega} \in \mathbf{D}_M$. According to the definition of $\mathbf{D}_M$ we know that $x_n = \mathbf{R}_{M,D}^n(\breve{\imath}^p)(x_{n+1})$. Then, we can distinguish two cases:

- If $x_{n+1} = inl\ t$ for some $t \in D$, then:

$$x_n\ =\ [\,inl, inr \circ M(\mathbf{R}_{M,D}^{n-1}(\breve{\imath}^p))\,]\ (inl\ t)\ =\ inl\ t$$

- If $x_{n+1} = inr\ m_{n+1}$ for some $m_{n+1} \in M(\mathbf{R}_{M,D}^n(\mathbf{O}))$ then:

$$x_n\ =\ [\,inl, inr \circ M(\mathbf{R}_{M,D}^{n-1}(\breve{\imath}^p))\,]\ (inr\ m_{n+1})\ =\ inr\ (M(\mathbf{R}_{M,D}^{n-1}(\breve{\imath}^p))\ m_{n+1})$$

**Theorem 11.8**. If $(x_n)_{n\in\omega} \in \mathbf{D}_M$ then, with the exception of $x_0$, all elements $x_n$ are either left or right summands. That is, exactly one of the following is true:

(a) $x_0 = \bot \ \wedge \ \exists t \in D.\ \forall n > 0.\ x_n = inl\ t$

(b) $\forall n \in \omega.\ \exists m_n \in M(\mathbf{R}^{n-1}_{M,D}(\mathbf{O})).\ x_n = inr\ m_n \ \wedge \ m_0 = \bot$
Moreover, $\forall n \in \omega.\ m_n = M(\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\ m_{n+1}$

**Proof:** Directly, from the previous remark. □

Let us now attempt to define the ep-pair $\breve{h}_{M,D} : \mathbf{R}_{M,D}(\mathbf{D}_M) \to \mathbf{D}_M$. The embedding part is easy. If $z \in D \oplus M(\mathbf{D}_M)$ then we define:

DEFINITION OF $\breve{h}_{M,D}$

$$\breve{h}^e_{M,D}\ z = (x_n)_{n\in\omega} \in \mathbf{D}_M \quad , \quad \text{where} \quad \begin{array}{l} x_0 = \bot_{\mathbf{o}} \\ x_n = [\,inl, inr \circ M(\breve{\mu}^p_{n-1})\,]\ z\ ,\ n > 0 \end{array}$$

The projection part is defined by distinguishing two cases, based on Theorem 11.8. Consider an element $x = (x_n)_{n\in\omega} \in \mathbf{D}_M$.

- If $\forall n > 0.\ x_n = inl\ t$, for some $t \in D$, then we define:

$$\breve{h}^p_{M,D}\ x \ = \ inl\ t$$

- If $\forall n \in \omega.\ x_n = inr\ m_n$, with $m_n \in M(\mathbf{R}^{n-1}_{M,D}(\mathbf{O}))$, then we define:

$$\breve{h}^p_{M,D}\ x \ = \ inr\ \left( \bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1})\ m_n \right)$$

First, notice that for all $n \in \omega$ we have $M(\breve{\mu}^e_{n-1}) : M(\mathbf{R}^{n-1}_{M,D}(\mathbf{O})) \to M(\mathbf{D}_M)$ and therefore $M(\breve{\mu}^e_{n-1})\ m_n \in M(\mathbf{D}_M)$. It must now be proved that the least upper bound exists. It suffices to show that elements $M(\breve{\mu}^e_{n-1})\ m_n$ form an $\omega$-chain, which is stated by the following theorem.

**Theorem 11.9**. $\forall n \in \omega.\ M(\breve{\mu}^e_{n-1})\ m_n \sqsubseteq M(\breve{\mu}^e_n)\ m_{n+1}$

**Proof:** We have:

$$M(\breve{\mu}^e_{n-1})\ m_n \ = \ M(\breve{\mu}^e_{n-1})\ (M(\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\ m_{n+1}) \ = \ M(\breve{\mu}^e_{n-1} \circ \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\ m_{n+1}$$

Since $M$ is locally-monotone, it suffices to show that $\breve{\mu}^e_{n-1} \circ \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p) \sqsubseteq \breve{\mu}^e_n$. Let $z \in \mathbf{R}^n_{M,D}(\mathbf{O})$ and $x = (x_m)_{m\in\omega} \in \mathbf{D}_M$ with:

$$x \ = \ \breve{\mu}^e_{n-1}\ (\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p)\ z)$$

Then, by definition of $\breve{\mu}^e_{n-1}$:

$$\begin{array}{ll} & x_m \\ = & \begin{cases} (\mathbf{R}^m_{M,D}(\breve{\imath}^p) \circ \ldots \mathbf{R}^{n-2}_{M,D}(\breve{\imath}^p))\ (\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p)\ z) & ,\ m \le n-1 \\ (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^e))\ (\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p)\ z) & ,\ m > n-1 \end{cases} \\ = & \begin{cases} (\mathbf{R}^m_{M,D}(\breve{\imath}^p) \circ \ldots \mathbf{R}^{n-2}_{M,D}(\breve{\imath}^p) \circ \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\ z & ,\ m \le n-1 \\ (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^e) \circ \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\ z & ,\ m > n-1 \end{cases} \end{array}$$

Also, let $y = (y_m)_{m\in\omega} \in \mathbf{D}_M$ with:

$$y \ = \ \breve{\mu}^e_n\ z$$

Then again:

$$y_m \;=\; \begin{cases} (\mathbf{R}^m_{M,D}(\breve{\imath}^p) \circ \ldots \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\; z & ,\; m \leq n-1 \\ (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \mathbf{R}^{n}_{M,D}(\breve{\imath}^e))\; z & ,\; m > n-1 \end{cases}$$

For $m \leq n-1$ it is obvious that $x_m = y_m$. For $m > n-1$ we have:

$$\begin{array}{ll} & x_m \\ = & (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \mathbf{R}^{n}_{M,D}(\breve{\imath}^e) \circ \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^e) \circ \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\; z \\ = & (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \mathbf{R}^{n}_{M,D}(\breve{\imath}^e) \circ \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^e \circ \breve{\imath}^p))\; z \\ \sqsubseteq & (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \mathbf{R}^{n}_{M,D}(\breve{\imath}^e) \circ \mathbf{R}^{n-1}_{M,D}(id))\; z \\ = & (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \mathbf{R}^{n}_{M,D}(\breve{\imath}^e))\; z \\ = & y_m \end{array}$$

□

$\breve{h}_{M,D}$ IS AN EP-PAIR

The next thing to do is to prove that $\breve{h}_{M,D}$ is an ep-pair, that is, prove that $\breve{h}^p_{M,D} \circ \breve{h}^e_{M,D} = id$ and $\breve{h}^e_{M,D} \circ \breve{h}^p_{M,D} \sqsubseteq id$. In fact, a stronger result is proved by the following two theorems.

**Theorem 11.10**. $\breve{h}^p_{M,D} \circ \breve{h}^e_{M,D} = id$

**Proof:** Both sides of the equation are functions of type $D \oplus M(\mathbf{R}_{M,D}) \to \oplus M(\mathbf{R}_{M,D})$. Proceed by case analysis on the argument of this function and using the definitions of $\breve{h}^e_{M,D}$ and $\breve{h}^p_{M,D}$. First case:

$$(\breve{h}^p_{M,D} \circ \breve{h}^e_{M,D})\;(inl\; t) \;=\; \breve{h}^p_{M,D}\;(\breve{h}^e_{M,D}\;(inl\; t)) \;=\; \breve{h}^p_{M,D}\;(x_n)_{n\in\omega}$$

where $x_0 = \bot$ and $x_n = inl\; t$ for $n > 0$. Then:

$$\breve{h}^p_{M,D}\;(x_n)_{n\in\omega} \;=\; inl\; t$$

Second case:

$$(\breve{h}^p_{M,D} \circ \breve{h}^e_{M,D})\;(inr\; m) \;=\; \breve{h}^p_{M,D}\;(\breve{h}^e_{M,D}\;(inr\; m)) \;=\; \breve{h}^p_{M,D}\;(x_n)_{n\in\omega}$$

where $x_n = inr\;(M(\breve{\mu}^p_{n-1})\; m)$ for $n \in \omega$. Then:

$$\begin{array}{ll} & \breve{h}^p_{M,D}\;(x_n)_{n\in\omega} \\ = & inr\;\left(\bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1})\;(M(\breve{\mu}^p_{n-1})\; m)\right) \\ = & inr\;\left(\bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1} \circ \breve{\mu}^p_{n-1})\; m\right) \\ = & inr\;\left(\left(\bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1} \circ \breve{\mu}^p_{n-1})\right)\; m\right) \\ = & \langle\; M \textit{ is locally continuous} \;\rangle \\ & inr\;\left(M\left(\bigsqcup_{n\in\omega} \breve{\mu}^e_{n-1} \circ \breve{\mu}^p_{n-1}\right)\; m\right) \\ = & \langle\; \breve{\mu} \textit{ is colimiting} \;\rangle \\ & inr\;(M(id)\; m) \\ = & inr\;(id\; m) \\ = & inr\; m \end{array}$$

□

**Theorem 11.11**. $\breve{h}^e_{M,D} \circ \breve{h}^p_{M,D} = id$

**Proof:** Both sides of the equation are functions of type $\mathbf{R}_{M,D} \to \mathbf{R}_{M,D}$. Consider an element $x = (x_n)_{n\in\omega} \in \mathbf{R}_{M,D}$. Proceed by case analysis based on Theorem 11.8 and using the definitions of $\breve{h}^e_{M,D}$ and $\breve{h}^p_{M,D}$. First case, if $x_0 = \bot$ and $\forall\, n > 0.\ \ x_n = inl\ t$, for some $t \in D$, then:

$$(\breve{h}^e_{M,D} \circ \breve{h}^p_{M,D})\ x \ = \ \breve{h}^e_{M,D}\ (\breve{h}^p_{M,D}\ x) \ = \ \breve{h}^e_{M,D}\ (inl\ t) \ = \ (y_n)_{n\in\omega}$$

where $y_0 = \bot$ and for all $n > 0$:

$$y_n \ = \ [\, inl, inr \circ M(\breve{\mu}^p_{n-1})\,]\ (inl\ t) \ = \ inl\ t$$

It is trivial to verify that $y = x$. Second case, if $\forall\, n \in \omega.\ \ x_n = inr\ m_n$, with $m_n \in M(\mathbf{R}^{n-1}_{M,D}(\mathbf{O}))$, then:

$$(\breve{h}^e_{M,D} \circ \breve{h}^p_{M,D})\ x \ = \ \breve{h}^e_{M,D}\ (\breve{h}^p_{M,D}\ x) \ = \ \breve{h}^e_{M,D}\ \left(inr\ \left(\bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1})\ m_n\right)\right) \ = \ (y_n)_{n\in\omega}$$

where $y_0 = \bot$ and for all $n > 0$:

$$\begin{aligned}
& y_n \\
= \ & [\, inl, inr \circ M(\breve{\mu}^p_{n-1})\,]\ \left(inr\ \left(\bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1})\ m_n\right)\right) \\
= \ & inr\ \left(M(\breve{\mu}^p_{n-1})\ \left(\bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1})\ m_n\right)\right) \\
= \ & \langle\ \textit{Continuity of } M(\breve{\mu}^p_{n-1})\ \rangle \\
& inr\ \left(\bigsqcup_{n'\in\omega} (M(\breve{\mu}^p_{n-1}) \circ M(\breve{\mu}^e_{n'-1}))\ m_{n'}\right) \\
= \ & inr\ \left(\bigsqcup_{n'\in\omega} M(\breve{\mu}^p_{n-1} \circ \breve{\mu}^e_{n'-1})\ m_{n'}\right) \\
= \ & \langle\ \textit{By the following Lemma 11.12 and because } M(\breve{\mu}^p_{n-1} \circ \breve{\mu}^e_{n'-1})\ m_{n'} \textit{ are an } \omega\textit{-chain}\ \rangle \\
& inr\ \left(\bigsqcup_{n'>n} m_n\right) \\
= \ & inr\ m_n
\end{aligned}$$

Again it is trivial to verify that $y = x$. □

**Lemma 11.12**. If $m_n = M(\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\ m_{n+1}$ for all $n \in \omega$ and $n' > n$, then:
$M(\breve{\mu}^p_{n-1} \circ \breve{\mu}^e_{n'-1})\ m_{n'} = m_n$

**Proof:** Using the definitions of $\breve{\mu}^p_{n-1}$ and $\breve{\mu}^e_{n'-1}$:

$$\begin{aligned}
& M(\breve{\mu}^p_{n-1} \circ \breve{\mu}^e_{n'-1})\ m_{n'} \\
= \ & M(\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p) \circ \ldots \circ \mathbf{R}^{n'-2}_{M,D}(\breve{\imath}^p))\ m_{n'} \\
= \ & (M(\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p)) \circ \ldots \circ M(\mathbf{R}^{n'-2}_{M,D}(\breve{\imath}^p)))\ m_{n'} \\
= \ & M(\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\ (\ldots\ M(\mathbf{R}^{n'-2}_{M,D}(\breve{\imath}^p))\ m_{n'}\ \ldots) \\
= \ & M(\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\ (\ldots\ m_{n'-1}\ \ldots) \\
= \ & \langle\ \textit{Repeating } n' - n \textit{ times in total}\ \rangle \\
& m_n
\end{aligned}$$

□

$\breve{h}_{M,D}$ IS MEDIATING EP-PAIR

What remains to be done is to prove that $\breve{h}_{M,D}$ is indeed a mediating ep-pair between cones $\breve{\mathbf{R}}_{M,D}(\breve{\mu})$ and $\breve{\mu}^-$. This is proved in the following two theorems.

**Theorem 11.13**. $\breve{h}^e_{M,D} \circ \mathbf{R}_{M,D}(\breve{\mu}^e_n) = \breve{\mu}^e_{n+1}$

**Proof:** Both sides of the equation are functions of type $\mathbf{R}^{n+1}_{M,D}(\mathbf{O}) \to \mathbf{D}_M$. Proceed by case analysis on the argument of this function. First case:

$$
\begin{array}{ll}
 & (\breve{h}^e_{M,D} \circ \mathbf{R}_{M,D}(\breve{\mu}^e_n))\ (inl\ t) \\
= & \breve{h}^e_{M,D}\ (\mathbf{R}_{M,D}(\breve{\mu}^e_n)\ (inl\ t)) \\
= & \breve{h}^e_{M,D}\ ([\,inl, inr \circ M(\breve{\mu}^e_n)\,]\ (inl\ t)) \\
= & \breve{h}^e_{M,D}\ (inl\ t) \\
= & (x_m)_{m\in\omega}
\end{array}
$$

where $x_0 = \bot$ and $x_m = inl\ t$ for $m > 0$. It can easily be verified that also $\breve{\mu}^e_{n+1}\ (inl\ t) = x$. Second case, starting from the left hand side:

$$
\begin{array}{ll}
 & (\breve{h}^e_{M,D} \circ \mathbf{R}_{M,D}(\breve{\mu}^e_n))\ (inr\ m) \\
= & \breve{h}^e_{M,D}\ (\mathbf{R}_{M,D}(\breve{\mu}^e_n)\ (inr\ m)) \\
= & \breve{h}^e_{M,D}\ ([\,inl, inr \circ M(\breve{\mu}^e_n)\,]\ (inr\ m)) \\
= & \breve{h}^e_{M,D}\ (inr\ (M(\breve{\mu}^e_n)\ m)) \\
= & (x_m)_{m\in\omega}
\end{array}
$$

where $x_0 = \bot$ and

$$x_m\ =\ inr\ (M(\breve{\mu}^p_{m-1})\ (M(\breve{\mu}^e_n)\ m))\ =\ inr\ (M(\breve{\mu}^p_{m-1} \circ \breve{\mu}^e_n)\ m)$$

Consider also the right hand side:

$$\breve{\mu}^e_{n+1}\ (inr\ m)\ =\ (y_m)_{m\in\omega}$$

Two cases are distinguished again. If $m - 1 \leq n$ then $\breve{\mu}^p_{m-1} \circ \breve{\mu}^e_n = \mathbf{R}^{m-1}_{M,D}(\breve{\imath}^p) \circ \ldots \circ \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p)$ and therefore:

$$
\begin{array}{ll}
 & x_m \\
= & inr\ (M(\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^p) \circ \ldots \circ \mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p))\ m) \\
= & (inr \circ M(\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^p)) \circ \ldots \circ M(\mathbf{R}^{n-1}_{M,D}(\breve{\imath}^p)))\ m \\
= & \langle\ \textit{Lemma 11.1 applied repeatedly}\ \rangle \\
 & (\mathbf{R}^m_{M,D}(\breve{\imath}^p) \circ \ldots \circ \mathbf{R}^n_{M,D}(\breve{\imath}^p) \circ inr)\ m \\
= & (\mathbf{R}^m_{M,D}(\breve{\imath}^p) \circ \ldots \circ \mathbf{R}^n_{M,D}(\breve{\imath}^p))\ (inr\ m) \\
= & \langle\ \textit{Definition of } \breve{\mu}^e_{n+1}\ \rangle \\
 & y_m
\end{array}
$$

On the other hand, if $m - 1 > n$ then $\breve{\mu}^p_{m-1} \circ \breve{\mu}^e_n = \mathbf{R}^{m-2}_{M,D}(\breve{\imath}^e) \circ \ldots \circ \mathbf{R}^n_{M,D}(\breve{\imath}^e)$ and therefore:

$$
\begin{array}{ll}
 & x_m \\
= & inr\ (M(\mathbf{R}^{m-2}_{M,D}(\breve{\imath}^e) \circ \ldots \circ \mathbf{R}^{n}_{M,D}(\breve{\imath}^e))\ m) \\
= & (inr \circ M(\mathbf{R}^{m-2}_{M,D}(\breve{\imath}^e)) \circ \ldots \circ M(\mathbf{R}^{n}_{M,D}(\breve{\imath}^e)))\ m \\
= & \langle\ \textit{Lemma 11.1 applied repeatedly}\ \rangle \\
 & (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \circ \mathbf{R}^{n+1}_{M,D}(\breve{\imath}^e) \circ inr)\ m \\
= & (\mathbf{R}^{m-1}_{M,D}(\breve{\imath}^e) \circ \ldots \circ \mathbf{R}^{n+1}_{M,D}(\breve{\imath}^e))\ (inr\ m) \\
= & \langle\ \textit{Definition of}\ \breve{\mu}^e_{n+1}\ \rangle \\
 & y_m
\end{array}
$$

Therefore, in general $x = y$. □

**Theorem 11.14**. $\mathbf{R}_{M,D}(\breve{\mu}^p_n) \circ \breve{h}^p_{M,D} = \breve{\mu}^p_{n+1}$

**Proof:** Both sides of the equation are functions of type $\mathbf{D}_M \to \mathbf{R}^{n+1}_{M,D}(\mathbf{O})$. Proceed by case analysis on the argument of this function, according to Theorem 11.8. Consider an arbitrary element $x = (x_n)_{n\in\omega} \in \mathbf{D}_M$. First case, if $x_0 = \bot$ and $\forall\, n > 0.\ \ x_n = inl\ t$, for some $t \in D$, then:

$$
\begin{array}{ll}
 & (\mathbf{R}_{M,D}(\breve{\mu}^p_n) \circ \breve{h}^p_{M,D})\ x \\
= & \mathbf{R}_{M,D}(\breve{\mu}^p_n)\ (\breve{h}^p_{M,D}\ x) \\
= & \mathbf{R}_{M,D}(\breve{\mu}^p_n)\ (inl\ t) \\
= & [\,inl, inr \circ M(\breve{\mu}^p_n)\,]\ (inl\ t) \\
= & inl\ t \\
= & x_{n+1} \\
= & \breve{\mu}^p_{n+1}\ x
\end{array}
$$

Second case, if $\forall\, n \in \omega.\ \ x_n = inr\ m_n$, with $m_n \in M(\mathbf{R}^{n-1}_{M,D}(\mathbf{O}))$, then:

$$
\begin{array}{ll}
 & (\mathbf{R}_{M,D}(\breve{\mu}^p_n) \circ \breve{h}^p_{M,D})\ x \\
= & \mathbf{R}_{M,D}(\breve{\mu}^p_n)\ (\breve{h}^p_{M,D}\ x) \\
= & \mathbf{R}_{M,D}(\breve{\mu}^p_n)\ \left(inr\left(\bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1})\ m_n\right)\right) \\
= & [\,inl, inr \circ M(\breve{\mu}^p_n)\,]\ \left(inr\left(\bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1})\ m_n\right)\right) \\
= & inr\left(M(\breve{\mu}^p_n)\ \left(\bigsqcup_{n\in\omega} M(\breve{\mu}^e_{n-1})\ m_n\right)\right) \\
= & \langle\ \textit{Similarly to the second case in the proof of Theorem 11.11}\ \rangle \\
 & inr\ m_{n+1} \\
= & x_{n+1} \\
= & \breve{\mu}^p_{n+1}\ x
\end{array}
$$

□

### 11.8.3 Special operations

BRIDGE TO INTERLEAVED COMPUTATIONS

The two functions $run$ and $step$ convert an interleaved computation of type $\mathsf{R}(M)(A)$ to a non interleaved computation of type $M(A)$ and vice-versa. The names of these functions indicate their behaviour. If an interleaved computation is viewed as a sequence of atomic steps, the first function *runs* this sequence without allowing other computation to intervene. The second function converts a whole computation to a single atomic *step* in an interleaved computation.

▶ $run \;:\; \mathsf{R}(M)(D) \to M(D)$

$run \;=\; fix\,(\lambda g.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; g\,] \circ \breve{h}^{p}_{M,D})$

▶ $step \;:\; M(D) \to \mathsf{R}(M)(D)$

$step \;=\; \lambda m.\, \breve{h}^{e}_{M,D}\,(inr\,(m \;*_{\mathsf{M}}\; (unit_{\mathsf{M}} \circ \breve{h}^{e}_{M,D} \circ inl)))$

The following theorem connects the behaviour of the two functions.

**Theorem 11.15**. $run \circ step = id$

**Proof:** According to the definitions of $step$ and $run$ and using the equational property of the least fixed point operator, we have:

$$
\begin{aligned}
& run\;(step\;m) \\
= \;& run\;(\breve{h}^{e}_{M,D}\,(inr\,(m \;*_{\mathsf{M}}\; (unit_{\mathsf{M}} \circ \breve{h}^{e}_{M,D} \circ inl)))) \\
= \;& fix\,(\lambda g.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; g\,] \circ \breve{h}^{p}_{M,D})\;(\breve{h}^{e}_{M,D}\,(inr\,(m \;*_{\mathsf{M}}\; (unit_{\mathsf{M}} \circ \breve{h}^{e}_{M,D} \circ inl)))) \\
= \;& [\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; fix\,(\lambda g.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; g\,] \circ \breve{h}^{p}_{M,D})\,] \\
& \quad (\breve{h}^{p}_{M,D}\,(\breve{h}^{e}_{M,D}\,(inr\,(m \;*_{\mathsf{M}}\; (unit_{\mathsf{M}} \circ \breve{h}^{e}_{M,D} \circ inl))))) \\
= \;& [\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; fix\,(\lambda g.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; g\,] \circ \breve{h}^{p}_{M,D})\,] \\
& \quad (inr\,(m \;*_{\mathsf{M}}\; (unit_{\mathsf{M}} \circ \breve{h}^{e}_{M,D} \circ inl))) \\
= \;& m \;*_{\mathsf{M}}\; (unit_{\mathsf{M}} \circ \breve{h}^{e}_{M,D} \circ inl) \;*_{\mathsf{M}}\; fix\,(\lambda g.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; g\,] \circ \breve{h}^{p}_{M,D}) \\
= \;& m \;*_{\mathsf{M}}\; (\lambda a.\, unit_{\mathsf{M}}\,(\breve{h}^{e}_{M,D}\,(inl\;a)) \;*_{\mathsf{M}}\; fix\,(\lambda g.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; g\,] \circ \breve{h}^{p}_{M,D})) \\
= \;& m \;*_{\mathsf{M}}\; (\lambda a.\, fix\,(\lambda g.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; g\,] \circ \breve{h}^{p}_{M,D})\;(\breve{h}^{e}_{M,D}\,(inl\;a))) \\
= \;& m \;*_{\mathsf{M}}\; (\lambda a.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; fix\,(\lambda g.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; g\,] \circ \breve{h}^{p}_{M,D})\,] \\
& \quad (\breve{h}^{p}_{M,D}\,(\breve{h}^{e}_{M,D}\,(inl\;a)))) \\
= \;& m \;*_{\mathsf{M}}\; (\lambda a.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; fix\,(\lambda g.\,[\,unit_{\mathsf{M}}, \lambda m.\, m \;*_{\mathsf{M}}\; g\,] \circ \breve{h}^{p}_{M,D})\,]\;(inl\;a)) \\
= \;& m \;*_{\mathsf{M}}\; (\lambda a.\, unit_{\mathsf{M}}\;a) \\
= \;& m \;*_{\mathsf{M}}\; unit_{\mathsf{M}} \\
= \;& m
\end{aligned}
$$

□

INTERLEAVING BEHAVIOUR

In order to define an interleaved computation of type $\mathsf{R}(M)(D)$, it is assumed that an *non-deterministic option* operator is defined for computations represented by monad $M$. This option operator is denoted as $\|_{\mathsf{M}}$ and has the type:

▶ $\cdot \,\|_{\mathsf{M}}\, \cdot \;:\; M(D) \times M(D) \to M(D)$

The subscript may be omitted if it can be deduced from the context.

Assuming the existance of operator $\|_{\mathsf{M}}$, it is possible to define an operator of the resumption monad transformer for performing the interleaving of computations. It is defined as follows:

▶ $\cdot \bowtie_{\mathsf{R}(M)} \cdot \;:\; \mathsf{R}(M)(A) \times \mathsf{R}(M)(B) \to \mathsf{R}(M)(A \times B)$

$$\begin{array}{l}\bowtie_{\mathsf{R}(M)} \;=\; \mathit{fix}\ (\lambda\, g.\ \lambda\, \langle x, y\rangle.\\ \quad [\lambda\, v_x.\ y\ *_{\mathsf{R}(M)}\ (\lambda\, v_y.\ \mathit{unit}_{\mathsf{R}(M)}\ \langle v_x, v_y\rangle), \lambda\, m_x.\\ \quad [\lambda\, v_y.\ x\ *_{\mathsf{R}(M)}\ (\lambda\, v_x.\ \mathit{unit}_{\mathsf{R}(M)}\ \langle v_x, v_y\rangle), \lambda\, m_y.\\ \quad \mathit{inr}\ (m_x\ *_{\mathsf{M}}\ (\lambda\, x'.\ \mathit{unit}_{\mathsf{M}}\ (g\ \langle x', y\rangle))\ \|\ m_y\ *_{\mathsf{M}}\ (\lambda\, y'.\ \mathit{unit}_{\mathsf{M}}\ (g\ \langle x, y'\rangle)))]])\end{array}$$

The subscript may be omitted if it can be deduced from the context. If one of the two computations does not require the execution of any atomic step, i.e. if one of the two computations has already been executed, then the other computation is executed and the two results are combined. Otherwise, if both computations require at least one atomic step, then there is a non-deterministic option of which computation will start executing.

## 11.9 Monad for expression semantics

DEFINITION

Monad $\mathsf{G}$ represents the computation of C expressions. Such computations can affect the program state and can be interleaved. For the definition of $\mathsf{G}$, the resumption monad transformer is applied on the continuation monad $\mathsf{C}$.

■ $g \;:\; \mathsf{G}(D) \;=\; \mathsf{R}(\mathsf{C})(D)$

The unit, bind operation and other properties of monad $\mathsf{G}$ are specified in terms of the corresponding operations obtained from the resumption monad transformer.

ERRORS

Again the top element of domain $\mathsf{G}(D)$ represents errors in computations. Non-termination and errors are again correctly propagated.

▶ $\mathit{error}_{\mathsf{G}} \;:\; \mathsf{G}(D)$

$\mathit{error}_{\mathsf{G}} \;=\; \top$

LIFTING

A number of polymorphic functions are needed for converting between values from various monad domains. Functions $\mathit{lift}_{\mathsf{C}\to\mathsf{G}}$ and $\mathit{lift}_{\mathsf{G}\to\mathsf{C}}$ are primitive, whereas all others are defined in terms of other lifting functions.

▶ $\mathit{lift}_{\mathsf{C}\to\mathsf{G}} \;:\; \mathsf{C}(A) \to \mathsf{G}(A)$

$\mathit{lift}_{\mathsf{C}\to\mathsf{G}} \;=\; \mathit{step}$

▶ $\mathit{lift}_{\mathsf{G}\to\mathsf{C}} \;:\; \mathsf{G}(A) \to \mathsf{C}(A)$

$\mathit{lift}_{\mathsf{G}\to\mathsf{C}} \;=\; \mathit{run}$

These two functions convert between interleaved and non-interleaved computations.

▶ $\mathit{lift}_{\mathsf{E}\to\mathsf{G}} \;:\; \mathsf{E}(A) \to \mathsf{G}(A)$

$\mathit{lift}_{\mathsf{E}\to\mathsf{G}} \;=\; \mathit{lift}_{\mathsf{C}\to\mathsf{G}} \circ \mathit{lift}_{\mathsf{V}\to\mathsf{C}} \circ \mathit{lift}_{\mathsf{E}\to\mathsf{V}}$

▶ $\mathit{lift}_{\mathsf{V}\to\mathsf{G}} \;:\; \mathsf{V}(A) \to \mathsf{G}(A)$

$\mathit{lift}_{\mathsf{V}\to\mathsf{G}} \;=\; \mathit{lift}_{\mathsf{C}\to\mathsf{G}} \circ \mathit{lift}_{\mathsf{V}\to\mathsf{C}}$

The following functions implement special operations for monad $\mathsf{G}$.

ACCESS THE STATE

▶ $getState_{\mathsf{G}} \;:\; \mathsf{G}(\mathbf{S})$

$getState_{\mathsf{G}} \;=\; lift_{\mathsf{C}\to\mathsf{G}}\; getState_{\mathsf{C}}$

▶ $setState_{\mathsf{G}} \;:\; (\mathbf{S}\to\mathbf{S})\to\mathsf{G}(\mathbf{S})$

$setState_{\mathsf{G}} \;=\; lift_{\mathsf{C}\to\mathsf{G}}\circ setState_{\mathsf{C}}$

These two functions are lifted versions of the corresponding functions for the continuation monad. They are used in the definition of the following functions.

READING VALUES

▶ $getValue_{m\mapsto\tau} \;:\; [\![m]\!]_{mem}\to\mathsf{G}([\![\tau]\!]_{dat})$

$getValue_{m\mapsto\tau} \;=\; \lambda\, d_m.\; getState_{\mathsf{G}} \;*\; (unit\circ(stateRead_{m\mapsto\tau}\; d_m))$

This function reads the value of an object that is stored in memory. It takes as parameter the address of the object and returns a computation resulting in the stored value. Parameters $m$ and $\tau$ must satisfy $\tau := datify\ m$. The resulting interleaved computation requires one step for accessing the program state.

WRITING VALUES

▶ $putValue_{\tau\mapsto m} \;:\; [\![m]\!]_{mem}\to[\![\tau]\!]_{dat}\to\mathsf{G}([\![\tau]\!]_{dat})$

$putValue_{\tau\mapsto m} \;=\; \lambda\, d_m.\;\lambda\, d.\; setState_{\mathsf{G}}\;(stateWrite_{\tau\mapsto m}\; d_m\; d)\;*\;(unit\circ(stateRead_{m\mapsto\tau}\; d_m))$

This function writes a value to an object stored in memory. It takes as parameters the address of the object and the value to be stored. Again, it must be $\tau := datify\ m$. The result is a computation which stores the value in the current program state and returns the value that was previously stored in the object. It requires one step for accessing the program state.

SEQUENCE POINTS

▶ $seqpt \;:\; \mathsf{G}(\mathbf{U})$

$seqpt \;=\; lift_{\mathsf{C}\to\mathsf{G}}\;(setState\; stateCommit\;*\;(\lambda\, s.\; unit\; u))$

This function returns a computation which generates a sequence point. The program state is changed by applying the function $stateCommit$. One step is required.

FULL EXPRESSIONS

▶ $fullExpression \;:\; \mathsf{G}(A)\to\mathsf{C}(A)$

$fullExpression \;=\; \lambda\, g.\; lift_{\mathsf{G}\to\mathsf{C}}\;(g\;*\;(\lambda\, d.\; seqpt\;*\;(\lambda\, u.\; unit\; d)))$

This function converts the interleaved computation, which represents a full expression, to a non-interleaved computation. A sequence point is generated after the full expression has been evaluated.

## 11.10 Environments

The domains defined in this section represent various kinds of dynamic environments. Such environments are either the dynamic equivalent of static environments, or provide information about aspects of the program that is not included in the program state, such as the meanings of defined functions.

### 11.10.1 Type environments

Dynamic type environments associate identifiers with their dynamic meanings. Since a static meaning of an identifier is required to be known before a dynamic meaning can be given, dynamic type environments are closely related to static type environment. Given a static type environment $e$, $[\![e]\!]_{Ent}$ is the domain of dynamic type environments that are related to $e$.

■ $\rho \;:\; [\![e]\!]_{Ent} \;=\; (I : \mathbf{Ide} \twoheadrightarrow [\![e \Downarrow I]\!]) \times [\![ \downarrow e]\!]_{Ent}$ DEFINITION

Just as static type environments, dynamic type environments are organized in a tree-like structure of nested scopes. In the first part of the product, each identifier is associated with its dynamic meaning. Notice the use of the dependent function here. The second part of the product represents the dynamic environment of the enclosing scope. It is equal to $\top$ in the case of the outermost scope.

Given a static type environment $e$, the dynamic meaning of identifier $I$ is an element of the domain $[\![ e \Downarrow I ]\!]$. This domain is defined as follows:

$$
\begin{array}{l}
[\![e \Downarrow I]\!] \;=\; \neg \mathit{isLocal}(e, I\ \mathit{ide}) \rightarrow \mathbf{U}\,, \\
\quad \mathbf{case}\ e[I\ \mathit{raw}\ \mathit{ide}]\ \mathbf{of} \\
\qquad \mathit{normal}\,[\phi] \;\Rightarrow\; [\![\phi]\!]_{den} \\
\qquad \mathbf{otherwise} \;\Rightarrow\; \mathbf{U}
\end{array}
$$

If $I$ is an identifier declared locally in $e$ and does not correspond to a *typedef*, then its dynamic meaning is an element of the dynamic domain that corresponds to its static denotable type. Otherwise, it is an element of the trivial domain $\mathbf{U}$.

The following operations are defined for dynamic type environments.

▶ $\cdot \uparrow \cdot \;:\; e : \mathbf{Ent} \ltimes [\![ \downarrow e]\!]_{Ent} \rightarrow [\![e]\!]_{Ent}$ OPEN SCOPE

$$
\begin{array}{l}
e \uparrow \rho \;=\; \mathbf{let}\ m_i = \lambda I.\ \neg \mathit{isLocal}(e, I\ \mathit{ide}) \rightarrow u\,, \\
\qquad\qquad\qquad\qquad \mathbf{case}\ e[I\ \mathit{raw}\ \mathit{ide}]\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\quad \mathit{normal}\,[\phi] \;\Rightarrow\; \top \\
\qquad\qquad\qquad\qquad\quad \mathbf{otherwise} \;\Rightarrow\; u \\
\qquad\quad \mathbf{in}\ \langle m_i, \rho \rangle
\end{array}
$$

This operation creates a dynamic type environment, taking as parameters the related static environment and the dynamic environment that corresponds to the enclosing scope. Local identifiers that do not correspond to a *typedef* are associated with the error value $\top$, reflecting the fact that they represent objects which have not been given an address yet. This is fixed by the dynamic semantics of declarations.

▶ $\cdot \downarrow \cdot \;:\; e : \mathbf{Ent} \ltimes [\![e]\!]_{Ent} \rightarrow [\![ \downarrow e]\!]_{Ent}$ CLOSE SCOPE

$$
e \downarrow \rho \;=\; \mathbf{let}\ \langle m, \rho_p \rangle = \rho\ \mathbf{in}\ \rho_p
$$

This operation returns the dynamic environment corresponding to the enclosing scope of $\rho$.

▶ $\mathit{create} \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \rightarrow \mathbf{Ide} \rightarrow \mathsf{V}([\![e]\!]_{Ent})$ CREATE OBJECT

$$
\begin{array}{l}
\mathit{create} \;=\; \lambda e.\ \lambda \rho.\ \lambda I. \\
\quad \mathbf{let}\ \mathit{create}_{den} \;:\; \phi : \mathbf{Type}_{den} \twoheadrightarrow [\![\phi]\!]_{den} \\
\qquad \mathit{create}_{den} = \lambda \phi.\ \mathbf{case}\ \phi\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \alpha \;\Rightarrow\; \mathit{create}_{obj}\ \alpha\ \mathit{newObject}_{\alpha}\ 0 \\
\qquad\qquad\qquad\qquad f \;\Rightarrow\; \mathit{newFunction} \\
\qquad \mathit{create}_{obj} \;:\; \alpha : \mathbf{Type}_{obj} \twoheadrightarrow \mathbf{Obj} \rightarrow \mathbf{Offset} \rightarrow [\![\alpha]\!]_{obj} \\
\qquad \mathit{create}_{obj} = \lambda \alpha.\ \lambda h.\ \lambda j.\ \mathbf{case}\ \alpha\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \mathit{obj}\,[\tau, q] \quad\;\; \Rightarrow\; \langle h, j \rangle \\
\qquad\qquad\qquad\qquad \mathit{array}\,[\alpha', n] \;\Rightarrow\; \lambda k.\ (k \geq 0) \wedge (k < n) \rightarrow \mathit{create}_{obj}\ \alpha'\ h\ (j + k \cdot \mathit{sizeof}(\alpha))\,,\ \top \\
\qquad \langle m_i, \rho_p \rangle \;= \rho \\
\quad \mathbf{in}\ \neg \mathit{isLocal}(e, I\ \mathit{ide}) \rightarrow \top\,, \\
\qquad\quad \mathbf{case}\ e[I\ \mathit{raw}\ \mathit{ide}]\ \mathbf{of} \\
\qquad\qquad \mathit{normal}\,[\phi] \;\Rightarrow\; \mathit{unit}\ \langle m_i\{I \mapsto \mathit{create}_{den}\ \phi\}, \rho_p \rangle \\
\qquad\qquad \mathbf{otherwise} \;\Rightarrow\; \top
\end{array}
$$

This function creates the object or function that corresponds with identifier $I$ in the given dynamic environment. The result is the updated dynamic environment. An error occurs if $I$ is not defined locally in the environment or if it is a *typedef*. If the object is an array, all of its elements are created.

ALLOCATE OBJECT ▶

$$
\begin{array}{l}
\mathit{allocate} \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Ide} \to \mathsf{C}(\mathbf{U}) \\
\mathit{allocate} \;=\; \lambda e.\, \lambda \rho.\, \lambda I. \\
\quad \mathbf{let}\ \mathit{allocate}_{den} \;:\; \phi : \mathbf{Type}_{den} \twoheadrightarrow [\![\phi]\!]_{den} \to \mathsf{C}(\mathbf{U}) \\
\qquad \mathit{allocate}_{den} = \lambda \phi.\, \lambda d.\, \mathbf{case}\ \phi\ \mathbf{of} \\
\qquad\qquad\qquad \alpha \;\Rightarrow\; \mathit{allocate}_{obj}\ \alpha\ d \\
\qquad\qquad\qquad f \;\Rightarrow\; \mathit{unit}\ u \\
\qquad \mathit{allocate}_{obj} \;:\; \alpha : \mathbf{Type}_{obj} \twoheadrightarrow [\![\alpha]\!]_{obj} \to \mathsf{C}(\mathbf{U}) \\
\qquad \mathit{allocate}_{obj} = \lambda \alpha.\, \lambda d.\, \mathbf{case}\ \alpha\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{obj}\,[\tau, q] \;\Rightarrow\; \mathit{setState}_{\mathsf{C}}\ (\mathit{stateAllocate}_{obj\,[\tau,q]}\ d)\ *\ (\lambda s.\, \mathit{unit}\ u) \\
\qquad\qquad\qquad \mathit{array}\,[\alpha', n] \;\Rightarrow\; \mathit{foldln}\,[0, n-1]\ (\lambda k.\, \mathit{allocate}_{obj}\ \alpha'(d\ k))\ (\mathit{unit}\ u) \\
\qquad \langle m_i, \rho_p \rangle \quad = \rho \\
\quad \mathbf{in}\ \neg \mathit{isLocal}(e, I\ \mathit{ide}) \to \top\,, \\
\qquad\quad \mathbf{case}\ e[I\ \mathit{raw}\ \mathit{ide}]\ \mathbf{of} \\
\qquad\qquad \mathit{normal}\,[\phi] \;\Rightarrow\; \mathit{allocate}_{den}\ \phi\ (m_i\ I) \\
\qquad\qquad \mathbf{otherwise} \;\Rightarrow\; \top
\end{array}
$$

This function allocates the object that corresponds with identifier $I$ in the current program state. The result is a computation with an unimportant result. An error occurs if $I$ is not defined locally in the environment or if it is a *typedef*. If the object is an array, all of its elements are allocated. If $I$ corresponds to a function, the program state remains unaltered.

DESTROY OBJECT ▶

$$
\begin{array}{l}
\mathit{destroy} \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Ide} \to \mathsf{C}(\mathbf{U}) \\
\mathit{destroy} \;=\; \lambda e.\, \lambda \rho.\, \lambda I. \\
\quad \mathbf{let}\ \mathit{destroy}_{den} \;:\; \phi : \mathbf{Type}_{den} \twoheadrightarrow [\![\phi]\!]_{den} \to \mathsf{C}(\mathbf{U}) \\
\qquad \mathit{destroy}_{den} = \lambda \phi.\, \lambda d.\, \mathbf{case}\ \phi\ \mathbf{of} \\
\qquad\qquad\qquad \alpha \;\Rightarrow\; \mathit{destroy}_{obj}\ \alpha\ d \\
\qquad\qquad\qquad f \;\Rightarrow\; \mathit{unit}\ u \\
\qquad \mathit{destroy}_{obj} \;:\; \alpha : \mathbf{Type}_{obj} \twoheadrightarrow [\![\alpha]\!]_{obj} \to \mathsf{C}(\mathbf{U}) \\
\qquad \mathit{destroy}_{obj} = \lambda \alpha.\, \lambda d.\, \mathbf{case}\ \alpha\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{obj}\,[\tau, q] \;\Rightarrow\; \mathit{setState}_{\mathsf{C}}\ (\mathit{stateDestroy}_{obj\,[\tau,q]}\ d)\ *\ (\lambda s.\, \mathit{unit}\ u) \\
\qquad\qquad\qquad \mathit{array}\,[\alpha', n] \;\Rightarrow\; \mathit{foldln}\,[0, n-1]\ (\lambda k.\, \mathit{destroy}_{obj}\ \alpha'(d\ k))\ (\mathit{unit}\ u) \\
\qquad \langle m_i, \rho_p \rangle \quad = \rho \\
\quad \mathbf{in}\ \neg \mathit{isLocal}(e, I\ \mathit{ide}) \to \top\,, \\
\qquad\quad \mathbf{case}\ e[I\ \mathit{raw}\ \mathit{ide}]\ \mathbf{of} \\
\qquad\qquad \mathit{normal}\,[\phi] \;\Rightarrow\; \mathit{destroy}_{den}\ \phi\ (m_i\ I) \\
\qquad\qquad \mathbf{otherwise} \;\Rightarrow\; \top
\end{array}
$$

This function deallocates the object that corresponds with identifier $I$ from the current program state. The result is a computation with an unimportant result. An error occurs if $I$ is not defined locally in the environment or if it is a *typedef*. If the object is an array, all of its elements are deallocated. If $I$ corresponds to a function, the program state remains unaltered.

LOOKUP ▶

$$
\begin{array}{l}
\mathit{lookup} \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to I : \mathbf{Ide} \twoheadrightarrow \mathsf{V}([\![e \Downarrow I]\!]) \\
\mathit{lookup} \;=\; \lambda e.\, \mathit{strict}_{\mathrm{I}}\ (\lambda \rho.\, \lambda I. \\
\quad \mathbf{let}\ \langle m, \rho_p \rangle = \rho \\
\quad \mathbf{in}\ \neg \mathit{isLocal}(e, I\ \mathit{ide}) \to \top\,, \\
\qquad\quad \mathbf{case}\ e[I\ \mathit{raw}\ \mathit{ide}]\ \mathbf{of} \\
\qquad\qquad \mathit{normal}\,[\phi] \;\Rightarrow\; m_i\ I \\
\qquad\qquad \mathbf{otherwise} \;\Rightarrow\; \top)
\end{array}
$$

This function returns the dynamic meaning of identifier $I$ in $\rho$. An error occurs if $I$ is not defined locally in the environment or if it is a *typedef*.

### 11.10.2 Member environments

Dynamic member environment associate identifiers that are members of structures or unions to their dynamic meanings. Again, the static member environments must be known. They come in two flavours. In the first case, the dynamic meanings of members are elements of dynamic data type domains. In the second, they are elements of dynamic member type domains. Thus, the first case corresponds to treating a member as the value that it contains, whereas the second corresponds to treating it as an l-value.

Given a static member environment $\pi$ and an identifier $I$ that is declared in $\pi$, the dynamic domains $[\![\pi \Downarrow I]\!]_{dat}$ and $[\![\pi \Downarrow I]\!]_{mem}$ that are defined below represent the two aforementioned flavours of dynamic meanings.

DEFINITION

$$
\begin{array}{l}
[\![\pi \Downarrow I]\!]_{dat} \;=\; \textbf{let}\ \langle n, m_n, m_i\rangle = \pi\ \textbf{in case}\ m_i\ I\ \textbf{of} \\
\quad \mathit{obj}\ [\tau, q] \;\Rightarrow\; [\![\tau]\!]_{dat} \\
\quad \mathit{bitfield}\ [\beta, q, n] \;\Rightarrow\; [\![\mathit{datify}\,\beta]\!]_{dat} \\
\quad \textbf{otherwise} \;\Rightarrow\; \mathbf{U}
\end{array}
$$

$$
\begin{array}{l}
[\![\pi \Downarrow I]\!]_{mem} \;=\; \textbf{let}\ \langle n, m_n, m_i\rangle = \pi\ \textbf{in} \\
\quad (m_i\ I = \top) \rightarrow \mathbf{U}\,,\ [\![m_i\ I]\!]_{mem}
\end{array}
$$

It should be noted that array members can only be given meanings of the second flavour.

LOOKUP

The following two functions are used for looking up the dynamic meaning of structure and union members respectively.

- $\mathit{structMember}_\pi \;:\; \mathbf{Addr} \rightarrow I : \mathbf{Ide} \twoheadrightarrow [\![\pi \Downarrow I]\!]_{mem}$
- $\mathit{unionMember}_\pi \;:\; \mathbf{Addr} \rightarrow I : \mathbf{Ide} \twoheadrightarrow [\![\pi \Downarrow I]\!]_{mem}$

Both functions take as parameters the address of the structure or union object and the identifier of the member. Their definition is implementation-defined and is omitted here.

### 11.10.3 Function prototypes

The dynamic domains related to function prototypes. Given a function prototype $p$, $[\![p]\!]_{Prot}$ is the domain of dynamic values for the actual parameters of functions whose prototype is $p$.

DEFINITION

$$[\![p]\!]_{Prot} \;=\; i : \mathbf{N} \twoheadrightarrow [\![p \Downarrow i]\!]$$

For each of the function's parameter a dynamic meaning is associated. That is, for a given function prototype $p$ and a given integer number $i$, $[\![p \Downarrow i]\!]$ is the domain of possible dynamic values for the $i$-th argument of $p$. This domain is defined below.

$$
\begin{array}{l}
[\![p \Downarrow i]\!] \;=\; \textbf{let}\ \langle n, m_p, b_{ell}\rangle = p\ \textbf{in} \\
\quad (1 \le i \le n) \rightarrow [\![m_p\ i]\!]_{dat}\,, \\
\quad b_{ell} \wedge (i > n) \rightarrow \mathbf{Arg}\,,\ \mathbf{I}
\end{array}
$$

Notice that the presence of ellipsis in function prototypes perplexes the definition of this domain. The dynamic meaning of actual parameters that fall in the part of the prototype after the ellipsis are elements of the special domain $\mathbf{Arg}$, defined as the coalesced sum of all dynamic data type domains.

■ $\mathbf{Arg} \;=\; \bigoplus_{\tau \in \mathbf{Type}_{dat}} [\![\tau]\!]_{dat}$

### 11.10.4 Function code environments

Function code environments associate function identifiers, i.e. elements of domain **Fun** to the dynamic meanings of their definitions. The use of pointers to functions in C, combined with type casting, enforces run-time checking of function types in the semantics of function calls. Thus, the static types of functions must be included in the dynamic environment.

DEFINITION ■ $\xi \;:\; \mathbf{Cod} \;=\; \mathbf{Fun} \to f : \mathbf{Type}_{fun} \ltimes [\![f]\!]_{fun}$

The definition of function code environments is straightforward. They are functions taking as parameter a function identifier. Notice the use of a dependent product in the result. The first part of the product is the function's static type and the second part is the dynamic meaning.

The following operations are defined for function code environments.

LOOKUP ▶ $\cdot[\cdot] \;:\; \mathbf{Cod} \times \mathbf{Fun} \to \mathsf{V}(f : \mathbf{Type}_{fun} \ltimes [\![f]\!]_{fun})$

$\xi[d_f] \;=\; (\xi\ d_f \neq \top) \to unit\ (\xi\ d_f)\,,\ error$

This operation returns the dynamic meaning of a given function in a given function code environment. An error occurs if the function is not defined.

UPDATE ▶ $\cdot[\cdot \mapsto \cdot] \;:\; \mathbf{Cod} \times \mathbf{Fun} \times (f : \mathbf{Type}_{fun} \ltimes [\![f]\!]_{fun}) \to \mathsf{V}(\mathbf{Cod})$

$\xi[d_f \mapsto w] \;=\; unit\ \xi\{d_f \mapsto w\}$

This operation updates the function code environment $\xi$ by setting the dynamic meaning of function $d_f$ to $w$. The result if the updated environment.

## 11.11 Scopes

MOTIVATION The modelling of nested scopes in dynamic type environments is not adequate for the developed semantics. C allows jumps of various kinds between different and even unrelated scopes. Furthermore, the standard requires that object allocation and deallocation must take place in jumps between different scopes. For these reasons, a special treatment of scopes is required in the dynamic semantics of C.

DEFINITION Every scope is identified by an element of domain **ScopeId**, i.e. an integer number. The top element of this domain corresponds to the outermost scope. Domain **Scope** represents information about scopes and their structure.

■ $\mathbf{ScopeId} \;=\; N$

■ $\eta \;:\; \mathbf{Scope} \;=\; \mathbf{ScopeId} \times (\mathbf{ScopeId} \to \mathbf{ScopeId}) \times (\mathbf{ScopeId} \to \mathbf{Ent})$
$\times(\mathbf{ScopeId} \to e : \mathbf{Ent} \twoheadrightarrow [\![ \downarrow e ]\!]_{Ent} \to \mathsf{C}([\![e]\!]_{Ent}))$
$\times(\mathbf{ScopeId} \to e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}([\![ \downarrow e ]\!]_{Ent}))$

The first part of the product in $\mathbf{Scope}$ is the identifier of the current scope. The second part is a function mapping each scope identifier to the identifier of the enclosing scope, representing thus the tree-like structure of scopes. The third part is a function which associates scopes with their static type environments. Finally, the fourth and fifth parts associate scopes with the actions of creating and destroying objects, i.e. with the notions of the dynamic meaning of declarations that are made in these scopes.

The following functions concerning scopes and scope information are useful in the specification of the semantics for C statements.

▶ $scopeEmpty_e \;:\; \mathbf{Scope}$ EMPTY SCOPE

$scopeEmpty_e \;=\; \langle \top, \top, \top\{\top \mapsto e\}, \top, \top \rangle$

An empty scope information contains only the outermost scope. The last two parts are $\top$ since the outermost scope contains no statements and jumps there are not allowed.

▶ $scopeGetId \;:\; \mathbf{Scope} \to \mathbf{ScopeId}$ GET IDENTIFIER

$scopeGetId \;=\; \lambda\,\eta.\ \mathbf{let}\ \langle n, m_p, m_e, m_c, m_d \rangle = \eta\ \mathbf{in}\ n$

This function returns the identifier of the current scope.

▶ $defineBlock \;:\; \mathbf{ScopeId} \to \mathbf{Ent} \to$ DEFINE BLOCK
$(e : \mathbf{Ent} \twoheadrightarrow [\![ \downarrow e ]\!]_{Ent} \to \mathsf{C}([\![ e ]\!]_{Ent})) \to$
$(e : \mathbf{Ent} \twoheadrightarrow [\![ e ]\!]_{Ent} \to \mathsf{C}([\![ \downarrow e ]\!]_{Ent})) \to \mathbf{Scope} \to \mathsf{V}(\mathbf{Scope})$

$defineBlock \;=\; \lambda\,m.\,\lambda\,e.\,\lambda\,f_c.\,\lambda\,f_d.\,\lambda\,\eta.\ \mathbf{let}\ \langle n, m_p, m_e, m_c, m_d \rangle = \eta\ \mathbf{in}$
$unit\ \langle m, m_p\{m \mapsto n\}, m_e\{m \mapsto e\}, m_c\{m \mapsto f_c\}, m_d\{m \mapsto f_d\} \rangle$

This function defines a new scope. Its parameters are the identifier for the new scope, its static type environment, the functions for the creation and destruction of local objects and the structure representing current scope information. The result is the updated scope information.

▶ $endBlock \;:\; \mathbf{Scope} \to \mathsf{V}(\mathbf{Scope})$ END BLOCK

$endBlock \;=\; \lambda\,\eta.\ \mathbf{let}\ \langle n, m_p, m_e, m_c, m_d \rangle = \eta\ \mathbf{in}\ unit\ \langle m_p\ n, m_p, m_e, m_c, m_d \rangle$

This function leaves the current scope and returns to the enclosing one.

▶ $inScope \;:\; \mathbf{ScopeId} \to (\mathbf{Scope} \to A) \to \mathbf{Scope} \to A$ IN SCOPE

$inScope \;=\; \lambda\,m.\,\lambda\,f.\,\lambda\,\eta.\ \mathbf{let}\ \langle n, m_p, m_e, m_c, m_d \rangle = \eta\ \mathbf{in}\ f\ \langle m, m_p, m_e, m_c, m_d \rangle$

This polymorphic function sets the current scope to the one identified by its first parameter.

▶ $scopeUse \;:\; \mathbf{Scope} \to (\mathbf{Scope} \to A) \to \mathbf{Scope} \to A$ USE SCOPE

$scopeUse \;=\; \lambda\,\eta.\,\lambda\,f.\,\lambda\,\eta'.\ f\ \eta$

This polymorphic function is mostly used on elements of domains of type $\mathsf{K}_\tau(D)$ and $\mathsf{L}_\tau(D)$, which have the general form $\mathbf{Scope} \to$ and are defined in Section 11.12. Viewed under this perspective, it returns a computation which the scope information given by the first parameter.

GO TO SCOPE

▶ $scopeGoto \;:\; \mathbf{Scope} \to \mathbf{ScopeId} \to e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}(e' : \mathbf{Ent} \ltimes [\![e']\!]_{Ent})$

$$
\begin{array}{lll}
scopeGoto & = & \lambda\eta.\,\lambda m.\,\lambda e.\,\lambda\rho. \\
\quad \textbf{let}\ \langle n, m_p, m_e, m_c, m_d\rangle = \eta & & \\
\qquad isAncestor & : & \mathbf{ScopeId} \to \mathbf{ScopeId} \to \mathbf{T} \\
\qquad isAncestor & = & \lambda m.\ fix\ (\lambda g.\,\lambda n.\ (m = n) \to true\,, \\
 & & \qquad\qquad (n = \top) \to false\,, \\
 & & \qquad\qquad g\ (m_p\ n)) \\
\qquad commonAncestor & : & \mathbf{ScopeId} \times \mathbf{ScopeId} \to \mathbf{ScopeId} \\
\qquad commonAncestor & = & fix\ (\lambda g.\,\lambda\langle m, n\rangle.\ (isAncestor\ m\ n) \to m\,, \\
 & & \qquad\qquad (isAncestor\ n\ m) \to n\,, \\
 & & \qquad\qquad g\ \langle m_p\ m, m_p\ n\rangle) \\
\qquad ascend & : & \mathbf{ScopeId} \times \mathbf{ScopeId} \to e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}(e' : \mathbf{Ent} \ltimes [\![e']\!]_{Ent}) \\
\qquad ascend & = & fix\ (\lambda g.\,\lambda\langle m, n\rangle.\,\lambda e.\,\lambda\rho.\ (m = n) \to unit\ \langle e, \rho\rangle\,, \\
 & & \qquad\qquad g\ \langle m, m_p\ n\rangle\ e\ \rho\ *\ (\lambda\langle e', \rho'\rangle. \\
 & & \qquad\qquad \textbf{let}\ e'' = m_e\ n \\
 & & \qquad\qquad \textbf{in}\ \ m_c\ n\ e''\ \rho'\ *\ (\lambda\rho''. \\
 & & \qquad\qquad\qquad unit\ \langle e'', \rho''\rangle))\ \ ) \\
\qquad descend & : & \mathbf{ScopeId} \times \mathbf{ScopeId} \to e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}(e' : \mathbf{Ent} \ltimes [\![e']\!]_{Ent}) \\
\qquad descend & = & fix\ (\lambda g.\,\lambda\langle m, n\rangle.\,\lambda e.\,\lambda\rho.\ (m = n) \to unit\ \langle e, \rho\rangle\,, \\
 & & \qquad\qquad m_d\ m\ e\ \rho\ *\ (\lambda\rho'. \\
 & & \qquad\qquad \textbf{let}\ e' = m_e\ (m_p\ m) \\
 & & \qquad\qquad \textbf{in}\ \ g\ \langle m_p\ m, n\rangle\ e'\ \rho')) \\
\qquad k & = & commonAncestor\ \langle n, m\rangle \\
\textbf{in}\ \ descend\ \langle n, k\rangle\ e\ \rho\ *\ (\lambda\langle e', \rho'\rangle. & & \\
\quad ascend\ \langle k, m\rangle\ e'\ \rho') & &
\end{array}
$$

This function performs a change in the static and dynamic environments, required when jumping between different scopes. It takes as parameters the current scope information, an identifier for a target scope and the current static and dynamic environments; it returns a computation resulting in the environments corresponding to the target scope. This computation ensures that all objects going out of scope are destroyed and all objects coming into scope are created. The tree-like structure used in the scope information is used for this purpose.

SCOPES AND CONTINUATIONS

■ $\mathbf{SC} \quad = \quad \mathbf{ScopeId} \times (e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{C})$

The domain $\mathbf{SC}$ represents a continuation that jumps to a different scope. The first part of the product is the identifier of the scope where the jump is made. The second part is a function which, given a static and dynamic environment for the scope identified by the first part, returns a continuation.

▶ $makeSC \;:\; \mathbf{ScopeId} \to (e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{C}) \to \mathbf{SC}$

$makeSC \;=\; \lambda n.\,\lambda f.\,\langle m, f\rangle$

This simple function creates an element of $\mathbf{SC}$ from its components.

▶ $convertSC \;:\; \mathbf{ScopeId} \to \mathbf{C} \to \mathbf{SC}$

$convertSC \;=\; \lambda n.\,\lambda c.\ makeSC\ m\ (\lambda e.\,\lambda\rho.\ c)$

This function converts a continuation to an element of $\mathbf{SC}$, using the given scope identifier for the target scope.

▶ $useContinuation \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Scope} \to \mathbf{SC} \to \mathsf{C}(\mathbf{U})$

$useContinuation \;=\; \lambda e.\, \lambda \rho.\, \lambda \eta.\, \lambda c.\, \mathbf{let}\ \langle m, f\rangle = c\ \mathbf{in}$
$\quad scopeGoto\ \eta\ m\ e\ \rho\ *\ (\lambda e', \rho'.\, escape\ (f\ e'\ \rho'))$

This function performs a jump using a continuation from a different scope. The current static and dynamic type environments are passed as parameters, together with the current scope and the continuation that will be used. In its definition, function *useContinuation* uses function *scopeGoto* to change to the proper environment for the target scope and then function *escape* to perform the jump. The result is a computation of unimportant result.

## 11.12 Monads for statement semantics

Two families of monads are defined in this section for representing the dynamic semantics of C statements. These monads are closely related and present many similarities. In brief, the family $\mathsf{K}$ of monads represents computations corresponding to the execution of statements, whereas the family $\mathsf{L}$ of monads is used for the extraction of other useful information concerning the execution of statements.

■ $k \;:\; \mathsf{K}_\tau(D) \;=\; \mathbf{Scope} \to \mathbf{SC} \times \mathbf{SC} \to ([\![\tau]\!]_{dat} \to \mathbf{SC}) \to \mathsf{C}(D)$ MONAD K

For a given data type $\tau$, monad $\mathsf{K}_\tau$ is used in the dynamic semantics of statements in the body of a function whose result type is $\tau$. Elements of domain $\mathsf{K}_\tau(D)$ are functions whose parameters are:

- The current scope information;
- A pair of continuations corresponding to *break* and *continue* statements;
- A function mapping possible results of the current function to corresponding continuations.

The result is a computation resulting in $D$. All parameters may be viewed as information that is passed to the computation.The unit of monad $\mathsf{K}_\tau$ is defined as:

▶ $unit_{\mathsf{K}} \;:\; D \to \mathsf{K}_\tau(D)$

$unit_{\mathsf{K}} \;=\; \lambda v.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, unit_{\mathsf{C}}\ v$

that is, the given parameters are simply ignored. The bind operator is defined as:

▶ $\cdot\ *_{\mathsf{K}}\ \cdot \;:\; \mathsf{K}_\tau(A) \times (A \to \mathsf{K}_\tau(B)) \to \mathsf{K}_\tau(B)$

$k\ *_{\mathsf{K}}\ f \;=\; \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, k\ \eta\ \langle c_b, c_c\rangle\ c_r\ *_{\mathsf{C}}\ (\lambda a.\, f\ a\ \eta\ \langle c_b, c_c\rangle\ c_r)$

which means that the values of the parameters are used throughout the whole computation. It is easy to see that this monad satisfies the three monad laws.

■ $z \;:\; \mathsf{L}_\tau(D) \;=\; \mathbf{Scope} \to \mathbf{SC} \times \mathbf{SC} \to ([\![\tau]\!]_{dat} \to \mathbf{SC}) \to (\mathbf{U} \to \mathbf{C}) \to D$ MONAD L

In a way similar to monad $\mathsf{K}_\tau$, monad $\mathsf{L}_\tau$ is defined for all data types $\tau$. Elements of type $\mathsf{L}_\tau(D)$ are functions which take as parameters all the those described above and also the following:

- A continuation corresponding to the statement's normal completion.

The result is not a computation, but only an element of type $D$. The type $\mathbf{U} \to \mathbf{C}$ has been chosen for the last parameter, instead of just $\mathbf{C}$, so that $\mathsf{K}_\tau(U)$ and $\mathsf{L}_\tau(\mathbf{C})$ are the same domain. This is a useful property that will be used later. The unit of monad $\mathsf{L}_\tau$ is defined as:

- $unit_{\mathsf{L}} \;:\; D \to \mathsf{L}_\tau(D)$
  $unit_{\mathsf{L}} \;=\; \lambda\, v.\, \lambda\, \eta.\, \lambda\, \langle c_b, c_c \rangle.\, \lambda\, c_r.\, \lambda\, \kappa.\, v$

ignoring all parameters, whereas the bind operator is defined as:

- $\cdot \ast_{\mathsf{L}} \cdot \;:\; \mathsf{L}_\tau(A) \times (A \to \mathsf{L}_\tau(B)) \to \mathsf{L}_\tau(B)$
  $z \ast_{\mathsf{L}} f \;=\; \lambda\, \eta.\, \lambda\, \langle c_b, c_c \rangle.\, \lambda\, c_r.\, \lambda\, \kappa.\, \mathbf{let}\ a = z\ \eta\ \langle c_b, c_c \rangle\ c_r\ \kappa\ \mathbf{in}\ f\ a\ \eta\ \langle c_b, c_c \rangle\ c_r\ \kappa$

Again it is easy to verify that the monad satisfies the three monad laws.

ERRORS Similarly to the previously defined monads, top elements of monads $\mathsf{K}_\tau$ and $\mathsf{L}_\tau$ represent the occurence of errors.

- $error_{\mathsf{K}} \;:\; \mathsf{K}_\tau(D)$
  $error_{\mathsf{K}} \;=\; \top$

- $error_{\mathsf{L}} \;:\; \mathsf{L}_\tau(D)$
  $error_{\mathsf{L}} \;=\; \top$

LIFTING The following functions are useful for converting between various kinds of monads. Primitive functions are $lift_{\mathsf{G}\to\mathsf{K}}$ and $lift_{\mathsf{V}\to\mathsf{L}}$ and all other functions are derived.

- $lift_{\mathsf{G}\to\mathsf{K}} \;:\; \mathsf{G}(A) \to \mathsf{K}_\tau(A)$
  $lift_{\mathsf{G}\to\mathsf{K}} \;=\; \lambda\, g.\, \lambda\, \eta.\, \lambda\, \langle c_b, c_c \rangle.\, \lambda\, c_r.\, fullExpression\ g$

- $lift_{\mathsf{V}\to\mathsf{L}} \;:\; \mathsf{V}(A) \to \mathsf{L}_\tau(A)$
  $lift_{\mathsf{V}\to\mathsf{L}} \;=\; unit_{\mathsf{L}}$

- $lift_{\mathsf{E}\to\mathsf{K}} \;:\; \mathsf{E}(A) \to \mathsf{K}_\tau(A)$
  $lift_{\mathsf{E}\to\mathsf{K}} \;=\; lift_{\mathsf{G}\to\mathsf{K}} \circ lift_{\mathsf{C}\to\mathsf{G}} \circ lift_{\mathsf{V}\to\mathsf{C}} \circ lift_{\mathsf{E}\to\mathsf{V}}$

- $lift_{\mathsf{E}\to\mathsf{L}} \;:\; \mathsf{E}(A) \to \mathsf{L}_\tau(A)$
  $lift_{\mathsf{E}\to\mathsf{L}} \;=\; lift_{\mathsf{V}\to\mathsf{L}} \circ lift_{\mathsf{E}\to\mathsf{V}}$

- $lift_{\mathsf{V}\to\mathsf{K}} \;:\; \mathsf{V}(A) \to \mathsf{K}_\tau(A)$
  $lift_{\mathsf{V}\to\mathsf{K}} \;=\; lift_{\mathsf{G}\to\mathsf{K}} \circ lift_{\mathsf{C}\to\mathsf{G}} \circ lift_{\mathsf{V}\to\mathsf{C}}$

- $lift_{\mathsf{C}\to\mathsf{K}} \;:\; \mathsf{C}(A) \to \mathsf{K}_\tau(A)$
  $lift_{\mathsf{C}\to\mathsf{K}} \;=\; lift_{\mathsf{G}\to\mathsf{K}} \circ lift_{\mathsf{C}\to\mathsf{G}}$

The following functions, most of which are polymorphic, define aspects of the execution behaviour of statements.

▶ $result_\tau \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to [\![\tau]\!]_{dat} \to \mathsf{K}_\tau(\mathbf{U})$
$result_\tau \;=\; \lambda e.\, \lambda \rho.\, \lambda d.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, useContinuation\ e\ \rho\ \eta\ (c_r\ d)$

FUNCTION RESULTS

This function terminates execution of the function body by returning the result value given by its third parameter. The first two parameters are the static and dynamic type environments. The result is a statement computation of an unimporant result.

▶ $funbody \;:\; \mathsf{K}_\tau(U) \to \mathsf{G}([\![\tau]\!]_{dat})$
$funbody \;=\; \lambda k.\, lift_{\mathsf{C}\to\mathsf{G}}\ (\lambda \kappa.\, k\ \top\ \langle \top, \top\rangle\ ((convertSC\ \top) \circ \kappa)\ (\lambda u.\, \kappa\ \top))$

This function converts a statement computation representing the body of a function to an expression computation. The result of the expression computation is the result that is returned by the function body.

▶ $use \;:\; \mathsf{L}_\tau(A) \to (A \to \mathsf{K}_\tau(\mathbf{U})) \to \mathsf{K}_\tau(\mathbf{U})$
$use \;=\; \lambda z.\, \lambda f.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, \lambda \kappa.\, f\ (z\ \eta\ \langle c_b, c_c\rangle\ c_r\ \kappa)\ \eta\ \langle c_b, c_c\rangle\ c_r\ \kappa$

COMBINING K AND L

This polymorphic function is the bridge between monads $\mathsf{K}_\tau$ and $\mathsf{L}_\tau$. It resembles the bind operator. The value that is hidden in the monad in the first parameter is applied to the function in the second parameter and the result is returned.

▶ $follow \;:\; \mathsf{L}_\tau(A) \to \mathsf{K}_\tau(\mathbf{U}) \to \mathsf{L}_\tau(A)$
$follow \;=\; \lambda z.\, \lambda f.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, \lambda \kappa.\, z\ \eta\ \langle c_b, c_c\rangle\ c_r\ (\lambda u.\, k\ \eta\ \langle c_b, c_c\rangle\ c_r\ \kappa)$

This polymorphic function specifies that the statement whose semantics in isolation is defined by the first parameter is followed, under normal order of execution, by the statement whose semantics is defined by the second parameter. The result is the updated semantics of the first parameter.

▶ $getBreak \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{K}_\tau(\mathbf{U})$
$getBreak \;=\; \lambda e.\, \lambda \rho.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, useContinuation\ e\ \rho\ \eta\ c_b$

BREAK AND CONTINUE

▶ $getContinue \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{K}_\tau(\mathbf{U})$
$getContinue \;=\; \lambda e.\, \lambda \rho.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, useContinuation\ e\ \rho\ \eta\ c_c$

These functions take as parameters the current static and dynamic type environments and return a statement computation that corresponds to execution of a *break* or *continue* statement respectively.

▶ $setBreak \;:\; \mathsf{L}_\tau(A) \to \mathsf{L}_\tau(A)$
$setBreak \;=\; \lambda z.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, \lambda \kappa.\, z\ \eta\ \langle convertSC\ (scopeGetId\ \eta)\ (\kappa\ u), c_c\rangle\ c_r\ \kappa$

▶ $setContinue \;:\; \mathsf{L}_\tau(A) \to \mathsf{L}_\tau(A)$
$setContinue \;=\; \lambda z.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, \lambda \kappa.\, z\ \eta\ \langle c_b, convertSC\ (scopeGetId\ \eta)\ (\kappa\ u)\rangle\ c_r\ \kappa$

These polymorphic functions take as parameters the semantics of a statement in isolation and set the continuation corresponding to the *break* and *continue* statements respectively to the continuation under normal order of execution.

## 11.13 Label environments

LABELS An additional type of dynamic environment, related to the semantics of statements, is also required. Domain **Lab** represents label environments which associate label identifiers to continuations. A fixing process similar to the one used for static type environments in Chapter 6 must be used for label environments, since (possibly non-terminating) loops can be defined using labels and the *goto* statement.

■ $\ell \;:\; \mathbf{Lab} \;=\; (\mathbf{Ide} \to \mathbf{SC}) \times \mathbf{T}$

The first part of the product is the function for the association of identifiers. The second part is a truth value which denotes whether the fixing process has started. The following functions are defined for managing label environments.

EMPTY LABELS ▶ $\ell_\circ \;:\; \mathbf{Lab}$

$\ell_\circ \;=\; \langle \bot, \mathit{false} \rangle$

This function returns an empty label environment. All label identifiers correspond to non-terminating continuations and the fixing process has not yet started.

INITIALIZE FIXING ▶ $\mathit{init\text{-}fix\text{-}L} \;:\; \mathbf{Lab} \to \mathsf{L}_\tau(\mathbf{Lab})$

$\mathit{init\text{-}fix\text{-}L} \;=\; \lambda \ell.\, \mathbf{let}\ \langle m_l, b_{fix} \rangle = \ell\ \mathbf{in}\ \neg b_{fix} \to \mathit{unit}\ \langle m_l, \mathit{true} \rangle,\ \mathit{error}$

This function initializes the fixing process. An error occurs if this has already been started.

FIX LABELS ▶ $\mathit{rec\text{-}L} \;:\; (\mathbf{Lab} \to \mathsf{L}_\tau(\mathbf{Lab})) \to \mathbf{Lab} \to \mathsf{L}_\tau(\mathbf{Lab})$

$\mathit{rec\text{-}L} \;=\; \lambda f.\, \lambda \ell.\, \mathbf{let}\ z = f\ \ell\ *\ \mathit{init\text{-}fix\text{-}L}\ \mathbf{in}\ \mathit{mclo}_{\mathsf{L}}\ z\ f$

This function performs the fixing of label environments. It uses the monadic closure operator and it similar in its definition to operator *init-fix* for type environments.

GET LABEL ▶ $\mathit{getLabel} \;:\; \mathbf{Ide} \to e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Lab} \to \mathsf{K}_\tau(\mathbf{U})$

$$\begin{array}{l}\mathit{getLabel} \;=\; \lambda I.\, \lambda e.\, \lambda \rho.\, \lambda \ell.\, \lambda \eta.\, \lambda \langle c_b, c_c \rangle.\, \lambda c_r. \\ \quad \mathbf{let}\ \mathit{getLabelRaw} \;:\; \mathbf{Ide} \to \mathbf{Lab} \to \mathbf{SC} \\ \quad\quad \mathit{getLabelRaw} = \lambda I.\, \lambda \ell.\, \mathbf{let}\ \langle m_l, b_{fix} \rangle = \ell\ \mathbf{in}\ b_{fix} \wedge (m_l\ I = \bot) \to \top,\ m_l\ I \\ \quad \mathbf{in}\ \mathit{useContinuation}\ e\ \rho\ \eta\ (\mathit{getLabelRaw}\ I\ \ell)\end{array}$$

This function takes as parameters an identifier, the current static and dynamic type environment and a label environment. It returns a statement computation, corresponding to the continuation associated with the identifier in the label environment. An error occurs if there is no such label identifier.

SET LABEL ▶ $\mathit{setLabel} \;:\; \mathbf{Ide} \to (e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{K}_\tau(\mathbf{U})) \to \mathbf{Lab} \to \mathsf{L}_\tau(\mathbf{Lab})$

$$\begin{array}{l}\mathit{setLabel} \;=\; \lambda I.\, \lambda k_f.\, \lambda \ell.\, \lambda \eta.\, \lambda \langle c_b, c_c \rangle.\, \lambda c_r.\, \lambda \kappa. \\ \quad \mathbf{let}\ \mathit{setLabelRaw} \;:\; [\![\tau']\!]_{dat} \to \mathbf{SC} \to \mathbf{Lab} \to \mathbf{Lab} \\ \quad\quad \mathit{setLabelRaw} = \lambda I.\, \lambda c'.\, \lambda \ell.\, \mathbf{let}\ \langle m_l, b_{fix} \rangle = \ell\ \mathbf{in}\ b_{fix} \vee (m_l\ I = \bot) \to \langle m_l\{I \mapsto c'\}, b_{fix} \rangle,\ \top \\ \quad \mathbf{in}\ \mathit{setLabelRaw}\ I\ (\mathit{makeSC}\ (\mathit{scopeGetId}\ \eta)\ (\lambda e_c.\, \lambda\_.\, \rho_c k_f\ e_c\ \rho_c\ \eta\ \langle c_b, c_c \rangle\ c_r\ \kappa))\ \ell\end{array}$$

This function updates the label environment by setting the continuation associated with identifier $I$ to the value given by $k_f$. An error occurs if the label identifier has already been declared and the fixing process is not under way.

Case labels, i.e. constant expressions which follow the *case* keyword in the body of *switch* statements, form one more kind of dynamic environment. However, no fixing process is required here, since there cannot be any loops caused by this kind of labels. Domain $\mathbf{Cas}_\tau$ is used for associating case labels with continuations. The data type $\tau$ is the type of the controlling expression of the *switch* statement. CASE LABELS

■ $\omega \;:\; \mathbf{Cas}_\tau \;=\; (\llbracket\tau\rrbracket_{dat} \to \mathbf{SC}) \times \mathbf{SC}$

The first part of the product is a function mapping the values of constant expressions to the corresponding continuations. The second part is the continuation that corresponds to the *default* label.

▶ $\omega_\circ \;:\; \mathbf{Cas}_\tau$ EMPTY CASES

$\omega_\circ \;=\; \top$

This function returns an empty case label environment. All continuations are set to erroneous values.

▶ $getCase_{\tau,\tau'} \;:\; \llbracket\tau'\rrbracket_{dat} \to e : \mathbf{Ent} \twoheadrightarrow \llbracket e\rrbracket_{Ent} \to \mathbf{Cas}_{\tau'} \to \mathsf{K}_\tau(\mathbf{U})$ GET CASE

$getCase_{\tau,\tau'} \;=\; \lambda d.\, \lambda e.\, \lambda \rho.\, \lambda \omega.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.$
$\quad \mathbf{let}\ getCaseRaw \;:\; \llbracket\tau'\rrbracket_{dat} \to \mathbf{Cas}_{\tau'} \to \mathbf{SC}$
$\qquad getCaseRaw = \lambda d.\, \lambda \omega.\, \mathbf{let}\ \langle m_l, c_d\rangle = \omega\ \mathbf{in}\ (m_l\ d \neq \top) \to m_l\ d,\ c_d$
$\qquad c \qquad\qquad = getCaseRaw\ d\ \omega$
$\quad \mathbf{in}\ \ (c = \top) \to unit\ u,\ useContinuation\ e\ \rho\ \eta\ c$

This function takes as parameters the value of the controlling expression, the current static and dynamic type environment and a case label environment. It returns a statement computation, corresponding to the continuation associated with the controlling expression in the given case label environment. A computation doing nothing is returned if the value of the controlling expression does not correspond to a case label and the *default* label has not been associated.

▶ $setCase_{\tau,\tau'} \;:\; \llbracket\tau'\rrbracket_{dat} \to (e : \mathbf{Ent} \twoheadrightarrow \llbracket e\rrbracket_{Ent} \to \mathsf{K}_\tau(\mathbf{U})) \to \mathbf{Cas}_{\tau'} \to \mathsf{L}_\tau(\mathbf{Cas}_{\tau'})$ SET CASE

$setCase_{\tau,\tau'} \;=\; \lambda d.\, \lambda k_f.\, \lambda \omega.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, \lambda \kappa.$
$\quad \mathbf{let}\ setCaseRaw \;:\; \llbracket\tau'\rrbracket_{dat} \to \mathbf{SC} \to \mathbf{Cas}_{\tau'} \to \mathbf{Cas}_{\tau'}$
$\qquad setCaseRaw = \lambda d.\, \lambda c'.\, \lambda \omega.\, \mathbf{let}\ \langle m_l, c_d\rangle = \omega\ \mathbf{in}\ (m_l\ d = \top) \to \langle m_l\{d \mapsto c'\}, c_d\rangle,\ \top$
$\quad \mathbf{in}\ \ setCaseRaw\ d\ (makeSC\ (scopeGetId\ \eta)\ (\lambda e_c.\, \lambda\_.\, \rho_c k_f\ e_c\ \rho_c\ \eta\ \langle c_b, c_c\rangle\ c_r\ \kappa))\ \omega$

This function updates the case label environment by setting the continuation associated with the given case expression to the value given by $k_f$. An error occurs if the same case expression has already been associated.

▶ $setDefault_{\tau,\tau'} \;:\; (e : \mathbf{Ent} \twoheadrightarrow \llbracket e\rrbracket_{Ent} \to \mathsf{K}_\tau(\mathbf{U})) \to \mathbf{Cas}_{\tau'} \to \mathsf{L}_\tau(\mathbf{Cas}_{\tau'})$ SET DEFAULT

$setDefault_{\tau,\tau'} \;=\; \lambda k_f.\, \lambda \omega.\, \lambda \eta.\, \lambda \langle c_b, c_c\rangle.\, \lambda c_r.\, \lambda \kappa.$
$\quad \mathbf{let}\ setDefaultRaw \;:\; \mathbf{SC} \to \mathbf{Cas}_{\tau'} \to \mathbf{Cas}_{\tau'}$
$\qquad setDefaultRaw = \lambda c'_d.\, \lambda \omega.\, \mathbf{let}\ \langle m_l, c_d\rangle = \omega\ \mathbf{in}\ (c_d = \top) \to \langle m_l, c'_d\rangle,\ \top$
$\quad \mathbf{in}\ \ setDefaultRaw\ (makeSC\ (scopeGetId\ \eta)\ (\lambda e_c.\, \lambda\_.\, \rho_c k_f\ e_c\ \rho_c\ \eta\ \langle c_b, c_c\rangle\ c_r\ \kappa))\ \omega$

This function updates the case label environment by setting the continuation associated with the *default* label to the value given by $k_f$. An error occurs if the *default* label has already been associated.

## 11.14 Auxiliary functions

SHIFT ▶ $shift \;:\; \mathbf{N} \to (\mathbf{N} \to A) \to \mathbf{N} \to A$

$shift \;=\; \lambda\, d.\, \lambda\, f.\, \lambda\, n.\; f\; (n + d)$

This polymorphic function shifts function $f$, whose domain is $\mathbf{N}$, by $d$ places to the left. That is, the $d$-th element of $f$ becomes the 0-th element of the resulting function.

ZERO MEMBER ▶ $zeroMember_m \;:\; [\![m]\!]_{mem} \to \mathsf{C}(\mathbf{U})$

$zeroMember_{obj\,[\tau,q]} \;=\; \lambda\, d_m.\; fullExpression\; (putValue_{\tau \mapsto obj\,[\tau,q]}\; d_m\; zeroValue_\tau)\; *\; (\lambda\, d.\; unit\; u)$

$zeroMember_{array\,[\alpha,n]} \;=\; \lambda\, d_f.\; zeroArray_\alpha\; d_f\; n$

$zeroMember_{bitfield\,[\beta,q,n]} \;=\; \lambda\, d_m.\; \mathbf{let}\; \tau = datify_{bit}\; \beta\; \mathbf{in}$
$\quad fullExpression\; (putValue_{\tau \mapsto bitfield\,[\beta,q,n]}\; d_m\; zeroValue_\tau)\; *\; (\lambda\, d.\; unit\; u)$

This function stores a zero value in the object whose address is specified by its first parameter. It is used in initializations.

ZERO ARRAY ▶ $zeroArray_\alpha \;:\; (\mathbf{N} \to [\![\alpha]\!]_{obj}) \to \mathbf{N} \to \mathsf{C}(\mathbf{U})$

$zeroArray_\alpha \;=\; \lambda\, d_f.\; \lambda\, n.\; (n \leq 0) \to unit\; u\;,$
$\quad zeroMember_\alpha\; (d_f\; 0)\; *\; (\lambda\, u.$
$\quad zeroArray_\alpha\; (shift\; 1\; d_f)\; (n-1))$

This function is used in the initialization of array elements that are not explicitly initialized.

ZERO STRUCTURE ▶ $zeroStruct_\pi \;:\; (I : \mathbf{Ide} \twoheadrightarrow [\![\pi \Downarrow I]\!]_{mem}) \to \mathsf{C}(\mathbf{U})$

$zeroStruct_\pi \;=\; \lambda\, d_{mf}.\; \mathbf{let}\; \langle n, m_n, m_i \rangle = \pi\; \mathbf{in}$
$\quad foldln\; [1, n]\; (\lambda\, k.\; \mathbf{let}\; I = m_n\; k\; \mathbf{in}\; zeroMember_{m_i\; I}\; (d_{mf}\; I))\; (unit\; u)$

This function is used in the initialization of structure members that are not explicitly initialized.

▶ $foldln\,[\cdot\,,\cdot\,] \;:\; (\mathbf{N} \times \mathbf{N}) \to (\mathbf{N} \to \mathsf{C}(\mathbf{U})) \to \mathsf{C}(\mathbf{U}) \to \mathsf{C}(\mathbf{U})$

$foldln\,[a,b] \;=\; \lambda\, f.\; \lambda\, c.\; (a \leq b) \to foldln\,[a+1,b]\; f\; (c\; *\; (\lambda\, u.\; f\; a))\;,\; c$

This function takes as parameters a range $[a, b]$ over the integer numbers, a function $f : \mathbf{N} \to \mathsf{C}(\mathbf{U})$ and a computation $c : \mathsf{C}(\mathbf{U})$. The result is the following computation:

$$c\; *\; (\lambda\, u_a.\; f\; a)\; *\; (\lambda\, u_{a+1}.\; f\; (a+1))\; *\; \ldots\; *\; (\lambda\, u_{b-1}.\; f\; (b-1))\; *\; (\lambda\, u_b.\; f\; b)$$

▶ $storeStringLit_\alpha \;:\; \textit{string-literal} \to (\mathbf{N} \to \mathbf{Addr}) \to \mathbf{N} \to \mathsf{C}(\mathbf{U})$

▶ $storeWideStringLit_\alpha \;:\; \textit{string-literal} \to (\mathbf{N} \to \mathbf{Addr}) \to \mathbf{N} \to \mathsf{C}(\mathbf{U})$

These two functions store the string literal (normal or wide) given by the first parameter to the memory locations that are given by the second parameter. The third parameter is the size of the character array that will hold the string literal. Type $\alpha$ is the type of the array's element. The result is a computation with an unimportant result.

# Chapter 12

# Dynamic semantics of expressions

CHAPTER OVERVIEW

This chapter defines the dynamic semantics of C expressions. Section 12.1 contains an introduction to the dynamic semantic functions and equations for expressions. The definition of semantics starts with the semantics of primary expressions in Section 12.2 and ends with the semantics of implicit coercions in Section 12.9.

## 12.1 Dynamic semantic functions and equations

SEMANTIC FUNCTIONS

In general, a dynamic semantic function maps well typed program phrases to dynamic semantic domains. A program phrase is well typed if there exists a typing derivation for it. The conclusion of this typing derivation specifies the phrase type, which is used to determine which dynamic semantic function is used. Thus, the definition of dynamic semantic functions follows not only the abstract syntax of C, but also the language's typing semantics.

Consider a non-terminal symbol *nt* which produces phrases that can be attributed the phrase type $\theta$. The dynamic semantics of a program phrase $P$ defined by this non-terminal symbol, with respect to type $\theta$, is denoted by $[\![ P ]\!]_\theta$. In this, it is assumed that a typing derivation exists, concluding in the judgement $e \vdash P : \theta$ for some environment $e$. This environment is usually a parameter of the dynamic semantic function.

In general, program phrases that can be attributed a given phrase type $\theta$ may be produced by different non-terminal symbols. For this reason, in the rest of Part IV, a line of the form $[\![ \theta ]\!] : D$ specifies that domain $D$ is used to represent the dynamic semantic meaning of program phrases that can be attributed phrase type $\theta$.

SEMANTIC EQUATIONS

Dynamic semantic equations are based on typing rules. Consider for example a rule of the form:

$$\frac{J_1 \quad \ldots \quad J_n}{e \vdash P : \theta} \quad (R)$$

Then, a dynamic semantic equation corresponding to this rule has the form:

*(R)* $\quad [\![ P ]\!]_\theta \; = \;$ *(expression which can use the dynamic semantics of phrases present in $J_i$)*

In the case of an equation corresponding to many typing rules sharing a common form, the common form of the rules is written first followed by the numbers of the typing rules. The equation is written next.

MULTIPLE DYNAMIC MEANINGS

As in the case of static semantic meanings, it is possible to have more than one dynamic semantic meanings for a given program phrase and phrase type. These meanings correspond to different aspects of execution. The multiple meanings are distinguished by prepending caligraphic letters, as in $\mathcal{X}[\![P]\!]_\theta$. The functions corresponding to each of the multiple meanings are defined separately.

DYNAMIC MEANING OF EXPRESSIONS

The dynamic semantic meaning of expressions is typically a function returning an expression computation. The dynamic semantic functions for program phrases that can be attributed expression phrase types are given below.

- $[\![\mathit{val}\,[\tau]]\!] \;:\; e : \mathbf{Ent} \to \mathsf{V}([\![\tau]\!]_{dat})$
- $[\![\mathit{lvalue}\,[m]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{G}([\![m]\!]_{mem})$
- $[\![\mathit{exp}\,[v]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{G}([\![v]\!]_{val})$

The meaning of a constant expression of type $\mathit{val}\,[\tau]$ is a function from the static type environment to an element of domain $\mathsf{V}([\![\tau]\!]_{dat})$. It cannot access any of the dynamic environments, nor the program state. The meanings of non-constant expressions also take the dynamic type environment and the function code enviroment as parameters and return interleaved computations that can affect the program state.

DYNAMIC MEANING OF ARGUMENTS

- $[\![\mathit{arg}\,[p]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{G}([\![p]\!]_{Prot})$

The dynamic semantic function for the non-terminal symbol *arguments*, with respect to the phrase type $\mathit{arg}\,[p]$, is a function taking as parameters the static and dynamic type environments and the function code environment and returning an interleaved computation of the arguments' dynamic values. The program state can be affected.

CONVERSION AS IF BY ASSIGNMENT

- $\mathcal{A}[\![\mathit{exp}\,[\tau']]\!]_{\mathit{exp}\,[\tau]} \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{G}([\![\tau]\!]_{dat})$

The alternative dynamic semantic function $\mathcal{A}$ corresponds to assignability typing judgements of the form $e \vdash E \gg \tau$. Its type is similar to the normal meaning for expressions, with respect to the phrase type $\mathit{exp}\,[\tau]$.

CONSTANTS

- $\mathcal{C}[\![\mathit{constant}]\!]_\tau \;:\; [\![\tau]\!]_{dat}$

This function specifies the dynamic meaning of constants of type $\tau$. Its complete definition is omitted in this thesis.

CONSTANT EXPRESSIONS

Provided that $e \vdash E : \mathit{val}\,[\tau]$ and $\mathit{isIntegral}\,(\tau)$, it is possible to define an alternative dynamic semantic meaning for constant integral expressions as follows:

- $\mathcal{IC}[\![\mathit{constant\text{-}expression}]\!] \;:\; \mathbf{Ent} \to \mathsf{E}(\mathbf{N})$

  $\mathcal{IC}[\![E]\!] \;=\; \lambda e.\, \mathit{lift}_{\mathsf{V}\to\mathsf{E}}\,([\![E]\!]_{\mathit{val}\,[\tau]}\, e)$

This meaning has been used in Part II and Part III of this thesis.

## 12.2 Primary expressions

$$\frac{\ldots}{e \vdash f : val\,[\tau]}$$ *(Rules: E1, E2 and E3)* FLOATING CONSTANTS

▲ $[\![f]\!]_{val\,[\tau]} = \lambda e.\ unit\ \mathcal{C}[\![f]\!]_\tau$

The dynamic meaning of floating constants is simply their value.

$$\frac{\ldots}{e \vdash n : val\,[\tau]}$$ *(Rules: E4, E5, E6, E7 and E8)* INTEGER CONSTANTS

▲ $[\![n]\!]_{val\,[\tau]} = \lambda e.\ unit\ \mathcal{C}[\![n]\!]_\tau$

Similarly with integer constants.

$$\frac{\ldots}{e \vdash c : val\,[\tau]}$$ *(Rules: E9 and E10)* CHARACTER CONSTANTS

▲ $[\![c]\!]_{val\,[\tau]} = \lambda e.\ unit\ \mathcal{C}[\![c]\!]_\tau$

Similarly with character constants.

*(E11)* $[\![s]\!]_{lvalue\,[array\,[obj\,[char,noqual],n]]} = \lambda e.\ \lambda\rho.\ \lambda\xi.$ STRING LITERALS
$\quad$ **let** $a_f = fromAddr_{array\,[obj\,[char,noqual],n]}\ \langle newObject_{array\,[obj\,[char,noqual],n]}, 0\rangle$
$\quad$ **in** $lift_{\mathsf{C}\to\mathsf{G}}\ (storeStringLit_{obj\,[char,noqual]}\ s\ a_f\ n)\ *\ (\lambda u.\ unit\ a_f)$

*(E12)* $[\![s]\!]_{lvalue\,[array\,[obj\,[wchar\_t,noqual],n]]} = \lambda e.\ \lambda\rho.\ \lambda\xi.$
$\quad$ **let** $a_f = fromAddr_{array\,[obj\,[wchar\_t,noqual],n]}\ \langle newObject_{array\,[obj\,[wchar\_t,noqual],n]}, 0\rangle$
$\quad$ **in** $lift_{\mathsf{C}\to\mathsf{G}}\ (storeWideStringLit_{obj\,[wchar\_t,noqual]}\ s\ a_f\ n)\ *\ (\lambda u.\ unit\ a_f)$

The dynamic meaning of string literals is a computation resulting in their address in memory. The computation affects the program state by storing the string literals in memory.

*(E13)* $[\![I]\!]_{lvalue\,[\alpha]} = \lambda e.\ \lambda\rho.\ \lambda\xi.\ lift_{\mathsf{V}\to\mathsf{G}}\ (lookup\ e\ \rho\ I)$ IDENTIFIERS

*(E14)* $[\![I]\!]_{exp\,[f]} = \lambda e.\ \lambda\rho.\ \lambda\xi.\ lift_{\mathsf{V}\to\mathsf{G}}\ (lookup\ e\ \rho\ I)$

*(E15)* $[\![I]\!]_{val\,[int]} = \lambda e.\ unit\ n$

The dynamic meaning of identifiers designating objects or functions is their associated value in the dynamic type environment. In the case of enumeration constants, it is just their value.

## 12.3 Postfix operators

*(E16)* $[\![E_1[E_2]]\!]_{lvalue\,[\alpha]} = [\![*(E_1+E_2)]\!]_{lvalue\,[\alpha]}$ ARRAY SUBSCRIPTS

The dynamic meaning of array subscripts is indirectly specified, as suggested in the standard, by means of the indirection operator and pointer arithmetic.

FUNCTION CALLS

*(E17)* $[\![E\,(\mathit{arguments})\,]\!]_{exp\,[\tau]} = \lambda e.\,\lambda\rho.\,\lambda\xi.$
$[\![E]\!]_{exp\,[ptr\,[func\,[\tau,p]]]}\ e\ \rho\ \xi \bowtie [\![\mathit{arguments}]\!]_{arg\,[p]}\ e\ \rho\ \xi\ *\ (\lambda\langle m_f, d_p\rangle.$
**case** $m_f$ **of**
$\quad \mathit{inl}\ d_f \Rightarrow \mathit{seqpt}\ *\ (\lambda u.$
$\quad\quad \mathit{lift}_{V\to G}\ \xi[d_f]\ *\ (\lambda\langle f, b_f\rangle.$
$\quad\quad \mathit{isCompatible}(f, \mathit{func}\,[\tau,p]) \to b_f\ d_p\ ,\ \mathit{error}))$
$\quad$ **otherwise** $\Rightarrow \mathit{error})$

The function pointer and the actual arguments are first evaluated and their interleaving is allowed. If the function pointer is not null, a sequence point is generated and the function is called with the given actual arguments. An error occurs if the type of the pointed function is not compatible with the type of the function pointer that was used to access it.

*(R1)* $[\![\epsilon]\!]_{arg\,[p_\circ]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\mathit{unit}\ \top$

*(R2)* $[\![\epsilon]\!]_{arg\,[p]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\mathit{unit}\ \top$

*(R3)* $[\![E\ ,\ \mathit{arguments}]\!]_{arg\,[p']} = \lambda e.\,\lambda\rho.\,\lambda\xi.$
$\mathcal{A}[\![E]\!]_{exp\,[\tau]}\ e\ \rho\ \xi \bowtie [\![\mathit{arguments}]\!]_{arg\,[p]}\ e\ \rho\ \xi\ *\ (\lambda\langle d, d_p\rangle.$
$\mathit{unit}\ (\mathit{shift}\ (-1)\ d_p)\{1 \mapsto d\})$

*(R4)* $[\![E\ ,\ \mathit{arguments}]\!]_{arg\,[p']} = \lambda e.\,\lambda\rho.\,\lambda\xi.$
$([\![E]\!]_{exp\,[\tau]}\ e\ \rho\ \xi\ *\ (\mathit{unit} \circ \mathit{cast}_{\tau\to\tau'})) \bowtie [\![\mathit{arguments}]\!]_{arg\,[p]}\ e\ \rho\ \xi\ *\ (\lambda\langle d, d_p\rangle.$
$\mathit{unit}\ (\mathit{shift}\ (-1)\ d_p)\{1 \mapsto d\})$

The dynamic meaning of actual arguments performs their interleaved evaluation. If the type of the corresponding formal argument is known, then an actual parameter is converted as if by assignment to the type of the formal argument. Otherwise, in the case of parameters passed in the ellipsis part of a prototype, the default argument promotions are performed.

MEMBER OPERATORS

*(E18)* $[\![E\,.\,I]\!]_{lvalue\,[m']} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{lvalue\,[obj\,[struct\,[t,\pi],q]]}\ e\ \rho\ \xi\ *\ (\lambda a.\,\mathit{unit}\ (\mathit{structMember}_\pi\ a\ I))$

*(E19)* $[\![E\,.\,I]\!]_{exp\,[\tau]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{exp\,[struct\,[t,\pi]]}\ e\ \rho\ \xi\ *\ (\lambda d_{mf}.\,\mathit{unit}\ (d_{mf}\ I))$

*(E20)* $[\![E\,.\,I]\!]_{lvalue\,[m']} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{lvalue\,[obj\,[union\,[t,\pi],q]]}\ e\ \rho\ \xi\ *\ (\lambda a.\,\mathit{unit}\ (\mathit{unionMember}_\pi\ a\ I))$

*(E21)* $[\![E\,.\,I]\!]_{exp\,[\tau]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{exp\,[struct\,[t,\pi]]}\ e\ \rho\ \xi\ *\ (\lambda d_{mf}.\,\mathit{unit}\ (d_{mf}\ I))$

The dynamic meaning of the dot operator is relatively simple. The address of the member or its stored value can easily be found.

*(E22)* $[\![E\text{->}I]\!]_{lvalue\,[m']} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{exp\,[ptr\,[obj\,[struct\,[t,\pi],q]]]}\ e\ \rho\ \xi\ *\ (\lambda m_a.$
**case** $m_a$ **of**
$\quad \mathit{inl}\ a \Rightarrow \mathit{unit}\ (\mathit{structMember}_\pi\ a\ I)$
$\quad$ **otherwise** $\Rightarrow \mathit{error})$

*(E23)* $[\![E\text{->}I]\!]_{lvalue\,[m']} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{exp\,[ptr\,[obj\,[union\,[t,\pi],q]]]}\ e\ \rho\ \xi\ *\ (\lambda m_a.$
**case** $m_a$ **of**
$\quad \mathit{inl}\ a \Rightarrow \mathit{unit}\ (\mathit{unionMember}_\pi\ a\ I)$
$\quad$ **otherwise** $\Rightarrow \mathit{error})$

The dynamic meaning of the arrow operator is a little more complicated. In case of a null pointer, an error must be generated.

## 12.4 Unary operators

$$\frac{e \vdash E : lvalue\,[m] \quad \ldots \quad \tau := datify\ m}{e \vdash op\ E : exp\,[\tau]} \qquad \textit{(Rules: E24, E25, E26 and E27)}$$

UNARY ASSIGNMENTS

▲ $[\![op\ E]\!]_{exp\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.$
$[\![E]\!]_{lvalue\,[m]}\ e\ \rho\ \xi\ *\ (\lambda d_m.$
$getValue_{m \mapsto \tau}\ d_m\ *\ (\lambda d.$
$\textbf{let}\ \langle d', v\rangle = [\![op]\!]_\tau\ d\ \textbf{in}\ putValue_{\tau \mapsto m}\ d_m\ d'\ *\ (\lambda u.\ unit\ v)))$

The l-value is first evaluated. The stored value is retrieved from memory and the required operation is performed. Then the updated value is stored in memory and the result is returned.

The dynamic semantic function for unary assignment operators is defined as follows, using the semantics of addition and subtraction. It is parameterized by the data type of the operand.

▶ $[\![\textit{unary-assignment}]\!]_\tau : [\![\tau]\!]_{dat} \to [\![\tau]\!]_{dat} \times [\![\tau]\!]_{dat}$

$[\![\texttt{++}\ (\textit{prefix})]\!]_\tau = \lambda d.\ \textbf{let}\ d' = [\![\texttt{+}]\!]_{\tau,\tau,\tau}\ \langle d, \mathcal{C}[\![\texttt{1}]\!]_\tau\rangle\ \textbf{in}\ \langle d', d'\rangle$
$[\![\texttt{++}\ (\textit{postfix})]\!]_\tau = \lambda d.\ \textbf{let}\ d' = [\![\texttt{+}]\!]_{\tau,\tau,\tau}\ \langle d, \mathcal{C}[\![\texttt{1}]\!]_\tau\rangle\ \textbf{in}\ \langle d', d\rangle$
$[\![\texttt{--}\ (\textit{prefix})]\!]_\tau = \lambda d.\ \textbf{let}\ d' = [\![\texttt{-}]\!]_{\tau,\tau,\tau}\ \langle d, \mathcal{C}[\![\texttt{1}]\!]_\tau\rangle\ \textbf{in}\ \langle d', d'\rangle$
$[\![\texttt{--}\ (\textit{postfix})]\!]_\tau = \lambda d.\ \textbf{let}\ d' = [\![\texttt{-}]\!]_{\tau,\tau,\tau}\ \langle d, \mathcal{C}[\![\texttt{1}]\!]_\tau\rangle\ \textbf{in}\ \langle d', d\rangle$

The parameter of these functions is the initial value and the result is the pair of the updated value and the returned value. In the case of postfix unary assignment operators, the returned value is the initial value. In the case of prefix unary assignment operators, the returned value is the updated value.

ADDRESS OPERATOR

*(E28)* $[\![\&E]\!]_{exp\,[ptr\,[\alpha]]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, [\![E]\!]_{lvalue\,[\alpha]}\ e\ \rho\ \xi\ *\ (unit \circ inl \circ toAddr_\alpha)$

*(E29)* $[\![\&E]\!]_{exp\,[ptr\,[f]]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, [\![E]\!]_{exp\,[f]}\ e\ \rho\ \xi\ *\ (unit \circ inl)$

The dynamic semantics of the address operator is simple. A null pointer can never be obtained.

INDIRECTION OPERATOR

*(E30)* $[\![{}^*E]\!]_{lvalue\,[\alpha]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, [\![E]\!]_{exp\,[ptr\,[\alpha]]}\ e\ \rho\ \xi\ *\ (\lambda m_a.$
$\textbf{case}\ m_a\ \textbf{of}$
$\quad inl\ a \Rightarrow unit\ a$
$\quad \textbf{otherwise} \Rightarrow error)$

*(E31)* $[\![{}^*E]\!]_{exp\,[f]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, [\![E]\!]_{exp\,[ptr\,[f]]}\ e\ \rho\ \xi\ *\ (\lambda m_f.$
$\textbf{case}\ m_a\ \textbf{of}$
$\quad inl\ d_f \Rightarrow unit\ d_f$
$\quad \textbf{otherwise} \Rightarrow error)$

In the dynamic meaning of the indirection operator, a check for null pointers is performed. An error occurs if a null pointer is dereferenced.

UNARY OPERATORS

The dynamic semantic function for unary operators is defined below, parameterized by the data type of the operand. The complete definition of this function is omitted in this thesis.

▶ $[\![\textit{unary-operator}]\!]_\tau : [\![\tau]\!]_{dat} \to [\![\tau]\!]_{dat}$

The following equations define the dynamic meanings of expressions using arithmetic unary operators.

$$\frac{e \vdash E : val\,[\tau] \quad \ldots}{e \vdash op\ E : val\,[\tau']} \qquad \textit{(Rules: E32, E34 and E36)}$$

▲ $[\![op\ E]\!]_{val\,[\tau']} = \lambda e.\, [\![E]\!]_{val\,[\tau]}\, e\ *\ (unit \circ [\![op]\!]_{\tau'} \circ cast_{\tau \mapsto \tau'})$

$$\frac{e \vdash E : exp\,[\tau] \quad \ldots}{e \vdash op\ E : exp\,[\tau']} \qquad \textit{(Rules: E33, E35 and E37)}$$

▲ $[\![op\ E]\!]_{exp\,[\tau']} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, [\![E]\!]_{exp\,[\tau]}\, e\, \rho\, \xi\ *\ (unit \circ [\![op]\!]_{\tau'} \circ cast_{\tau \mapsto \tau'})$

The dynamic meaning of expressions using the unary plus, minus or bitwise negation operators are simple.

*(E38)* $[\![\,!E]\!]_{\theta} = [\![E{==}0]\!]_{\theta}$

Logical negation is defined in terms of equality to zero, as suggested in the standard.

SIZEOF OPERATOR

*(E39)* $[\![\texttt{sizeof}\ E]\!]_{val\,[size\_t]} = \lambda e.\, unit\ sizeof(m)$
*(E41)* $[\![\texttt{sizeof}\ E]\!]_{val\,[size\_t]} = \lambda e.\, unit\ sizeof(\tau)$
*(E42)* $[\![\texttt{sizeof(}T\texttt{)}]\!]_{val\,[size\_t]} = \lambda e.\, unit\ sizeof(\alpha)$

The dynamic meaning of the *sizeof* operator is also very simple.

## 12.5 Cast operators

*(E43)* $[\![\texttt{(}T\texttt{)}\ E]\!]_{exp\,[void]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, [\![E]\!]_{exp\,[\tau]}\, e\, \rho\, \xi\ *\ (\lambda d.\, unit\ u)$

In the case of casting to *void*, the value of an expression is simply discarded.

*(E44)* $[\![\texttt{(}T\texttt{)}\ E]\!]_{val\,[\tau']} = \lambda e.\, [\![E]\!]_{val\,[\tau]}\, e\ *\ (unit \circ cast_{\tau \mapsto \tau'})$
*(E45)* $[\![\texttt{(}T\texttt{)}\ E]\!]_{exp\,[\tau']} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, [\![E]\!]_{exp\,[\tau]}\, e\, \rho\, \xi\ *\ (unit \circ cast_{\tau \mapsto \tau'})$

Otherwise, a type casting is performed.

## 12.6 Binary operators

MEANINGS OF BINARY OPERATORS

Three dynamic meanings for binary operators are defined. The primary meaning of a binary operator is a function taking two parameters of types $\tau_1$ and $\tau_2$ and returning a result of type $\tau$. All data types are given as parameters. The complete definition of this function is omitted in this thesis.

► $[\![\textit{binary-operator}]\!]_{\tau_1,\tau_2,\tau} : [\![\tau_1]\!]_{dat} \times [\![\tau_2]\!]_{dat} \to [\![\tau]\!]_{dat}$

The second alternative dynamic meaning for binary operators models their application on constant values. The parameters $\tau_1$ and $\tau_2$ specify the data types of the two operands before conversion. The parameters $\tau_1'$ and $\tau_2'$ specify the types after the necessary conversions have taken place. Finally, the parameter $\tau$ specifies the data type of the result.

▶ $\mathcal{V}[\![\textit{binary-operator}]\!]_{\tau_1,\tau_2,\tau_1',\tau_2',\tau} \;:\; \mathsf{V}([\![\tau_1]\!]_{dat}) \times \mathsf{V}([\![\tau_2]\!]_{dat}) \to \mathsf{V}([\![\tau]\!]_{dat})$

$$\begin{array}{l}\mathcal{V}[\![op]\!]_{\tau_1,\tau_2,\tau_1',\tau_2',\tau} \;=\; \lambda\langle v_1, v_2\rangle.\\ \quad v_1 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_1 \mapsto \tau_1'}) \;*\; (\lambda d_1.\\ \quad v_2 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_2 \mapsto \tau_2'}) \;*\; (\lambda d_1.\\ \quad \mathit{unit}\ ([\![op]\!]_{\tau_1',\tau_2',\tau}\ \langle d_1, d_2\rangle))))\end{array}$$

The definition of this meaning is simple and uniform for all binary operators.

The third alternative dynamic meaning for binary operators models their application on non-constant values, i.e. expression computations. The purpose of the parameters was explained above.

▶ $\mathcal{E}[\![\textit{binary-operator}]\!]_{\tau_1,\tau_2,\tau_1',\tau_2',\tau} \;:\; \mathsf{G}([\![\tau_1]\!]_{dat}) \times \mathsf{G}([\![\tau_2]\!]_{dat}) \to \mathsf{G}([\![\tau]\!]_{dat})$

Notice that the definition of this meaning is not uniform for all binary operators.

$$\begin{array}{l}\mathcal{E}[\![\&\&]\!]_{\tau_1,\tau_2,\tau_1',\tau_2',\tau} \;=\; \lambda\langle g_1, g_2\rangle.\\ \quad g_1 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_1 \mapsto \tau_1'}) \;*\; (\lambda d_1.\ \mathit{seqpt} \;*\; (\lambda u.\ \mathit{checkBoolean}_{\tau_1'}(d_1) \to\\ \qquad g_2 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_2 \mapsto \tau_2'}) \;*\; (\lambda d_2.\ \mathit{checkBoolean}_{\tau_2'}(d_2) \to \mathit{unit}\ \mathcal{C}[\![\mathtt{1}]\!]_\tau\ ,\ \mathit{unit}\ \mathcal{C}[\![\mathtt{0}]\!]_\tau)\ ,\\ \qquad \mathit{unit}\ \mathcal{C}[\![\mathtt{0}]\!]_\tau))\\ \mathcal{E}[\![||]\!]_{\tau_1,\tau_2,\tau_1',\tau_2',\tau} \;=\; \lambda\langle g_1, g_2\rangle.\\ \quad g_1 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_1 \mapsto \tau_1'}) \;*\; (\lambda d_1.\ \mathit{seqpt} \;*\; (\lambda u.\ \mathit{checkBoolean}_{\tau_1'}(d_1) \to\\ \qquad \mathit{unit}\ \mathcal{C}[\![\mathtt{1}]\!]_\tau\ ,\\ \qquad g_2 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_2 \mapsto \tau_2'}) \;*\; (\lambda d_2.\ \mathit{checkBoolean}_{\tau_2'}(d_2) \to \mathit{unit}\ \mathcal{C}[\![\mathtt{1}]\!]_\tau\ ,\ \mathit{unit}\ \mathcal{C}[\![\mathtt{0}]\!]_\tau)))\end{array}$$

In the case of the logical binary operators, the left operand is evaluated first and a sequence point is generated. If the final result has not been determined, the second operand is also evaluted. The result of the expression is one of the integer values 1 or 0.

$$\begin{array}{l}\mathcal{E}[\![,]\!]_{\tau_1,\tau_2,\tau_1',\tau_2',\tau} \;=\; \lambda\langle g_1, g_2\rangle.\\ \quad g_1 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_1 \mapsto \tau_1'}) \;*\; (\lambda d_1.\ \mathit{seqpt} \;*\; (\lambda u.\\ \quad g_2 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_2 \mapsto \tau_2'})))\end{array}$$

In the case of the comma operator, the left operand is first evaluated, its value is discarded and a sequence point is generated. Then, the second operand is evaluated and its result is the result of the whole expression.

$$\begin{array}{l}\mathcal{E}[\![op]\!]_{\tau_1,\tau_2,\tau_1',\tau_2',\tau} \;=\; \lambda\langle g_1, g_2\rangle.\\ \quad \mathbf{let}\ g_1' = g_1 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_1 \mapsto \tau_1'})\\ \qquad g_2' = g_2 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_2 \mapsto \tau_2'})\\ \quad \mathbf{in}\ g_1' \bowtie g_2' \;*\; (\mathit{unit} \circ [\![op]\!]_{\tau_1',\tau_2',\tau})\end{array}$$

Finally, in the case of all other operators, the evaluation of the two operands is interleaved.

DYNAMIC EQUATIONS

The following equations define the dynamic semantics that correspond to various typing rules for expressions with binary operators.

$$\frac{e \vdash E_1 \;:\; \mathit{val}\,[\tau_1] \qquad e \vdash E_2 \;:\; \mathit{val}\,[\tau_2] \qquad \ldots}{e \vdash E_1\ \mathit{op}\ E_2 \;:\; \mathit{val}\,[\tau']}$$

*(Rules: E46, E48, E50, E52, E56, E90, E92 and E94)*

▲ $[\![E_1\ \mathit{op}\ E_2]\!]_{\mathit{val}\,[\tau']} \;=\; \lambda e.\ \mathcal{V}[\![op]\!]_{\tau_1,\tau_2,\tau',\tau',\tau'}\ \langle [\![E_1]\!]_{\mathit{val}\,[\tau_1]}\ e, [\![E_2]\!]_{\mathit{val}\,[\tau_2]}\ e\rangle$

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad e \vdash E_2 : val\,[\tau_2] \quad \ldots \quad \tau_1' := intPromote\ \tau_1 \quad \tau_2' := intPromote\ \tau_2}{e \vdash E_1\ op\ E_2 : val\,[\tau_1']}$$

*(Rules: E60 and E62)*

▲ $[\![E_1\ op\ E_2]\!]_{val\,[\tau_1']} = \lambda e.\ \mathcal{V}[\![op]\!]_{\tau_1,\tau_2,\tau_1',\tau_2',\tau_1'}\ \langle [\![E_1]\!]_{val\,[\tau_1]}\ e, [\![E_2]\!]_{val\,[\tau_2]}\ e\rangle$

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad e \vdash E_2 : val\,[\tau_2] \quad \ldots \quad \tau' := arithConv\,\langle \tau_1, \tau_2\rangle}{e \vdash E_1\ op\ E_2 : val\,[int]}$$

*(Rules: E64, E67, E70, E73, E76 and E83)*

▲ $[\![E_1\ op\ E_2]\!]_{val\,[int]} = \lambda e.\ \mathcal{V}[\![op]\!]_{\tau_1,\tau_2,\tau',\tau',int}\ \langle [\![E_1]\!]_{val\,[\tau_1]}\ e, [\![E_2]\!]_{val\,[\tau_2]}\ e\rangle$

$$\frac{e \vdash E_1 : val\,[\tau_1] \quad e \vdash E_2 : val\,[\tau_2] \quad \ldots}{e \vdash E_1\ op\ E_2 : val\,[int]}$$

*(Rules: E96 and E99)*

▲ $[\![E_1\ op\ E_2]\!]_{val\,[int]} = \lambda e.\ \mathcal{V}[\![op]\!]_{\tau_1,\tau_2,\tau_1,\tau_2,int}\ \langle [\![E_1]\!]_{val\,[\tau_1]}\ e, [\![E_2]\!]_{val\,[\tau_2]}\ e\rangle$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad \ldots}{e \vdash E_1\ op\ E_2 : exp\,[\tau']}$$

*(Rules: E47, E49, E51, E53, E57, E91, E93 and E95)*

▲ $[\![E_1\ op\ E_2]\!]_{exp\,[\tau']} = \lambda e.\ \lambda \rho.\ \lambda \xi.\ \mathcal{E}[\![op]\!]_{\tau_1,\tau_2,\tau',\tau',\tau'}\ \langle [\![E_1]\!]_{exp\,[\tau_1]}\ e\ \rho\ \xi, [\![E_2]\!]_{exp\,[\tau_2]}\ e\ \rho\ \xi\rangle$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad \ldots \quad \tau_1' := intPromote\ \tau_1 \quad \tau_2' := intPromote\ \tau_2}{e \vdash E_1\ op\ E_2 : exp\,[\tau_1']}$$

*(Rules: E61 and E63)*

▲ $[\![E_1\ op\ E_2]\!]_{exp\,[\tau_1']} = \lambda e.\ \lambda \rho.\ \lambda \xi.\ \mathcal{E}[\![op]\!]_{\tau_1,\tau_2,\tau_1',\tau_2',\tau_1'}\ \langle [\![E_1]\!]_{exp\,[\tau_1]}\ e\ \rho\ \xi, [\![E_2]\!]_{exp\,[\tau_2]}\ e\ \rho\ \xi\rangle$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad \ldots \quad \tau' := arithConv\,\langle \tau_1, \tau_2\rangle}{e \vdash E_1\ op\ E_2 : exp\,[int]}$$

*(Rules: E65, E68, E71, E74, E77 and E84)*

▲ $[\![E_1\ op\ E_2]\!]_{exp\,[int]} = \lambda e.\ \lambda \rho.\ \lambda \xi.\ \mathcal{E}[\![op]\!]_{\tau_1,\tau_2,\tau',\tau',int}\ \langle [\![E_1]\!]_{exp\,[\tau_1]}\ e\ \rho\ \xi, [\![E_2]\!]_{exp\,[\tau_2]}\ e\ \rho\ \xi\rangle$

$$\frac{e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad \ldots}{e \vdash E_1\ op\ E_2 : exp\,[\tau']}$$

*(Rules: E54, E55, E58, E59, E66, E69, E72, E75, E78, E79, E80, E85, E86, E87, E98, E101 and E115)*

▲ $[\![E_1\ op\ E_2]\!]_{exp\,[\tau']} = \lambda e.\ \lambda \rho.\ \lambda \xi.\ \mathcal{E}[\![op]\!]_{\tau_1,\tau_2,\tau_1,\tau_2,\tau'}\ \langle [\![E_1]\!]_{exp\,[\tau_1]}\ e\ \rho\ \xi, [\![E_2]\!]_{exp\,[\tau_2]}\ e\ \rho\ \xi\rangle$

In the case of arithmetic or simple pointer operations, the only thing to determine from the typing rules above are the parameters for the various data types.

*(E97)* $[\![E_1 \&\& E_2]\!]_{val\,[int]} = \lambda e.\ [\![E_1]\!]_{val\,[\tau_1]}\ e\ *\ (\lambda d_1.\ \neg checkBoolean_{\tau_1}(d_1) \rightarrow unit\ \mathcal{C}[\![0]\!]_\tau\ ,\ error)$

*(E100)* $[\![E_1 \,||\, E_2]\!]_{val\,[int]} = \lambda e.\ [\![E_1]\!]_{val\,[\tau_1]}\ e\ *\ (\lambda d_1.\ checkBoolean_{\tau_1}(d_1) \rightarrow unit\ \mathcal{C}[\![1]\!]_\tau\ ,\ error)$

These two equations define the short-circuit semantics of the logical boolean operators for constant values.

*(E81)* $[\![E_1 == E_2]\!]_{exp\,[int]} = \lambda e.\, \lambda \rho.\, \lambda \xi.$
$[\![E_1]\!]_{exp\,[ptr\,[\phi_1]]}\, e\, \rho\, \xi\, *\, (\lambda d.\, checkBoolean_{ptr\,[\phi_1]}(d) \to unit\, \mathcal{C}[\![\mathtt{0}]\!]_{int}\,,\ unit\, \mathcal{C}[\![\mathtt{1}]\!]_{int})$

*(E82)* $[\![E_1 == E_2]\!]_{exp\,[int]} = \lambda e.\, \lambda \rho.\, \lambda \xi.$
$[\![E_2]\!]_{exp\,[ptr\,[\phi_2]]}\, e\, \rho\, \xi\, *\, (\lambda d.\, checkBoolean_{ptr\,[\phi_2]}(d) \to unit\, \mathcal{C}[\![\mathtt{0}]\!]_{int}\,,\ unit\, \mathcal{C}[\![\mathtt{1}]\!]_{int})$

*(E88)* $[\![E_1 \mathtt{!=} E_2]\!]_{exp\,[int]} = \lambda e.\, \lambda \rho.\, \lambda \xi.$
$[\![E_1]\!]_{exp\,[ptr\,[\phi_1]]}\, e\, \rho\, \xi\, *\, (\lambda d.\, checkBoolean_{ptr\,[\phi_1]}(d) \to unit\, \mathcal{C}[\![\mathtt{1}]\!]_{int}\,,\ unit\, \mathcal{C}[\![\mathtt{0}]\!]_{int})$

*(E89)* $[\![E_1 \mathtt{!=} E_2]\!]_{exp\,[int]} = \lambda e.\, \lambda \rho.\, \lambda \xi.$
$[\![E_2]\!]_{exp\,[ptr\,[\phi_2]]}\, e\, \rho\, \xi\, *\, (\lambda d.\, checkBoolean_{ptr\,[\phi_2]}(d) \to unit\, \mathcal{C}[\![\mathtt{1}]\!]_{int}\,,\ unit\, \mathcal{C}[\![\mathtt{0}]\!]_{int})$

These four equations define the semantics of pointer comparison when one of the operands is the null pointer constant.

## 12.7 Conditional operator

The dynamic meaning of the conditional operator is a function taking as its first parameter the controlling expression and as its second parameter the pair of alternative expressions. All parameters are interleaved computations, and so is the result. It is parameterized by the data types of the controlling expression, the alternatives and the result.

▶ $cond_{\tau,\tau_1,\tau_2,\tau'} : \mathsf{G}([\![\tau]\!]_{dat}) \to \mathsf{G}([\![\tau_1]\!]_{dat}) \times \mathsf{G}([\![\tau_2]\!]_{dat}) \to \mathsf{G}([\![\tau']\!]_{dat})$

$cond_{\tau,\tau_1,\tau_2,\tau'} = \lambda g.\, \lambda \langle g_1, g_2 \rangle.$
$g\, *\, (\lambda d.\, seqpt\, *\, (\lambda u.\, checkBoolean_\tau(d) \to g_1\, *\, (unit \circ cast_{\tau_1 \mapsto \tau'})\,,\ g_2\, *\, (unit \circ cast_{\tau_2 \mapsto \tau'})))$

The controlling expression is first evaluated and a sequence point is generated. Then, depending on the value of the controlling expression, one of the alternatives is evaluated and the result is returned, after a cast to the appropriate type.

*(E102)* $[\![E ? E_1 : E_2]\!]_{val\,[\tau']} = \lambda e.$
$[\![E]\!]_{val\,[\tau]}\, e\, *\, (\lambda d.\, checkBoolean_\tau(d) \to [\![E_1]\!]_{val\,[\tau_1]}\, e\, *\, (unit \circ cast_{\tau_1 \mapsto \tau'})\,,\ error)$

*(E103)* $[\![E ? E_1 : E_2]\!]_{val\,[\tau']} = \lambda e.$
$[\![E]\!]_{val\,[\tau]}\, e\, *\, (\lambda d.\, \neg checkBoolean_\tau(d) \to [\![E_2]\!]_{val\,[\tau_2]}\, e\, *\, (unit \circ cast_{\tau_2 \mapsto \tau'})\,,\ error)$

These equations define the short-circuit semantics of the conditional operator in the case of constant values.

$$\frac{e \vdash E : exp\,[\tau] \quad e \vdash E_1 : exp\,[\tau_1] \quad e \vdash E_2 : exp\,[\tau_2] \quad \ldots}{e \vdash E ? E_1 : E_2 : exp\,[\tau']}$$

*(Rules: E104, E105, E106, E107, E108, E111 and E112)*

▲ $[\![E ? E_1 : E_2]\!]_{exp\,[\tau']} = \lambda e.\, \lambda \rho.\, \lambda \xi.$
$cond_{\tau,\tau_1,\tau_2,\tau'}\, ([\![E]\!]_{exp\,[\tau]}\, e\, \rho\, \xi)\, \langle [\![E_1]\!]_{exp\,[\tau_1]}\, e\, \rho\, \xi, [\![E_2]\!]_{exp\,[\tau_2]}\, e\, \rho\, \xi \rangle$

In case of all sorts of alternatives except null pointers, the only things to determine from the typing rules are the type parameters.

*(E109)* $[\![E ? E_1 : E_2]\!]_{exp\,[ptr\,[\phi_1]]} = \lambda e.\, \lambda \rho.\, \lambda \xi.$
$cond_{\tau,ptr\,[\phi_1],ptr\,[\phi_1],ptr\,[\phi_1]}\, ([\![E]\!]_{exp\,[\tau]}\, e\, \rho\, \xi)\, \langle [\![E_1]\!]_{exp\,[\tau_1]}\, e\, \rho\, \xi, unit\, (inr\, u) \rangle$

*(E110)* $[\![E ? E_1 : E_2]\!]_{exp\,[ptr\,[\phi_2]]} = \lambda e.\, \lambda \rho.\, \lambda \xi.$
$cond_{\tau,ptr\,[\phi_2],ptr\,[\phi_2],ptr\,[\phi_2]}\, ([\![E]\!]_{exp\,[\tau]}\, e\, \rho\, \xi)\, \langle unit\, (inr\, u), [\![E_2]\!]_{exp\,[\tau_2]}\, e\, \rho\, \xi \rangle$

When one of the alternatives is a null pointer, it is implicitly converted to the type of the other alternative.

## 12.8 Assignment operators

SIMPLE ASSIGNMENT

*(E113)* $[\![E_1{=}E_2]\!]_{exp\,[\tau]} \;=\; \lambda\, e.\, \lambda\, \rho.\, \lambda\, \xi.\, [\![=]\!]_{m,\tau}\, \langle [\![E_1]\!]_{lvalue\,[m]}\; e\; \rho\; \xi, \mathcal{A}[\![E_2]\!]_{exp\,[\tau]}\; e\; \rho\; \xi \rangle$

The dynamic semantics of simple assignment is straightforward. It is defined in terms of the meaning of the simple assignment operator. This meaning is a function, taking as parameters the two expressions corresponding to the l-value and the r-value. It is parameterized by the member type $m$ of the l-value and the data type $\tau$ of the r-value, which is the same as the type of the result. It is required that $\tau := \mathit{datify}\ m$.

▶ $[\![=]\!]_{m,\tau} \;:\; \mathsf{G}([\![m]\!]_{mem}) \times \mathsf{G}([\![\tau]\!]_{dat}) \to \mathsf{G}([\![\tau]\!]_{dat})$

$$
\begin{aligned}
[\![=]\!]_{m,\tau} \;=\; & \lambda\, \langle g_1, g_2 \rangle. \\
& g_1 \bowtie g_2 \;*\; (\lambda\, \langle d_m, d \rangle. \\
& \mathit{putValue}_{\tau \mapsto m}\; d_m\; d \;*\; (\lambda\, d'. \\
& \mathit{unit}\; d))
\end{aligned}
$$

The l-value and the r-value are evaluated, allowing interleaving. Then, the evaluated value is stored in the evaluated location. The result is the stored value.

COMPOSITE ASSIGNMENT

The definition of composite assignment is probably the most complicated part of the dynamic semantics of C expressions. The reason is that the left side of the assignment must only be evaluated once. For this reason, typing rule E114 cannot be used directly. A typing derivation with more levels must be used instead. This derivation has the general form:

$$
\dfrac{\dfrac{e \vdash E_1 : \mathit{lvalue}\,[m] \quad \ldots}{e \vdash E_1 : \mathit{exp}\,[\tau_1]} \qquad \dfrac{e \vdash E_2 : \mathit{exp}\,[\tau_2] \quad \ldots}{\dfrac{e \vdash E_1\ \mathit{op}\ E_2 : \mathit{exp}\,[\tau']}{e \vdash E_1\ \mathit{op}\ E_2 \gg \tau} \quad \ldots}}{\dfrac{e \vdash E_1 : \mathit{lvalue}\,[m] \quad \ldots}{\dfrac{e \vdash E_1 = E_1\ \mathit{op}\ E_2 : \mathit{exp}\,[\tau]}{e \vdash E_1\ \mathit{op}{=}\ E_2 : \mathit{exp}\,[\tau]}}}
$$

where:

- The implicit coercion rule at the fifth (higher) level is one of C1 or C3;
- The arithmetic operation rule at the fourth level is one of E47, E49, E51, E53, E54, E57, E58, E61, E63, E91, E93 or E95;
- The assignability rule at the third level is one of A1, A2, A3, A4 or A5;
- The rule at the second level is E113; and
- The rule at the first level is E114.

The dynamic semantic equation that corresponds to this derivation is the following:

*(E114)* $[\![E_1\ \mathit{op}{=}\ E_2]\!]_{exp\,[\tau]} \;=\; \lambda\, e.\, \lambda\, \rho.\, \lambda\, \xi.\, [\![\mathit{op}{=}]\!]_{m,\tau,\tau_2,\tau',\tau',\tau'}\; \langle [\![E_1]\!]_{lvalue\,[m]}\; e\; \rho\; \xi, [\![E_2]\!]_{exp\,[\tau_2]}\; e\; \rho\; \xi \rangle$

MEANING OF BINARY ASSIGNMENT

The dynamic semantic meaning of composite binary assignment operators is a function similar to the one for the simple assignment operator. However, the number of parameters is larger.

▶ $[\![\textit{binary-assignment}]\!]_{m,\tau_1,\tau_2,\tau_1',\tau_2',\tau} \;:\; \mathsf{G}([\![m]\!]_{mem}) \times \mathsf{G}([\![\tau_2]\!]_{dat}) \to \mathsf{G}([\![\tau]\!]_{dat})$

$$
\begin{array}{l}
[\![op{=}]\!]_{m,\tau_1,\tau_2,\tau_1',\tau_2',\tau} \;=\; \lambda\langle g_1, g_2\rangle.\\
\quad \textbf{let}\ g_1' = g_1 \;*\; (\lambda d_m.\\
\qquad\qquad \mathit{getValue}_{m\mapsto\tau_1}\ d_m \;*\; (\lambda d.\\
\qquad\qquad \mathit{unit}\ \langle d_m, \mathit{cast}_{\tau_1\mapsto\tau_1'}\ d\rangle))\\
\qquad g_2' = g_2 \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau_2\mapsto\tau_2'})\\
\quad \textbf{in}\ \ g_1' \bowtie g_2' \;*\; (\lambda\langle\langle d_m, d_1\rangle, d_2\rangle.\\
\qquad \textbf{let}\ d = \mathit{cast}_{\tau\mapsto\tau_1}\ ([\![op]\!]_{\tau_1',\tau_2',\tau}\ \langle d_1, d_2\rangle)\\
\qquad \textbf{in}\ \ \mathit{putValue}_{\tau_1\mapsto m}\ d_m\ d \;*\; (\lambda d'.\ \mathit{unit}\ d))
\end{array}
$$

The l-value expression, the contents stored there and the r-value expression are evaluated allowing interleaving. Then, the result is calculated and stored in memory.

CONVERSION AS IF BY ASSIGNMENT

An alternative dynamic semantic meaning for expressions of type $\mathit{exp}\ [\tau']$ is defined, corresponding to assignability typing rules.

$$\frac{e \vdash E : \mathit{exp}\,[\tau] \quad \ldots}{e \vdash E \gg \tau'} \qquad \textit{(Rules: A1, A2, A3, A4 and A5)}$$

▲ $\mathcal{A}[\![E]\!]_{\mathit{exp}\,[\tau']} \;=\; \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{\mathit{exp}\,[\tau]}\ e\ \rho\ \xi \;*\; (\mathit{unit} \circ \mathit{cast}_{\tau\mapsto\tau'})$

In all cases except null pointer constants, a simple type casting is necessary to convert the expression's value.

*(A6)* $\mathcal{A}[\![E]\!]_{\mathit{exp}\,[\mathit{ptr}\,[\phi']]} \;=\; \lambda e.\,\lambda\rho.\,\lambda\xi.\,\mathit{unit}\ (\mathit{inr}\ u)$

The case of null pointer constants is even simpler.

## 12.9 Implicit coercions

*(C1)* $[\![E]\!]_{\mathit{exp}\,[\tau]} \;=\; \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{\mathit{lvalue}\,[\mathit{obj}\,[\tau,q]]}\ e\ \rho\ \xi \;*\; \mathit{getValue}_{\mathit{obj}\,[\tau,q]\mapsto\tau}$

*(C3)* $[\![E]\!]_{\mathit{exp}\,[\tau]} \;=\; \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{\mathit{lvalue}\,[\mathit{bitfield}\,[\beta,q,n]]}\ e\ \rho\ \xi \;*\; \mathit{getValue}_{\mathit{bitfield}\,[\beta,q,n]\mapsto\tau}$

Implicit coercion from a given object or bit-field designator requires the reading of the contents of the given object or bit-field.

*(C2)* $[\![E]\!]_{\mathit{exp}\,[\mathit{ptr}\,[\alpha]]} \;=\; \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{\mathit{lvalue}\,[\mathit{array}\,[\alpha,n]]}\ e\ \rho\ \xi \;*\; (\mathit{unit} \circ \mathit{inl} \circ \mathit{toAddr}_{\mathit{array}\,[\alpha,n]})$

*(C4)* $[\![E]\!]_{\mathit{exp}\,[\mathit{ptr}\,[f]]} \;=\; \lambda e.\,\lambda\rho.\,\lambda\xi.\,[\![E]\!]_{\mathit{exp}\,[f]}\ e\ \rho\ \xi \;*\; (\mathit{unit} \circ \mathit{inl})$

Implicit coercion from arrays or functions to pointers is simple. It can never produce a null pointer.

*(C5)* $[\![E]\!]_{\mathit{exp}\,[\tau]} \;=\; \lambda e.\,\lambda\rho.\,\lambda\xi.\,\mathit{lift}_{\mathsf{V}\to\mathsf{G}}\ ([\![E]\!]_{\mathit{val}\,[\tau]}\ e)$

Finally, implicit coercion from a constant value to a non constant value requires a simple lifting between monads $\mathsf{V}$ and $\mathsf{G}$.

# Chapter 13

# Dynamic semantics of declarations

CHAPTER OVERVIEW

This chapter defines the dynamic semantics of C declarations, including external declarations and translation units. A large number of alternative dynamic meanings are given to declarations, representing different aspects of their execution behaviour. Section 13.1 presents the semantics of external declarations and Section 13.2 the semantics of declarations.

## 13.1 External declarations

TRANSLATION UNITS

▶ $[\![ tunit ]\!] \;:\; \mathsf{C}([\![ int ]\!]_{dat})$

*(X1)*
$$
\begin{aligned}
&[\![ \textit{external-declaration-list} ]\!]_{tunit} \;=\\
&\quad lift_{\mathsf{E}\to\mathsf{C}}\,(rec\;\{\!|\textit{external-declaration-list}|\!\}\;e_\circ)\;*\;(\lambda e.\\
&\quad [\![ \textit{external-declaration-list} ]\!]_{xdecl}\;e\;(e \uparrow \top)\;*\;(\lambda \rho.\\
&\quad lift_{\mathsf{V}\to\mathsf{C}}\,(mfix_{\mathsf{V}}\,(\mathcal{C}[\![ \textit{external-declaration-list} ]\!]\;e\;\rho))\;*\;(\lambda \xi.\\
&\quad \mathcal{I}[\![ \textit{external-declaration-list} ]\!]\;e\;\rho\;\xi\;*\;(\lambda u.\\
&\quad lift_{\mathsf{E}\to\mathsf{G}}\;e[\text{“}main\text{”}\;ide]\;*\;(\lambda \delta.\;\mathbf{case}\;\delta\;\mathbf{of}\\
&\qquad normal\;[func\,[int, p_\circ]] \Rightarrow lift_{\mathsf{V}\to\mathsf{C}}\,(lookup\;e\;\rho\;\text{“}main\text{”})\;*\;(\lambda d_m.\\
&\qquad\qquad\qquad lift_{\mathsf{V}\to\mathsf{C}}\,(\xi[d_m])\;*\;(\lambda \langle f_m, b_m\rangle.\\
&\qquad\qquad\qquad isCompatible(f_m, func\,[int, p_\circ]) \to lift_{\mathsf{G}\to\mathsf{C}}\,(b_m\;\top),\;error))\\
&\qquad \mathbf{otherwise} \qquad\quad \Rightarrow error)))))
\end{aligned}
$$

The dynamic meaning of translation units is a computation resulting in the value of type *int* that is returned from function *main*. The static type environment is first calculated and the objects of the outermost scope are created and allocated. Following that, the function code environment is calculated as the monadic least fixed point of the $\mathcal{C}$-meaning for all external declarations. The fixed point is necessary so that recursive function calls are allowed. Following that, objects of the outermost scope are initialized and function *main* is called. It is assumed that *main* has the type “`int main ();`” and for this reason no actual parameters are passed. An error occurs if no *main* function has been defined or if its type is not compatible with “`int main ();`”.

EXTERNAL DECLARATIONS

▶ $[\![ xdecl ]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![ e ]\!]_{Ent} \to \mathsf{C}([\![ e ]\!]_{Ent})$

*(X2)*
$$
\begin{aligned}
&[\![ \textit{external-declaration external-declaration-list} ]\!]_{xdecl} \;=\; \lambda e.\\
&\quad [\![ \textit{external-declaration} ]\!]_{xdecl}\;e\;;\;[\![ \textit{external-declaration-list} ]\!]_{xdecl}\;e
\end{aligned}
$$

*(X3)*
$$
\begin{aligned}
&[\![ \textit{declaration} ]\!]_{xdecl} \;=\; \lambda e.\,\lambda \rho.\\
&\quad lift_{\mathsf{V}\to\mathsf{C}}\,([\![ \textit{declaration} ]\!]_{decl}\;e\;\rho)\;*\;(\lambda \rho'.\;\mathcal{A}[\![ \textit{declaration} ]\!]_{decl}\;e\;\rho'\;*\;(\lambda u.\;unit\;\rho'))
\end{aligned}
$$

*(X4)*
$$
\begin{aligned}
&[\![ \textit{declaration-specifiers declarator } \{\textit{ declaration-list statement-list }\} ]\!]_{xdecl} \;=\; \lambda e.\,\lambda \rho.\\
&\quad lift_{\mathsf{V}\to\mathsf{C}}\,([\![ \textit{declarator} ]\!]_{dtor\,[func\,[\tau, p]]}\;e\;\rho)
\end{aligned}
$$

The primary dynamic meaning for external declarations is a function that updates the dynamic type environment. In the case of object declarations, the objects must be created and allocated, whereas in

the case of function declarations, functions need only be created.

EXTERNAL DECLARATIONS $\mathcal{I}$-MEANING

$$\blacktriangleright \quad \mathcal{I}[\![xdecl]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{C}(\mathbf{U})$$

*(X2)*
$$\mathcal{I}[\![\textit{external-declaration external-declaration-list}]\!]_{xdecl} = \lambda e.\, \lambda \rho.\, \lambda \xi.\; \mathcal{I}[\![\textit{external-declaration}]\!]_{xdecl}\; e\; \rho\; \xi \;*\; (\lambda u.\, \mathcal{I}[\![\textit{external-declaration-list}]\!]_{xdecl}\; e\; \rho\; \xi)$$

*(X3)*
$$\mathcal{I}[\![\textit{declaration}]\!]_{xdecl} = \mathcal{I}[\![\textit{declaration}]\!]_{decl}$$

*(X4)*
$$\mathcal{I}[\![\textit{declaration-specifiers declarator } \{\textit{ declaration-list statement-list } \}]\!]_{xdecl} = \lambda e.\, \lambda \rho.\, \lambda \xi.\; unit\; ndu$$

The $\mathcal{I}$-meaning for external declarations is useful for the initialization of objects of the outermost scope. In the case of function definitions, no initialization is necessary, because of deviation D-5 in Section 2.3. This would not be true if static objects were supported inside function bodies.

EXTERNAL DECLARATIONS $\mathcal{C}$-MEANING

$$\blacktriangleright \quad \mathcal{C}[\![xdecl]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{V}(\mathbf{Cod})$$

*(X2)*
$$\mathcal{C}[\![\textit{external-declaration external-declaration-list}]\!]_{xdecl} = \lambda e.\, \lambda \rho.\, \lambda \xi.\; \mathcal{C}[\![\textit{external-declaration}]\!]_{xdecl}\; e\; \rho\; ;\; \mathcal{C}[\![\textit{external-declaration-list}]\!]_{xdecl}\; e\; \rho$$

*(X3)*
$$\mathcal{C}[\![\textit{declaration}]\!]_{xdecl} = \lambda e.\, \lambda \rho.\, \lambda \xi.\; unit\; \xi$$

The $\mathcal{C}$-meaning for external declarations is used for the definition of functions. It is a function which takes as parameters the current static and dynamic type environment and the current function code environment and returns an updated function code environment.

*(X4)*
$$\begin{aligned}
&\mathcal{C}[\![\textit{declaration-specifiers declarator } \{\textit{ declaration-list statement-list } \}]\!]_{xdecl} = \lambda e.\, \lambda \rho.\, \lambda \xi. \\
&\quad lift_{\mathsf{E}\to\mathsf{V}}\; (rec\; (\mathcal{F}\{\!|\textit{declarator}|\!\}\; ;\; \{\!|\textit{declaration-list}|\!\})\; (\uparrow e)) \;*\; (\lambda e'. \\
&\quad \mathcal{G}[\![\textit{declarator}]\!]_{dtor\,[func\,[\tau,p]]}\; e\; \rho \;*\; (\lambda d_f. \\
&\quad \textbf{let}\; b_f = \lambda d_p.\; funbody\; ( \\
&\qquad lift_{\mathsf{C}\to\mathsf{K}}\; (\mathcal{F}_A[\![\textit{declarator}]\!]_{dtor\,[func\,[\tau,p]]}\; e'\; (e' \uparrow \rho)\; d_p) \;*\; (\lambda \rho'. \\
&\qquad lift_{\mathsf{V}\to\mathsf{K}}\; (defineBlock\; 0\; e' \\
&\qquad\quad (\lambda e_c.\, \lambda \rho_c.\; lift_{\mathsf{V}\to\mathsf{C}}\; ([\![\textit{declaration-list}]\!]_{decl}\; e_c\; (e_c \uparrow \rho_c)) \;*\; (\lambda \rho_c'. \\
&\qquad\qquad \mathcal{A}[\![\textit{declaration-list}]\!]_{decl}\; e_c\; \rho_c' \;*\; (\lambda u. \\
&\qquad\qquad unit\; \rho_c'))) \\
&\qquad\quad (\lambda e_c.\, \lambda \rho_c.\; \mathcal{R}[\![\textit{declaration-list}]\!]_{decl}\; e_c\; \rho_c \;*\; (\lambda \rho_c'. \\
&\qquad\qquad \mathcal{F}_R[\![\textit{declarator}]\!]_{dtor\,[func\,[\tau,p]]}\; e_c\; \rho_c' \;*\; (\lambda \rho_c''. \\
&\qquad\qquad unit\; (e_c \downarrow \rho_c'')))) \\
&\qquad\quad scopeEmpty_e) \;*\; (\lambda \eta. \\
&\qquad lift_{\mathsf{V}\to\mathsf{K}}\; (\mathcal{P}[\![\textit{statement-list}]\!]_{stmt\,[\tau]}\; e'\; \eta) \;*\; (\lambda \eta'. \\
&\qquad scopeUse\; \eta'\; ( \\
&\qquad\quad lift_{\mathsf{V}\to\mathsf{K}}\; ([\![\textit{declaration-list}]\!]_{decl}\; e'\; \rho') \;*\; (\lambda \rho''. \\
&\qquad\quad lift_{\mathsf{C}\to\mathsf{K}}\; (\mathcal{A}[\![\textit{declaration-list}]\!]_{decl}\; e'\; \rho'') \;*\; (\lambda u_1. \\
&\qquad\quad lift_{\mathsf{C}\to\mathsf{K}}\; (\mathcal{I}[\![\textit{declaration-list}]\!]_{decl}\; e'\; \rho''\; \xi) \;*\; (\lambda u_2. \\
&\qquad\quad use\; (rec\text{-}L\; (\mathcal{L}[\![\textit{statement-list}]\!]_{stmt\,[\tau]}\; e'\; \rho''\; \xi)\; \ell_\circ)\; (\lambda \ell. \\
&\qquad\qquad [\![\textit{statement-list}]\!]_{stmt\,[\tau]}\; e'\; \rho''\; \xi\; \ell \;*\; (\lambda u. \\
&\qquad\qquad lift_{\mathsf{C}\to\mathsf{K}}\; (\mathcal{R}[\![\textit{declaration-list}]\!]_{decl}\; e'\; \rho'') \;*\; (\lambda \rho'''. \\
&\qquad\qquad lift_{\mathsf{C}\to\mathsf{K}}\; (\mathcal{F}_R[\![\textit{declarator}]\!]_{dtor\,[func\,[\tau,p]]}\; e'\; \rho''') \;*\; (\lambda \rho''''. \\
&\qquad\qquad unit\; u)))))))))))) \\
&\quad \textbf{in}\;\; \xi[d_f \mapsto \langle func\,[\tau,p], b_f\rangle]))
\end{aligned}$$

The $\mathcal{C}$-meaning of function definitions is the most complicated part of the dynamic semantics of declarations. First the static type environment for the function's body is calculated and the function identifier is looked up in the dynamic type environment. Following that, the dynamic semantics of the function's body is defined and the function code environment is updated.

The semantics of the function's body requires a conversion from a statement computation to an expression computation, performed by function $funbody$. Execution starts by the calculation of the

body's dynamic type environment, in which the declaration of formal parameters must be taken into account. This is performed by the $\mathcal{F}_A$-meaning of the function declarator. Following that, a new scope for the function's body is defined, starting from the empty scope information that corresponds to the outermost scope. Upon entering the function's scope, the objects to be created are those declared inside the body; upon leaving it, the objects to be destroyed also include the function's formal parameters. Subsequently, scope information is updated by including enclosed scopes defined in the function's body. This is performed by the $\mathcal{P}$-meaning of the body's statements.

After all environments and scope information have been determined, the objects declared inside the body need to be created, allocated and initialized. The label environment is then calculated, using the function *rec-L* and starting with an empty label environment. The statements in the function's body are then executed, followed by the destruction of the objects declared in the body and the formal parameters. The result of the body's execution is the result obtained from the execution of the body's statements.

## 13.2 Declarations

DECLARATIONS

▶ $[\![\, decl \,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![ e ]\!]_{Ent} \to \mathsf{V}([\![ e ]\!]_{Ent})$

*(D1)* $[\![ \epsilon ]\!]_{decl} \;=\; \lambda e.\, unit$

*(D2)* $[\![ \textit{declaration declaration-list} ]\!]_{decl} \;=\; \lambda e.\, [\![ \textit{declaration} ]\!]_{decl}\; e\; ;\; [\![ \textit{declaration-list} ]\!]_{decl}\; e$

*(D3)* $[\![ \textit{declaration-specifiers init-declarator-list ;} ]\!]_{decl} \;=\; [\![ \textit{init-declarator-list} ]\!]_{idtor}$

DECLARATIONS $\mathcal{A}$-MEANING

▶ $\mathcal{A}[\![\, decl \,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![ e ]\!]_{Ent} \to \mathsf{C}(\mathbf{U})$

*(D1)* $\mathcal{A}[\![ \epsilon ]\!]_{decl} \;=\; \lambda e.\, \lambda \rho.\, unit\; u$

*(D2)* $\mathcal{A}[\![ \textit{declaration declaration-list} ]\!]_{decl} \;=\; \lambda e.\, \lambda \rho.\, \mathcal{A}[\![ \textit{declaration} ]\!]_{decl}\; e\; \rho \;*\; (\lambda u.\, \mathcal{A}[\![ \textit{declaration-list} ]\!]_{decl}\; e\; \rho)$

*(D3)* $\mathcal{A}[\![ \textit{declaration-specifiers init-declarator-list ;} ]\!]_{decl} \;=\; \mathcal{A}[\![ \textit{init-declarator-list} ]\!]_{idtor}$

DECLARATIONS $\mathcal{R}$-MEANING

▶ $\mathcal{R}[\![\, decl \,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![ e ]\!]_{Ent} \to \mathsf{C}([\![ e ]\!]_{Ent})$

*(D1)* $\mathcal{R}[\![ \epsilon ]\!]_{decl} \;=\; \lambda e.\, unit$

*(D2)* $\mathcal{R}[\![ \textit{declaration declaration-list} ]\!]_{decl} \;=\; \lambda e.\, \mathcal{R}[\![ \textit{declaration} ]\!]_{decl}\; e\; ;\; \mathcal{R}[\![ \textit{declaration-list} ]\!]_{decl}\; e$

*(D3)* $\mathcal{R}[\![ \textit{declaration-specifiers init-declarator-list ;} ]\!]_{decl} \;=\; \mathcal{R}[\![ \textit{init-declarator-list} ]\!]_{idtor}$

DECLARATIONS $\mathcal{I}$-MEANING

▶ $\mathcal{I}[\![\, decl \,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![ e ]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{C}(\mathbf{U})$

*(D1)* $\mathcal{I}[\![ \epsilon ]\!]_{decl} \;=\; \lambda e.\, \lambda \rho.\, \lambda \xi.\, unit\; u$

*(D2)* $\mathcal{I}[\![ \textit{declaration declaration-list} ]\!]_{decl} \;=\; \lambda e.\, \lambda \rho.\, \lambda \xi.$
$\mathcal{I}[\![ \textit{declaration} ]\!]_{decl}\; e\; \rho\; \xi \;*\; (\lambda u.\, \mathcal{I}[\![ \textit{declaration-list} ]\!]_{decl}\; e\; \rho\; \xi)$

*(D3)* $\mathcal{I}[\![ \textit{declaration-specifiers init-declarator-list ;} ]\!]_{decl} \;=\; \mathcal{I}[\![ \textit{init-declarator-list} ]\!]_{idtor}$

The primary semantic meaning of declarations simply creates the declared objects and functions. The $\mathcal{A}$-meaning is used for the allocation of objects, whereas the $\mathcal{R}$-meaning is responsible for their deallocation and destruction. Finally, the $\mathcal{I}$-meaning performs the declared objects' initialization.

### 13.2.1 Declarators

DECLARATORS WITH INITIALIZERS

▶ $[\![idtor]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{V}([\![e]\!]_{Ent})$

*(D4)* $[\![\epsilon]\!]_{idtor} \;=\; \lambda e.\; unit$

*(D5)* $[\![\textit{init-declarator init-declarator-list}]\!]_{idtor} \;=\; \lambda e.\; [\![\textit{init-declarator}]\!]_{idtor}\; e \;;\; [\![\textit{init-declarator-list}]\!]_{idtor}\; e$

*(D6)* $[\![\textit{declarator}]\!]_{idtor} \;=\; [\![\textit{declarator}]\!]_{dtor\,[\phi]}$

*(D7)* $[\![\textit{declarator = initializer}]\!]_{idtor} \;=\; [\![\textit{declarator}]\!]_{dtor\,[\phi]}$

DECLARATORS WITH INITIALIZERS $\mathcal{A}$-MEANING

▶ $\mathcal{A}[\![idtor]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}(\mathbf{U})$

*(D4)* $\mathcal{A}[\![\epsilon]\!]_{idtor} \;=\; \lambda e.\; \lambda \rho.\; unit\; u$

*(D5)* $\mathcal{A}[\![\textit{init-declarator init-declarator-list}]\!]_{idtor} \;=\; \lambda e.\; \lambda \rho.$
$\mathcal{A}[\![\textit{init-declarator}]\!]_{idtor}\; e\; \rho \;*\; (\lambda u.\; \mathcal{A}[\![\textit{init-declarator-list}]\!]_{idtor}\; e\; \rho)$

*(D6)* $\mathcal{A}[\![\textit{declarator}]\!]_{idtor} \;=\; \mathcal{A}[\![\textit{declarator}]\!]_{dtor\,[\phi]}$

*(D7)* $\mathcal{A}[\![\textit{declarator = initializer}]\!]_{idtor} \;=\; \mathcal{A}[\![\textit{declarator}]\!]_{dtor\,[\phi]}$

DECLARATORS WITH INITIALIZERS $\mathcal{R}$-MEANING

▶ $\mathcal{R}[\![idtor]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}([\![e]\!]_{Ent})$

*(D4)* $\mathcal{R}[\![\epsilon]\!]_{idtor} \;=\; \lambda e.\; unit$

*(D5)* $\mathcal{R}[\![\textit{init-declarator init-declarator-list}]\!]_{idtor} \;=\; \lambda e.\; \mathcal{R}[\![\textit{init-declarator}]\!]_{idtor}\; e \;;\; \mathcal{R}[\![\textit{init-declarator-list}]\!]_{idtor}\; e$

*(D6)* $\mathcal{R}[\![\textit{declarator}]\!]_{idtor} \;=\; \mathcal{R}[\![\textit{declarator}]\!]_{dtor\,[\phi]}$

*(D7)* $\mathcal{R}[\![\textit{declarator = initializer}]\!]_{idtor} \;=\; \mathcal{R}[\![\textit{declarator}]\!]_{dtor\,[\phi]}$

DECLARATORS WITH INITIALIZERS $\mathcal{I}$-MEANING

▶ $\mathcal{I}[\![idtor]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{C}(\mathbf{U})$

*(D4)* $\mathcal{I}[\![\epsilon]\!]_{idtor} \;=\; \lambda e.\; \lambda \rho.\; \lambda \xi.\; unit\; u$

*(D5)* $\mathcal{I}[\![\textit{init-declarator init-declarator-list}]\!]_{idtor} \;=\; \lambda e.\; \lambda \rho.\; \lambda \xi.$
$\mathcal{I}[\![\textit{init-declarator}]\!]_{idtor}\; e\; \rho\; \xi \;*\; (\lambda u.\; \mathcal{I}[\![\textit{init-declarator-list}]\!]_{idtor}\; e\; \rho\; \xi)$

*(D6)* $\mathcal{I}[\![\textit{declarator}]\!]_{idtor} \;=\; \lambda e.\; \lambda \rho.\; \lambda \xi.\; unit\; u$

*(D7)* $\mathcal{I}[\![\textit{declarator = initializer}]\!]_{idtor} \;=\; \lambda e.\; \lambda \rho.\; \lambda \xi.$
$lift_{\mathsf{V} \to \mathsf{G}}\; (\mathcal{G}[\![\textit{declarator}]\!]_{dtor\,[\alpha]}\; e\; \rho) \;*\; (\lambda d.\; [\![\textit{initializer}]\!]_{init\,[\alpha]}\; e\; \rho\; \xi\; d)$

The alternative meanings of declarators with initializers serve the same purpose as the corresponding meanings of declarations. In the initialization of objects whose declaration includes an initializer, the dynamic semantics of the initializer is used.

DECLARATORS

▶ $[\![dtor\,[\phi]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{V}([\![e]\!]_{Ent})$

*(D8)* $[\![I]\!]_{dtor\,[\phi]} \;=\; \lambda e.\; \lambda \rho.\; create\; e\; \rho\; I$

*(D9)* $[\![I]\!]_{dtor\,[\phi]} \;=\; \lambda e.\; \lambda \rho.\; unit\; \rho$

*(D10)* $[\![\textit{declarator [ constant-expression ]}]\!]_{dtor\,[\phi]} \;=\; [\![\textit{declarator}]\!]_{dtor\,[\phi]}$

*(D11)* $[\![\textit{declarator [ ]}]\!]_{dtor\,[\phi]} \;=\; [\![\textit{declarator}]\!]_{dtor\,[\phi]}$

*(D12)* $[\![\textit{* type-qualifier declarator}]\!]_{dtor\,[\phi]} \;=\; [\![\textit{declarator}]\!]_{dtor\,[\phi]}$

*(D13)* $[\![\textit{declarator ( parameter-type-list )}]\!]_{dtor\,[\phi]} \;=\; [\![\textit{declarator}]\!]_{dtor\,[\phi]}$

*(D14)* $[\![\textit{declarator ( parameter-type-list )}]\!]_{dtor\,[\phi]} \;=\; [\![\textit{declarator}]\!]_{dtor\,[\phi]}$

The purpose of the primary meaning for declarators has been discussed before. Declarators corresponding to a *typedef* are simply ignored.

DECLARATORS $\mathcal{G}$-MEANING

▶ $\mathcal{G}[\![\, dtor\,[\phi]\,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{V}([\![\phi]\!]_{den})$

*(D8)* $\mathcal{G}[\![I]\!]_{dtor\,[\phi]} = \lambda e.\, \lambda \rho.\, lookup\ e\ \rho\ I$

*(D9)* $\mathcal{G}[\![I]\!]_{dtor\,[\phi]} = \lambda e.\, \lambda \rho.\, error$

*(D10)* $\mathcal{G}[\![declarator\ [\ constant\text{-}expression\ ]\,]\!]_{dtor\,[\phi]} = \mathcal{G}[\![declarator]\!]_{dtor\,[\phi]}$

*(D11)* $\mathcal{G}[\![declarator\ [\ ]\,]\!]_{dtor\,[\phi]} = \mathcal{G}[\![declarator]\!]_{dtor\,[\phi]}$

*(D12)* $\mathcal{G}[\![*\ type\text{-}qualifier\ declarator]\!]_{dtor\,[\phi]} = \mathcal{G}[\![declarator]\!]_{dtor\,[\phi]}$

*(D13)* $\mathcal{G}[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[\phi]} = \mathcal{G}[\![declarator]\!]_{dtor\,[\phi]}$

*(D14)* $\mathcal{G}[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[\phi]} = \mathcal{G}[\![declarator]\!]_{dtor\,[\phi]}$

The $\mathcal{G}$-meaning for declarators is used in order to extract their denoted value from the dynamic type environment. An error occurs if the declarator corresponds to a *typedef*.

DECLARATORS $\mathcal{A}$-MEANING

▶ $\mathcal{A}[\![\, dtor\,[\phi]\,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}(\mathbf{U})$

*(D8)* $\mathcal{A}[\![I]\!]_{dtor\,[\phi]} = \lambda e.\, \lambda \rho.\, allocate\ e\ \rho\ I$

*(D9)* $\mathcal{A}[\![I]\!]_{dtor\,[\phi]} = \lambda e.\, \lambda \rho.\, unit\ u$

*(D10)* $\mathcal{A}[\![declarator\ [\ constant\text{-}expression\ ]\,]\!]_{dtor\,[\phi]} = \mathcal{A}[\![declarator]\!]_{dtor\,[\phi]}$

*(D11)* $\mathcal{A}[\![declarator\ [\ ]\,]\!]_{dtor\,[\phi]} = \mathcal{A}[\![declarator]\!]_{dtor\,[\phi]}$

*(D12)* $\mathcal{A}[\![*\ type\text{-}qualifier\ declarator]\!]_{dtor\,[\phi]} = \mathcal{A}[\![declarator]\!]_{dtor\,[\phi]}$

*(D13)* $\mathcal{A}[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[\phi]} = \mathcal{A}[\![declarator]\!]_{dtor\,[\phi]}$

*(D14)* $\mathcal{A}[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[\phi]} = \mathcal{A}[\![declarator]\!]_{dtor\,[\phi]}$

The $\mathcal{A}$ meaning of declarators serves the purpose of allocating objects. Declarators corresponding to a *typedef* are ignored.

DECLARATORS $\mathcal{R}$-MEANING

▶ $\mathcal{R}[\![\, dtor\,[\phi]\,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}([\![e]\!]_{Ent})$

*(D8)* $\mathcal{R}[\![I]\!]_{dtor\,[\phi]} = \lambda e.\, \lambda \rho.\, destroy\ e\ \rho\ I$

*(D9)* $\mathcal{R}[\![I]\!]_{dtor\,[\phi]} = \lambda e.\, unit$

*(D10)* $\mathcal{R}[\![declarator\ [\ constant\text{-}expression\ ]\,]\!]_{dtor\,[\phi]} = \mathcal{R}[\![declarator]\!]_{dtor\,[\phi]}$

*(D11)* $\mathcal{R}[\![declarator\ [\ ]\,]\!]_{dtor\,[\phi]} = \mathcal{R}[\![declarator]\!]_{dtor\,[\phi]}$

*(D12)* $\mathcal{R}[\![*\ type\text{-}qualifier\ declarator]\!]_{dtor\,[\phi]} = \mathcal{R}[\![declarator]\!]_{dtor\,[\phi]}$

*(D13)* $\mathcal{R}[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[\phi]} = \mathcal{R}[\![declarator]\!]_{dtor\,[\phi]}$

*(D14)* $\mathcal{R}[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[\phi]} = \mathcal{R}[\![declarator]\!]_{dtor\,[\phi]}$

The $\mathcal{R}$ meaning of declarators serves the purpose of deallocating and destroying objects. Declarators corresponding to a *typedef* are again ignored.

DECLARATORS $\mathcal{F}_A$-MEANING

▶ $\mathcal{F}_A[\![\, dtor\,[func\,[\tau, p]]\,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to [\![p]\!]_{Prot} \to \mathsf{G}([\![e]\!]_{Ent})$

*(D8)* $\mathcal{F}_A[\![I]\!]_{dtor\,[func\,[\tau,p]]} = \lambda e.\, \lambda \rho.\, \lambda d_p.\, error$

*(D9)* $\mathcal{F}_A[\![I]\!]_{dtor\,[func\,[\tau,p]]} = \lambda e.\, \lambda \rho.\, \lambda d_p.\, error$

*(D10)* $\mathcal{F}_A[\![declarator\ [\ constant\text{-}expression\ ]\,]\!]_{dtor\,[func\,[\tau,p]]} = \mathcal{F}_A[\![declarator]\!]_{dtor\,[func\,[\tau,p]]}$

*(D11)* $\mathcal{F}_A[\![declarator\ [\ ]\,]\!]_{dtor\,[func\,[\tau,p]]} = \mathcal{F}_A[\![declarator]\!]_{dtor\,[func\,[\tau,p]]}$

*(D12)* $\mathcal{F}_A[\![*\ type\text{-}qualifier\ declarator]\!]_{dtor\,[func\,[\tau,p]]} = \mathcal{F}_A[\![declarator]\!]_{dtor\,[func\,[\tau,p]]}$

*(D13)* $\mathcal{F}_A[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[func\,[\tau,p]]} = \mathcal{F}_A[\![declarator]\!]_{dtor\,[func\,[\tau,p]]}$

*(D14)* $\mathcal{F}_A[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[func\,[\tau,p]]} = [\![parameter\text{-}type\text{-}list]\!]_{prot\,[p]}$

The $\mathcal{F}_A$-meaning is only defined for function declarators. An error occurs if it is used on other types of declarators. It is a function which takes as parameters the static and dynamic type environments and the dynamic values of the function's actual parameters. It returns a computation resulting in the updated dynamic type environment, where the formal parameters have been created. Also, this computation initializes the values of the formal parameters to the given values of the actual parameters.

DECLARATORS $\mathcal{F}_R$-MEANING

▶ $\mathcal{F}_R[\![\, dtor\,[func\,[\tau, p]]\,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}([\![e]\!]_{Ent})$

*(D8)* $\mathcal{F}_R[\![\, I\,]\!]_{dtor\,[func\,[\tau,p]]} = \lambda e.\, \lambda \rho.\, error$

*(D9)* $\mathcal{F}_R[\![\, I\,]\!]_{dtor\,[func\,[\tau,p]]} = \lambda e.\, \lambda \rho.\, error$

*(D10)* $\mathcal{F}_R[\![declarator\ [\ constant\text{-}expression\ ]\,]\!]_{dtor\,[func\,[\tau,p]]} = \mathcal{F}_R[\![declarator]\!]_{dtor\,[func\,[\tau,p]]}$

*(D11)* $\mathcal{F}_R[\![declarator\ [\ ]\,]\!]_{dtor\,[func\,[\tau,p]]} = \mathcal{F}_R[\![declarator]\!]_{dtor\,[func\,[\tau,p]]}$

*(D12)* $\mathcal{F}_R[\![*\ type\text{-}qualifier\ declarator]\!]_{dtor\,[func\,[\tau,p]]} = \mathcal{F}_R[\![declarator]\!]_{dtor\,[func\,[\tau,p]]}$

*(D13)* $\mathcal{F}_R[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[func\,[\tau,p]]} = \mathcal{F}_R[\![declarator]\!]_{dtor\,[func\,[\tau,p]]}$

*(D14)* $\mathcal{F}_R[\![declarator\ (\ parameter\text{-}type\text{-}list\ )\,]\!]_{dtor\,[func\,[\tau,p]]} = \mathcal{R}[\![parameter\text{-}type\text{-}list]\!]_{prot\,[p]}$

The $\mathcal{F}_R$-meaning is again only defined for function declarators and an error occurs otherwise. It is a function which takes as parameters the static and dynamic type environments and returns a computation resulting in the updated dynamic type environment, where the formal parameters have been deallocated and destroyed.

### 13.2.2 Function prototypes and parameters

PARAMETER TYPE LISTS

▶ $[\![\, prot\,[p]\,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to [\![p]\!]_{Prot} \to \mathsf{G}([\![e]\!]_{Ent})$

*(D15)* $[\![\, \epsilon\,]\!]_{prot\,[p_\circ]} = \lambda e.\, \lambda \rho.\, \lambda d_p.\, unit\ \rho$

*(D16)* $[\![\, \ldots\,]\!]_{prot\,[p]} = \lambda e.\, \lambda \rho.\, \lambda d_p.\, unit\ \rho$ *(Inaccurate.)*

*(D17)* $[\![parameter\text{-}declaration\ parameter\text{-}type\text{-}list]\!]_{prot\,[p']} = \lambda e.\, \lambda \rho.\, \lambda d_p.$
$[\![parameter\text{-}declaration]\!]_{par\,[\tau]}\ e\ \rho\ (d_p\ 1)\ *\ (\lambda \rho'.\, [\![parameter\text{-}type\text{-}list]\!]_{prot\,[p]}\ e\ \rho'\ (shift\ 1\ d_p))$

The primary dynamic meaning of parameter type lists serves the same purpose as the $\mathcal{F}_A$-meaning for function declarators, which was defined above. The given semantics is inaccurate as far as the semantics of function prototypes containing ellipsis is concerned.

PARAMETER TYPE LISTS $\mathcal{R}$-MEANING

▶ $\mathcal{R}[\![\, prot\,[p]\,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}([\![e]\!]_{Ent})$

*(D15)* $\mathcal{R}[\![\, \epsilon\,]\!]_{prot\,[p_\circ]} = \lambda e.\, \lambda \rho.\, unit\ \rho$

*(D16)* $\mathcal{R}[\![\, \ldots\,]\!]_{prot\,[p]} = \lambda e.\, \lambda \rho.\, unit\ \rho$ *(Innacurate.)*

*(D17)* $\mathcal{R}[\![parameter\text{-}declaration\ parameter\text{-}type\text{-}list]\!]_{prot\,[p']} = \lambda e.$
$[\![parameter\text{-}declaration]\!]_{par\,[\tau]}\ e\ ;\ [\![parameter\text{-}type\text{-}list]\!]_{prot\,[p]}\ e$

The $\mathcal{R}$-meaning of parameter type lists serves the same purpose as the $\mathcal{F}_R$-meaning for function declarators, defined above. Again, the given semantics is inaccurate as far as the semantics of function prototypes containing ellipsis is concerned.

PARAMETER DECLARATIONS

▶ $[\![\, par\,[\tau]\,]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to [\![\tau]\!]_{dat} \to \mathsf{G}([\![e]\!]_{Ent})$

*(D18)* $[\![declaration\text{-}specifiers\ declarator]\!]_{par\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda d.$
$lift_{\mathsf{V}\to\mathsf{G}}\ ([\![declarator]\!]_{dtor\,[obj\,[\tau,q]]}\ e\ \rho)\ *\ (\lambda \rho'.$
$lift_{\mathsf{C}\to\mathsf{G}}\ (\mathcal{A}[\![declarator]\!]_{dtor\,[obj\,[\tau,q]]}\ e\ \rho')\ *\ (\lambda u.$
$lift_{\mathsf{V}\to\mathsf{G}}\ (\mathcal{G}[\![declarator]\!]_{dtor\,[obj\,[\tau,q]]}\ e\ \rho')\ *\ (\lambda a.$
$putValue_{\tau \mapsto obj\,[\tau,q]}\ a\ d\ *\ (\lambda d'.$
$unit\ \rho'))))$

The purpose of the primary dynamic meaning for parameter declarations has already been discussed. The formal parameter is created, allocated and initialized by the value of the corresponding actual parameter.

PARAMETER DECLARATIONS $\mathcal{R}$-MEANING

▶ $\mathcal{R}[\![par\,[\tau]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathsf{C}([\![e]\!]_{Ent})$

*(D18)* $\mathcal{R}[\![\textit{declaration-specifiers declarator}]\!]_{par\,[\tau]} \;=\; \mathcal{R}[\![\textit{declarator}]\!]_{dtor\,[obj\,[\tau,q]]}$

The purpose of the $\mathcal{R}$-meaning for parameter declarations has already been discussed. The formal parameter is deallocated and destroyed.

### 13.2.3 Initializations

INITIALIZATION

The primary dynamic meaning of initializations is a function which takes as parameters the current static and dynamic type environments, the function code environment and the address of the object to be initialized. It returns a computation with an unimportant result which initializes the given object to the value contained in the initializer.

▶ $[\![init\,[m]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to [\![m]\!]_{mem} \to \mathsf{C}(\mathbf{U})$

*(I1)* $$[\![E]\!]_{init\,[obj\,[\tau,q]]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda a.\; fullExpression\;(\\ \mathcal{A}[\![E]\!]_{exp\,[\tau]}\; e\;\rho\;\xi\;*\;(\lambda d.\\ putValue_{\tau\mapsto obj\,[\tau,q]}\; a\; d\;*\;(\lambda d'.\\ unit\;u)))$$

*(I2)* $$[\![E]\!]_{init\,[bitfield\,[\beta,q,n]]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda a_b.\; fullExpression\;(\\ \mathcal{A}[\![E]\!]_{exp\,[\tau]}\; e\;\rho\;\xi\;*\;(\lambda d.\\ putValue_{\tau\mapsto bitfield\,[\beta,q,n]}\; a_b\; d\;*\;(\lambda d'.\\ unit\;u)))$$

For the initialization of simple objects and bit-field members the initializing expression is evaluated and the result is converted as if by assignment to the appropriate data type. The obtained value is then stored in memory.

*(I3)* $[\![E]\!]_{init\,[array\,[\alpha,n]]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda a_f.\; storeStringLit_\alpha\; E\; a_f\; n$

*(I4)* $[\![E]\!]_{init\,[array\,[\alpha,n]]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda a_f.\; storeWideStringLit_\alpha\; E\; a_f\; n$

The dynamic semantics of the initialization of character and wide character arrays by means of string and wide string literals is defined by using the $storeStringLit_\alpha$ and $storeWideStringLit_\alpha$ functions.

*(I5)* $$[\![\{\;\textit{initializer-list}\;\}]\!]_{init\,[array\,[\alpha,n]]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda a_f.\\ [\![\textit{initializer-list}]\!]_{init\text{-}a\,[\alpha]}\; e\;\rho\;\xi\;a_f\;n$$

*(I6)* $$[\![\{\;\textit{initializer-list}\;\}]\!]_{init\,[obj\,[struct\,[t,\pi],q]]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda a.\\ [\![\textit{initializer-list}]\!]_{init\text{-}s\,[\alpha]}\; e\;\rho\;\xi\;(structMember_\pi\; a)$$

*(I7)* $$[\![\{\;\textit{initializer-list}\;\}]\!]_{init\,[obj\,[union\,[t,\pi],q]]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda a.\\ [\![\textit{initializer-list}]\!]_{init\text{-}u\,[\alpha]}\; e\;\rho\;\xi\;(unionMember_\pi\; a)$$

Aggregate objects can also be initialized by bracketed lists of initializers. The type of the object determines the semantic function that will be used.

ARRAY INITIALIZATION

▶ $[\![\mathit{init\text{-}a}\,[\alpha]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to (\mathbf{N} \to [\![\alpha]\!]_{obj}) \to \mathbf{N} \to \mathsf{C}(\mathbf{U})$

*(I8)* $[\![\mathit{initializer}]\!]_{\mathit{init\text{-}a}\,[\alpha]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda a_f.\, \lambda n.\, (n > 0) \to$
$[\![\mathit{initializer}]\!]_{\mathit{init}\,[\alpha]}\; e\; \rho\; \xi\; (a_f\; 0)\; *\; (\lambda u.$
$\mathit{zeroArray}_{\alpha}\; (\mathit{shift}\; 1\; a_f)\; (n-1))\,,\; \mathit{error}$

*(I9)* $[\![\mathit{initializer\ initializer\text{-}list}]\!]_{\mathit{init\text{-}a}\,[\alpha]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda a_f.\, \lambda n.\, (n > 0) \to$
$[\![\mathit{initializer}]\!]_{\mathit{init}\,[\alpha]}\; e\; \rho\; \xi\; (a_f\; 0)\; *\; (\lambda u.$
$[\![\mathit{initializer\text{-}list}]\!]_{\mathit{init\text{-}a}\,[\alpha]}\; e\; \rho\; \xi\; (\mathit{shift}\; 1\; a_f)\; (n-1))\,,\; \mathit{error}$

The dynamic meaning of array initialization is defined iteratively for each of the array's elements. In case the number of initializers contained in the list is smaller than the number of array elements, the remaining elements are initialized to zero values.

STRUCTURE INITIALIZATION

▶ $[\![\mathit{init\text{-}s}\,[\pi]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to (I : \mathbf{Ide} \twoheadrightarrow [\![\pi \Downarrow I]\!]_{mem}) \to \mathsf{C}(\mathbf{U})$

*(I10)* $[\![\mathit{initializer}]\!]_{\mathit{init\text{-}s}\,[\pi]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda d_{mf}.$
$[\![\mathit{initializer}]\!]_{\mathit{init}\,[m]}\; e\; \rho\; \xi\; (d_{mf}\; I)\; *\; (\lambda u.$
$\mathit{zeroStruct}_{\pi'}\; d_{mf})$

*(I11)* $[\![\mathit{initializer\ initializer\text{-}list}]\!]_{\mathit{init\text{-}s}\,[\pi]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda d_{mf}.$
$[\![\mathit{initializer}]\!]_{\mathit{init}\,[m]}\; e\; \rho\; \xi\; (d_{mf}\; I)\; *\; (\lambda u.$
$[\![\mathit{initializer\text{-}list}]\!]_{\mathit{init\text{-}s}\,[\pi']}\; e\; \rho\; \xi\; d_{mf})$

The dynamic meaning of structure initialization is defined iteratively for each of the structure's members. In case the number of initializers contained in the list is smaller than the number of the structure's members, the remaining members are initialized to zero values.

UNIT INITIALIZATION

▶ $[\![\mathit{init\text{-}u}\,[\pi]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to (I : \mathbf{Ide} \twoheadrightarrow [\![\pi \Downarrow I]\!]_{mem}) \to \mathsf{C}(\mathbf{U})$

*(I12)* $[\![\mathit{initializer}]\!]_{\mathit{init\text{-}u}\,[\pi]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda d_{mf}.\, [\![\mathit{initializer}]\!]_{\mathit{init}\,[m]}\; e\; \rho\; \xi\; (d_{mf}\; I)$

Finally, the dynamic meaning of union initialization is defined as the dynamic meaning of initializing the union's first element.

# Chapter 14

# Dynamic semantics of statements

CHAPTER OVERVIEW

This chapter defines the dynamic semantics of statements. An introduction to the four alternative dynamic meanings of statements is given in Section 14.1. Subsequently, the dynamic semantics of statement lists is defined in Section 14.2 and the dynamic semantics of statements is defined in Section 14.3.

## 14.1 Dynamic functions

PRIMARY MEANING

Four alternative dynamic meanings are defined for statements and statement lists of phrase type $stmt\,[\tau]$. Each corresponds to a different aspect of their execution. The primary dynamic meaning of statements is a function returning a statement computation. Its parameters are the current static and dynamic type environments, the function code environment and the label environment.

- $[\![stmt\,[\tau]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathbf{Lab} \to \mathsf{K}_\tau(\mathbf{U})$

$\mathcal{P}$-MEANING

The second alternative dynamic meaning for statements is used for the extraction of information concerning scopes. It is a function, taking as parameters the current static environment and scope information and returning the updated scope information.

- $\mathcal{P}[\![stmt\,[\tau]]\!] \;:\; \mathbf{Ent} \to \mathbf{Scope} \to \mathsf{L}_\tau(\mathbf{Scope})$

$\mathcal{L}$-MEANING

The third alternative meaning is used for updating the label environment, adding information about the labels that are defined in the statement. It is a function which takes as parameters the current static and dynamic environments, the function code environment and an initial label environment and returns the updated label environment.

- $\mathcal{L}[\![stmt\,[\tau]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathbf{Lab} \to \mathsf{L}_\tau(\mathbf{Lab})$

$\mathcal{C}$-MEANING

Finally, the fourth alternative dynamic meaning is used in a similar way for updating the case label environment. It is again a function taking as parameters the current static and dynamic environments, the function code environment and the label environment. It also takes the data type $\tau'$ of the controlling expression for the case label environment and an initial case label environment of type $\mathbf{Cas}_{\tau'}$. The result is the updated case label environment, which additionally contains information about case labels defined in the statement.

- $\mathcal{C}[\![stmt\,[\tau]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathbf{Lab} \to \tau' : \mathbf{Type}_{dat} \twoheadrightarrow \mathbf{Cas}_{\tau'} \to \mathsf{L}_\tau(\mathbf{Cas}_{\tau'})$

## 14.2 Statement lists

EMPTY STATEMENT LIST

$$\textit{(S1)} \quad [\![\epsilon]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\,\lambda \ell.\; unit\; u$$
$$\textit{(S1)} \quad \mathcal{P}[\![\epsilon]\!]_{stmt\,[\tau]} = \lambda e.\; unit$$
$$\textit{(S1)} \quad \mathcal{L}[\![\epsilon]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\; unit$$
$$\textit{(S1)} \quad \mathcal{C}[\![\epsilon]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\,\lambda \ell.\,\lambda \tau_c.\; unit$$

The dynamic semantics of empty statement lists is very simple, for all alternative meanings.

NON-EMPTY STATEMENT LIST

$$\textit{(S2)} \quad [\![\textit{statement statement-list}]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\,\lambda \ell.\; [\![\textit{statement}]\!]_{stmt\,[\tau]}\; e\; \rho\; \xi\; \ell\; *\; (\lambda u.\; [\![\textit{statement-list}]\!]_{stmt\,[\tau]}\; e\; \rho\; \xi\; \ell)$$
$$\textit{(S2)} \quad \mathcal{P}[\![\textit{statement statement-list}]\!]_{stmt\,[\tau]} = \lambda e.\; \mathcal{P}[\![\textit{statement}]\!]_{stmt\,[\tau]}\; e\; ;\; \mathcal{P}[\![\textit{statement-list}]\!]_{stmt\,[\tau]}\; e$$
$$\textit{(S2)} \quad \mathcal{L}[\![\textit{statement statement-list}]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\,\lambda \ell.\; \textit{follow}\; (\mathcal{L}[\![\textit{statement}]\!]_{stmt\,[\tau]}\; e\; \rho\; \xi\; \ell)\; ([\![\textit{statement-list}]\!]_{stmt\,[\tau]}\; e\; \rho\; \xi\; \ell)\; *\; (\lambda \ell'.\; \mathcal{L}[\![\textit{statement-list}]\!]_{stmt\,[\tau]}\; e\; \rho\; \xi\; \ell')$$
$$\textit{(S2)} \quad \mathcal{C}[\![\textit{statement statement-list}]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\,\lambda \ell.\,\lambda \tau_c.\,\lambda \omega.\; \textit{follow}\; (\mathcal{C}[\![\textit{statement}]\!]_{stmt\,[\tau]}\; e\; \rho\; \xi\; \ell\; \tau_c\; \omega)\; ([\![\textit{statement-list}]\!]_{stmt\,[\tau]}\; e\; \rho\; \xi\; \ell)\; *\; (\lambda \omega'.\; \mathcal{C}[\![\textit{statement-list}]\!]_{stmt\,[\tau]}\; e\; \rho\; \xi\; \ell\; \tau_c\; \omega')$$

The dynamic semantic meanings of non-empty statement lists is formed by appropriately sequencing the meanings of their components. The first two equations are simple. In the last two, the auxiliary function *follow* is used, to indicate that the statement whose dynamic information is given by its first parameter is followed, under normal order of execution, by the statement list whose primary dynamic meaning is given by its second parameter.

## 14.3 Statements

The following sections define the dynamic semantics of all kinds of C statements, according to the same classification that is followed in the standard.

### 14.3.1 Empty and expression statements

EMPTY STATEMENT

$$\textit{(S3)} \quad [\![\,;\,]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\,\lambda \ell.\; unit\; u$$
$$\textit{(S3)} \quad \mathcal{P}[\![\,;\,]\!]_{stmt\,[\tau]} = \lambda e.\; unit$$
$$\textit{(S3)} \quad \mathcal{L}[\![\,;\,]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\; unit$$
$$\textit{(S3)} \quad \mathcal{C}[\![\,;\,]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\,\lambda \ell.\,\lambda \tau_c.\; unit$$

As in the case of empty statement lists, the dynamic semantics of empty statements is very simple.

EXPRESSION STATEMENT

$$\textit{(S4)} \quad [\![\textit{expression}\;;]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\,\lambda \ell.\; lift_{\mathsf{G}\to\mathsf{K}}\,([\![\textit{expression}]\!]_{exp\,[\tau']}\; e\; \rho\; \xi)\; *\; (\lambda d.\; unit\; u)$$
$$\textit{(S4)} \quad \mathcal{P}[\![\textit{expression}\;;]\!]_{stmt\,[\tau]} = \lambda e.\; unit$$
$$\textit{(S4)} \quad \mathcal{L}[\![\textit{expression}\;;]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\; unit$$
$$\textit{(S4)} \quad \mathcal{C}[\![\textit{expression}\;;]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda \rho.\,\lambda \xi.\,\lambda \ell.\,\lambda \tau_c.\; unit$$

The primary dynamic meaning for expression statements needs only convert the expression computation to a statement computation, discarding the result. The other alternative meanings are easily defined.

### 14.3.2 Compound statement

*(S5)* $[\![\{\ \textit{id declaration-list statement-list}\ \}]\!]_{stmt\,[\tau]} = \lambda e.\ \lambda \rho.\ \lambda \xi.\ \lambda \ell.$ BLOCKS
$\quad lift_{\mathsf{E}\to\mathsf{K}}\ (rec\ \{\!|\textit{declaration-list}|\!\}\ (\uparrow e))\ *\ (\lambda e'.$
$\quad inScope\ id\ ($
$\quad\quad lift_{\mathsf{V}\to\mathsf{K}}\ ([\![\textit{declaration-list}]\!]_{decl}\ e'\ (e' \uparrow \rho))\ *\ (\lambda \rho'.$
$\quad\quad lift_{\mathsf{C}\to\mathsf{K}}\ (\mathcal{A}[\![\textit{declaration-list}]\!]_{decl}\ e'\ \rho')\ *\ (\lambda u_1.$
$\quad\quad lift_{\mathsf{C}\to\mathsf{K}}\ (\mathcal{I}[\![\textit{declaration-list}]\!]_{decl}\ e'\ \rho'\ \xi)\ *\ (\lambda u_2.$
$\quad\quad [\![\textit{statement-list}]\!]_{stmt\,[\tau]}\ e'\ \rho'\ \xi\ \ell\ *\ (\lambda u.$
$\quad\quad lift_{\mathsf{C}\to\mathsf{K}}\ (\mathcal{R}[\![\textit{declaration-list}]\!]_{decl}\ e'\ \rho')\ *\ (\lambda \rho''.$
$\quad\quad unit\ u)))))))$

In the primary dynamic meaning for the compound statement, the static type environment for the body is first calculated and a change of the current scope is performed. Following that, objects defined within the block are created, allocated and initialized, using three forms of alternative dynamic meanings for declarations. The statement list is then executed, and following that, the defined objects are destroyed.

*(S5)* $\mathcal{P}[\![\{\ \textit{id declaration-list statement-list}\ \}]\!]_{stmt\,[\tau]} = \lambda e.\ \lambda \eta.$
$\quad lift_{\mathsf{E}\to\mathsf{V}}\ (rec\ \{\!|\textit{declaration-list}|\!\}\ (\uparrow e))\ *\ (\lambda e'.$
$\quad defineBlock\ id\ e'$
$\quad\quad (\lambda e_c.\ \lambda \rho_c.\ lift_{\mathsf{V}\to\mathsf{C}}\ ([\![\textit{declaration-list}]\!]_{decl}\ e_c\ (e_c \uparrow \rho_c))\ *\ (\lambda \rho_c'.$
$\quad\quad\quad\quad \mathcal{A}[\![\textit{declaration-list}]\!]_{decl}\ e_c\ \rho_c'\ *\ (\lambda u.\ unit\ \rho_c')))$
$\quad\quad (\lambda e_c.\ \lambda \rho_c.\ \mathcal{R}[\![\textit{declaration-list}]\!]_{decl}\ e_c\ \rho_c\ *\ (\lambda \rho_c'.\ unit\ (e_c \downarrow \rho_c')))$
$\quad\quad \eta\ *\ (\lambda \eta'.$
$\quad \mathcal{P}[\![\textit{statement-list}]\!]_{stmt\,[\tau]}\ e'\ \eta'\ *\ endBlock))$

Updating the scope information for compound statements requires the definition of a new scope. The dynamic semantics of the declaration is used for specifying the objects that need to be created when the scope is entered and destroyed when the scope is left. Finally, the scope information is further updated by statements in the block's body.

*(S5)* $\mathcal{L}[\![\{\ \textit{id declaration-list statement-list}\ \}]\!]_{stmt\,[\tau]} = \lambda e.\ \lambda \rho.\ \lambda \xi.\ \lambda \ell.$
$\quad lift_{\mathsf{E}\to\mathsf{L}}\ (rec\ \{\!|\textit{declaration-list}|\!\}\ (\uparrow e))\ *\ (\lambda e'.$
$\quad inScope\ id\ ($
$\quad\quad lift_{\mathsf{V}\to\mathsf{L}}\ ([\![\textit{declaration-list}]\!]_{decl}\ e'\ (e' \uparrow \rho))\ *\ (\lambda \rho'.$
$\quad\quad follow\ (\mathcal{L}[\![\textit{statement-list}]\!]_{stmt\,[\tau]}\ e'\ \rho'\ \xi\ \ell)$
$\quad\quad\quad (lift_{\mathsf{C}\to\mathsf{K}}\ (\mathcal{R}[\![\textit{declaration-list}]\!]_{decl}\ e'\ \rho')\ *\ (\lambda \rho''.\ unit\ u)))))$

*(S5)* $\mathcal{C}[\![\{\ \textit{id declaration-list statement-list}\ \}]\!]_{stmt\,[\tau]} = \lambda e.\ \lambda \rho.\ \lambda \xi.\ \lambda \ell.\ \lambda \tau_c.\ \lambda \omega.$
$\quad lift_{\mathsf{E}\to\mathsf{L}}\ (rec\ \{\!|\textit{declaration-list}|\!\}\ (\uparrow e))\ *\ (\lambda e'.$
$\quad inScope\ id\ ($
$\quad\quad lift_{\mathsf{V}\to\mathsf{L}}\ ([\![\textit{declaration-list}]\!]_{decl}\ e'\ (e' \uparrow \rho))\ *\ (\lambda \rho'.$
$\quad\quad follow\ (\mathcal{C}[\![\textit{statement-list}]\!]_{stmt\,[\tau]}\ e'\ \rho'\ \xi\ \ell\ \tau_c\ \omega)$
$\quad\quad\quad (lift_{\mathsf{C}\to\mathsf{K}}\ (\mathcal{R}[\![\textit{declaration-list}]\!]_{decl}\ e'\ \rho')\ *\ (\lambda \rho''.\ unit\ u)))))$

The meanings $\mathcal{L}$ and $\mathcal{C}$ for block statements are very similar. The only thing to notice is that the statement list is followed, under normal order of execution, by the destruction of locally defined objects.

### 14.3.3 Selection statements

IF-THEN STATEMENT

*(S6)* $[\![\texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda\ell.$
$\quad lift_{\mathsf{G}\to\mathsf{K}}\,([\![\textit{expression}]\!]_{exp\,[\tau']}\ e\ \rho\ \xi)\ *\ (\lambda d.$
$\quad checkBoolean_{\tau'}(d) \to [\![\textit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell,\ unit\ u)$

*(S6)* $\mathcal{P}[\![\texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{P}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S6)* $\mathcal{L}[\![\texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{L}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S6)* $\mathcal{C}[\![\texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{C}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

In the case of the *if* statement with no *else* clause, the definition of dynamic semantics is very simple. In the primary meaning, the condition is evaluated and, if it is true, the *then* clause is executed. Otherwise, nothing is done. In the alternative statements, the meaning of the *if* statement is trivially the meaning of its *then* clause.

IF-THEN-ELSE STATEMENT

*(S7)* $[\![\texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement}_1\ \texttt{else}\ \textit{statement}_2]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda\ell.$
$\quad lift_{\mathsf{G}\to\mathsf{K}}\,([\![\textit{expression}]\!]_{exp\,[\tau']}\ e\ \rho\ \xi)\ *\ (\lambda d.$
$\quad checkBoolean_{\tau'}(d) \to [\![\textit{statement}_1]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell,\ [\![\textit{statement}_2]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)$

*(S7)* $\mathcal{P}[\![\texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement}_1\ \texttt{else}\ \textit{statement}_2]\!]_{stmt\,[\tau]} = \lambda e.$
$\quad \mathcal{P}[\![\textit{statement}_1]\!]_{stmt\,[\tau]}\ e\ ;\ \mathcal{P}[\![\textit{statement}_2]\!]_{stmt\,[\tau]}\ e$

*(S7)* $\mathcal{L}[\![\texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement}_1\ \texttt{else}\ \textit{statement}_2]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda\rho.\,\lambda\xi.$
$\quad \mathcal{L}[\![\textit{statement}_1]\!]_{stmt\,[\tau]}\ e\ ;\ \mathcal{L}[\![\textit{statement}_2]\!]_{stmt\,[\tau]}\ e$

*(S7)* $\mathcal{C}[\![\texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement}_1\ \texttt{else}\ \textit{statement}_2]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda\ell.\,\lambda\tau_c.$
$\quad \mathcal{C}[\![\textit{statement}_1]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell\ \tau_c\ ;\ \mathcal{C}[\![\textit{statement}_2]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell\ \tau_c$

The case of *if* statements containing *else* clauses is slightly more complex. In the primary meaning, the *else* clause is executed if the condition fails. In other alternatives, the dynamic meanings of both clauses have to be appropriately combined.

SWITCH STATEMENT

*(S8)* $[\![\texttt{switch (}\ \textit{expression}\ \texttt{)}\ \textit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda\ell.$
$\quad lift_{\mathsf{G}\to\mathsf{K}}\,([\![\textit{expression}]\!]_{exp\,[\tau']}\ e\ \rho\ \xi)\ *\ (\lambda d.$
$\quad use\ (setBreak\ (\mathcal{C}[\![\textit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell\ \tau''\ \omega_\circ))\ (\lambda\omega.$
$\quad\quad getCase_{\tau,\tau''}\ (cast_{\tau'\mapsto\tau''}\ d)\ e\ \rho\ \omega))$

The primary dynamic meaning of the *switch* statement first evaluates the controlling expression. Following that, the case label environment for the body is calculated, starting with an empty environment. Execution depends on the value of the controlling expression. Notice that within the body the *break* statement completes the execution of the *switch* statement.

*(S8)* $\mathcal{P}[\![\texttt{switch (}\ \textit{expression}\ \texttt{)}\ \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{P}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S8)* $\mathcal{L}[\![\texttt{switch (}\ \textit{expression}\ \texttt{)}\ \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{L}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S8)* $\mathcal{C}[\![\texttt{switch (}\ \textit{expression}\ \texttt{)}\ \textit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\,\lambda\rho.\,\lambda\xi.\,\lambda\ell.\,\lambda\tau_c.\ unit$

Alternative meanings for the *switch* statement are easy to define. The only thing to notice is that its $\mathcal{C}$-meaning does not enter a new switch statement, when calculating case labels. Case labels are always associated with the smallest enclosing *switch* statement.

### 14.3.4 Labeled statements

CASE LABEL

*(S9)* $[\![\texttt{case}\ \textit{constant-expression} : \textit{statement}]\!]_{stmt\,[\tau]} = [\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S9)* $\mathcal{P}[\![\texttt{case}\ \textit{constant-expression} : \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{P}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S9)* $\mathcal{L}[\![\texttt{case}\ \textit{constant-expression} : \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{L}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S9)* $\mathcal{C}[\![\texttt{case}\ \textit{constant-expression} : \textit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, \lambda \omega.$
$\quad lift_{\mathsf{V}\to\mathsf{L}}\,([\![\textit{constant-expression}]\!]_{val\,[\tau']}\ e) * (unit \circ cast_{\tau'\mapsto\tau_c}) * (\lambda d.$
$\quad setCase_{\tau,\tau_c}\ d\ (\lambda e_c.\, \lambda \rho_c.\, [\![\textit{statement}]\!]_{stmt\,[\tau]}\ e_c\ \rho_c\ \xi\ \ell)\ \omega * (\lambda \omega'.$
$\quad \mathcal{C}[\![\textit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell\ \tau_c\ \omega'))$

The dynamic semantics of *case* labels is straightforward, except for the case of the $\mathcal{C}$-meaning. In this case, the label needs to be defined in the case label environment Otherwise, *case* labels are ignored.

DEFAULT LABEL

*(S10)* $[\![\texttt{default} : \textit{statement}]\!]_{stmt\,[\tau]} = [\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S10)* $\mathcal{P}[\![\texttt{default} : \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{P}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S10)* $\mathcal{L}[\![\texttt{default} : \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{L}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S10)* $\mathcal{C}[\![\texttt{default} : \textit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, \lambda \omega.$
$\quad setDefault_{\tau,\tau_c}\ (\lambda e_c.\, \lambda \rho_c.\, [\![\textit{statement}]\!]_{stmt\,[\tau]}\ e_c\ \rho_c\ \xi\ \ell)\ \omega * (\lambda \omega'.$
$\quad \mathcal{C}[\![\textit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell\ \tau_c\ \omega')$

In a similar way, the dynamic semantics of *default* labels is straightforward. In the case of the $\mathcal{C}$-meaning, the label again needs to be defined in the case label environment. Otherwise, *default* labels are ignored.

IDENTIFIER LABELS

*(S11)* $[\![I : \textit{statement}]\!]_{stmt\,[\tau]} = [\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S11)* $\mathcal{P}[\![I : \textit{statement}]\!]_{stmt\,[\tau]} = \mathcal{P}[\![\textit{statement}]\!]_{stmt\,[\tau]}$

*(S11)* $\mathcal{L}[\![I : \textit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.$
$\quad setLabel\ I\ (\lambda e_c.\, \lambda \rho_c.\, [\![\textit{statement}]\!]_{stmt\,[\tau]}\ e_c\ \rho_c\ \xi\ \ell)\ \ell * (\lambda \ell'.$
$\quad \mathcal{L}[\![\textit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell')$

*(S11)* $\mathcal{C}[\![I : \textit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, \lambda \omega.\, [\![\textit{statement}]\!]_{stmt\,[\tau]}$

In the case of the dynamic semantics for identifier labels, the focus of attention is on the $\mathcal{L}$-meaning, where the identifier label must be defined. Otherwise, identifier labels are simply ignored.

### 14.3.5 Iteration statements

The semantics of iteration statements is defined by using the auxiliary function $loop_{\tau,\tau'}$, where the parameter $\tau$ determines the data type of the result that can be returned by *return* statements in the loop's body and parameter $\tau'$ determines the data type of the loop's condition. The function takes three parameters: an expression computation that corresponds to the loop's condition, a statement computation that corresponds to the loop's body and an expression computation that corresponds to additional code that must be executed between successive iterations of the loop. The result is the statement computation for the whole loop.

▶ $loop_{\tau,\tau'} : \mathsf{G}([\![\tau']\!]_{dat}) \to \mathsf{K}_\tau(\mathbf{U}) \to \mathsf{G}(\mathbf{U}) \to \mathsf{K}_\tau(\mathbf{U})$

$loop_{\tau,\tau'} = \lambda g_c.\, \lambda k_b.\, \lambda g_s.\, fix\ (\lambda k.$
$\quad lift_{\mathsf{G}\to\mathsf{K}}\ g_c * (\lambda d.$
$\quad checkBoolean_{\tau'}(d) \to setBreak\ (follow\ (setContinue\ k_b)\ (lift_{\mathsf{G}\to\mathsf{K}}\ g_s * (\lambda u.\, k))),\ unit\ u))$

The least fixed point operator is used for the definition of the loop's dynamic semantics. The loop's condition is first evaluated. If it is true, then the body is executed, followed by the additional code, and the loop is repeated. Otherwise, nothing is done. Inside the loop's body, statements *break* and *continue* may alter the normal order of execution.

WHILE STATEMENT

*(S12)* $[\![\texttt{while ( } \mathit{expression} \texttt{ ) } \mathit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.$
$\quad loop_{\tau,\tau'}\ ([\![\mathit{expression}]\!]_{exp\,[\tau']}\ e\ \rho\ \xi)\ ([\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)\ (unit\ u)$

*(S12)* $\mathcal{P}[\![\texttt{while ( } \mathit{expression} \texttt{ ) } \mathit{statement}]\!]_{stmt\,[\tau]} = \mathcal{P}[\![\mathit{statement}]\!]_{stmt\,[\tau]}$

*(S12)* $\mathcal{L}[\![\texttt{while ( } \mathit{expression} \texttt{ ) } \mathit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.$
$\quad \textbf{let}\ k = loop_{\tau,\tau'}\ ([\![\mathit{expression}]\!]_{exp\,[\tau']}\ e\ \rho\ \xi)\ ([\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)\ (unit\ u)$
$\quad \textbf{in}\ \mathit{follow}\ (\mathcal{L}[\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)\ k$

*(S12)* $\mathcal{C}[\![\texttt{while ( } \mathit{expression} \texttt{ ) } \mathit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, \lambda \omega.$
$\quad \textbf{let}\ k = loop_{\tau,\tau'}\ ([\![\mathit{expression}]\!]_{exp\,[\tau']}\ e\ \rho\ \xi)\ ([\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)\ (unit\ u)$
$\quad \textbf{in}\ \mathit{follow}\ (\mathcal{C}[\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell\ \tau_c\ \omega)\ k$

The simplest type of loop is the *while* statement, which is directly defined using the $loop_{\tau,\tau'}$ function. No additional code is required to be executed between iterations. In the $\mathcal{L}$ and $\mathcal{C}$-meanings, execution of the loop's body is followed by a repetition of the loop.

DO-WHILE STATEMENT

*(S13)* $[\![\texttt{do } \mathit{statement} \texttt{ while ( } \mathit{expression} \texttt{ ) ; }]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.$
$\quad \textbf{let}\ k_b = [\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell$
$\qquad k = loop_{\tau,\tau'}\ ([\![\mathit{expression}]\!]_{exp\,[\tau']}\ e\ \rho\ \xi)\ k_b\ (unit\ u)$
$\quad \textbf{in}\ setBreak\ (\mathit{follow}\ (setContinue\ k_b)\ k)$

*(S13)* $\mathcal{P}[\![\texttt{do } \mathit{statement} \texttt{ while ( } \mathit{expression} \texttt{ ) ; }]\!]_{stmt\,[\tau]} = \mathcal{P}[\![\mathit{statement}]\!]_{stmt\,[\tau]}$

*(S13)* $\mathcal{L}[\![\texttt{do } \mathit{statement} \texttt{ while ( } \mathit{expression} \texttt{ ) ; }]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.$
$\quad \textbf{let}\ k = loop_{\tau,\tau'}\ ([\![\mathit{expression}]\!]_{exp\,[\tau']}\ e\ \rho\ \xi)\ ([\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)\ (unit\ u)$
$\quad \textbf{in}\ \mathit{follow}\ (\mathcal{L}[\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)\ k$

*(S13)* $\mathcal{C}[\![\texttt{do } \mathit{statement} \texttt{ while ( } \mathit{expression} \texttt{ ) ; }]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, \lambda \omega.$
$\quad \textbf{let}\ k = loop_{\tau,\tau'}\ ([\![\mathit{expression}]\!]_{exp\,[\tau']}\ e\ \rho\ \xi)\ ([\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)\ (unit\ u)$
$\quad \textbf{in}\ \mathit{follow}\ (\mathcal{C}[\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell\ \tau_c\ \omega)\ k$

In the dynamic semantics for *do* statements the loop's body is executed first, making the necessary adjustments for the *break* and *continue* statements. Then, the full loop is executed in the way defined by function $loop_{\tau,\tau'}$. No additional code is required to be executed between iterations.

FOR STATEMENT

*(S14)* $[\![\texttt{for ( } \mathit{expr\text{-}opt}_1 \texttt{ ; } \mathit{expr\text{-}opt}_2 \texttt{ ; } \mathit{expr\text{-}opt}_3 \texttt{ ) } \mathit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.$
$\quad lift_{\mathtt{G}\to\mathtt{K}}\ ([\![\mathit{expr\text{-}opt}_1]\!]_{exp\,[\tau_1]}\ e\ \rho\ \xi)\ *\ (\lambda d.$
$\quad loop_{\tau,\tau'}\ ([\![\mathit{expr\text{-}opt}_2]\!]_{exp\,[\tau_2]}\ e\ \rho\ \xi)\ ([\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)$
$\qquad ([\![\mathit{expr\text{-}opt}_3]\!]_{exp\,[\tau_3]}\ e\ \rho\ \xi\ *\ (\lambda d.\ unit\ u)))$

*(S14)* $\mathcal{P}[\![\texttt{for ( } \mathit{expr\text{-}opt}_1 \texttt{ ; } \mathit{expr\text{-}opt}_2 \texttt{ ; } \mathit{expr\text{-}opt}_3 \texttt{ ) } \mathit{statement}]\!]_{stmt\,[\tau]} = \mathcal{P}[\![\mathit{statement}]\!]_{stmt\,[\tau]}$

*(S14)* $\mathcal{L}[\![\texttt{for ( } \mathit{expr\text{-}opt}_1 \texttt{ ; } \mathit{expr\text{-}opt}_2 \texttt{ ; } \mathit{expr\text{-}opt}_3 \texttt{ ) } \mathit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.$
$\quad \textbf{let}\ k = loop_{\tau,\tau'}\ ([\![\mathit{expr\text{-}opt}_2]\!]_{exp\,[\tau_2]}\ e\ \rho\ \xi)\ ([\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)$
$\qquad\qquad ([\![\mathit{expr\text{-}opt}_3]\!]_{exp\,[\tau_3]}\ e\ \rho\ \xi\ *\ (\lambda d.\ unit\ u)))$
$\qquad k' = lift_{\mathtt{G}\to\mathtt{K}}\ ([\![\mathit{expr\text{-}opt}_3]\!]_{exp\,[\tau_3]}\ e\ \rho\ \xi\ *\ (\lambda d.\ k))$
$\quad \textbf{in}\ \mathit{follow}\ (\mathcal{L}[\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)\ k'$

*(S14)* $\mathcal{C}[\![\texttt{for ( } \mathit{expr\text{-}opt}_1 \texttt{ ; } \mathit{expr\text{-}opt}_2 \texttt{ ; } \mathit{expr\text{-}opt}_3 \texttt{ ) } \mathit{statement}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, \lambda \omega.$
$\quad \textbf{let}\ k = loop_{\tau,\tau'}\ ([\![\mathit{expr\text{-}opt}_2]\!]_{exp\,[\tau_2]}\ e\ \rho\ \xi)\ ([\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell)$
$\qquad\qquad ([\![\mathit{expr\text{-}opt}_3]\!]_{exp\,[\tau_3]}\ e\ \rho\ \xi\ *\ (\lambda d.\ unit\ u)))$
$\qquad k' = lift_{\mathtt{G}\to\mathtt{K}}\ ([\![\mathit{expr\text{-}opt}_3]\!]_{exp\,[\tau_3]}\ e\ \rho\ \xi\ *\ (\lambda d.\ k))$
$\quad \textbf{in}\ \mathit{follow}\ (\mathcal{C}[\![\mathit{statement}]\!]_{stmt\,[\tau]}\ e\ \rho\ \xi\ \ell\ \tau_c\ \omega)\ k'$

Finally, in the case of the *for* statement, function $loop_{\tau,\tau'}$ is again used, this time with additional code between successive iterations. In the $\mathcal{L}$ and $\mathcal{C}$ meanings, the loop's body is followed by the execution of the additional code and a new iteration of the loop.

### 14.3.6 Jump statements

*(S16)* $[\![\texttt{break ;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, getBreak\ e\ \rho$ — BREAK STATEMENT

*(S16)* $\mathcal{P}[\![\texttt{break ;}]\!]_{stmt\,[\tau]} = \lambda e.\, unit$

*(S16)* $\mathcal{L}[\![\texttt{break ;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, unit$

*(S16)* $\mathcal{C}[\![\texttt{break ;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, unit$

*(S15)* $[\![\texttt{continue ;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, getContinue\ e\ \rho$ — CONTINUE STATEMENT

*(S15)* $\mathcal{P}[\![\texttt{continue ;}]\!]_{stmt\,[\tau]} = \lambda e.\, unit$

*(S15)* $\mathcal{L}[\![\texttt{continue ;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, unit$

*(S15)* $\mathcal{C}[\![\texttt{continue ;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, unit$

The dynamic semantics of the *break* and *continue* statements is very simple. The appropriate continuation is taken from the statement computation.

*(S17)* $[\![\texttt{goto}\ I\ \texttt{;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, getLabel\ I\ e\ \rho\ \ell$ — GOTO STATEMENT

*(S17)* $\mathcal{P}[\![\texttt{goto}\ I\ \texttt{;}]\!]_{stmt\,[\tau]} = \lambda e.\, unit$

*(S17)* $\mathcal{L}[\![\texttt{goto}\ I\ \texttt{;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, unit$

*(S17)* $\mathcal{C}[\![\texttt{goto}\ I\ \texttt{;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, unit$

The semantics of the *goto* statement is also easy. The appropriate continuation is found from the given label environment.

*(S18)* $[\![\texttt{return ;}]\!]_{stmt\,[void]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, result_{void}\ e\ \rho\ u$ — RETURN STATEMENT

*(S18)* $[\![\texttt{return ;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, result_{\tau}\ e\ \rho\ \top$

*(S18)* $\mathcal{P}[\![\texttt{return ;}]\!]_{stmt\,[\tau]} = \lambda e.\, unit$

*(S18)* $\mathcal{L}[\![\texttt{return ;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, unit$

*(S18)* $\mathcal{C}[\![\texttt{return ;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, unit$

*(S19)* $[\![\texttt{return}\ expression\ \texttt{;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, lift_{\mathsf{G} \to \mathsf{K}}(\mathcal{A}[\![expression]\!]_{exp\,[\tau]}\ e\ \rho\ \xi)\ *\ (result_{\tau}\ e\ \rho)$

*(S19)* $\mathcal{P}[\![\texttt{return}\ expression\ \texttt{;}]\!]_{stmt\,[\tau]} = \lambda e.\, unit$

*(S19)* $\mathcal{L}[\![\texttt{return}\ expression\ \texttt{;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, unit$

*(S19)* $\mathcal{C}[\![\texttt{return}\ expression\ \texttt{;}]\!]_{stmt\,[\tau]} = \lambda e.\, \lambda \rho.\, \lambda \xi.\, \lambda \ell.\, \lambda \tau_c.\, unit$

The semantics of the two flavours of the *return* statement is again straightforward. In the case of a simple return, a distinction is made between functions whose return types are *void* and other functions. For the former, the value $u$ is returned as the function's result. For the latter, the erroneous value $\top$ is returned. If this value is used in the calling function, an error occurs. In the case of returned expressions, conversions as if by assignment apply.

## 14.4 Optional expressions

*(S20)* $[\![\epsilon]\!]_{val\,[int]} = \lambda e.\, \mathcal{C}[\![\texttt{1}]\!]_{int}$ — MISSING EXPRESSION

The dynamic semantics of a missing optional expression is the same as if an integer constant was present, with the value 1.

# Part V

# Epilogue

# Chapter 15

# Implementation

CHAPTER OVERVIEW

This chapter investigates possible ways to implement a denotational semantics, using a general purpose programming language. In Section 15.1 the problem is discussed and the denotational semantics of a small but purposefully complex functional programming language is defined and used as an example in the rest of the chapter. Section 15.2 explores the advantages and disadvantages of the functional programming paradigm for the given task, using the languages Standard ML and Haskell, whereas Section 15.3 does the same for the object-oriented programming paradigm, using the language C++. Finally, in Section 15.4 an implementation of the proposed semantics for the C programming language is outlined, using Haskell as the target language.

## 15.1 Definition of the problem

IMPLEMENTATION OF SEMANTICS

Although not in the spirit of operational semantics, a denotational description of a programming language defines directly an execution model, that is, an abstract interpreter for the programming language. This execution model can be implemented by translating the denotational semantics to a program, written in some target language. It is not always easy to implement such an interpreter by using a general purpose programming language as the target. It is much harder to implement an efficient interpreter, but this subject is not touched in this thesis. Using an appropriate target language is very important and can greatly reduce the complexity of this task. Several languages have been suggested and used for this purpose, with more or less success. It seems that typed functional programming languages are more suitable, as suggested in [Watt86], where ML is used as a meta-language for denotational descriptions. Non-strict functional languages like Haskell also present significant advantages. Imperative programming languages have also been suggested, such as Algol 68 [Paga79] and Pascal [Alli83, Alli85], with considerably less success. A more thorough presentation of related work is made in Section 16.4.

In order to formulate a denotational description, a meta-language has to be employed. A variation of the typed $\lambda$-calculus over Scott domains has been used in this thesis for defining the semantics of C, and the same meta-language is considered in this chapter. Regardless of the target language, the implementation of a denotational description requires a means of translating abstract syntax, domain definitions and semantic equations into code. A complete interpreter would also require a parser and a translator from concrete to abstract syntax, but these are not covered in this chapter.

EXAMPLE

For simplicity reasons, a language simpler than C is used as an example in this chapter. This language is purely functional and features high-order functions, recursion by means of a `fix` operator, advanced control constructs, such as `abort`, `call/cc` and a control delimiter or *prompt*, written as #. It is called PFLC (Pure Functional Language with Control). The complexity of this exam-

ple serves the purpose of using as many elements of the meta-language as possible, in its semantic description.

ABSTRACT SYNTAX

The abstract syntax of PFLC is defined as follows. A program is just an expression that must be evaluated. Expressions are integer or boolean constants, bound identifiers, conditionals, applications of binary or unary operators, lambda abstractions, function applications, fixed points or applications of control operators. A variety of binary and unary operators is supported.

- *program* ::= *expression*
- *expression* ::= *n* | `true` | `false` | *I* | `if` *expression* `then` *expression* `else` *expression*
  | *expression binary-operator expression* | *unary-operator expression*
  | `lambda` *I* `.` *expression* | *expression expression* | `fix` *I* `.` *expression*
  | `abort` *expression* | `call/cc` *expression* | `#` *expression*
- *binary-operator* ::= `+` | `-` | `*` | `/` | `=` | | `<>` | `<` | `>` | `<=` | `>=` | `and` | `or`
- *unary-operator* ::= `-` | `not`

CONTROL OPERATORS

All control operators have been inspired from the work of Sitaram and are described in detail in [Sita90]. A brief description is attempted here. An evaluation context can be understood as an expression with a hole in it; the value of the whole context depends on the value that is placed in the hole. At any time during the execution of a program, there is stack of evaluation contexts and an expression that is currently evaluated. The context that is at the top of the stack is called the *current evaluation context* and the result of the currently evaluated expression is the value placed its hole. In the beginning of the program, the stack contains just one context which consists of a single hole; the currently evaluated expression is the whole program.

Expression "`abort` *E*" aborts the evaluation of the current context and replaces it with expression *E*. As an example, consider the function *F* defined by the term:

```
lambda x. lambda y. 7 + (if y=0 then abort 42 else x+y)
```

With this in mind, expression "*F* 1 2" results in the value 10, i.e. $7+1+2$, whereas expression "*F* 1 0" results in the value 42, since `abort` is triggered.

In expression "`call/cc` *E*", it is expected that *E* is a function, taking as argument an abstracted form of the surrounding evaluation context, i.e. a *continuation.* It leaves the evaluation context intact. On application, this continuation replaces the current evaluation context. Consider the following example, in which function *G* is defined by the term:

```
lambda n. 2 + call/cc (lambda c. n + (if n=0 then c 40 else n))
```

In this example, expression "*G* 1" gives the result 4, i.e. $2 + 1 + 1$, whereas expression "*G* 0" gives the result 42, i.e. $2 + 40$. In the second case, the context *c* surrounding the `call/cc` operator is used with the value 40 in its hole.

Finally, the control delimiter constraints control manipulation by restricting the current evaluation context. When expression "# *E*" is evaluated, a new evaluation context is pushed to the stack, consisting of a single hole. Apart from that, evaluation of *E* continues as usual. The effect of the control delimiter is illustrated by the following example. Consider the function *H* defined below, which is identical to the function *F* in the example for *abort*, except for the prompt:

```
lambda x. lambda y. 7 + #(if y=0 then abort 42 else x+y)
```

In this case however, evaluation of "*H* 1 0" results in the value 49, i.e. $7 + 42$, because the abort operation was restricted by the control delimiter to the interior of the parentheses.

**Figure 15.1:** The denotational semantics of PFLC.

▶ $\mathcal{P}[\![\mathit{program}]\!] \;:\; \mathbf{V}$

$\mathcal{P}[\![\mathit{expression}]\!] \;=\; \mathcal{E}[\![\mathit{expression}]\!]\;(\lambda\, I.\; \top_{\mathbf{V}})\;\mathit{id}$

▶ $\mathcal{E}[\![\mathit{expression}]\!] \;:\; \mathbf{Env} \to \mathbf{K} \to \mathbf{V}$

$\mathcal{E}[\![n]\!]\;\rho\;\kappa \;=\; \kappa\; n$

$\mathcal{E}[\![\texttt{true}]\!]\;\rho\;\kappa \;=\; \kappa\; \mathit{true}$

$\mathcal{E}[\![\texttt{false}]\!]\;\rho\;\kappa \;=\; \kappa\; \mathit{false}$

$\mathcal{E}[\![I]\!]\;\rho\;\kappa \;=\; \kappa\;(\rho\; I)$

$\mathcal{E}[\![\texttt{if}\ \mathit{expression}\ \texttt{then}\ \mathit{expression}_1\ \texttt{else}\ \mathit{expression}_2]\!]\;\rho\;\kappa \;=$
$\quad \mathcal{E}[\![\mathit{expression}]\!]\;\rho\;(\lambda\, e.\; (e\,|\,\mathbf{T}) \to \mathcal{E}[\![\mathit{expression}_1]\!]\;\rho\;\kappa\,,\; \mathcal{E}[\![\mathit{expression}_2]\!]\;\rho\;\kappa)$

$\mathcal{E}[\![\mathit{expression}_1\ \mathit{binary\text{-}operator}\ \mathit{expression}_2]\!]\;\rho\;\kappa \;=$
$\quad \mathcal{B}[\![\mathit{binary\text{-}operator}]\!]\;(\mathcal{E}[\![\mathit{expression}_1]\!]\;\rho)\;(\mathcal{E}[\![\mathit{expression}_2]\!]\;\rho)\;\kappa$

$\mathcal{E}[\![\mathit{unary\text{-}operator}\ \mathit{expression}]\!]\;\rho\;\kappa \;=\; \mathcal{U}[\![\mathit{unary\text{-}operator}]\!]\;(\mathcal{E}[\![\mathit{expression}]\!]\;\rho)\;\kappa$

$\mathcal{E}[\![\texttt{lambda}\ I\ \texttt{.}\ \mathit{expression}]\!]\;\rho\;\kappa \;=\; \kappa\;(\phi\;(\lambda\, p.\; \mathcal{E}[\![\mathit{expression}]\!]\;\rho\{I \mapsto p\}))$

$\mathcal{E}[\![\mathit{expression}_1\ \mathit{expression}_2]\!]\;\rho\;\kappa \;=\; \mathcal{E}[\![\mathit{expression}_1]\!]\;\rho\;(\lambda\, e_1.\; \mathcal{E}[\![\mathit{expression}_2]\!]\;\rho\;(\lambda\, e_2.\; \delta\; e_1\; e_2\; \kappa))$

$\mathcal{E}[\![\texttt{fix}\ I\ \texttt{.}\ \mathit{expression}]\!]\;\rho\;\kappa \;=\; \kappa\;(\phi\;(\mathit{fix}\;(\lambda\, w.\; \delta\;(\mathcal{E}[\![\mathit{expression}]\!]\;\rho\{I \mapsto \phi\; w\}\;\mathit{id}))))$

$\mathcal{E}[\![\texttt{abort}\ \mathit{expression}]\!]\;\rho\;\kappa \;=\; \mathcal{E}[\![\mathit{expression}]\!]\;\rho\;\mathit{id}$

$\mathcal{E}[\![\texttt{call/cc}\ \mathit{expression}]\!]\;\rho\;\kappa \;=\; \mathcal{E}[\![\mathit{expression}]\!]\;\rho\;(\lambda\, e.\; \delta\; e\;(\phi\;(\lambda\, p.\; \lambda\, \kappa'.\; \kappa\; p))\;\kappa)$

$\mathcal{E}[\![\#\ \mathit{expression}]\!]\;\rho\;\kappa \;=\; \kappa\;(\mathcal{E}[\![\mathit{expression}]\!]\;\rho\;\mathit{id})$

▶ $\mathcal{B}[\![\mathit{binary\text{-}operator}]\!] \;:\; (\mathbf{K} \to \mathbf{V}) \to (\mathbf{K} \to \mathbf{V}) \to \mathbf{K} \to \mathbf{V}$

$\mathcal{B}[\![+]\!]\;f_1\;f_2\;\kappa \;=\; f_1\;(\lambda\, e_1.\; f_2\;(\lambda\, e_2.\; \kappa\;((e_1\,|\,\mathbf{N}) + (e_2\,|\,\mathbf{N}))))$

*etc.*

▶ $\mathcal{U}[\![\mathit{unary\text{-}operator}]\!] \;:\; (\mathbf{K} \to \mathbf{V}) \to \mathbf{K} \to \mathbf{V}$

$\mathcal{U}[\![\texttt{-}]\!]\;f\;\kappa \;=\; f\;(\lambda\, e.\; \kappa\;(-(e\,|\,\mathbf{N})))$

*etc.*

▶ $\phi \;:\; (\mathbf{V} \to \mathbf{K} \to \mathbf{V}) \to \mathbf{V}$

$\phi\; f \;=\; (\mathit{up} \circ \mathit{strict})\; f$

▶ $\delta \;:\; \mathbf{V} \to (\mathbf{V} \to_s \mathbf{K} \to \mathbf{V})$

$\delta\; e \;=\; \mathit{down}\;(e\,|\,\mathbf{F})$

SEMANTICS

The following domains are used in the definition of the semantics of PFLC. The primitive domains $\mathbf{N}$, $\mathbf{T}$ and $\mathbf{Ide}$ represent integer numbers, truth values and identifiers respectively. PFLC is an untyped language. Values are distinguished in basic values, i.e. integers and booleans, and functions. The environment is a simple mapping from identifiers to values.

| | | | | | |
|---|---|---|---|---|---|
| ■ | | $\mathbf{B}$ | $=$ | $\mathbf{N} \oplus \mathbf{T}$ | *(basic values)* |
| ■ | | $\mathbf{F}$ | $=$ | $(\mathbf{V} \to_s \mathbf{K} \to \mathbf{V})_\perp$ | *(function values)* |
| ■ | | $\mathbf{V}$ | $=$ | $\mathbf{B} \oplus \mathbf{F}$ | *(values)* |
| ■ | $\rho$ : | $\mathbf{Env}$ | $=$ | $\mathbf{Ide} \to \mathbf{V}$ | *(environments)* |
| ■ | $\kappa$ : | $\mathbf{K}$ | $=$ | $\mathbf{V} \to \mathbf{V}$ | *(continuations)* |

The semantic functions for PFLC are defined in Figure 15.1, where $n : \mathbf{N}$ and $I : \mathbf{Ide}$. A simple continuation semantics is used. The auxiliary functions $\phi$ and $\delta$ facilitate the management of function values. The semantics itself is not further discussed here, since it is only given as an example. The reader is referred to [Moss90] for an introduction to the denotational semantics of purely functional languages, and to [Sita90] for the semantics of control operators.

## 15.2 The functional programming paradigm

THE DIRECT APPROACH

Typed functional programming languages present many similarities with the meta-languages that are used in specifying denotational semantics. Functions, products and sums are directly supported in most typed functional languages and the least fixed point operator can be rather easily implemented. Bottom and top elements need special treatment. However, bottom elements can simply be omitted if they represent non-termination, as is usually the case. The correct representation of non-termination is a problem that cannot be solved in any implementation of denotational semantics: the interpreter will simply not terminate in the case of non-termination... Moreover, top elements modelling error conditions can be represented by exceptions, which are supported in most typed functional programming languages. The abstract syntax can usually be represented very naturally by appropriate data types. With all this in mind, a direct translation from a denotational semantics to a functional program is not only feasible, but relatively easy. For moderately sized languages, this is the most efficient way to create a rapid prototype of an interpreter. The approach that was sketched above is discussed in Section 15.2.1, using Standard ML as the target language.

THE STRUCTURED APPROACH

As the size of the semantic description grows, the disadvantages of the direct translation approach multiply. The main problems are that the resulting functional program is not well-structured and that the details of the translation, which must be repeated over and over, are easily forgotten. For these reasons, a second more well-structured approach is suggested for large semantic descriptions. This approach exploits features of the target language such as modules, signatures, functors or type classes, which improve the modularity of the resulting functional program. Furthermore, the mathematical objects which participate in the semantics are modelled in a consistent way and thus the development of the functional program is facilitated. This second approach is further discussed in Section 15.2.2, using Haskell as the target language.

### 15.2.1 Standard ML

THE LANGUAGE

*Standard ML* is a strongly-typed eager functional language [Harp86, Harp89, Miln90, Miln91]. In a relatively short time, it has become remarkably popular in the functional programming community and a number of very good compilers are currently available. Standard ML features a static polymorphic type system, exceptions, mutable variables and arrays, modules and abstract types. In this section a direct translation of the semantics of PFLC to a Standard ML program will be attempted. The structured approach could also have been used.

OVERVIEW

The direct translation of the semantics of PFLC results in a Standard ML program of approximately 200 lines. Nearly 40 lines were required for the implementation of the abstract syntax and 150 lines for that of the semantics. The implementation of the semantics was relatively straightforward and its size is surprisingly small, considering that the implementation of a pretty-printer for the abstract syntax required an additional 100 lines of code.

ABSTRACT SYNTAX

The implementation of the abstract syntax was completely straightforward. The following data types were defined for the abstract syntax of expressions. All constructors end with a prime symbol, in order to be distinguished from type names.

```
datatype Expr =
    Int' of int
  | True'
  | False'
```

```
  | Ident' of Ide
  | BinOp' of Expr * BinOp * Expr
  | UnOp' of UnOp * Expr
  | If' of Expr * Expr * Expr
  | Lambda' of Ide * Expr
  | App' of Expr * Expr
  | Fix' of Ide * Expr
  | Abort' of Expr
  | CallCC' of Expr
  | Prompt' of Expr
```

SEMANTIC DOMAINS

The definition of semantic domains presents some difficulties. The main problem is that the definition of types in Standard ML does not allow mutually recursive definitions. Thus, mutual recursion had to be eliminated. Bottom and top elements were omitted from semantic domains, and lifted domains were not used. An exception was defined for the representation of all top elements.

```
type T = bool
type N = int

datatype B = Number' of N | Boolean' of T

datatype V = Basic' of B | Function' of V -> (V -> V) -> V

type K   = V -> V
type F   = V -> K -> V
type Env = Ide -> V

exception TOP
```

LEAST FIXED POINTS

The least fixed point operator cannot be implemented in Standard ML using its equational property. The following definition leads to non-termination every time `fix` is called, due to Standard ML's eager evaluation:

```
fun fix f = f (fix f)
```

Unfortunately, it does not seem possible to implement a least fixed point operator of the polymorphic type $(A \to A) \to A$. However, it is possible to construct such an operator of the more specific type $((A \to B) \to (A \to B)) \to A \to B$, and this is adequate for the semantics of PFLC. The straigthforward approach is:

```
fun fix f x = f (fix f) x
```

but the following version suggested in [Gunt92] can also be used:

```
local datatype 'a fix = FUN of 'a fix -> 'a
      fun NUF (FUN x) = x
in    fun fix f =
        (fn x => (fn y => f((NUF x) x) y))
          (FUN (fn x => (fn y => f((NUF x) x) y)))
end
```

In case the more general type of the least fixed point operator is required, the direct translation approach is not possible and a different way of circumventing this problem must be found.

SEMANTICS The translation of the semantic equations is again straightforward. A small excerpt is given below. The resemblance between this code and Figure 15.1 is striking.

```
fun semP E = semE E (fn I => raise TOP) (fn e => e)

and semE (Int'(n)) rho kappa = kappa (Basic' (Number' n))
  | semE True' rho kappa = kappa (Basic' (Boolean' true))
  | semE (Ident'(I)) rho kappa = kappa (rho I)
  | semE (If'(E, E1, E2)) rho kappa =
    semE E rho (fn Basic' (Boolean' t) =>
                if t then semE E1 rho kappa else semE E2 rho kappa)
  | semE (App'(E1, E2)) rho kappa =
    semE E1 rho (fn e1 => semE E2 rho (fn e2 => delta e1 e2 kappa))
  | semE (Abort'(E)) rho kappa = semE E rho (fn e => e)
  | semE (CallCC'(E)) rho kappa =
    semE E rho (fn e => delta e (phi (fn p => fn kappa' => kappa p)) kappa)
  | ...

and phi f = Function' f

and delta (Function' f) = f
  | delta _ = raise TOP
```

EXAMPLE In order to use the implemented semantics, one needs only to write a program like the following, using an implementation of Standard ML:[1]

```
(*
 * fact = fix (lambda f. lambda n. if n = 0 then 1 else n * f(n-1))
 *)

val fact = Fix'("f", Lambda'("n",
              If'(BinOp'(Ident'("n"), Equals', Int'(0)),
                 Int'(1),
                 BinOp'(Ident'("n"), Times',
                    App'(Ident'("f"), BinOp'(Ident'("n"), Minus', Int'(1)))))))


(*
 *  Evaluate the expression "fact 7" using the semantics of PFLC
 *)

val main = semP (App'(fact, Int'(7)))
```

This program calculates the factorial of 7, using a recursive definition. The result of the evaluation of "`main`" is the value "`Basic' (Number' 5040)`".

### 15.2.2 Haskell and related languages

THE LANGUAGE *Haskell* belongs to the family of strongly-typed lazy purely functional languages [Huda96, Pete97]. Its popularity increases steadily and there are currently many efficient compilers. Haskell provides non-strict semantics, a static polymorphic type system, algebraic data types, modules, monads and monadic I/O and a rich system of primitive data types. A variation of Haskell is *Gofer*, which extends

[1] The Standard ML of New Jersey, available from `http://cm.bell-labs.com/cm/cs/what/smlnj/`, provides a very good compiler and interpreter for Standard ML. Other good implementations also exist.

the language by adding type classes with multiple parameters but does not support modules [Jone94]. In this section, the structured approach towards the implementation of the semantics of PFLC is presented. Of course, the direct approach could also have been used, resulting in a much smaller and simpler Haskell program of approximately 350 lines (30 for the syntax, 200 lines for the semantics and 120 lines for parsing and pretty-printing).

OVERVIEW

The structured translation of the semantics of PFLC is a program of approximately 470 lines. The implementation of the abstract syntax requires 35 lines, that of the semantics requires 135 lines and 400 lines are needed for the implementation of the framework for domains. The basic class defined in this framework is the `Domain` class:

```
class Domain a where
   (<<=) :: a -> a -> Bool
   isBottom :: a -> Bool
   bottom   :: a
   isTop    :: a -> Bool
   top      :: a
```

where operator `<<=` represents the domain ordering. Notice that for bottom and top elements two members are needed: one for accessing these elements and one for checking if a given element is bottom or top. This is necessary since in general Haskell types do not provide an equality operator.

DOMAIN CONSTRUCTORS

Domain constructors produce instances of the `Domain` class, like the ones defined below. Notice that domain ordering for functions is not computable and that flat domains are based on equality types.

```
-- Function domains

instance (Domain a, Domain b) => Domain (a -> b) where
   f <<= g     = error "Cannot compare functions with <<="
   isBottom f  = error "Cannot compare function to bottom"
   bottom      = \a -> bottom
   isTop f     = error "Cannot compare function to top"
   top         = \a -> top

-- Product domains

instance (Domain a, Domain b) => Domain (a, b) where
   (x1, x2) <<= (y1, y2)  = (x1 <<= y1) && (x2 <<= y2)
   isBottom (x, y) = isBottom x && isBottom y
   bottom = (bottom, bottom)
   isTop (x, y) = isTop x && isTop y
   top = (top, top)

-- Flat domains

data (Eq a) => FlatDomain a = FlatBottom | FlatElement a | FlatTop

instance (Eq a) => Domain (FlatDomain a) where
   FlatBottom      <<= _               = True
   (FlatElement x) <<= (FlatElement y)  = x == y
   _               <<= FlatTop         = True
   _               <<= _               = False
   isBottom FlatBottom = True
   isBottom _          = False
   bottom              = FlatBottom
   isTop FlatTop       = True
```

```
   isTop _             = False
   top                 = FlatTop

-- Coalesced sums

data SumDomain a b = SumBottom | SumLeft a | SumRight b | SumTop

instance (Domain a, Domain b) => Domain (SumDomain a b) where
   SumBottom    <<= _             = True
   (SumLeft x)  <<= (SumLeft y)   = x <<= y
   (SumRight x) <<= (SumRight y)  = x <<= y
   _            <<= SumTop        = True
   _            <<= _             = False
   isBottom SumBottom = True
   isBottom _         = False
   bottom             = SumBottom
   isTop SumTop       = True
   isTop _            = False
   top                = SumTop
```

SPECIAL OPERATIONS

Special operations for some types of domains may also be required, as in the case of coalesced sums. Functions `inl` and `inr` are used to insert of elements in the coalesced sums, whereas functions `outl` and `outr` are used to extract elements.

```
inl :: (Domain a, Domain b) => a -> SumDomain a b
inl x = if isBottom x then SumBottom else
        if isTop x then SumTop else SumLeft x

inr :: (Domain a, Domain b) => b -> SumDomain a b
inr x = if isBottom x then SumBottom else
        if isTop x then SumTop else SumRight x

outl :: (Domain a, Domain b) => SumDomain a b -> a
outl (SumLeft x)  = x
outl (SumRight x) = top
outl SumBottom    = bottom
outl SumTop       = top

outr :: (Domain a, Domain b) => SumDomain a b -> b
outr (SumLeft x)  = top
outr (SumRight x) = x
outr SumBottom    = bottom
outr SumTop       = top
```

LEAST FIXED POINTS

Because Haskell is a lazy language, the least fixed point operator can be defined in a very simple way, according to its equational property. Its type is the most general that can be achieved. The requirement "`Domain a`" is not really necessary.

```
fix :: (Domain a) => (a -> a) -> a
fix f = f (fix f)
```

ABSTRACT SYNTAX

The abstract syntax of PFLC as defined in Haskell is very similar to the definition for Standard ML that was given in the previous section. The same convention with prime symbols for data constructors holds. A small excerpt is given below:

```
data Expr =
   Int' Int
 | True'
 | False'
 | Ident' String
 | BinOp' (Expr, BinOp, Expr)
 | UnOp' (UnOp, Expr)
 | If' (Expr, Expr, Expr)
 | Lambda' (String, Expr)
 | App' (Expr, Expr)
 | Fix' (String, Expr)
 | Abort' Expr
 | CallCC' Expr
 | Prompt' Expr
```

SEMANTIC DOMAINS

The definition for the semantic domains deviates slightly from the corresponding definition in Standard ML. The framework for representing domains is used here, and the auxiliary domains `T` and `N` are defined as flat domains. The mutual recursion needs not be eliminated in the case of Haskell. However, Haskell does not allow mutual recursion on type synonyms, and this is the reason why the `newtype` declaration is needed, in the case of `V`. The newly defined type must also be made an instance of class `Domain` and an additional data constructor must be used.

```
type T   = FlatDomain Bool
type N   = FlatDomain Int

type    B   = SumDomain N T
type    F   = LiftedDomain (V -> K -> V)
newtype V   = V' (SumDomain B F)
type    Env = Ide -> V
type    K   = V -> V

instance Domain V where
   (V' x) <<= (V' y)  = x <<= y
   isBottom (V' x)    = isBottom x
   bottom             = V' bottom
   isTop (V' x)       = isTop x
   top                = V' top
```

In order to avoid using a series of insert and extract operations for coalesced sum domains, a number of more abstract such operations are defined. Examples are `inBV` and `outVB` that are defined below:

```
inBV :: B -> V
inBV = V . inl

outVB :: V -> B
outVB (V x) = outl x
```

SEMANTICS

The semantic equations are translated almost directly from the meta-language to Haskell. The small deviations are due to the notation for flat domains and coalesced sums. As far as coalesced sums are concerned, it is important to remember that the omission of explicit insertors and extractors, e.g. *inl* and *outl*, from the meta-language is a notational convention. The operators should normally be there. An excerpt of the definition of the semantics of PFLC in Haskell is given below:

```
semP :: Expr -> V
semP expr = semE expr (\i -> top) id

semE :: Expr -> Env -> K -> V
semE (Int' x) rho kappa = kappa (inNV (FlatElement x))
semE True' rho kappa = kappa (inTV (FlatElement True))
semE (Ident' i) rho kappa = kappa (rho (FlatElement i))
semE (If' (expr, expr1, expr2)) rho kappa =
  semE expr rho (\e ->
     nif (outVT e) (semE expr1 rho kappa, semE expr2 rho kappa))
semE (App' (expr1, expr2)) rho kappa =
   semE expr1 rho (\e1 -> semE expr2 rho (\e2 -> delta e1 e2 kappa))
semE (Abort' expr) rho kappa = semE expr rho id
semE (CallCC' expr) rho kappa =
   semE expr rho (\e -> delta e (phi (\p -> \kappa' -> kappa p)) kappa)
...

phi :: (V -> K -> V) -> V
phi f = inFV (up (strict_D f))

delta :: V -> V -> K -> V
delta x = down (outVF x)
```

EXAMPLE The implemented semantics can be used for the evaluation of the same program that recursively calculates the factorial of 7. The program is given below, and its result, ignoring the presence of the trivial data constructor `V'`, is the value `inl(inl(5040))`.

```
-- fact = fix (lambda f. lambda n. if n = 0 then 1 else n * f(n-1))

fact :: Expr
fact = Fix'("f", Lambda'("n",
          If'(BinOp'(Ident'("n"), Equals', Int'(0)),
             Int'(1),
             BinOp'(Ident'("n"), Times',
                App'(Ident'("f"), BinOp'(Ident'("n"), Minus', Int'(1)))))))

-- Evaluate the expression "fact 7" using the semantics of PFLC

main = semP (App'(fact, Int'(7)))
```

## 15.3 The object-oriented paradigm

C++ IS NOT FUNCTIONAL In this section, the object-oriented programming paradigm is applied to the problem of implementing denotational descriptions and C++ is suggested as the target implementation language. The results of this research have been presented in [Papa96a] and [Papa96b]. It is clear that C++ is not a natural choice for this problem domain, lacking lexical closures, expressible functions and fully operational high-order functions. However, it is shown that object-orientation is a useful tool in this problem domain and some of the many difficulties imposed by C++ are overcome.

In the rest of this section, an object-oriented framework is suggested in several versions, providing a type-safe implementation for the $\lambda$-calculus over Scott domains. However, it is clear that the achieved results are not as elegant, nor as efficient in terms of execution time as those of a possible implementation in a more suitable language, such as ML. It is also clear that the presence of features such as high-order functions, currying, partial binding and the $\lambda$-notation determines the suitability

of a programming language for implementing denotational descriptions. Such features are inherent in the functional programming paradigm, and this partly explains why ML is a more appropriate choice in this problem domain.

In order to overcome the drawbacks of C++, an attempt is made to integrate such features in the proposed framework. However, in contrast to other approaches towards the same goal [Rose92, Klag93, Sarg93, Dami94, Watt94, Kuhn95, Lauf95, Rams95], the integration suggested here does not require any extensions to the language. Unfortunately, there is a tradeoff between natural description using pure C++ and performance. This dilemma is resolved in this section at the expense of performance, on the grounds that implementations of denotational descriptions are commonly used for the study of programming languages, in a context where performance is of little importance.

USING C++

Further analysis of the problem's requirements, inspired by techniques of object-oriented analysis and design, leads to the following remarks:

- Domains should be represented by classes, whose objects would represent elements, encapsulating data and operations.
- Representing compound domains (i.e. domains of functions, products, lifted domains, etc.) by class templates, parameterized by the type of their components seems to be a very appropriate choice. By proper representation of operators over such domains by function templates, it is possible to create a type-safe framework for element expressions. Furthermore, in this way it is possible to overcome the shortcoming of C++ as far as high-order functions are concerned.
- It does not seem necessary to treat syntactic domains in a different way. However, expressing syntactic domains by using products and sums complicates the implementation of semantic equations. It might be more appropriate to implement a special coalesced sum domain for the representation of syntactic domains.
- In order to naturally represent $\lambda$-abstractions, it is necessary to find a way of expressing unnamed functions. The definition of a C++ function for each use of the $\lambda$-abstraction cannot be considered a natural choice. For the expression of binding variables in abstractions, lack of lexical closures in C++ expressions must be overcome.

SECTION OVERVIEW

In Section 15.3.1 an untyped version of the framework is presented, implementing an untyped version of the meta-language. In Section 15.3.2 the same framework is extended by the introduction of types to the meta-language, and the use of templates is exploited to make it type-safe. Section 15.3.3 describes an elementary preprocessor that can be used as a front end to overcome the lack of function closures. In Section 15.3.4 several extensions to C++ are considered, that would be beneficial for the implementation of denotational semantics, and a last version of the suggested framework is presented, using the GNU C++ extensions. In Section 15.3.5 the full example is given and discussed. Finally, Section 15.3.6 summarizes and evaluates the results of this approach.

### 15.3.1 Untyped version

ENVELOPES

The first implementation in C++ to be considered is for an untyped version of the meta-language. The *envelope / letter* idiom [Copl92] is used, in order to achieve method polymorphism and at the same time alleviate the memory-management problems that result from the use of object pointers. An *envelope* class is used for the representation of Scott domain elements, whereas several *letter* classes

are used for the representation of various operations on such elements. The envelope class `Element` is defined as:

```
class Element {
   private:
      ElementImpl * const impl;

   public:
      Element (ElementImpl * const ei) : impl(ei) { }
      Element (const Element & e) : impl(e.impl->copy()) { }
     ~Element () { delete impl; }

      Element operator () (const Element & arg) const {
         return Element(new ApplicationImpl(impl, arg.impl));
      }

      friend Element lambda (const Element & exp) {
         return Element(new AbstractionImpl(exp.impl));
      }

      friend Element arg (int db) {
         return Element(new ParameterImpl(db));
      }

      friend Element fix (const Element & exp) {
         return Element(new FixImpl(exp.impl));
      }

      ...

      friend Element evaluate (const Element & e) {
         return Element(e.impl->evaluate());
      }
};
```

where functions `lambda` and `arg` are used for the creation of $\lambda$-abstraction elements, `operator ()` is used for the creation of function applications and function `fix` is used for the implementation of the least fixed point operator. In addition, method `evaluate` is used for the evaluation of elements, by using evaluation rules of $\lambda$-calculus over Scott domains.

LETTERS

A set of *letter* classes are used for element implementations, derived from the abstract letter class `ElementImpl`. In order to differentiate between implementation classes in run-time, a dynamic type casting mechanism is required. Although *run-time type inference* (RTTI) has long been suggested as part of the proposed C++ standard, it is not generally supported by compilers, at least in a portable manner. For this reason, a custom version of RTTI is used here, which defines a virtual `whatIs` method and a static `isMember` method for all concrete classes. The definition of these two is facilitated by using two special macros:

```
#define RTTI_ABSTRACT                                        \
public:                                                      \
  virtual const char * whatIs () const = 0;

#define RTTI_SIGNATURE(AbstractType, sig)                    \
public:                                                      \
  virtual const char * whatIs () const { return sig; }       \
  static int isMember (const AbstractType & x)               \
     { return strcmp(x.whatIs(), sig) == 0; }
```

where string signatures are used for clarity, instead of defining a special enumeration for classes.

Class `ElementImpl` is defined as:

```
class ElementImpl {
   RTTI_ABSTRACT

   public:
               ElementImpl () { }
      virtual ~ElementImpl () { }
      virtual  ElementImpl * copy () const = 0;
      virtual  ElementImpl * subst (int db, const ElementImpl * val) const = 0;
      virtual  ElementImpl * incFV (int t = 0) const = 0;
      virtual  ElementImpl * evaluate () const = 0;
};
```

where method `copy` implements the duplication of an object, method `subst` is used for textual substitution, method `incFV` will be explained later and method `evaluate` is used for the evaluation of elements.

Concrete letter classes can be defined for the implementation of domain operations, such as $\lambda$-abstractions, function applications or the *fix* operator. A letter class must be defined for bottom elements. Furthermore, a letter class must be defined for the implementation of function parameters. For this purpose *De Bruijn indices* are used instead of named dummies [dBru72].[2] De Bruijn indices facilitate the definition and implementation of textual substitution. Of the aforementioned letter classes, `AbstractionImpl` can be defined as:

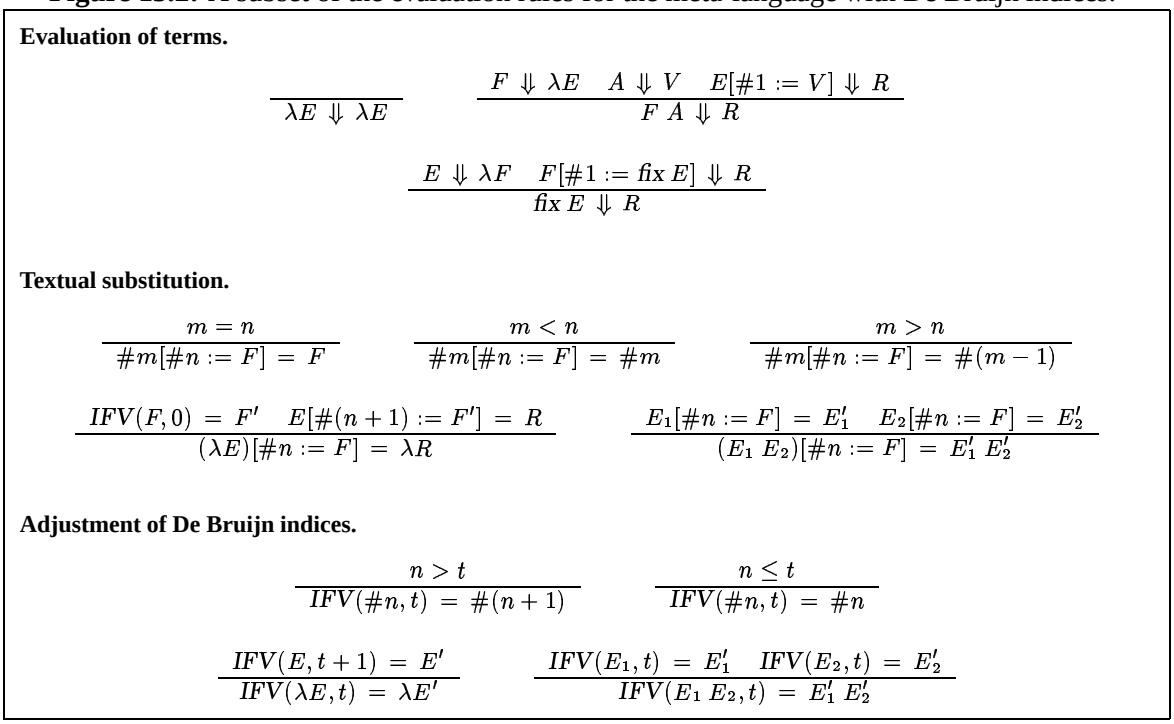
```
class AbstractionImpl: public ElementImpl {
   RTTI_SIGNATURE(ElementImpl, "AbstractionImpl")

   private:
      ElementImpl * const expression;

   public:
      AbstractionImpl (const ElementImpl * exp): expression(exp->copy()) { }

      virtual ~AbstractionImpl () { delete expression; }

      virtual  ElementImpl * copy () const;
      virtual  ElementImpl * subst (int db, const ElementImpl * val) const;
      virtual  ElementImpl * incFV (int t = 0);
      virtual  ElementImpl * evaluate ();

      ElementImpl * AbstractionImpl::apply (const ElementImpl * arg) const;
};
```

EVALUATION RULES

Methods `evaluate` and `subst` must be implemented according to a set of evaluation rules for the meta-language. Various evaluation strategies are possible and the strategy that is adopted here uses left-most reduction and call-by-value function application. A subset of the evaluation rules that are used is given in Figure 15.2. These rules correspond to the subset of the meta-language containing $\lambda$-abstractions, function applications and the *fix* operator. The notation $E \Downarrow V$ represents the evaluation relation and is read as "$E$ evaluates to $V$", whereas $E[\#n := F]$ denotes textual substitution and is read as "$E$ where dummy $n$ is substituted with $F$". Method `incFV` implements the adjustment of De Bruijn indices, that is necessary for substituting within $\lambda$-abstractions.

[2] De Bruijn indices are denoted as $\#n$, where $n \geq 1$.

**Figure 15.2:** A subset of the evaluation rules for the meta-language with De Bruijn indices.




For example, the implementation of some of these methods for the letter classes that were previously mentioned is given below:

```
ElementImpl * ApplicationImpl::evaluate () const {
   ElementImpl * eFun   = function->evaluate();
   ElementImpl * eArg   = argument->evaluate();
   ElementImpl * result = (AbstractionImpl::isMember(*eFun))
                           ? ((const AbstractionImpl *) eFun)->apply(arg)
                           : new BottomImpl();

   delete eFun;
   delete eArg;
   return result;
}

ElementImpl * FixImpl::evaluate () const {
   ElementImpl * eExp   = expression->evaluate();
   ElementImpl * result = (AbstractionImpl::isMember(*eExp))
                           ? ((const AbstractionImpl *) eExp)->apply(this)
                           : new BottomImpl();

   delete eExp;
   return result;
}

ElementImpl * AbstractionImpl::apply (const ElementImpl * arg) const {
   ElementImpl * applied = expression->subst(1, arg);
   ElementImpl * result  = applied->evaluate();
```

```
   delete applied;
   return result;
}

ElementImpl * AbstractionImpl::subst (int db, const ElementImpl * val) const {
   ElementImpl * fVal   = val->incFV();
   ElementImpl * sExp   = expression->subst(db+1, fVal);
   ElementImpl * result = new AbstractionImpl(sExp);

   delete fVal;
   delete sExp;
   return result;
}

ElementImpl * ParameterImpl::incFV (int t) const {
   return new ParameterImpl(deBruijn > t ? deBruijn+1 : deBruijn);
}
```

EXAMPLE

As an example, consider the expression $(\lambda\, x.\ \lambda\, y.\ x\ y)\ (\lambda\, x.\ x)\ 42$, which evaluates to $42$. This expression is written as $(\lambda\lambda \#2\ \#1)\ (\lambda \#1)\ 42$, when using De Bruijn indices. The evaluation of this expression is performed by the following C++ code:

```
Element x = lambda(lambda(arg(2)(arg(1))))(lambda(arg(1)))(Integer(42));
cout << x.evaluate() << endl;
```

where it is assumed that a class `Integer` has been derived from `Element`, an implementation for integer numbers has been written, as well as an `operator <<` for printing elements. Note that, although the evaluated element contains an implementation for the integer number $42$, its type is `Element` and not `Integer`, as it would be expected. This is due to the untypedness of this version of the framework and is corrected in the next section.

REMARKS

At this point, a few remarks are made about the suggested framework.

- A term of the meta-language is represented by elements whose implementations follow the term's structure. Functions are implemented as objects of the class `AbstractionImpl` and can be directly expressed in a fairly natural way without defining functions. However, terms of the meta-language can only be evaluated at run-time by explicit calls to method `evaluate`. No optimizations can be made at compile-time, in contrast to possible implementations in functional languages such as ML.

- It is possible to improve the efficiency of the framework by implementing part of the evaluation process in element constructors, such as `lambda` and `operator ()`. In this way, it is possible to reduce the size of element implementations. An improved framework contains a `simplify` method for element implementations, which performs all possible evaluations without evaluating least fixed point operations. Thus, invocation of this method will always terminate, something which is not true for method `evaluate`.

- The memory management scheme for element implementations results in a heavy use of operators `new` and `delete`. By overloading these operators and by changing accordingly the `copy` method, it is possible to implement a smarter memory management scheme that would not copy element implementations when not necessary (e.g. by keeping reference counters). Better results can be achieved by using a proper garbage collector for C++.

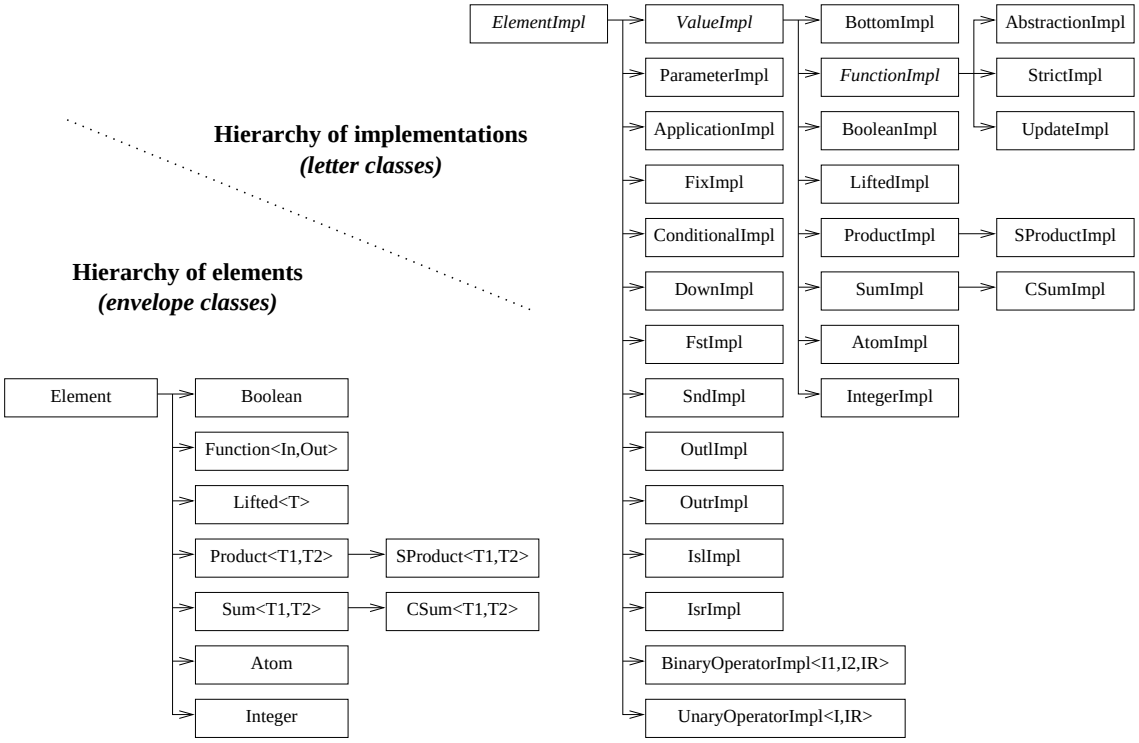



**Figure 15.3:** Class hierarchies for domain elements and implementations.

### 15.3.2 Type-safe version

TYPE SAFETY

Although the untyped version the framework succeeds in implementing the numerous operations on Scott domains in a fairly natural way, it fails to represent the Scott domains themselves. As an attempt to provide the elements with type information, it is possible to derive classes from `Element` but operations on such elements do not propagate the type information. Furthermore, the untyped version is only able to diagnose semantic errors in element expressions (such as application of an argument to an element that is not a function) at run-time.

It is possible to create a type-safe version of the framework, implementing a typed version of the meta-language. This version represents Scott domains as classes derived from `Element`, assigns type information to elements and propagates this type information correctly and consistently in operations. In addition, it is able to detect type errors at compile-time, by using the type system of C++. A set of classes and class templates are derived from `Element`, as shown in Figure 15.3. The various class templates represent domain constructors and are parameterized by the types of their operands. The same figure also shows the complete hierarchy of implementations.

TEMPLATES

In the type-safe version, functions such as `evaluate` must be replaced by function templates, propagating the correct type information. The definition of `evaluate` becomes:

```
template<class T>
T evaluate (const T & e) {
   return T(e.getImplementation()->evaluate());
}
```

The class template `Function<In, Out>` represents function domains. It is defined as:

```
#define FUN(In, Out) Function< In, Out >

template<class In, class Out>
class Function : public Element {
   public:
      Function (ElementImpl * ei) : Element(ei) { }
      Function (const Function<In, Out> & f) : Element(f) { }
      Out operator () (const In & arg) const;
};
```

and the corresponding domain operators, that is function application and the least fixed point operator, are defined as:

```
template<class In, class Out>
Out Function<In, Out>::operator () (const In & arg) const {
   return Out(new ApplicationImpl(getImplementation(), arg.getImplementation()));
}

template<class In, class Out>
FUN(In, Out) fix (const FUN(FUN(In, Out), FUN(In, Out)) & exp) {
   return FUN(In, Out)(new FixImpl(exp.getImplementation()));
}
```

TYPES OF PARAMETERS

In the typed version of $\lambda$-notation it is necessary to specify the type of the parameter. Also, in order to make the framework type-safe, it is also necessary to explicitly specify the type of each parameter's instance, since the C++ compiler cannot deduce the type of an expression such as `arg(1)`. This is a possible source of errors, since there is no way of checking whether the types of parameter instances are consistent with the types specified in the corresponding $\lambda$-expressions. Therefore, the $\lambda$-notation that is used in the type-safe version is not as simple as that of the untyped one. The two macros `lambda` and `arg` hide the ugly implementation details from the user. The use of `operator |=` for the implementation of `lambda` is justified because this operator is right associative and has a very low precedence. An empty class template has to be defined (`LambdaOperator<In>`) just to provide the parameter's type to `operator |=`.

```
#define lambda(T) (LambdaOperator< T >(0)) |=
#define arg(T, n) (T(new ParameterImpl(n)))

template<class In>
class LambdaOperator {
   public:
      LambdaOperator (int) { }
};

template<class In, class Out>
FUN(In, Out) operator |= (const LambdaOperator<In> & l, const Out & exp) {
   return FUN(In, Out)(new AbstractionImpl(exp.getImplementation()));
}
```

EXAMPLE

As an example, consider again the expression $(\lambda\, x : \mathbf{N} \to \mathbf{N}.\ \lambda\, y : \mathbf{N}.\ x\ y)\ (\lambda\, x : \mathbf{N}.\ x)\ 42$, which evaluates to 42. This expression is written as $(\lambda^{\mathbf{N}\to\mathbf{N}}\, \lambda^{\mathbf{N}}\, \#2\ \#1)\ (\lambda^{\mathbf{N}}\, \#1)\ 42$, in the typed meta-language, using De Bruijn indices. Note that the type of each parameter's instance is determined by the type specified by the corresponding $\lambda$-notation. The evaluation of this expression is performed by the following C++ code. The evaluated element is of the expected type `Integer`.

```
Integer x = (lambda(FUN(Integer, Integer)) lambda(Integer)
                arg(FUN(Integer, Integer), 2)(arg(Integer, 1)))
             (lambda(Integer) arg(Integer, 1))
             (Integer(42));
cout << x.evaluate() << endl;
```

REMARKS

It should be noted at this point that two problems still remain in the type-safe version of the framework. Both are due to the type system of C++:

- Type unification in templates does not work as expected in some compilers. In GNU C++, for instance, type unification fails when the formal parameter is a (reference to a) class template and the actual parameter is a subclass of the class template. The same type unification succeeds in Borland C++.[3]
- It is difficult to define recursive domains, such as $\mathbf{Tree} = \mathbf{N} + (\mathbf{Tree} \times \mathbf{Tree})$. The obvious definition would be something like:

  ```
  class Tree;
  typedef SUM(Integer, PROD(Tree, Tree)) Tree;
  ```

  only this does not work, neither in GNU C++ nor in Borland C++. It seems that the only way to overcome this problem is the following definition, which is problematic because of the previous remark:

  ```
  class Tree : public SUM(Integer, PROD(Tree, Tree))
  {
     public:
        Tree (const SUM(Integer, PROD(Tree, Tree)) & t) :
           SUM(Integer, PROD(Tree, Tree))(t) { }
  };
  ```

### 15.3.3 Preprocessor

NEED FOR PREPROCESSOR

The framework that has been suggested in the previous section is capable of expressing denotational descriptions in a fairly natural way. However, element expressions contain redundant information and this is a possible source of errors. The redundant information is the type of parameter instances. Consider the expression $\lambda\, x : \mathbf{N}.\ (\lambda\, x : \mathbf{N}.\ x)\ x$, which is written as:

```
lambda(Integer) (lambda(Integer) arg(Integer, 1))(arg(Integer, 1))
```

There are two difficulties with the representation of such an expression: (i) named dummies in $\lambda$-abstractions have to be replaced by De Bruijn indices, and (ii) the types of parameter instances have to be explicitly specified, although this information is redundant. The former is a matter of choice, simplifying the framework and improving its performance; on the other hand, named dummies provide a more natural way of representing $\lambda$-abstractions. The latter, however, is a problem of the implementation, due to the choice of C++ as the implementation language.

PREPROCESSOR

After a couple of unsuccessful attempts to overcome these two difficulties by using the C++ preprocessor, it seems that this is not possible, because C++ lacks lexical closures and variable scoping within expressions.[4] It is possible, however, to implement an external preprocessor for the conversion of $\lambda$-notations with named dummies to De Bruijn indices and explicitly typed parameter instances. The complete code of such a preprocessor is given below. It consists of 42 lines of *flex* code and 124 lines of *bison* code, including full error handling, and correctly translates the extensions used in a program, ignoring all the rest. The implementation of the preprocessor is elementary and a total of 8 tokens and 4 nonterminal symbols are used.

**Lexical analyzer** (`preproc.l`)

```
%{
   #include <string.h>

   #define YYSTYPE char *
   #include "preproc.tab.h"

   #define COPY do { yylval = strdup(yytext); } while(0)
   #define MORE yymore()
   #define NL   (lineNo++)

   int lineNo = 1;
%}

%x COMMENT STRING PREPROC

%%

"lambda"                {                          return TK_LAMBDA; }
[\(\)\.:]               {                          return yytext[0]; }

"/*"                    { MORE;     BEGIN(COMMENT);                  }
<COMMENT>[^*\n]+        { MORE;                                      }
<COMMENT>\n+            { MORE; NL;                                  }
<COMMENT>"*"+[^*/\n]    { MORE;                                      }
<COMMENT>"*"+\n         { MORE; NL;                                  }
<COMMENT>"*"+"/"        { COPY;     BEGIN(INITIAL); return TK_WHITE;  }
"//"[^\n]*\n            { COPY; NL;                 return TK_WHITE;  }

\"                      { MORE;     BEGIN(STRING);                   }
<STRING>[^"\\\n]+       { MORE;                                      }
<STRING>\\.             { MORE;                                      }
<STRING>\"              { COPY;     BEGIN(INITIAL); return TK_OTHER;  }

^[ \t]*#                { MORE;     BEGIN(PREPROC);                  }
<PREPROC>[^\\\n]+       { MORE;                                      }
<PREPROC>\\.            { MORE;                                      }
<PREPROC>\\\n           { MORE; NL;                                  }
<PREPROC>\n             { COPY; NL; BEGIN(INITIAL); return TK_OTHER;  }

[A-Za-z0-9_]+           { COPY;                     return TK_ID;     }
[ \t]+                  { COPY;                     return TK_WHITE;  }
\n                      { COPY; NL;                 return TK_WHITE;  }
```

[3] It is hoped that the emergence of the standard for C++ will eliminate this kind of problems.

[4] See section 15.3.4 for an implementation using the set of GNU C++ extensions.

```
.                       { COPY;                     return TK_OTHER;  }

%%
```

**Parser** (`preproc.y`)

```
%{
   #include <assert.h>
   #include <stdlib.h>
   #include <stdio.h>
   #include <string.h>
   #define YYSTYPE char *

   struct stack {
      char         * name;
      char         * type;
      struct stack * next;
   };

   struct stack * dummies = NULL;

   void push (struct stack ** s, char * name, char * type) {
      struct stack * node = (struct stack *) (malloc(sizeof(struct stack)));

      node->name = name;
      node->type = type;
      node->next = *s;
      *s = node;
   }

   void pop (struct stack ** s) {
      struct stack * node = *s;

      assert(node != NULL);
      *s = node->next;
      free(node->name);
      free(node->type);
      free(node);
   }

   void search (struct stack * s, const char * name) {
      int i;

      for (i = 1 ; s != NULL ; i++) {
         if (strcmp(s->name, name) == 0) {
            printf("ARG(%s, %d)", s->type, i);
            return;
         }
         s = s->next;
      }
      printf("%s", name);
   }
%}

%token TK_ID, TK_LAMBDA, TK_OTHER, TK_WHITE

%%
```

```
expression:
   expression '(' { printf("("); } expression ')' { printf(")"); }
 | expression lambda
 | expression ':' { printf(":"); }
 | expression '.' { printf("."); }
 | expression TK_ID    { search(dummies, $2); free($2); }
 | expression TK_OTHER { printf("%s", $2);    free($2); }
 | expression TK_WHITE { printf("%s", $2);    free($2); }
 | /* empty */
;

lambda:
   TK_LAMBDA white '(' white TK_ID white ':' type '.' {
      printf("LAMBDA(%s)(", $8);
      push(&dummies, $5, $8);
      free($2); free($4); free($6);
   }
   expression ')' {
      printf(")");
      pop(&dummies);
   }
;

type:
   type '(' type ')' {
      $$ = (char *) (malloc(strlen($1) + strlen($3) + 3));
      strcpy($$, $1); free($1); strcat($$, "(");
      strcat($$, $3); free($3); strcat($$, ")");
   }
 | type TK_ID {
      $$ = (char *) (malloc(strlen($1) + strlen($2) + 1));
      strcpy($$, $1); free($1);
      strcat($$, $2); free($2);
   }
 | type TK_OTHER {
      $$ = (char *) (malloc(strlen($1) + strlen($2) + 1));
      strcpy($$, $1); free($1);
      strcat($$, $2); free($2);
   }
 | type TK_WHITE {
      $$ = (char *) (malloc(strlen($1) + 2));
      strcpy($$, $1);  free($1);
      strcat($$, " "); free($2);
   }
 | /* empty */ {
      $$ = (char *) (malloc(1));
      *($$) = '\0';
   }
;

white:
   white TK_WHITE {
      $$ = (char *) (malloc(strlen($1) + strlen($2) + 1));
      strcpy($$, $1); free($1);
      strcat($$, $2); free($2);
   }
```

```
 | /* empty */ {
      $$ = (char *) (malloc(1));
      *($$) = '\0';
   }
;

%%

int yyerror (const char * s) {
   extern int lineNo;

   fprintf(stderr, "line %d: %s\n", lineNo, s);
   abort();
}

int main () { return yyparse(); }
```

EXAMPLE By using the preprocessor, element expression $\lambda\, x : \mathbf{N}.\ (\lambda\, x : \mathbf{N}.\ x)\ x$ can be written as:

```
lambda(x : Integer. lambda(y : Integer. y)(x))
```

No redundant information is given and this notation is much clearer and more natural than the one given in the beginning of this section.

### 15.3.4 Extensions

GNU EXTENSIONS It is possible to enhance the object-oriented framework for the implementation of denotational semantics by using a set of extensions to C++. The set of GNU extensions even makes the preprocessor unnecessary, without affecting the readability of element expressions. Two extensions that are particularly useful are:

- *Statement expressions*: compound statements within parentheses can appear within expressions. The last statement in the compound statement determines the value of the whole construct. This extension is particularly useful because it allows the definition of new scopes for variables inside expressions.
- *Operator typeof*: a compile-time operator, referring to the type of an expression, which is never evaluated. It can be used in any type expression and is very useful in combination with statement expressions.

NAMED DUMMIES The first and obvious improvement to the framework, using the two GNU extensions that were mentioned above, is that the meta-language can now use named dummies instead of De Bruijn indices in $\lambda$-abstractions. The set of evaluation rules must be revised and a subset of the new evaluation rules is given in Figure 15.4. The only point that needs special treatment is textual substitution in $\lambda$-abstractions. It is resolved by the renaming dummies whenever necessary.

DOMAIN CONSTRUCTORS A second important improvement is that a class template for domains themselves can be defined, namely `Domain<E>`, in addition to classes for domain elements. This allows the modelling of domain constructors as C++ operators, and the use of such *domain expressions* in element expressions, whenever this is required. The type parameter `E` that is used in this class template represents the type of domain elements. It is then possible to define:

**Figure 15.4:** A subset of the evaluation rules for the meta-language with named dummies.

**Evaluation of terms.**

$$\frac{}{\lambda I.\, E \Downarrow \lambda I.\, E} \qquad \frac{F \Downarrow \lambda I.\, E \quad A \Downarrow V \quad E[I := V] \Downarrow R}{F\ A \Downarrow R}$$

$$\frac{E \Downarrow \lambda I.\, F \quad F[I := \mathit{fix}\, E] \Downarrow R}{\mathit{fix}\, E \Downarrow R}$$

**Textual substitution.**

$$\frac{I = I'}{I'[I := F] = F} \qquad \frac{I \neq I'}{I'[I := F] = I'} \qquad \frac{E_1[I := F] = E_1' \quad E_2[I := F] = E_2'}{(E_1\ E_2)[I := F] = E_1'\ E_2'}$$

$$\frac{I = I'}{(\lambda I'.\, E)[I := F] = \lambda I'.\, E} \qquad \frac{I \neq I' \quad I' \notin FV(F) \quad E[I := F] = R}{(\lambda I'.\, E)[I := F] = \lambda I'.\, R}$$

$$\frac{I \neq I' \quad I' \in FV(F) \quad E[I' := I''][I := F] = R \quad I'' : \text{new dummy}}{(\lambda I'.\, E)[I := F] = \lambda I''.\, R}$$

**Free variables.**

$$\frac{}{FV(I) = \{I\}} \qquad \frac{FV(E) = S}{FV(\lambda I.\, E) = S - \{I\}} \qquad \frac{FV(E_1) = S_1 \quad FV(E_2) = S_2}{FV(E_1\ E_2) = S_1 \cup S_2}$$

```
Domain<Integer> N;
Domain<Boolean> T;
```

The class template `Domain<E>` is defined as:

```
template<class E>
class Domain {
   public:
      Domain () { }
      E * nullInstance () const { return NULL; }
      E bottomInstance () const { return E(new BottomImpl()); }
      E operator () (const E & e) { return e; }
};

#define bottom(T) ((T).bottomInstance())

template<class E1, class E2>
Domain<FUN(E1, E2)> operator |= (const Domain<E1> & d1, const Domain<E2> & d2) {
   return Domain<FUN(E1, E2)>;
}

template<class E1, class E2>
Domain<PROD(E1, E2)> operator * (const Domain<E1> & d1, const Domain<E2> & d2) {
   return Domain<PROD(E1, E2)>;
}

#define OBJ(T) typeof( *((T).nullInstance()) )
```

where `operator |=` and `operator *` represent respectively the domain constructors for functions and products and `operator ()` simplifies the expression of domain elements, allowing expressions such as `N(42)`. The macro `bottom` returns the bottom element of a domain, whereas the macro `OBJ(T)` returns the type of a given domain's element. Finally, the $\lambda$-notation can be implemented as:[5]

```
#define lambda(v, T, E) (lambdaOperator(#v, T,              \
                        ({ OBJ(T) v(new ParameterImpl(#v)); \
                           new typeof( E )(E); })))

template<class In, class Out>
FUN(In, Out) lambdaOperator (const char * dummy, const Domain<In> &, Out * exp) {
   FUN(In, Out) result(new AbstractionImpl(dummy, exp->getImplementation()));

   delete exp;
   return result;
}
```

EXAMPLE

As an example, consider again the expression $(\lambda\, x : \mathbf{N} \to \mathbf{N}.\ \lambda\, y : \mathbf{N}.\ x\ y)\ (\lambda\, x : \mathbf{N}.\ x)\ 42$. This time, the evaluation of this expression is performed by the following C++ code:

```
OBJ(N) x = (lambda(x, N|=N, lambda(y, N, x(y))))(lambda(x, N, x))(N(42))
cout << x.evaluate() << endl;
```

ADDITIONAL EXTENSIONS

A third possible extension, the presence of which would change the framework radically, is *unnamed functions* as suggested in [Breu88]. Unnamed functions would render unnecessary the definition of a special class for $\lambda$-abstractions and would much simplify the hierarchy of implementations. They would significantly narrow the gap between C++ and functional languages. Unfortunately, unnamed functions have not been adopted, to the best of the author's knowledge, in any popular C++ compiler, although their implementation does not seem particularly problematic.

### 15.3.5 Example

THE PROGRAM

An interpreter for PFLC, based on the denotational semantics of Section 15.1 has been developed in three variations, using the type-safe framework with De Bruijn indices, the custom preprocessor and the framework based on GNU extensions. In all cases, the framework itself was implemented in a header file of approximately 3,000 lines. The size of the programs that implement the semantics of PFLC in the three cases do not differ significantly. Execution time is the same for the first two cases, and it slightly increases in the third case.

The program consisted of approximately 1,000 lines of code, including a number of test programs. A significant part of the C++ implementation (approximately 350 lines) was devoted to the implementation of syntactic domains. It was decided not to use coalesced sums and products for this purpose, in order to simplify the equations. From this experience, it is now clear that the implementation of syntactic domains in the suggested framework is inadequate.

A FRAGMENT

The following code is a part of the implementation of PFLC's semantic equations, in the form that was suggested in section 15.3.3. Syntactic and semantic domains were defined earlier in the program and the problem of recursive domains, such as $\mathbf{V}$, was handled by defining empty classes, as discussed in section 15.3.2.

[5] A pointer type for the result of the statement expression had to be used, because of a bug in GNU C++ statement expressions.

```
V semP (const Expr & E) {
   return semE(E) (lambda(I: Ide. BOTTOM(V))) (lambda(e: V. e));
}

FUN(Env, FUN(K, V)) semE (const Expr & E) {
   if (E.is("Int'"))
      return lambda(rho: Env. lambda(kappa: K.
         kappa(V::inl(B::inl(semN(E[1]))))));

   else if (E.is("BinOp'")) {
      return lambda(rho: Env. lambda(kappa: K.
         semBO(E[2])(semE(E[1])(rho))(semE(E[3])(rho))(kappa)));
   }

   else if (E.is("Lambda'"))
      return lambda(rho: Env. lambda(kappa: K.
         kappa(phi(lambda(p: V. semE(E[2])(update(rho, E[1], p)))))));

   else if (E.is("Fix'"))
      return lambda(rho: Env. lambda(kappa: K.
         kappa(phi(fix(lambda(w: FUN(V, FUN(K, V)).
            delta(semE(E[2])(update(rho, E[1], phi(w)))(lambda(e: V. e)))))))));

   else if (E.is("Abort'"))
      return lambda(rho: Env. lambda(kappa: K.
         semE(E[1])(rho)(lambda(e: V. e))));

   ...
}
```

The scheme for accessing syntactic domains (method `is` and `operator []`) is a simplification of the one that was actually used. In fact, this scheme cannot be implemented in a type-safe way without dynamic type casting in function `semE` and a hierarchy of class templates for syntactic domains.

EVALUATION

Except for the (rather clumsy) implementation of syntactic domains, the implementation of an interpreter for PFLC by using the suggested framework was entirely successful and demonstrates the ability of C++ to implement denotational descriptions in a natural way. The performance of the interpreter is much lower than the performance of an implementation in a functional language. However, this was expected since C++ hardly supports the functional programming paradigm, which is so natural in this problem domain, and performance was sacrificed for ease-of-representation. As discussed in section 15.3.4, it is believed that the extension of C++ with a single feature (unnamed functions) would make it possible to achieve a performance comparable to functional languages.

### 15.3.6 Discussion

The variations of a type-safe framework for the implementation of denotational descriptions in C++ that were suggested in the previous sections exploit the object-oriented programming paradigm. Pure C++ can be used, although a set of extensions is definitely valuable. The suggested framework could be translated to other object-oriented languages supporting inheritance, polymorphism and generic types.

The main criteria in the evaluation of this framework are *expressiveness* and *performance.* Concerning the first criterion, the framework provides a natural way of expressing denotational descriptions. Some drawbacks of this approach, imposed by C++, have been already discussed. Compared

with other general purpose programming languages that have been suggested for the same purpose, this framework is inferior to implementations in functional languages. The object-oriented programming paradigm results in more natural denotational descriptions than possible implementations using imperative programming languages.

Performance issues in the implementation of this framework have been consciously neglected. Performance is reduced because of three factors:

- The memory management of C++ is poor for the requirements of this problem domain. This can be alleviated by using a garbage collector and overloading operators `new` and `delete`.
- $\lambda$-abstractions and high-order functions are managed by the programmer instead of the compiler, resulting in poor optimizations when compared to those that are performed by the compiler of a functional programming language. This can only be solved by extending C++ with an unnamed function feature, as discussed earlier.
- A large number of virtual methods is required and their calls reduce performance. This problem is inherent in the object-oriented programming paradigm with C++.

Nevertheless, performance is not considered to be a very significant factor in the evaluation of the framework. Implementations of denotational descriptions are mainly used as experimental execution models for the study of programming languages. In this context, performance is seldom an important issue.

## 15.4 Implementation of the proposed semantics for C

EVALUATION PROCESS

A significant effort has been made to evaluate the proposed denotational semantics for the C programming language. In this task, the major issue was to assess how complete and accurate the developed semantics is. Unfortunately, there is no systematic way for such an evaluation, since there is simply no way to compare a formal system of this complexity against an informal specification, such as the ANSI C standard. For this reason, an implementation of an interpreter corresponding to the developed semantics has been tested instead, using some test suites for C implementations that were available.

CHOICE OF LANGUAGE

An earlier version of the developed semantics was first implemented using SML as the target language. Although that version of the semantics did not use monads, a number of problems were envountered in the implementation. Later, SML was abandoned and Haskell was used instead, mainly because it has a richer type system, more flexible syntax, elegant support for monads and also because lazy evaluation avoids a number of non-termination problems.

CURRENT IMPLEMENTATION

The current implementation consists of approximately 15,000 lines of Haskell code, which are distributed roughly as follows: 3,000 lines for the static semantics, 3,000 lines for the typing semantics, 5,000 lines for the dynamic semantics, 3,000 lines for parsing and pretty-printing and 1,000 more lines of general code and code related to testing. As it was expected, the implementation is very slow and this presents a serious handicap in the yet unfinished evaluation process, significantly limiting the size of test programs. Although the evaluation of the developed semantics is still under way and minor bugs are waiting to be fixed, the results indicate that the developed semantics is complete and accurate to a great extent, with respect to the ANSI C standard.

# Chapter 16

# Related work

CHAPTER OVERVIEW

This chapter presents related work in the main research fields that are addressed by this thesis and, whenever applicable, compares it with the present work. Section 16.1 covers the field of defining the semantics of real programming languages and Section 16.2 specializes in the semantics of the C programming language. Section 16.3 presentes related work in the use of monads in denotational semantics. Finally, Section 16.4 briefly outlines related work in the implementation of denotational semantics.

## 16.1 Semantics of real programming languages

The semantics of many popular programming languages have been formally specified during the last 30 years. Various formalisms have been used for this purpose. However, in most cases the formalizations are not complete and features that are hard to formalize are often omitted. Only few real programming languages have been given formal semantics, even incomplete to some extent, as part of their official definitions. Scheme and ML are probably the only examples of such languages. In the search for formal specification of real programming languages, the author must greatfully acknowledge the invaluable help that he has received from the research of Baumann (`baumann@ifi.unizh.ch`) at the University of Zürich, who gathered a large number of references to relevant current literature in 1995.[1] In the following list of references, attempts to formalize the semantics of C have been omitted, as they are separately presented in Section 16.2.

- **Denotational semantics** have been used for the formal specification of:
  - Sequential *Ada*, in the work of Pedersen [Pede80].
  - *Algol 60* and *Pascal*, in the work of Bjørner and Jones [Bjor82a, Bjor82b], using the VDM formalism.
  - *Scheme*, in the work of a large research group, resulting in a publication edited by Clinger and Rees [Abel91]. The denotational specification is part of the IEEE standard [IEEE91].
  - *Smalltalk-80*, in the work of Wolczko [Wolc87].
- Variations of **operational semantics** have been used for the specification of:
  - *Standard ML*, as part of the language's definition by Milner, Tofte and Harper [Miln90, Miln91, Kahr93], using natural semantics.

[1] Unfortunately, the results of this research have now apparently disappeared from the Internet. They were previously accessible from the URL: `http://www.ifi.unizh.ch/groups/baumann/sol.html`.

  - *Eiffel*, in the work of Attali, Caromel and Oudshoorn [Atta93], using again natural semantics.
  - *Scheme*, in the work of Honsell, Pravato and Ronchi della Rocca [Hons95], using structured operational semantics.

- **Axiomatic semantics** have been used for the specification of:
  - *Pascal*, in the seminal work of Hoare and Wirth [Hoar73], probably the earliest attempt to a formal specification of a real programming language.

- **Abstract state machines**, formerly known as evolving algebras, have been used for the specification of large subsets of many languages, including:
  - *Ada*, in the work of Morris and Pottinger [Morr90].
  - *Cobol*, in the work of Vale [Vale93].
  - *C++*, in the work of Wallace [Wall93, Wall95].
  - *Modula-2*, in the work of Gurevich and Morris [Gure88, Morr88].
  - *Oberon*, in the work of Kutter and Pierantonio [Kutt97b, Kutt97a].
  - *Occam*, in the work of Gurevich and Moss [Gure90] and also in that of Börger, Durdanović and Rosenzweig [Borg94a, Borg96].
  - *Prolog*, in the work of Börger and Rosenzweig [Borg94b].
  - *Smalltalk*, in the work of Blakley [Blak92].

- Finally, **action semantics** have been used for the formalization of the semantics of:
  - *Pascal*, in the work of Mosses and Watt [Moss93].
  - *Standard ML*, in the work of Watt [Watt87], which was one of the earliest applications of action semantics.

## 16.2 Formal semantics of C

Until recently, C has not been a very popular language as far as the formalization of its semantics is concerned. However, after 1990, significant research has been conducted concerning semantic aspects of the language, mainly because of its popularity and its wide applications. In general, the majority of the suggested formalizations focuses on subsets of the language and avoids to address the most complicated issues, such as side effects in expressions, unspecified evaluation order and sequence points.

- The earliest formal approach to the semantics of C is given in the work of **Sethi**, where the semantics of pre-ANSI C declarations is mainly addressed [Seth80]. This appoach uses denotational semantics and makes a number of simplifications, the most important being a requirement for left-to-right evaluation of expressions. This work, although largely incomplete with respect to the ANSI C standard, has significantly influenced the present thesis.

- In a different paper [Seth83], **Sethi** addresses the semantics of C's control structures using again denotational semantics and introducing pipes as a notational variation for combining functions. This work is part of Sethi's research in the area of semantics-directed compiling. Left-to-right evaluation of expressions is again enforced and the declaration of variables is not allowed in compound statements.

- The work of **Gurevich** and **Huggins** defines a formal semantics for C using the formalism of evolving algebras, or abstract state machines [Gure93b]. However, in a number of cases, the proposed semantics is not accurate with respect to the standard. The semantics of expression evaluation is based on two mistaken assumptions: (i) that no interleaving is allowed in the evaluation of subexpressions, and (ii) that side effects take place at the same time that they are generated.

- **Black** and **Windley** have proposed a high-level axiomatic semantics for programming languages with side effects in expressions [Blac96]. The semantics is formalized as a set of inference rules for assignments, *while* loops and function calls. The inference rules distinguish between pre-evaluation and post-evaluation side effects. Furthermore, an implementation of the proposed semantics in the HOL theorem prover is used in the same work for the verification of a secure HTTP daemon, consisting of about 100 lines of C. The proposed semantics is not complete, with respect to the standard, and in an alltogether different level of abstraction from the work presented in this thesis.

- In the work of **Cook** and **Subramanian** an operational semantics for C is developed in the theorem prover Nqthm [Cook94b, Subr96]. The proposed semantics can be used for the verification of simple C programs in the theorem prover. However, it is incomplete and inaccurate to a large extent. Restrictions on C's type system allow only the type *int*, arrays of *int* and functions returning *void* or *int*. Only a small subset of C's control statements is allowed, excluding *switch*, *do*, *for* and jump statements but *return*. Furthermore, restrictions on operators are enforced and left-to-right evaluation order is assumed.

- **Cook**, **Cohen** and **Redmond** have also developed a denotational semantics for C in an unpublished work [Cook94a]. The semantics is based on a temporal logic defined by the authors. Although left-to-right evaluation is assumed in this work, the authors suggest how this can be remedied. However, it is not clear whether the suggestion allows for interleaving and there is no treatment of sequence points.

- An operational semantics for C has been sketched in terms of a random access machine, as a part of the **MATHS project** in California State University. The proposed semantics does not apparently cover declarations and is vague in parts related to the semantics of expressions and statements. Because of the formalism used by the authors, a comparison of this work to the present thesis is not easy.

- Finally, the work of **Norrish** describes an operational semantics for ANSI C, which has been fully defined in the HOL theorem prover and has been given the name Cholera [Norr97]. The operational semantics uses small-step reductions and follows the tradition of the formal definition of SML [Miln90]. A derived axiomatic logic is also developed in the same work as a set of theorems that can be proved in HOL using the operational model [Norr96]. The axiomatic logic is useful in verifying properties of C programs

To the best of the author's knowledge, the work of Norrish is the only approach that formalizes accurately C's unspecified order of evaluation and sequence points. The language that is specified by Cholera does not support *switch* and *goto* statements, nor string literals. Its type system lacks qualified types, unions and bit-fields. Moreover, no dynamic memory allocation is possible and a stack-based memory model is assumed, which is probably a restriction with respect to the standard.

Overall, the operational semantics of Norrish specifies accurately a language that is a subset of the one specified in the present thesis. His work follows an alltogether different path towards the same goal. For this reason, a more thorough comparison of his operational semantics with the denotational semantics defined here would be beneficial for both ends.

The author of this thesis knows of no other denotational approach to the semantics of the C programming language.

## 16.3 The use of monads in denotational semantics

The concept of monads comes from category theory. Monads have been proposed by Moggi as a useful structuring tool for denotational semantics [Mogg89, Mogg90]. Moggi demonstrated the use of monads for representing different aspects of computations, defined monads for programming language features such as state, exceptions and continuations, and presented call-by-value and call-by-name semantics for the $\lambda$-calculus. In a short time, the idea of monads became very popular in the functional programming community as a way of structuring functional programs and simulate non-functional features. The work of Wadler [Wadl92, Wadl95b, Wadl95a] played a very important role in this direction.

In the last few years, research related to the application of monads in denotational semantics has focused on the combination of monads to structure semantic interpreters. Monad transformers, which were also first proposed by Moggi [Mogg90] as "holes" inside monads, and other similar constructs have attracted the attention of many researchers. In the work of Steele [Stee94], pseudomonads were proposed as a way of building interpreters out of smaller parts. The first modular interpreter based on monad transformers was a system with the name Semantic Lego, written in Scheme by Espinosa [Espi95]. In this work Espinosa first raised the issue of lifting, and proposed stratification as an alternative. In the work of Liang, Hudak and Jones [Lian95b], monad transformers are demonstrated to succesfully modularize semantic interpreters and the lifting of several monad operations is investigated.

Powerdomains were first proposed by Plotkin [Plot76] as the domain-theoretic analogue to powersets. They have been used, in many variations, to model the semantics of programming languages with non-deterministic features, or parallelism that can be treated in a non-deterministic way. The convex powerdomain has been proved to be a monad in the category of complete partial orders and continuous functions [Gunt92]. However, the author of this thesis has not found a complete, clear and concise definition of this monad and its operations in literature.

The technique of resumptions has been used to model the semantics of interleaved execution in programming languages. An extensive treatment can be found in the book of de Bakker and de Vink [dBak96], where many variations of domains for modelling resumptions are defined and their

properties are explored.[2] In this work, a family of similar domains is defined including a domain for resumptions over a direct semantics which satisfies the isomorphism $D \simeq A + (S \to \mathcal{P}(S \times D))$, where $\mathcal{P}$ is a powerdomain constructor. However, no generalization is attempted to different kinds of resumptions. Such a generalization can be achieved by defining a resumption monad transformer and is attempted in the present thesis.

## 16.4 Implementation of denotational semantics

Denotational semantics can be considered as an abstract execution model of programming languages. By rewriting the semantic equations in an appropriate target programming language and by attaching a front-end for lexical analysis and parsing and a back-end for communication with the environment, a working implementation of such an abstract interpreter is possible. However, this rewriting of denotational semantics is not easy. The choice of target language is a crucial issue and several general-purpose languages have been suggested. It seems that functional languages are more suitable, since semantic equations are typically written in meta-languages influenced by the $\lambda$-calculus.

The list below indicates some general-purpose programming languages that have been used for the implementation of denotational semantics. However, it does not include the meta-languages of semantics-oriented specification systems such as SIS or VDMSL.

- **Algol 68** was the first imperative programming language to be suggested for this purpose, in the work of Pagan [Paga79]. An extension to the language was suggested by Pagan, in order to allow partial parametrization in functions and facilitate the implementation of semantics.
- **Pascal** has also been used for the same purpose in the work of Allison [Alli83, Alli85]. Although Pascal is a more modest language than Algol 68, this approach circumvents many of the difficulties indicated by Pagan. However, it is clear from this work that the introduction of extensions in Pascal could significantly simplify the task.
- **Lisp** and **Scheme** were probably the first from the family of functional languages to be used for the implementation of denotational semantics. Their main advantage is that, being of a functional nature, they are much closer to the meta-languages that are commonly used in semantics.
- **ML** was suggested as a much more natural choice for the implementation of denotational semantics, in the work of Watt [Watt86]. ML also belongs in the family of functional programming languages. Furthermore, in contrast to Lisp, ML is strongly-typed and this property proves extremely valuable in detecting errors in semantic equations.
- **Haskell** and other lazy functional languages have also been used recently for the implementation of denotational semantics. Lazy languages seem to circumvent non-termination problems, arising from the direct translation of semantic equations in eager functional programming languages. Haskell and its sibling **Gofer** are also strongly typed.

The current trend is towards the use of strongly-typed functional languages, such as ML and Haskell. Although these languages are not perfectly suited for this purpose, as discussed in Chapter 15, in most cases they prove to be adequate for the implementation of denotational semantics.

[2] The terminology is slightly different, however, as such domains are called “branching domains” in the book of de Bakker and de Vink and the term “resumption” is used in a different way.”

# Chapter 17

# Conclusion

CHAPTER OVERVIEW

This thesis presents the results of the author's research on the formal semantics of the ANSI C programming language. In this concluding chapter, a recapitulation of the thesis is first attempted in Section 17.1, where the accomplishments and contribution are briefly summarized. In Section 17.2 directions for future research are presented. Finally, Section 17.3 contains a few closing remarks.

## 17.1 Summary

ACCOMPLISHMENTS

A formal semantics for the ANSI C programming language has been developed in this thesis. Emphasis has been given primarily on the issues of accuracy and completeness, and secondarily on simplicity. The denotational approach has been followed. In order to improve the modularity and elegance of the semantics and facilitate its development, monads and monad transformers have been used, representing different aspects of computations. The semantics is divided in three distinct phases:

- Static semantics;
- Typing semantics; and
- Dynamic semantics.

Among the possible applications of a formal semantics for C, one should first mention its possible use as a precise, abstract and implementation-independent standard for the language; a point of reference for implementers and advanced programmers. However, its most important application is probably as a formal basis for reasoning about the correctness of C programs.

The developed semantics, together with a simple module for syntactic analysis, forms an abstract interpreter for C programs. A direct implementation of such an abstract interpreter has been developed, using Haskell as the implementation language. The implementation has been used in order to test the formal semantics and assess its accuracy and completeness. Although this evaluation process is still under way, the results so far have shown that the developed semantics is satisfactorily complete and accurate, with respect to the standard.

CONTRIBUTION

The main contribution of this work is the developed semantics itself. The author knows of no other denotational semantics for ANSI C that is as accurate and complete as the one presented here. With respect to these two properties, the developed semantics is also superior to all other proposed formal semantics for C, regardless of the used formalism. This work is a demonstration that a programming language as useful in practice and as inherently complicated as C can nontheless be given a formal semantics.

Another significant contribution is the application of monads and monad transformers for the specification of a real programming language. The use of monads over the category of domains and

continuous functions simply enhances the modularity and elegance of the semantics, without requiring changes in the mathematical foundations. Throughout this thesis, it has been demonstrated that the use of monad notation indeed achieves its purpose.

Furthermore, interesting results have been achieved in an attempt to model the interleaving of computations and non-determinism using monad notation. The resumption monad transformer, which is defined and investigated in this thesis, in conjunction with the convex powerdomain monad provides a basis for specifying the semantics of programming languages supporting non-determinism and execution interleaving or parallelism.

Experimentation with the implementation of the abstract interpreter has also led to interesting results. The advantages and disadvantages of using lazy and eager functional languages, such as Haskell and Standard ML, for the implementation of denotational semantics have been explored. Moreover, it has been demonstrated that the implementation of denotational semantics is also possible in object-oriented languages with generic types like C++, although such languages are not a natural choice.

## 17.2 Future research

Several directions for extensions and future investigations are naturally suggested by this research, ranging in the author's mind from "definitely feasible" to "almost science fictional". They are presented in the following list.

- The primary focus of the author's research in the near future will be the *evaluation* and *improvement* of the developed semantics, and it is estimated that significant effort will be made for the improvement of the existing implementation, in order to facilitate the testing process. Accuracy and completeness remain the primary objectives. To achieve the first, a number of corrections in the semantics will be required, most of which will address problems that have not yet been identified. To achieve the second, an effort will be made to withdraw some of the deviations presented in Section 2.3. Support for static objects is the first planned addition. Support for dynamic object allocation is the second. A third planned addition, of significantly higher complexity, is the extension of the semantics to also cover a large subset of C's standard library.

- Further research is required in order to investigate the properties and applications of the *resumption monad transformer*, which can model arbitrary interleaved computations.

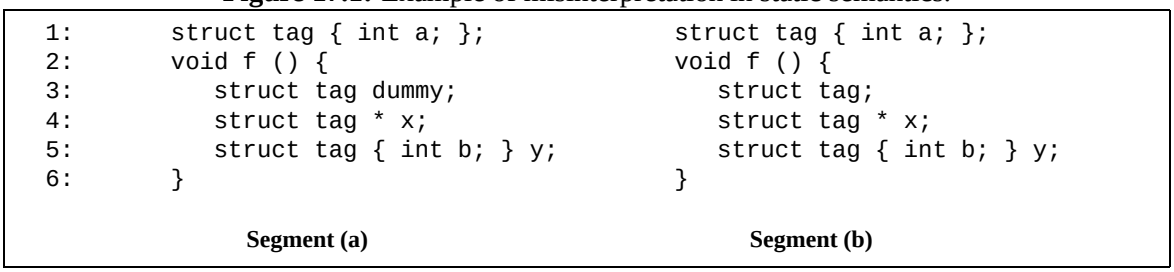
- The implementation of the developed semantics gave rise to an interesting question:

  > *What are the characteristics of a programming language that make it suitable for implementing denotational specifications, especially using monad notation?*

  Based on the experience that has been gained from the implementation of the semantics using the functional and object-oriented programming paradigms, the desired features of an implementation language for denotational semantics will be investigated. It is hoped that the results of this research will narrow the gap between the two programming paradigms and will be profitable for the programming community.

- Switching to a less theoretical track, a different research direction aims at the study of the *practical applications* that the developed formal semantics for C may have in the software development process. Program verification, debugging and understanding will be considered as possible application areas, as well as proving the correctness of program transformations.

**Figure 17.1:** Example of misinterpretation in static semantics.



```
1:      struct tag { int a; };          struct tag { int a; };
2:      void f () {                     void f () {
3:         struct tag dummy;               struct tag;
4:         struct tag * x;                 struct tag * x;
5:         struct tag { int b; } y;        struct tag { int b; } y;
6:      }                               }
```

**Segment (a)** **Segment (b)**

- Along the same line, the applications of the developed formal semantics in *compiler construction* remain to be explored. Major goals of related research has always been the transformation of the formal semantics to correct compilers or, at least, the development of correctness proofs for existing compilers.
- A final direction for future research aims at studying and specifying the semantics of the object-oriented languages that descend from C, such as *C++* and *Java*. These two languages play a very important role in the contemporary software industry and the situation is unlikely to change for many years. The introduction of object-oriented features is bound to produce a large number of changes, and it is expected that the formal semantics of C++ or Java will be orders of magnitude more complex than that of C.

## 17.3 Closing remarks

COMMON MISINTERPRETATIONS

C is probably the most widely spread programming language in today's software industry. The author believes that there is a large number of programmers who are confident of their understanding of the C language, but whose understanding is unfortunately subjective and incorrect, i.e. they do not understand the language in the way that is intended in the standard. He supports his opinion by presenting three simple programs, which are sources of common misinterpretations among C programmers. In all cases, however, all doubts vanish when one reads the standard carefully. The examples are taken respectively from the areas of static, typing and dynamic semantics of C, as these are distinguished in the present thesis.

IN STATIC SEMANTICS

Consider the two small program segments shown in Figure 17.1. The two segments differ only in the presence of identifier *dummy* in line 3. This identifier is never used within function *f* and therefore, one might assume that the two segments are equivalent. A question which may seem easy at first is:

*What are the members of the structure pointed by* x*?"*

Possible candidates are obviously *a* and *b*, depending on which of the declarations for *tag* is in effect at line 4. But which one is it? The correct answer is that the two programs are not equivalent and the only member of the structure is *a*, in the case of segment (a), and *b* in the case of segment (b). The rationale behind this answer can be found in §6.5.2.3 of the standard and has been discussed in Section 5.3.

IN TYPING SEMANTICS

Next, consider the following program segment, which is intended to increase the contents of a variable representing the number of counted men or women, depending on the value of a boolean flag.

```
int countMen = 0, countWomen = 0;
(sexFlag ? countMen : countWomen)++;
```

The question now is:

*Is this program segment legal?*

The answer depends on whether a conditional expression is an l-value or not. A footnote in §6.3.15 of the standard states that it is not, thus invalidating the above segment. However, popular C compilers (e.g. GNU C) treat this as an extension to the standard and by default allow such constructs. The developed typing semantics is very specific about this. Rule E104 in Section 8.1.12 states that the result of the conditional operator is not an l-value.

IN DYNAMIC SEMANTICS

As a third example, consider one of the most infamous C expressions:

```
x = x++
```

together with the question:

*Is this expression legal and, if yes, what are the contents of variable* x *after its execution?*

Now many answers are possible. The correct answer is that this expression leads to undefined behaviour, since it violates the restriction in §6.3 of the standard according to which "between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression". The developed dynamic semantics for C expressions gives the same answer, although this cannot be very easily verified.

WHEN THE STANDARD FAILS

One might argue here that, although the presented program segments are simple, they do not often occur in practice. Although this is rather true, it is very probable that any given C programmer will eventually run into a similar case. Moreover, one might argue that since the informal standard dictates the answers, the real problem is the programmers' incompetence. In reply to that, let us consider a case where the standard itself is not at all clear and many possible interpretations exist. Consider the following simple C program:

```
int r = 0;
int f (int x) {
   return (r = a);
}
int main () {
   return (f(1) + f(2), r);
}
```

This program presents again a situation that will eventually arise in practice. A natural question is:

*Is this program legal and, if yes, what is the number returned by* main *after its execution?*

Similar programs and questions are often discussed in the `comp.std.c` newsgroup by distinguished researchers, programmers and even members of the ANSI C standardization committee, invariably leading to the expression of numerous contradictory opinions and no conclusions reached. Although a technical discussion will be avoided here, two sound answers corresponding to different possible interpretations of the standard are the following. The approach taken by the developed dynamic semantics corresponds to the second answer.

- The program is not legal because $r$ is modified twice between successive sequence points.
- The program is legal and its result may be 1 or 2, but it is unspecified which one.

The result from all this discussion is that programmers' incompetence, although a significant problem on its own right, is not solely responsible for misunderstandings. Responsibility lies in the standard as well. C is an inherently complicated language and simply cannot be defined informally, using natural language, without introducing ambiguities. Informal texts are valuable as introductions to the language and for educational purposes. However, the author believes that the definition of the language, the standard itself, must be formal. After all, C is very often used to program applications of a very delicate nature, where software failure may have disastrous results. In this context, misunderstandings about the programming language cannot be allowed.

A VIEW OF THE FUTURE

After approximately half a century of experience with computers and software, it is widely accepted by now that software development is an engineering discipline, as emphasized by the terms *software engineering* and *systems engineering* that are commonly used to describe it. Typically, all engineering disciplines are based on theoretical, mathematical foundations. In the case of software engineering however, there is still a remarkably wide gap between the formal techniques for program development, devised and advocated by academics in universities, and the techniques and methods used in practice by software engineers. The author believes that an important goal for Computer Science is to narrow this gap and bring together these two communities, for the benefit of both. One way to achieve this goal is through the formal study of tools and techniques used by software engineers, and the formal definition of the C programming language happily falls into this research area. The author believes that future tools used in the software development process will be based on fully formal definitions of programming languages and wishes to see the work that has been presented in this thesis as one small step in this direction.

# Bibliography


[Abel91] H. Abelson et al., "Revised 4 Report on the Algorithmic Language Scheme", *Lisp Pointers*, vol. 4, no. 3, pp. 1–55, July 1991.

[Alli83] L. Allison, "Programming Denotational Semantics", *Computer Journal*, vol. 26, no. 2, pp. 164–174, 1983.

[Alli85] L. Allison, "Programming Denotational Semantics II", *Computer Journal*, vol. 28, no. 5, pp. 480–486, 1985.

[Alli86] L. Allison, *A Practical Introduction to Denotational Semantics*, Cambridge University Press, New York, NY, 1986.

[ANSI89a] American National Standards Institute, New York, NY, *American National Standard for Information Systems: Programming Language C, ANSI X3.159-1989*, 1989.

[ANSI89b] American National Standards Institute, New York, NY, *Rationale for American National Standard for Information Systems: Programming Language C*, 1989, Supplement to ANSI X3.159-1989.

[ANSI90] American National Standards Institute, New York, NY, *ANSI/ISO 9899-1990, American National Standard for Programming Languages: C*, 1990, Revision and redesignation of ANSI X3.159-1989.

[ANSI94] American National Standards Institute, New York, NY, *Technical Corrigendum Number 1 to ANSI/ISO 9899-1990 American National Standard for Programing Languages: C*, 1994.

[Aspe91] A. Asperti and G. Longo, *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1991.

[Atta93] I. Attali, D. Caromel and M. Oudshoorn, "A Formal Definition of the Dynamic Semantics of the Eiffel Language", *Australian Computer Science Communications*, vol. 14, no. 1, pp. 109–119, February 1993.

[Barr96] M. Barr and C. Wells, *Category Theory for Computing Science*, Prentice-Hall International Series in Computer Science, Prentice Hall, New York, NY, 2nd edition, 1996.

[Bjor82a] D. Bjørner and C. B. Jones, "Algol 60", in *Formal Specification and Software Development*, chapter 6, pp. 141–173, Prentice Hall, Englewood Cliffs, NJ, 1982.

[Bjor82b] D. Bjørner and C. B. Jones, "Pascal", in *Formal Specification and Software Development*, chapter 7, pp. 175–251, Prentice Hall, Englewood Cliffs, NJ, 1982.

[Blac96] P. E. Black and P. J. Windley, "Inference Rules for Programming Languages with Side Effects in Expressions", in *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, pp. 51–60, Turku, Finland, 26–30 August 1996, Springer Verlag.

[Blak92] B. Blakley, *A Smalltalk Evolving Algebra and its Uses*, Ph.D. thesis, University of Michigan, Ann Arbor, MI, 1992.

[Bodw82] J. Bodwin, L. Bradley, K. Kanda, D. Litle and U. Pleban, "Experience with an Experimental Compiler Generator Based on Denotational Semantics", in *Proceedings of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pp. 216–229, June 1982.

[Borg94a] E. Börger, I. Durdanovic and D. Rosenzweig, "Occam: Specification and Compiler Correctness, Part I: Simple Mathematical Interpreters", in B. Pehrson and I. Simon, editors, *Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*, pp. 489–508, North-Holland, 1994.

[Borg94b] E. Börger and D. Rosenzweig, "A Mathematical Definition of Full Prolog", *Science of Computer Programming*, 1994.

[Borg96] E. Börger and I. Durdanovic, "Correctness of Compiling Occam to Transputer Code", *Computer Journal*, vol. 39, no. 1, pp. 52–92, 1996.

[Breu88] T. Breuel, "Lexical Closures for C++", in *Proceedings of the USENIX C++ Conference*, pp. 293–304, Denver, CO, October 1988.

[Brow92] D. F. Brown, H. Moura and D. A. Watt, "ACTRESS: An Action Semantics Directed Compiler Generator", in *Proceedings of the 4th International Conference on Compiler Construction*, vol. 641 of *Lecture Notes in Computer Science*, pp. 95–109, New York, NY, 1992, Springer Verlag.

[Cook94a] J. Cook, E. Cohen and T. Redmond, "A Formal Denotational Semantics for C", Technical Report 409D, Trusted Information Systems, September 1994.

[Cook94b] J. Cook and S. Subramanian, "A Formal Semantics for C in Nqthm", Technical Report 517D, Trusted Information Systems, October 1994.

[Copl92] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

[Cox91] B. J. Cox and A. J. Novobilski, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, MA, 2nd edition, 1991.

[Dami94] L. Dami, *Software Composition: Towards an Integration of Functional and Object Oriented Approaches*, Ph.D. thesis, Université de Genève, April 1994.

[dBak80] J. de Bakker, *Mathematical Theory of Program Correctness*, International Series in Computer Science, Prentice Hall, Englewood Cliffs, NJ, 1980.

[dBak96] J. de Bakker and E. de Vink, *Control Flow Semantics*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1996.

[dBru72] N. G. de Bruijn, "Lambda-Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation", *Indagationes Mathematicae*, vol. 34, pp. 381–392, 1972.

[Dijk76] E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ, 1976.

[Elli90] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.

[Espi95] D. A. Espinosa, *Semantic Lego*, Ph.D. thesis, Columbia University, Department of Computer Science, 1995.

[Fior96] M. P. Fiore, A. Jung, E. Moggi, P. O'Hearn, J. Riecke, G. Rosolini and I. Stark, "Domains and Denotational Semantics: History, Accomplishments and Open Problems", Technical Report CSR-96-2, University of Birmingham, School of Computer Science, January 1996.

[Gaud81] M. C. Gaudel, "Compiler Generation from Formal Definition of Programming Languages: A Survey", in *Formalization of Programming Concepts*, vol. 107 of *Lecture Notes in Computer Science*, pp. 96–114, Springer Verlag, 1981.

[Geha89] N. H. Gehani and W. D. Roome, *Concurrent C*, Silicon Press, Summit, NJ, 1989.

[Gogu91] J. A. Goguen, "A Categorical Manifesto", *Mathematical Structures in Computer Science*, vol. 1, pp. 49–68, 1991.

[Gord79] M. J. C. Gordon, *The Denotational Descriptions of Programming Languages*, Springer Verlag, Berlin, Germany, 1979.

[Gosl96] J. Gosling, B. Joy and G. L. Steele, Jr., *The Java Language Specification*, Addison-Wesley, Reading, MA, 1996.

[Grie81] D. Gries, *The Science of Programming*, Springer Verlag, New York, NY, 1981.

[Gunt90] C. A. Gunter and D. S. Scott, "Semantic Domains", in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, chapter 12, pp. 633–674, Elsevier Science Publishers B.V., 1990.

[Gunt92] C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1992.

[Gure88] Y. Gurevich and J. Morris, "Algebraic Operational Semantics and Modula-2", in E. Börger, H. Kleine Büning and M. M. Richter, editors, *Proceedings of the 1st Workshop on Computer Science Logic (CSL'87)*, vol. 329 of *Lecture Notes in Computer Science*, pp. 81–101, Springer Verlag, 1988.

[Gure90] Y. Gurevich and L. S. Moss, "Algebraic Operational Semantics and Occam", in E. Börger, H. Kleine Büning and M. M. Richter, editors, *Proceedings of the 3rd Workshop on Computer Science Logic (CSL'89)*, vol. 440 of *Lecture Notes in Computer Science*, pp. 176–192, Springer Verlag, 1990.

[Gure93a] Y. Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", in G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pp. 266–292, World Scientific, 1993.

[Gure93b] Y. Gurevich and J. K. Huggins, "The Semantics of the C Programming Language", in E. Börger et al., editors, *Selected Papers from CSL'92 (Computer Science Logic)*, vol. 702 of *Lecture Notes in Computer Science*, pp. 274–308, Springer Verlag, New York, NY, 1993.

[Gure95] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide", in E. Börger, editor, *Specification and Validation Methods*, pp. 9–36, Oxford University Press, 1995.

[Harb95] S. P. Harbison and G. L. Steele, Jr., *C: A Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, 4th edition, 1995.

[Harp86] R. Harper, D. MacQueen and R. Milner, "Standard ML", Technical Report ECS-LFCS-86-12, University of Edinburgh, Laboratory for Foundations of Computer Science, March 1986.

[Harp89] R. Harper, "Introduction to Standard ML", Technical Report ECS-LFCS-86-14, University of Edinburgh, Laboratory for Foundations of Computer Science, January 1989, Revised edition.

[Henn90] M. Hennessy, *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*, John Wiley and Sons, New York, NY, 1990.

[Hoar69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, October 1969.

[Hoar73] C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL", *Acta Informatica*, vol. 2, pp. 335–355, 1973.

[Hons95] F. Honsell, A. Pravato and S. Ronchi della Rocca, "Structured Operational Semantics of the Language SCHEME", Technical report, University of Torino, Department of Informatics, 1995.

[Huda96] P. Hudak, J. Fasel and J. Peterson, "A Gentle Introduction to Haskell", Technical Report YALEU/DCS/RR-901, Yale University, Department of Computer Science, May 1996.

[IEEE91] Institute of Electrical and Electronics Engineers, New York, NY, *IEEE Standard for the Scheme Programming Language, IEEE Standard 1178-1990*, 1991.

[John73] S. C. Johnson and B. W. Kernighan, "The Programming Language B", Technical Report 8, AT&T Bell Laboratories, January 1973.

[Jone80] N. D. Jones and D. A. Schmidt, "Compiler Generation from Denotational Semantics", in N. D. Jones, editor, *Semantics-Directed Compiler Generation*, vol. 94 of *Lecture Notes in Computer Science*, pp. 71–93, Springer Verlag, Berlin, Germany, 1980.

[Jone94] M. P. Jones, "The Implementation of the Gofer Functional Programming System", Research Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, May 1994.

[Kahr93] S. Kahr, "Mistakes and Ambiguities in the Definition of Standard ML", Technical Report ECS-LFCS-93-257, University of Edinburgh, Department of Computer Science, 1993.

[Kern78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1978.

[Kern88] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1988.

[Klag93] H. Klagges, "A Functional Language Interpreter Integrated into the C++ Language System", Master's thesis, Balliol College, University of Oxford, Oxford University Computing Laboratory, September 1993.

[Kuhn95] T. Kühne, "Parameterization Versus Inheritance", in C. Mingins and B. Meyer, editors, *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific'94)*, pp. 235–245, London, 1995, Prentice Hall, For correct version ask author; proceedings contain corrupted version.

[Kutt97a] P. W. Kutter, "Dynamic Semantics of the Oberon Programming Language", TIK-Report 25, ETH Zürich, February 1997.

[Kutt97b] P. W. Kutter and A. Pierantonio, "The Formal Specification of Oberon", *Journal of Universal Computer Science*, vol. 3, no. 5, pp. 443–503, 1997.

[Lauf95] K. Läufer, "A Framework for Higher-Order Functions in C++", in *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, pp. 103–116, Monterey, CA, 26–29 June 1995.

[Lee87] P. Lee and U. Pleban, "A Realistic Compiler Generator Based on High-Level Semantics", in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 284–295, 1987.

[Lian95a] S. Liang, "A Modular Semantics for Compiler Generation", Technical Report YALEU/DCS/TR-1067, Yale University, Department of Computer Science, February 1995.

[Lian95b] S. Liang, P. Hudak and M. Jones, "Monad Transformers and Modular Interpreters", in *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, San Francisco, CA, January 1995.

[Lian96] S. Liang and P. Hudak, "Modular Denotational Semantics for Compiler Construction", in *Proceedings of the European Symposium on Programming*, April 1996.

[Miln76] R. E. Milne and C. Stachey, *A Theory of Programming Language Semantics*, Chapman and Hall, London, UK, 1976.

[Miln90] R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.

[Miln91] R. Milner and M. Tofte, *Commentary on Standard ML*, MIT Press, Cambridge, MA, 1991.

[Mitc90] J. C. Mitchell, "Type Systems for Programming Languages", in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, chapter 8, pp. 365–458, Elsevier Science Publishers B.V., 1990.

[Mitc96] J. C. Mitchell, *Foundations for Programming Languages*, MIT Press, Cambridge, MA, 1996.

[Mogg89] E. Moggi, "Computational Lambda Calculus and Monads", in *IEEE Symposium on Logic in Computer Science*, pp. 14–23, 1989.

[Mogg90] E. Moggi, "An Abstract View of Programming Languages", Technical Report ECS-LFCS-90-113, University of Edinburgh, Laboratory for Foundations of Computer Science, 1990.

[Morr88] J. Morris, *Algebraic Operational Semantics for Modula-2*, Ph.D. thesis, University of Michigan, Ann Arbor, MI, 1988.

[Morr90] J. Morris and G. Pottinger, "Ada-Ariel Semantics", Technical report, Odyssey Research Associates, July 1990.

[Moss76] P. D. Mosses, "Compiler Generation Using Denotational Semantics", in *Mathematical Foundations of Computer Science*, vol. 45 of *Lecture Notes in Computer Science*, pp. 436–441, Springer Verlag, Berlin, Germany, 1976.

[Moss90] P. D. Mosses, "Denotational Semantics", in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, chapter 11, pp. 577–631, Elsevier Science Publishers B.V., 1990.

[Moss92] P. D. Mosses, *Action Semantics*, vol. 26 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, New York, NY, 1992.

[Moss93] P. D. Mosses and D. A. Watt, "Pascal Action Semantics", 1993.

[Norr96] M. Norrish, "Derivation of Verification Rules for C from Operational Definitions", in J. von Wright, J. Grundy and J. Harrison, editors, *Supplementary Proceedings of TPHOLs '96*, no. 1 in TUCS General Publications, pp. 69–75, Turku Center for Computer Science, August 1996.

[Norr97] M. Norrish, "An Abstract Dynamic Semantics for C", Technical Report TR-421, University of Cambridge, Computer Laboratory, May 1997.

[Paga79] F. G. Pagan, "Algol 68 as a Metalaguage for Denotational Semantics", *Computer Journal*, vol. 22, no. 1, pp. 63–66, 1979.

[Pals92] J. Palsberg, "A Provably Correct Compiler Generator", in B. Krieg-Brückner, editor, *Proceedings of the 4th European Symposium on Programming (ESOP'92)*, vol. 582 of *Lecture Notes in Computer Science*, pp. 418–434, New York, NY, 1992, Springer Verlag.

[Papa96a] N. S. Papaspyrou, "A Framework for Programming Denotational Semantics in C++", *ACM SIGPLAN Notices*, vol. 31, no. 8, pp. 16–25, August 1996.

[Papa96b] N. S. Papaspyrou, "A Framework for Programming Denotational Semantics in C++", Technical Report CSD-SW-TR-5-96, National Technical University of Athens, Software Engineering Laboratory, June 1996.

[Paul82] L. Paulson, "A Semantics-Directed Compiler Generator", in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 224–233, 1982.

[Pede80] J. S. Pedersen, "A Formal Semantics Definition of Sequential Ada", in D. Bjørner and O. N. Oest, editors, *Towards a Formal Description of Ada*, vol. 98 of *Lecture Notes in Computer Science*, pp. 213–308, Springer Verlag, New York, NY, 1980.

[Pete97] J. Peterson and K. Hammond (editors), *Report on the Programming Language Haskell*, version 1.4 edition, March 1997, Available from `http://haskell.org/`.

[Pett92] M. Pettersson and P. Fritzson, "DML: A Meta-Language and System for the Generation of Practical and Efficient Compilers from Denotational Specifications", in *Proceedings of the 1992 International Conference on Computer Languages*, pp. 127–136, 1992.

[Pier90] B. C. Pierce, "A Taste of Category Theory for Computer Scientists", Technical Report CMU-CS-90-113R, Carnegie Mellon University, School of Computer Science, September 1990.

[Pier91] B. Pierce, *Basic Category Theory for Computer Scientists*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1991.

[Pleb88] U. F. Pleban and P. Lee, "An Automatically Generated, Realistic Compiler for an Imperative Programming Language", in *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 222–227, Atlanta, GA, June 1988.

[Plot76] G. D. Plotkin, "A Powerdomain Construction", *SIAM Journal on Computing*, vol. 5, pp. 452–487, 1976.

[Rams95] J. D. Ramsdell, "CST: C State Transformers", *ACM SIGPLAN Notices*, vol. 30, no. 12, pp. 32–36, December 1995.

[Rich79] M. Richards and C. Whitbey-Stevens, *BCPL: The Language and its Compiler*, Cambridge University Press, Cambridge, UK, 1979.

[Ritc93] D. M. Ritchie, "The Development of the C Language", *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 201–208, March 1993, Preprints of the Second ACM SIGPLAN History of Programming Language (HOPL II).

[Rose92] J. R. Rose and H. Muller, "Integrating the Scheme and C Languages", in *Conference Record of the ACM Symposium on Lisp and Functional Programming*, pp. 247–259, San Francisco, CA, 1992.

[Sarg93] J. Sargeant, "United Functions and Objects: An Overview", Technical Report UMCS-93-1-4, University of Manchester, Department of Computer Science, 1993.

[Schm86] D. A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon, Newton, MA, 1986.

[Scot71] D. Scott and C. Strachey, "Towards a Mathematical Semantics for Computer Languages", in *Proceedings of the Symposium on Computers and Automata*, pp. 19–46, Brooklyn, NY, 1971, Polytechnic Press.

[Scot82] D. S. Scott, "Domains for Denotational Semantics", in *International Colloquium on Automata, Languages and Programs*, vol. 140 of *Lecture Notes in Computer Science*, pp. 577–613, Berlin, Germany, 1982, Springer Verlag.

[Seth80] R. Sethi, "A Case Study in Specifying the Semantics of a Programming Language", in *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pp. 117–130, January 1980.

[Seth83] R. Sethi, "Control Flow Aspects of Semantics-Directed Compiling", *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 4, pp. 554–595, October 1983.

[Sita90] D. Sitaram and M. Felleisen, "Reasoning with Continuations II: Full Abstraction for Models of Control", in M. Wand, editor, *Conference Record of the ACM Symposium on Lisp and Functional Programming*, pp. 161–175, ACM Press, 1990.

[Stee94] G. L. Steele, Jr., "Building Interpreters by Composing Monads", in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1994.

[Stoy77] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.

[Stro91] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 2nd edition, 1991.

[Subr96] S. Subramanian and J. V. Cook, "Mechanical Verification of C Programs", in *Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Practice*, January 1996, Extended abstract.

[Tenn76] R. D. Tennent, "The Denotational Semantics of Programming Languages", *Communications of the ACM*, vol. 19, no. 8, pp. 437–453, August 1976.

[Tenn91] R. D. Tennent, *Semantics of Programming Languages*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Vale93] M. Vale, "The Evolving Algebra Semantics of COBOL, Part 1: Programs and Control", Technical Report CSE-TR-162-93, University of Michigan, EECS Department, Ann Arbor, MI, 1993.

[Wadl92] P. Wadler, "The Essence of Functional Programming", in *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL'92)*, January 1992.

[Wadl94] P. Wadler, "Monads and Composable Continuations", *Lisp and Symbolic Computation*, vol. 7, no. 1, pp. 39–56, January 1994.

[Wadl95a] P. Wadler, "How to Declare an Imperative", in J. Lloyd, editor, *Proceedings of the International Logic Programming Symposium (ILPS'95)*, MIT Press, December 1995.

[Wadl95b] P. Wadler, "Monads for Functional Programming", in J. Jeuring and E. Meijer, editors, *Proceedings of the Båstad Spring School on Advanced Functional Programming*, vol. 925 of *Lecture Notes in Computer Science*, Springer Verlag, New York, NY, May 1995.

[Wall93] C. Wallace, "The Semantics of the C++ Programming Language", Technical Report CSE-TR-190-93, University of Michigan, Department of Electrical Engineering and Computer Science, December 1993.

[Wall95] C. Wallace, "The Semantics of the C++ Programming Language", in E. Börger, editor, *Specification and Validation Methods*, pp. 131–164, Oxford University Press, 1995.

[Watt86] D. A. Watt, "Executable Semantic Descriptions", *Software Practice and Experience*, vol. 16, no. 1, pp. 13–43, January 1986.

[Watt87] D. A. Watt, "An Action Semantics of Standard ML", in *Proceedings of the 3rd Workshop on the Mathematical Foundations of Programming Language Semantics*, vol. 298 of *Lecture Notes in Computer Science*, pp. 572–598, New York, NY, 1987, Springer Verlag.

[Watt94] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach and R. S. Sutor, "A First Report on the A# Compiler", in *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, July 1994.

[Wins93] G. Winskel, *The Formal Semantics of Programming Languages*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1993.

[Wolc87] M. Wolczko, "Semantics of Smalltalk-80", in *European Conference on Object-Oriented Programming (ECOOP'87)*, vol. 276 of *Lecture Notes in Computer Science*, pp. 108–120, New York, NY, 1987, Springer Verlag.

# Index of notation

## Category theory

| | | |
|---|---|---|
| $\mathsf{C}$ | category | 27 |
| $dom(f)$ | domain object of arrow $f$ | 27 |
| $codom(f)$ | codomain object of arrow $f$ | 27 |
| $f : x \to y$ | arrow with domain $x$ and codomain $y$ | 27 |
| $f \circ g$ | composition of arrows $f$ and $g$ | 27 |
| $id_x$ | identity arrow on object $x$ | 27 |
| $F : \mathsf{C} \to \mathsf{D}$ | functor from category $\mathsf{C}$ to category $\mathsf{D}$ | 28 |
| $F \circ G$ | composition of functors $F$ and $G$ | 28 |
| $id_{\mathsf{C}}$ | identity functor on category $\mathsf{C}$ | 28 |
| $F^n$ | composition of functor $F$ with itself $n$ times | 28 |
| $\eta : F \dot{\to} G$ | natural transformation between functors $F$ and $G$ | 29 |
| $\eta_x$ | arrow on object $x$ induced by natural transformation $\eta$ | 29 |
| $\alpha \circ \beta$ | composition of natural transformations $\alpha$ and $\beta$ | 29 |
| $\beta \circ F$ | composition of natural transformation $\beta$ and functor $F$ | 29 |
| $F \dashv U$ | functors $F$ and $U$ are adjoint | 30 |
| $x \times y$ | product of $x$ and $y$ | 30 |
| $\langle f_1, f_2 \rangle$ | product mediating arrow between $f_1$ and $f_2$ | 30 |
| $x + y$ | sum of $x$ and $y$ | 30 |
| $[\, f_1, f_2 \,]$ | sum mediating arrow between $f_1$ and $f_2$ | 30 |
| $x_1 \times x_2 \times \ldots \times x_n$ | finite product of $x_1, x_2, \ldots x_n$ | 31 |
| $x_1 + x_2 + \ldots + x_n$ | finite sum of $x_1, x_2, \ldots x_n$ | 31 |

## Domain theory

| | | |
|---|---|---|
| $\sqsubseteq_D$ | ordering relation on poset $D$ | 33 |
| $\bigsqcup P$ | least upper bound of subset $P$ of a poset | 33 |
| $\downarrow x$ | set of elements below $x$ in poset $D$ | 33 |
| $\mathsf{K}(D)$ | the set of compact elements of cpo $D$ | 34 |
| $N \triangleleft D$ | subset $N$ is normal in poset $D$ | 34 |
| $\bot_D, \top_D$ | bottom and top elements of domain $D$ | 34 |

| | | |
|---|---|---|
| $S^{\mathrm{o}}$ | flat domain induced by set $S$ | 34 |
| $\mathbf{O}$ | domain with one element $\bot$ | 34 |
| $\mathbf{I}$ | domain with two elements $\bot$ and $\top$ | 34 |
| $\mathbf{U}$ | domain with three elements $\bot$, $u$ and $\top$ | 34 |
| $\mathbf{T}$ | truth value domain $\{ \mathit{true}, \mathit{false} \}^{\mathrm{o}}$ | 34 |
| $\mathbf{N}$ | flat domain of integer numbers | 34 |
| $\omega$ | poset of integer numbers under $\leq$ relation | 34 |
| $(x_n)_{n \in \omega}$ | chain of elements $x_i$ in poset $D$ | 34 |
| $\mathit{fix}$ | least fixed point operator | 35 |
| $\mathit{clo}$ | closure operator | 35 |
| $D \times E$ | product of domains $D$ and $E$ | 35 |
| $\langle x, y \rangle$ | pair of elements in a product domain | 35 |
| $\langle f_1, f_2 \rangle$ | product function | 35 |
| $D \otimes E$ | smash product of domains $D$ and $E$ | 36 |
| $D + E$ | separated sum of domains $D$ and $E$ | 36 |
| $[\, f_1, f_2 \,]$ | sum selection function | 36 |
| $D \oplus E$ | coalesced sum of domains $D$ and $E$ | 36 |
| $D_\bot$ | lifted domain | 37 |
| $x : D \twoheadrightarrow F(x)$ | dependent function domain | 37 |
| $x : D \ltimes F(x)$ | dependent product domain | 37 |
| $\mathsf{Dom}$ | category of domains and continuous functions | 37 |
| $\mathsf{Dom}^{ep}$ | category of domains and ep-pairs | 37 |
| $\mathsf{SId}$ | category of continuous semi-lattice domains and continuous homomorphisms | 37 |
| $U$ | forgetful functor between categories $\mathsf{SId}$ and $\mathsf{Dom}$ | 37 |
| $\Delta : I \to \mathsf{C}$ | diagram indexed by poset $I$ over category $\mathsf{C}$ | 38 |
| $\mu : \Delta \to x$ | cone over diagram $\Delta$ | 38 |
| $f : \mu \to \nu$ | mediating arrow between cones $\mu$ and $\nu$ | 38 |
| $\Delta^-$ | diagram defined by shifting $\Delta$ | 38 |
| $\mu^-$ | cone defined by shifting $\mu$ | 38 |
| $\mathcal{P}_f^*(S)$ | set of finite non-empty subsets of $S$ | 39 |
| $\sqsubseteq^\natural$ | (convex) powerdomain ordering relation on $\mathcal{P}_f^*(D)$ | 39 |
| $D^\natural$ | (convex) powerdomain of $D$ | 39 |
| $f^\natural$ | powerdomain function | 39 |
| $\{\!\{ x \}\!\}$ | powerdomain singleton for element $x$ | 39 |
| $s \uplus^\natural t$ | powerdomain union of $s$ and $t$ | 39 |
| $P$ | (convex) powerdomain functor | 40 |

| | | |
|---|---|---|
| $w^*$ | application of associative, communtative and idempotent binary relation $*$ to all elements of finite non-empty set $w$ | 40 |
| $\mathsf{P}$ | (convex) powerdomain monad | 40 |
| $ext^\natural$ | powerdomain function extension operator | 41 |

## Monads

| | | |
|---|---|---|
| $\langle M, \eta, \mu \rangle$ | monad with endofunctor $M$, unit $\eta$ and join $\mu$ | 31 |
| $\langle M, unit, * \rangle$ | alternative monad notation, where $*$ is the bind operator | 33 |
| $f \ ; \ g$ | monadic inverse composition of functions | 44 |
| $mfix$ | monadic least fixed point operator | 44 |
| $mclo$ | monadic closure operator | 44 |

## Meta-language

| | | |
|---|---|---|
| $e \triangleright x : D$ | well formed phrase $e$ denotes element $x$ of domain $D$ | 41 |
| $e \rightarrow e_1 \ , \ e_2$ | conditional construct | 41 |
| $\lambda \, I . \, e$ | function abstraction | 42 |
| $e_1 \ e_2$ | function application | 42 |
| **let** ... **in** ... | let structure | 43 |
| **case** ... **of** ... | case structure | 43 |
| $f\{x \mapsto y\}$ | function update | 44 |

## Static semantics

| | | |
|---|---|---|
| **Ide** | domain of identifiers | 47 |
| **TagType** | domain of tag types | 48 |
| **Tag** | domain of tags | 48 |
| $\mathbf{Type}_{dat}$ | domain of data types | 48 |
| **Qual** | domain of qualifiers | 48 |
| $\mathbf{Type}_{obj}$ | domain of object types | 49 |
| $\mathbf{Type}_{fun}$ | domain of function types | 49 |
| $\mathbf{Type}_{den}$ | domain of denotable types | 49 |
| $\mathbf{Type}_{mem}$ | domain of member types | 49 |
| $\mathbf{Type}_{bit}$ | domain of bit-field types | 49 |
| $\mathbf{Type}_{val}$ | domain of value types | 49 |
| $\mathbf{Type}_{ide}$ | domain of identifier types | 49 |
| $\mathbf{Type}_{phr}$ | domain of phrase types | 49 |
| $\mathsf{E}$ | error monad | 50 |
| **Ent** | domain of type environments | 51, 88 |

| | | |
|---|---|---|
| $e_\circ$ | empty type environment | 52, 88 |
| $e[I\ ide]$ | lookup ordinary identifier $I$ in $e$ | 52 |
| $e[I\ raw\ ide]$ | raw lookup ordinary identifier $I$ in $e$ | 52, 88 |
| $e[I\ tag\ \sigma]$ | lookup tag $I$ of type $\sigma$ in $e$ | 52 |
| $e[I\ raw\ tag]\sigma$ | raw lookup tag $I$ of type $\sigma$ in $e$ | 52, 88 |
| $e[I\ tagID\ \sigma]$ | get tag $I$ of type $\sigma$ in $e$ | 53 |
| $e[I \mapsto ide\ \delta]$ | update ordinary identifier $I$ with $\delta$ in $e$ | 53, 88 |
| $e[I \mapsto tag\ \tau]$ | update tag $I$ with $\tau$ in $e$ | 53, 88 |
| $e[I \mapsto fresh\ tag\ \sigma]$ | create a fresh tag $I$ of type $\sigma$ in $e$ | 53, 89 |
| $\uparrow e$ | open a new empty scope enclosed in $e$ | 53, 89 |
| $\downarrow e$ | close scope $e$ | 54, 89 |
| $isLocal(e, I\ ide)$ | check if ordinary identifier $I$ is local in $e$ | 54, 89 |
| $isLocal(e, I\ ide)$ | check if tag $I$ is local in $e$ | 54, 89 |
| **Enum** | domain of enumeration environments | 54 |
| $\epsilon_\circ$ | empty enumeration environment | 54 |
| $\epsilon[I \mapsto n]$ | update identifier $I$ with value $n$ in $\epsilon$ | 54 |
| **Memb** | domain of member environments | 55 |
| $\pi_\circ$ | empty member environment | 55 |
| $\pi[I]$ | lookup identifier $I$ in $\pi$ | 55 |
| $\pi[I \mapsto m]$ | append identifier $I$ of type $m$ to $\pi$ | 55 |
| $\Downarrow \pi$ | decompose $\pi$ by extracting the first member | 55 |
| **Prot** | domain of function prototypes | 56 |
| $p_\circ$ | empty function prototype | 56 |
| $p \ll \tau$ | append parameter of type $\tau$ to $p$ | 56 |
| $\tau \prec p$ | prepend parameter of type $\tau$ to $p$ | 56 |
| $\tau_1 \subseteq \tau_2$ | data type $\tau_1$ is included in $\tau_2$ | 62 |
| $c \in \tau$ | constant $c$ can be represented by data type $\tau$ | 62 |
| $q_1\ \&\ q_2$ | combine type qualifiers $q_1$ and $q_2$ | 67 |
| $q_1 \subseteq q_2$ | check if qualifier $q_1$ is included in $q_2$ | 67 |

## Typing semantics

| | | |
|---|---|---|
| $e \vdash phrase : \theta$ | *phrase* is attributed type $\theta$ in type environment $e$ (main typing relation) | 95 |
| $v := z$ | the static semantic valuation $z \in \mathsf{E}(D)$ produces the (non-error) value $v \in D$ | 96 |
| $e \vdash I \triangleleft \delta$ | identifier $I$ is associated with type $\delta$ in type environment $e$ | 96 |
| $\pi \vdash I \triangleleft m$ | identifier $I$ is associated with type $m$ in member environment $\pi$ | 97 |

| | | |
|---|---|---|
| $e \vdash E \gg \tau$ | expression $E$ can be assigned to an object of type $\tau$. | 97 |
| $e \vdash E = \texttt{NULL}$ | expression $E$ is a null pointer constant | 97 |
| $e \vdash T \equiv \phi$ | type name $T$ denotes type $\phi$ in type environment $e$ | 97 |
| $e \not\vdash \mathit{phrase} : \theta$ | it is not possible to derive $e \vdash \mathit{phrase} : \theta$ | 97 |

## Dynamic semantics

| | | |
|---|---|---|
| $\mathbf{Obj}$ | domain of object identifiers | 127 |
| $\mathbf{Fun}$ | domain of function identifiers | 128 |
| $\mathbf{Addr}$ | domain of addresses | 128 |
| $\mathbf{Offset}$ | domain of object offsets (in bytes) | 128 |
| $\mathbf{BitOfs}$ | domain of bit-field offsets (in bits) | 128 |
| $[\![\tau]\!]_{dat}$ | dynamic domain corresponding to data type $\tau$ | 129 |
| $[\![\alpha]\!]_{obj}$ | dynamic domain corresponding to object type $\alpha$ | 129 |
| $[\![f]\!]_{fun}$ | dynamic domain corresponding to function type $f$ | 129 |
| $[\![\phi]\!]_{den}$ | dynamic domain corresponding to denotable type $\phi$ | 129 |
| $[\![m]\!]_{mem}$ | dynamic domain corresponding to member type $m$ | 129 |
| $[\![v]\!]_{val}$ | dynamic domain corresponding to value type $v$ | 129 |
| $\mathbf{VC}$ | domain of characters | 129 |
| $\mathbf{VSC}$ | domain of signed characters | 129 |
| $\mathbf{VUC}$ | domain of unsigned characters | 129 |
| $\mathbf{VSS}$ | domain of short integers | 129 |
| $\mathbf{VUC}$ | domain of unsigned short integers | 129 |
| $\mathbf{VSI}$ | domain of integers | 129 |
| $\mathbf{VUI}$ | domain of unsigned integers | 129 |
| $\mathbf{VSL}$ | domain of long integers | 129 |
| $\mathbf{VUL}$ | domain of unsigned long integers | 129 |
| $\mathbf{VF}$ | domain of low precision real numbers | 129 |
| $\mathbf{VD}$ | domain of normal precision real numbers | 129 |
| $\mathbf{VLD}$ | domain of high precision real numbers | 129 |
| $\mathbf{VE}_{\epsilon}$ | domain of enumerated values of type $\epsilon$ | 129 |
| $\mathsf{V}$ | value monad | 130 |
| $\mathsf{P}$ | powerdomain monad | 130 |
| $\mathbf{S}$ | domain of program states | 131 |
| $\mathbf{C}$ | domain of continuations | 132 |
| $\mathbf{A}$ | domain of program answers | 132 |
| $\mathsf{C}$ | continuation monad | 133 |

| | | |
|---|---|---|
| $c_1 \parallel_{\mathsf{C}} c_2$ | non-deterministic option between elements of domain $\mathsf{C}(D)$ | 134 |
| $\mathsf{R}$ | resumption monad transformer | 134, 136 |
| $r_1 \bowtie_{\mathsf{R(M)}} r_2$ | interleaving of computations produced by monad $\mathsf{R}(M)$ | 145 |
| $\mathsf{G}$ | monad of expression computations | 145 |
| $[\![e]\!]_{Ent}$ | domain of dynamic type environments based on $e$ | 147 |
| $[\![e \Downarrow I]\!]$ | dynamic domain corresponding to the type of $I$ in $e$ | 147 |
| $e \uparrow \rho$ | open a new dynamic type environment based on $e$ enclosed in $\rho$ | 147 |
| $e \downarrow \rho$ | close the dynamic environment $\rho$ based on $e$ | 147 |
| $[\![\pi \Downarrow I]\!]_{dat}$ | dynamic domain corresponding to the data type of $I$ in $\pi$ | 149 |
| $[\![\pi \Downarrow I]\!]_{mem}$ | dynamic domain corresponding to the member type of $I$ in $\pi$ | 149 |
| $[\![p]\!]_{Prot}$ | domain of dynamic function prototypes based on $p$ | 149 |
| $[\![p \Downarrow i]\!]$ | dynamic domain corresponding to the type of the $i$-th argument in $p$ | 149 |
| $\mathbf{Arg}$ | domain of arguments of unspecified type | 150 |
| $\mathbf{Cod}$ | domain of code environments | 150 |
| $\xi[d_f]$ | lookup function $d_f$ in $\xi$ | 150 |
| $\xi[d_f \mapsto w]$ | update function $d_f$ with $w$ in $\xi$ | 150 |
| $\mathbf{ScopeId}$ | domain of scope identifiers | 150 |
| $\mathbf{Scope}$ | domain of scope information | 150 |
| $\mathbf{SC}$ | domains of scope-bound continuations | 152 |
| $\mathsf{K}_\tau$ | monad of statement computations | 153 |
| $\mathsf{L}_\tau$ | monad of statement auxiliary information | 153 |
| $\mathbf{Lab}$ | domain of label environments | 156 |
| $\ell_\circ$ | empty label environment | 156 |
| $\mathbf{Cas}_\tau$ | domain of case label environments | 157 |
| $\omega_\circ$ | empty case label environment | 157 |

# Index of terms

# Index of functions

**Dynamic semantic functions**

**Auxiliary functions**