Designing and Prototyping a Functor Language Using Denotational Semantics

Diplomarbeit

zur Erlangung des akademischen Grades
"Diplom-Ingenieur"
in der Studienrichtung
Technische Mathematik

eingereicht von

Wolfgang Stöcher

angefertigt am
Forschungsinstitut RISC-Linz
Institut für Mathematik
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

eingereicht bei
Doz. Dr. Heinrich Rolletschek

Mai 1997

gesponsort von: Fonds zur Förderung der Wissenschaftlichen Forschung,
Österreichische National Bank (FWF-Nr P11414-TEC, ÖNB-Nr FWF442)
online verfügbar (Postscript und HTML) unter:
`http://www.risc.uni-linz.ac.at/people/wstoech/diploma`

Abstract

Functors are parameterized modules. Especially for mathematical software the concept of functors has increasing importance. We develop here step by step the typing rules and denotational semantics of an imperative core language with a powerful type system, which is then extended by introducing modules and functors. Small example programs are given and tested with straight forward implementations of the type checker written in PROLOG and an interpreter written in SML.

The main achievements of this thesis are the following:

Modularity and Genericity: The typing and semantic rules are structured into self-contained units and generic in many ways. The module system is essentially independent from the core language and orthogonal to the type system. The given rules are generic in the particular set of predefined types in the core language.

Implementation Guide: Although the rules are quite general, they still can serve as a guide for implementing an interpreter or compiler. This fact is proven by our prototype implementation closely sticking to the rules described in this thesis.

Contents

List of Figures

Chapter 1

Introduction

This report was written in the frame of the "High-Performance Generic Programming (HPGP)" project pursued at RISC-Linz. The aim of the HPGP-project is to develop a higher-order functor language together with a generic compilation system for the construction of high-performance mathematical software libraries (see also (Hong and Schreiner, 1996)). Here we deal with the development of the semantics for a (not yet higher-order) functor language.

1.1 Motivation and Background

There is a long tradition of developing software for Computer Algebra at RISC-Linz. Let us mention the projects with most significance and influence to the current HPGP project and their software:

SACLIB: This is a C library of algorithms for exact arithmetic for computing with polynomials over integers, rationals and algebraic numbers. SACLIB was initiated by G. Collins and constantly improved by himself and his students. It contains the most efficient algorithms known at the time of its development (Buchberger and Collins, 1993). Its implementation is mainly based on untyped lists. The library was originally used for a software system solving the problem of quantifier elimination over real closed fields (Collings, 1975),(Hong, 1989).

GRÖBNER: This package is also implemented in C, using SACLIB and supports already some kind of polymorphism by precompiler macros (Buchberger and Windsteiger, 1993). It implements several strategies for computing a Gröbner Basis of a polynomial ideal (Buchberger et al., 1982), (Becker and Weispfenning, 1993).

Sturm: This is a set of C++ libraries implementing algorithms for algebraic constraint solving, initiated by H. Hong (Hong et al., 1994). It makes extensive use of C++ templates. The problems that arose in this project gave the main input for the ideas and the motivation of the current HPGP project.

Other Implementations: There are numerous other packages for computer algebra implemented in various languages and environments. Some of them use a high-level computer algebra system like Maple or Mathematica. However, when computing time becomes a critical issue, again C implementations of the corresponding algorithms are used. A prominent example is the Paule-Schorn implementation (Paule and Schorn, 1995) of Zeilberger's algorithm for proving binomial coefficient identities in Mathematica that uses an efficient linear solver implemented in C and linked to the Mathematica kernel via MathLink (Aichinger, 1997).

1.2 Lessons Learned

From these and other projects we learned several things that are important for implementing practically useful software systems for doing mathematics:

- In order to achieve maximal performance a programming language that can be compiled to machine code should be used.
- The already well-structured knowledge of mathematics actually demands a *polymorphic implementation*. Most theorems and algorithms in mathematics are not restricted to a specific domain but do only depend on the overall structure of the involved entities. In mathematics new domains are usually constructed from simpler domains in a well-defined way which is best modeled by the *concept of functors* (see the next section).
- A *static type system* is a valuable aid for the programmer. Again it is the well-structured knowledge in mathematics that not only supports but almost demands this approach. A static type system could eventually play an important role in proving the correctness of implementations.
- To ensure maximal productivity the development system should fulfill some more constraints: minimal compilation times, automatic handling of file dependencies and library construction as far as possible and others; these constraints, however, will not concern us in this report.

1.3 Functors in Mathematics and Computer Science

As mentioned before, many domains in mathematics can be viewed as constructed from simpler domains via a functor. Take for instance the construction of a quotient field from a ring. A can be described by a carrier set together with operations that must fulfill certain properties. The quotient field, i. e. its carrier set and operations, can be defined in terms of the underlying ring, i. e. the ring's carrier set and operations. The properties of the quotient field can be derived from the properties of a ring and the way the quotient field is defined.

Functors in computer science are parameterized modules. After having introduced modules and abstract types (like done in Modula and C++), it turned out that many modules can be defined in an analogous way and that parameterizing a module by other modules can save a lot of work. This lead, for instance, to the (unsatisfying) concept of templates in C++, and to the well worked out module system of SML. Whereas templates in C++ are more or less handled by text replacement, functors in SML can be type checked and compiled separately. A powerful type system ensures high flexibility and still maximal security. The interface is described via signatures, an analogon to the types of values on the level of modules. Signatures can be seen as the equivalent in computer science to abstract domain definitions (like that of a ring) in mathematics. The signatures described in this thesis as well as in SML, however, only cover the carrier set and the operations, but not the properties. We will discuss briefly other approaches and further extensions planned in the HPGP project in Section 6.4.

Functors have lately gained also attraction from the side of automated theorem proving. (Buchberger, 1996) introduces the concept of functors as a means not only for defining computational structures but also as a means for structuring knowledge bases in systems for theorem proving. It is shown how the concept of a functor in a style similar to SML functors can be implemented in Mathematica with some additional flexibility in the definition of the carriers of domains. These ideas are pursued in the current Theorema project (Buchberger, 1997), in which a frame for "doing mathematics", i.e. computing, solving, and proving, is implemented. The notion of functor is the crucial unifying principle.

1.4 Aims

The main aim of this report is, loosely speaking, to develop the semantics of a functor language structured in such a way that it becomes essentially independent of the semantics of the underlying core language. The semantics should be defined in such a way that it guides a corresponding implementation. Our work includes the design of adequate language constructs and the development of the corresponding typing rules. We restrict ourselves here to abstract syntax and the core functionality in order to gain an understanding of the concepts and build the basis for the implementation of a compiler.

The more concrete goals for the rules in this report were:

C/PASCAL-like Core Language: The core language should contain the essential constructs of an imperative core language with a static type system.

Independent Functor Language: The functor language (also referred to as the *module system*) should be independent from the core language. This should make it possible to put the functor language on top of any programming language. It turned out that this was a very demanding goal and it is one of the major achievements of our work. The functor language described in this thesis is based on the typing and semantic rules of the core language, but structured in such a way that it can be easily adopted.

Guide for Implementing: The rules developed here should be useful for implementing a compiler later on. This demands an algorithmic approach with as little indeterminism as possible. Although we studied the semantics of SML (Milner et al., 1990) with great pleasure and appreciate this work, we felt that this approach was too abstract for our intentions. While the rules given there are very elegantly abstracting the meaning of the language from other issues like identifier naming and store handling, it takes a lot of informal commentaries (Milner and Tofte, 1991) to understand the details and how to implement the rules. Here we hope to be much closer to an actual implementation of a compiler and have the rules still readable.

Generic in Predefined Types: Although the rules should be as concrete as possible, we still tried to abstract from certain issues. The particular set of predefined types of the language should not influence the rules.

Designing for Efficiency: There are several issues that have to be considered, in order to make the rules applicable efficiently in a real-life development environment. In order to allow separate compilation, and still enable efficient building of a program from components, information such as the size of the representation of abstract types needs to be forwarded to other modules. This allows more direct access to the structure of an abstract type, instead of putting a pointer on top of each type (thus enforcing uniform size, as done in SML).

1.5 Achievements

While working on the typing and semantic rules also several other issues arose and some of them found a satisfying solution. The rules are generic with respect to several issues, not only predefined types.

1.5.1 Genericity

The given rules are generic with respect to several aspects:

Predefined Types: We structure the type domain by introducing several layers of kinds of types. Especially the typing rules make extensive use of these different layers. The bottom layer is the one of predefined types. The operations on it are not explicitely specified, but handled in a generic way. Subsection 3.3.5 describes how to add a predefined type.

Store Implementation: Conventional models of a store like a list of values or a mapping from locations to values are too specific and usually do not perform practical memory management, anyway. We handle the store as an "abstract datatype" that is specified by an "interface". This frees us from also dealing with memory management which would involve issues such as strategies for dynamic allocation. See Section 3.1 for the details.

Namespaces: The way identifiers are organized in namespaces is only reflected by a single operator, namely $\uplus$ that is used for context and environment construction. Its definition decides whether the same name may be used for several kinds of objects (types, functions, and others), whether identifiers may be overloaded or just used to denote one object (see Subsection 2.5).

1.5.2 Implementation Reference

This issue actually cannot be evaluated before a compiler has been implemented. Part of it, however, has already been done. This report was developed in parallel with the implementation of a type-checker for the language *Cora* (Stöcher, 1995), that implements the constructs described in Chapter 3 and part of the constructs of Chapter 4. The feedback in both directions was very helpful. Furthermore, prototypes of a type-checker and an interpreter were implemented using the given rules in a rather straight forward way. Again the feedback in both directions was very important: the rules gave the structure of the prototype implementation, the implementation itself helped to reveal holes and errors in the rules. Actually the implementation is in complete correspondence to the rules and auxiliary functions and predicates given in this report. The implementation is described in Chapter 7, the source code can be found in Appendix B.

1.5.3 Variables and Call By Reference

We basically follow the ideas presented in (Schmidt, 1994). There the distinction between left-hand side and right-hand side of a variable is overcome by introducing the @ operator for looking up a variable's value. We go one step further by removing this construct from the language again and let the type-checker automatically reintroduce it. These scheme also plays together well with the handling of call by reference (see Section 3.4 and Section 3.5).

1.5.4 Type System

The type system presented in this thesis allows static and therefore strong type checking; it is mainly developed in Chapter 4. The type system is characterized by the type domains and relations for type equivalence and type structure. Apart from these issues it is independent from the constructs presented in the other chapters. The type system also plays together well with the module system. Only the definition of type equivalence needs to be extended for handling long identifiers and type constraints.

1.5.5 Module System

The module system concentrates on abstracting and parameterizing declarations, thus introducing modules and functors, respectively. It is essentially independent from the core language and orthogonal to the type system. Signatures are needed for specifying the parameters of functors and controling the visibility of components in a module. We managed to treat them rather independently from the core issues in defining modules and functors.

1.6 The Purpose of Denotational Semantics

With the increasing complexity of current programming languages also the use of formal methods for defining the semantics has become more important (as done for SML in (Milner et al., 1990)). Formal methods allow the detection of design errors already at an early stage and are a good guide towards a simple design. At the same time a formal specification of the semantics can be used as a reference manual for the implementer of an interpreter or a compiler for that language. It thus serves as a contract between the designer, the implementor and the user of a language. The semantics of the functor language in this thesis are given in terms of *Denotational Semantics* ((Schmidt, 1986), (Schmidt, 1994)).

The resulting formal semantic rules allow **rapid prototyping**. Writing denotational semantics is in some sense similar to (functional) programming: One has to define

- auxiliary functions ("subroutines"),
- domains ("datatypes")
- and eventually the semantics ("program")
- in formal syntax ("programming language").

Therefore the same problems in developing appear:

- definitions are not correctly applied ("type errors"),
- wholes in the concept ("missing declarations")
- and lots of other small mistakes and errors like typos ("syntax errors", "bugs").

While in programming one has the compiler and test runs to assure oneself about the integrity and correctness of one's concepts, mere denotational semantics written on paper are hard to check for correctness and the designed language constructs are even harder to check for usefulness.

Therefore the design of language constructs benefits from the feedback of the implementation of those constructs in a compiler or interpreter and and of the application in programs. Since the denotational semantics look (and actually are) like programs we tried to find a simple way to make them executable. It turned out that an implementation of the typing rules in PROLOG and the actual semantics in SML can be done very fast. While the development of the typing and semantic rules was very demanding, they could be implemented rather straight forwardly.

1.7 What follows

Roughly speaking our approach can be summarized as "going from a core language towards a module system". After having laid the basis for working with

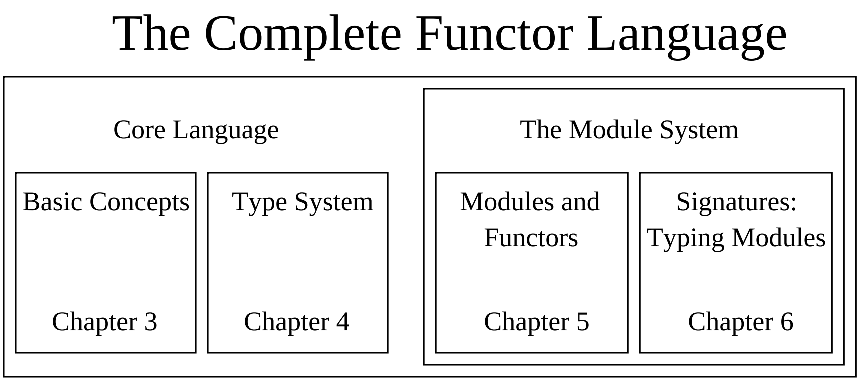

Figure 1.1: Structure of the Functor Language

denotational semantics, we start with the basic concepts of a C-like imperative core language and then extend it with a powerful type system (Chapters 3, 4). As a the next step we introduce the basic concepts of a module system with functors; the syntax and basic ideas are strongly influenced by SML, the typing and semantic rules have been developed independently, however (Chapter 5). With the last step the module system is completed by typing the modules via signatures (Chapter 6). See also Figure 1.1 for an overview.

More detailed this report is structured as follows:

Denotational Semantics (Chapter 2): This is an introduction to the method of denotational semantics. We discuss its relevance to language design and give a brief overview over the history and the basic results. Via simple examples we introduce the methodology of denotational semantics step by step and the notation used in this report. Also the basic auxiliary functions are defined.

Imperative Core Language (Chapter 3): In this chapter we develop the denotational semantics of the core language including error handling and I/O. The development of the type system is delayed until to the next chapter. A basic type system, however, is needed anyway and the type domain is embedded into the syntax instead of extra type expressions. For the imperative core language we define and state the properties '*Unicity of Typing*' and '*Soundness of Typing*', and roughly show how to prove them. These properties are also traced further on when the language is extended in the following chapters.

Core Type System (Chapter 4): Here the full type system is developed, together with type expressions in the syntax. The type system covers most features of common imperative languages like C or PASCAL. Some features, however, go beyond.

Modules and Functors (Chapter 5): Introducing modules and functors is now a rather simple application of the *abstraction principle*. The concept is similar to the modules system in SML. Signatures, like before types, are at first only introduced as a type domain and embedded as themselves into the syntax.

Signatures (Chapter 6): Here we give the syntax and typing rules for signatures and discuss their handling in more detail. Also a construct for expressing type constraints (sharing) is introduced.

Making the Rules Executable (Chapter 7): This chapter explains the implementation of the rapid prototypes. We used Prolog for the type-checker, and SML for the interpreter.

Reference Manual (Appendix A): In the appendix the complete syntax, typing and semantic rules for the whole language are summarized.

Prototype (Appendix B): This chapter of the appendix contains the source-code listings of the prototype implementations.

Chapter 2

Denotational Semantics

2.1 Language Design

The design of a programming language is not a straightforward, deterministic process. There are many issues involved that need constant feedback from the implementers of compilers or interpreters and from the users programming in that language. The following issues are involved:

natural and unambiguous syntax: Although this issue might seem trivial, it should not be neglected, since it is of considerable importance for the acceptance of the language among programmers. Still we are convinced that the underlying concepts formalized by denotational semantics are more important for the understanding of a language. Therefore we will not deal with this issue in this thesis.

clarity and orthogonality of constructs: The language constructs must have a well-defined meaning. Their effects should be as local as possible and not constrained by exceptions or influenced by other constructs. This becomes extremely important when writing and maintaining large pieces of software. Formalizing the semantics shows what information needs to be passed between different constructs.

"good" type system: Depending on the area of application a suitable type system should be developed. The type system is a valuable tool in early error detection. Of course, there is always a trade-off between flexibility and security. Software for graphical user-interfaces has other demands than mathematical software. Whereas inheritance and dynamic typing will be adequate in the former case, static type checking together with parametric polymorphism will suit best in the latter case.

Usually a successful language has gone through several stages or has even evolved from several other experimental languages that will never gain broader attention. New features have to be implemented, tested for their usefulness and interaction with other features, changed or even replaced by other constructs until they become a fixed part of the language. A well documented example of the development of a programming language over several years is the functional programming language SML (Standard Meta-Language). The constructs presented in this report are to be understood as part of such an evolving process.

2.2 History and Basic Results

The field of "Denotational Semantics" was created by Christopher Strachey in the 1960's. His key intention could be summarized as separating the language definition from its implementation. He strived towards formalizing the semantics of programming languages by attaching a "meaning" to a program in a mathematical framework. In those days it was not always clear how to express such a meaning (see for instance the introductory words in (Stoy, 1997)). His work and ideas were soon continued and extended by many others, among them Dana Scott, who developed the theory of the least fixed points. We will only briefly mention the necessary results in order to understand the construction of compound and recursive domains. What follows is actually a brief summary of the presentation in the introductory text (Schmidt, 1986), where all the details can be found.

2.2.1 Compound Domains

There are three basic ways of constructing new domains from existing ones: product, disjoint union and function space. We will also consider two special constructions, namely finite lists and lifting, that can be reduced to the three basic constructions. Each construction carries with it a set of operations for assembling and disassembling elements of the compound domain.

Product

The product construction takes two or more component domains and builds a domain of tuples from the components. In mathematics this is referred to as the "Cartesian product", in computer science and programming languages this is usually referred to as a "record".

Let $a \in A$ and $b \in B$. We simply write (a, b) to denote an element of $A \times B$. In order to extract the components of an element $c \in A \times B$ we will mostly use *pattern matching* instead of access functions, i. e. we write $c = (a, b)$ and then simply say a instead of something like $c \downarrow 1$. Otherwise we prefer symbolic names for the components with record notation: $r.s$ denotes the store (third) component of the runtime information $r \in \mathit{Value}^* \times \mathit{Value}^* \times \mathit{Store}$. But this we will state explicitly for each domain anyway.

Disjoint Union

The construction for unioning two or more domains into one domain is *disjoint union* (or *sum*). In programming languages this is usually referred to as a "union".

For domains A and B the domain $A+B$ denotes a collection whose members are the elements of A and the elements of B, labeled to mark their origins. The classic representation of this labeling is the ordered pair $(0, a)$ for $a \in A$ and $(1, b)$ for $b \in B$. Usually one defines assembly operations like

$\mathrm{in}A : A \rightarrow A + B$,

$\mathrm{in}A(a) = (0, a)$.

The disassembly operation is usually defined via a case distinction and pattern matching. Let $c \in A+B$, and let $f(x) = e_1$ and $g(x) = e_2$ be definitions of $f : A \rightarrow D$ and $g : B :\rightarrow D$, then

case c of

$\mathrm{is}A(x) :\ e_1\ |$

$\mathrm{is}B(x) :\ e_2$

represents a value in D.

For simplicity we will omit these functions since it will always be clear from the context when a value has to be embedded into a corresponding disjoint union or the other way around.

Finite Lists

An important example combining both, the product and disjoint union construction, is the construction of finite lists. Given a domain A, the domain A^* of all finite lists over elements of A can be constructed as follows:

$$A^* := \{()\} + A + A \times A + \ldots$$

Function Space

The function space builder is fundamental to all semantic definitions.

For domains A and B, the function space builder, denoted by the symbol $\rightarrow$, creates the domain $A \rightarrow B$, a collection of functions from domain A do codomain B. The associated disassembly operation is just function application:

$\_(\_) : (A \rightarrow B) \times A \rightarrow B$,

which takes $f \in A \rightarrow B$ and an $a \in A$ and produces $f(a) \in B$.

An important property of function domains is the principle of extensionality: for any f and g in $A \rightarrow B$, if for all $a \in A, f(a) = g(a)$, then $f = g$. In other words: functions are understood in terms of their argument-answer behavior.

The assembly operator can best be expressed in the notation of the lambda-calculus: An abstraction of the form $f = (\lambda x.e)$ is in $A \rightarrow B$, if e is an expression containing occurrences of an identifier x, such that whenever a value $a \in A$ replaces the occurrences of x in e, the resulting value $[a/x]e \in B$. Instead of writing $f = (\lambda x.e)$ we will use the short-hand notation $f(x) = e$.

Lifting and Strictness

The meaning of a program may also be undefined. This happens when, looking at the semantics from an operational point of view, the program does not terminate. In order to also model this behavior we introduce the element $\perp$ (read "bottom") to represent undefinedness. The addition of $\perp$ to a domain can itself be formalized as a domain-building operation.

For domain A, the lifting domain builder $(.)_\perp$ creates the domain $A_\perp$, a collection of the members of A plus an additional distinguished element $\perp$. The elements of A in $A_\perp$ are called *proper elements*; $\perp$ is the *improper element.*

It would go to far here to give formal definitions of a disassembly or assembly operator, but let us remark briefly on strictness. An operation that maps a $\perp$ argument to a $\perp$ answer is called *strict*; otherwise it is called *non-strict.* The common programming languages (more precise: their implementations or semantics) like C, PASCAL, etc. are mostly strict: whenever part of the execution does not terminate, the whole program does not terminate. The only exception is the evaluation of boolean expressions like $a\&b$, where b is not evaluated if a evaluates to false. Hence the evaluation of $a\&b$ might terminate, even if the evaluation of b does not, and therefore the operator & is non-strict.

2.2.2 Recursive Domains

It is tempting to define the domain of lists over a domain A as $\mathit{Alist} = \mathit{Nil} + (A \times \mathit{Alist})$. This definition describes the internal structure of the lists better than the A^* domain. It clearly shows why operations such as *cons*, *isNil*, *head* and *tail* are essential for assembling and disassembling lists. Furthermore, when coming to more complex domain definitions, e. g. domains that recursively depend on each other, a non-recursive definition would be very cumbersome. It is a lot easier to write something like

$A = F(B), \quad B = G(A),$

instead of

$A_{n+1} = F(B_n), \quad B_n = G(A_n)$ for $n \geq 0$,

where A_0 and B_0 are non-recursive, and finally

$A = A_0 + A_1 + \cdots, \quad B = B_0 + B_1 + \cdots.$

However, recursive definitions are not as harmless as they might seem on a first glance. Let us for simplicity look at an example of a recursive function definition for $f : \mathbb{N} \to \mathbb{N}_\perp$:

$$f(x) = \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } f(x+1).$$

Viewing this definition as a program, it is clear that $f(0) = 1$ and for $x > 0$ the value of $f(x)$ is undefined, i. e.

$$f(x) = \begin{cases} 1, & x = 0 \\ \perp, & x > 0 \end{cases} .$$

Viewing the recursive definition of f as an equation in f for arbitrary x, we have suddenly an infinite number of solutions for f. One other solution, for instance, would be $f(x) \equiv 1$. Analogously a recursive domain definition might actually have more than one domain satisfying it.

This problem is resolved by the *least fixed point* theory. One can introduce a partial ordering among all solutions (*fixed points*) of a recursive definition and show that they have a common minimum with respect to that ordering that is also a fixed point. This least fixed point is exactly the solution we would

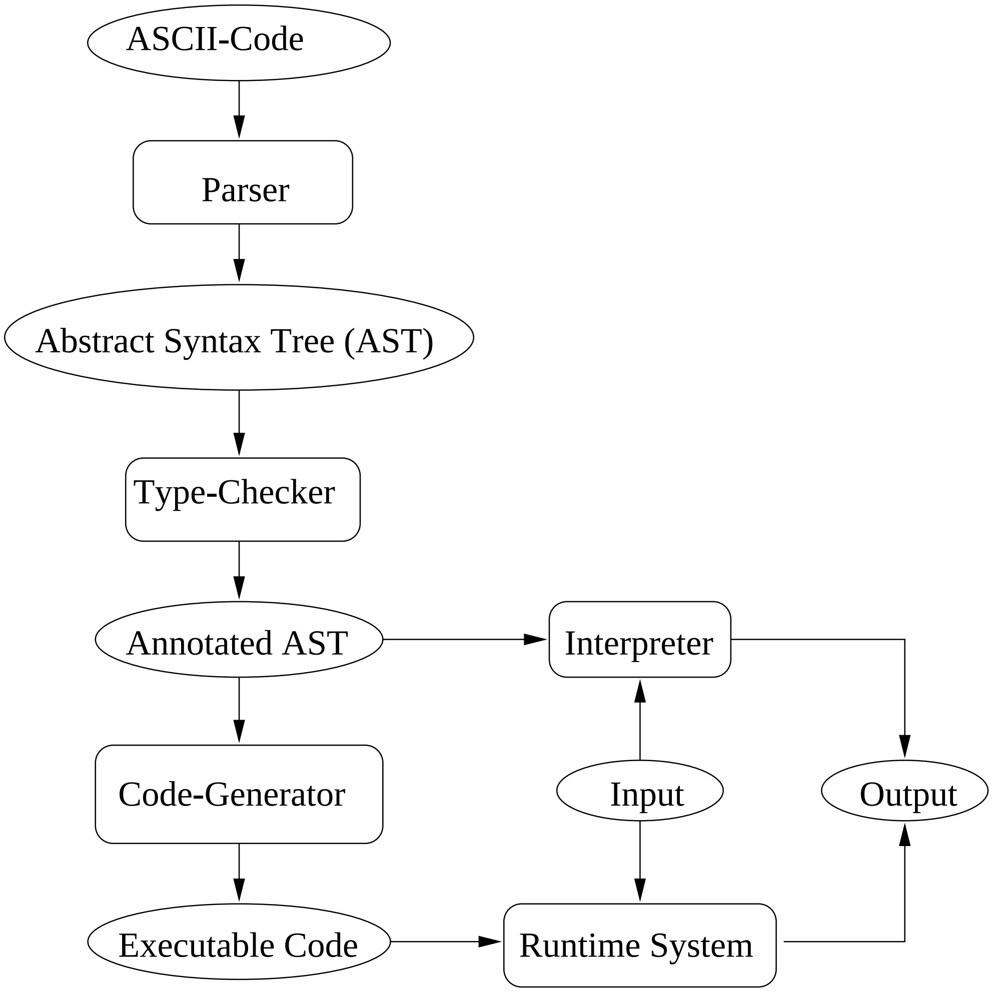

Figure 2.1: Translation and Evaluation of Programs

expect from an operational point of view. This method also works for mutually recursive definitions.

The theory of least fixed points is due to D. Scott. It is based on the theory of lattices (partially ordered set where any two elements have a minimal upper bound and a maximal lower bound) and continuous functions on them (see (Stoy, 1997) and (Schmidt, 1986)).

2.3 The Rules Defining a Language

Usually a program goes through three major steps until its output has been computed: it is parsed into an abstract syntax tree, then or maybe already at the same time type-checked yielding an annotated abstract syntax tree. This annotated tree can be evaluated by an interpreter. For reasons of efficiency the annotated syntax tree is usually translated into machine code before evaluation ("execution"), but this is not essential. See Figure 2.1 for an overview of these processes.

These three major steps are reflected by three sets of rules: the syntax rules, the type inference rules ("typing rules") and the semantic rules ("semantics"). We do not want to discuss syntactic issues in this report, therefore we assume that the programs are already given as abstract syntax trees. The definition of the abstract syntax is actually contained in the typing rules, but for reasons of convenience we state them explicitly; furthermore the abstract syntax as we give it in this report is close to concrete syntax.

2.3.1 Syntax Rules

The (abstract) syntax rules are given in BNF grammar. For a few domains that have a list-like structure and where the single elements can be treated more or less in parallel (e. g. parameter lists), we will use a more convenient notation. These rules specify how abstract syntax trees may be built.

Let us consider the following simple functional language (the concepts will be more clear when not introducing a store at the same time):

I	...	the syntax domain of identifiers
N	...	the syntax domain of numerals (0,1,...)
E	::=	I | N | -E | E_1+E_2 | E_1=E_2 |
		if E_1 **then** E_2 **else** E_3 | **let** I=E_1 **in** E_2

The intention should be obvious: numerals can be negated, summed up, compared or assigned locally to an identifier; the **if**-construct allows branching. In the following subsection we will develop some of the corresponding typing and semantic rules.

2.3.2 Type Inference Rules

Not all abstract syntax trees allowed by the above syntax rules make sense as a program. It is, for instance, not assured that identifiers appearing in expressions have been declared before or are used in a place according to their type. In the above sample language several unwanted expressions can be formed when just looking at the syntax rules, for instance (1=2)+3[1] will not make sense (unless an implicit conversion from a truth value to an integer value is defined). The reason is a typing failure: an expression of type "boolean" appears in a place where an expression of type "integer" is expected.

The typing rules tell which abstract syntax trees are well typed, i. e. well formed programs. Therefore each node in the abstract syntax tree is annotated with a type according to those rules. In the above sample language one would for

[1]Note that the parenthesis are not part of the syntax but show how the expression is grouped.

instance demand that the two summands in $\mathrm{E}_1 + \mathrm{E}_2$ must be of type integer. Formally we could write this as

$$(\mathrm{E}1)\ \frac{\mathrm{E}_1 : int \qquad \mathrm{E}_2 : int}{\mathrm{E}_1 + \mathrm{E}_2 : int},$$

where $\mathrm{E} : int$ means that expression E is of type integer. Read the above typing rule as: "If (expression) E_1 is of type integer and (expression) E_2 is of type integer then $\mathrm{E}_1 + \mathrm{E}_2$ is of type integer."

The rules for numerals are the following ones:

$$(\mathrm{N}1)\ \mathrm{N} : num\ \text{, if } \mathrm{N} \in \{0, 1, \ldots\}$$

$$(\mathrm{E}2)\ \frac{\mathrm{N} : num}{\mathrm{E} : int}$$

For dealing with identifiers we need an additional concept, namely that of a *context*. Since the type of an identifier depends on a previous declaration, the type bindings $(\mathrm{I} : \tau)$ are collected in a type assignment $\pi = \{\mathrm{I}_1 : \tau_1, \ldots\}$, a finite set of type bindings that is also referred to as the *context*. We will use Greek lower letters like τ as typed variables for type domains. In the case of the sample language above τ stands for *int* or *bool*. The type of an expression actually does not only depend on its structure, but also on the types of previously defined identifiers. This is expressed by writing:

$$\pi \vdash \mathrm{E} : \tau$$

Read the above formula as: "Within context π (the expression) E has type τ." The above typing rule for adding expressions now actually looks like this:

$$(\mathrm{E}1)\ \frac{\pi \vdash \mathrm{E}_1 : int \qquad \pi \vdash \mathrm{E}_2 : int}{\pi \vdash \mathrm{E}_1 + \mathrm{E}_2 : int}$$

The following two rules show how type assignments are constructed and used:

$$(\mathrm{E}3)\ \frac{\pi \vdash \mathrm{E}_1 : \tau_1 \qquad \pi \uplus \{\mathrm{I} : \tau_1\} \vdash \mathrm{E}_2 : \tau_2}{\pi \vdash \mathbf{let}\ \mathrm{I} = \mathrm{E}_1\ \mathbf{in}\ \mathrm{E}_2 : \tau_2} \qquad (\mathrm{E}4)\ \frac{(\mathrm{I} : \tau) \in \pi}{\pi \vdash \mathrm{I} : \tau}$$

In rule (E3) a local identifier I is bound to an expression E_1. This identifier may be used in expression E_2. To ensure that I is used in a correct place it gets the type of expression E_1 attached and is then added to the context π. One more issue arises here: a simple union $\pi \cup \{\mathrm{I} : \tau_1\}$ would not be correct, since in nested **let**-blocks one might use the same identifier bound to expressions of different types and then the type assignment would contain two different type bindings for one identifier. In order to avoid this the operator $\uplus$ for the combination of contexts is introduced (see (Schmidt, 1994)), such that in $\pi_1 \uplus \pi_2$ the type bindings in π_2 overwrite corresponding type assignments in π_1. A formal definition is given in Subsection 2.5.

2.3.3 Semantic Rules

The meaning of a program we define as, loosely speaking, a function from the input to the output (of course, the domains for input and output still need to be defined). The semantic rules tell how this function is constructed from the program's annotated abstract syntax tree. As we will see, the group of rules of each syntax domain can be viewed as one function.

In the above described language of simple expressions there is actually no input and so the meaning of a program is here just a value. Since we only have integers and booleans, the set of values can be defined like this:

$Value = \mathbb{Z} + \mathbb{B}$,
where $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$
and $\mathbb{B} = \{true, false\}$

Let us start with the numerals. The meaning of a numeral N is just the corresponding integer it represents. We will not go into details such as the syntactic representation, but assume that has been settled. Then we can write

$$\mathcal{N}(\mathrm{N}) = n,$$

where n is the integer value represented by N. The function $\mathcal{N}$ maps N to n, thus is it as mapping from the syntax domain of numerals into $\mathbb{Z}$. Now let us turn to expressions. The value of the expression $\mathrm{E}_1{+}\mathrm{E}_2$ can be easily defined as

$$\mathcal{E}(\mathrm{E}_1{+}\mathrm{E}_2) = \mathcal{E}(\mathrm{E}_1) + \mathcal{E}(\mathrm{E}_2).$$

The function $\mathcal{E}$ is a mapping from the syntax domain of expressions to *Value* defined above. Hence we would have to apply the corresponding disassembly operator for disjoint unions to $\mathcal{E}(\mathrm{E}_1)$ and $\mathcal{E}(\mathrm{E}_2)$ before adding them. But since it is clear from the context in what domains the objects are, we will omit the assembly and disassembly operators.

Note that in the above equation the first $+$ in $\mathrm{E}_1{+}\mathrm{E}_2$ is a syntactic object, whereas the $+$ on the right side denotes the usual addition of integers. In order to avoid confusion between syntactical objects and others, the syntax is usually put inside '$[\![\,]\!]$' parentheses. Rewriting the above equation accordingly and omitting the now superfluous '()' parentheses we get:

$$\mathcal{E}[\![\mathrm{E}_1{+}\mathrm{E}_2]\!] = \mathcal{E}[\![\mathrm{E}_1]\!] + \mathcal{E}[\![\mathrm{E}_2]\!]$$

Now when using the annotated syntax, as constructed by the typing rules, for defining the meaning, i. e. $\mathcal{E}[\![\mathrm{N} : num]\!]$ and $\mathcal{E}[\![\mathrm{E}_1 : int]\!]$, it is clear from looking at the phrase which function is applied. Therefore the semantic rules are usually written like this (cf. (Schmidt, 1986), (Schmidt, 1994)):

$$[\![\mathrm{E}_1{+}\mathrm{E}_2 : int]\!] = [\![\mathrm{E}_1 : int]\!] + [\![\mathrm{E}_2 : int]\!]$$

Before we can give the rules involving identifiers we have to talk about how to handle the binding of an identifier to a value. For typing phrases we introduced a context containing the type bindings of identifiers. The analogous concept for evaluating phrases is called *environment*. An environment e is simply a set of identifier bindings and is passed as an argument to the semantic functions. It is usually denoted the following way:

$$e = \{\mathrm{I}_1 = v_1, \ldots\}$$

Now we can give the rules involving identifiers:

$$\begin{aligned}&[\![\pi \vdash \mathbf{let}\ \mathrm{I} = \mathrm{E}_1\ \mathbf{in}\ \mathrm{E}_2 : \tau_2]\!]\ e = [\![\pi \uplus \{\mathrm{I} = v\} \vdash \mathrm{E}_2 : \tau_2]\!]\ (e \uplus \{\mathrm{I} = v\}),\\ &\qquad \text{where } v = [\![\pi \vdash \mathrm{E}_1 : \tau_1]\!]\ e\\ &[\![\pi \vdash \mathrm{I} : \tau]\!]\ e = v, \quad \text{if } (I = v) \in e\end{aligned}$$

The meaning function for expressions does not evaluate directly to a value anymore, but to a function from the environment to the set of values:

$$\mathcal{E} = [\![\pi \vdash . : \tau]\!] : \mathit{Expr} \to \mathit{Env} \to \mathit{Value},$$

where *Expr* and *Env* denote the set of all annotated expression phrases and the set of environments, respectively. Thus, when also passing the environment e, we again get a value.

In the rule for the **let** construct we first evaluate expression E_1 to the value v. Then we bind the identifier I to this value and add it to the environment e. The operator $\uplus$ works on environments in an analogous way as it does on contexts (see before). Note also that in

$$[\![\pi \uplus \{\mathrm{I} = v\} \vdash \mathrm{E}_2 : \tau_2]\!]\ (e \uplus \{\mathrm{I} = v\})$$

the context π and the environment e are changed accordingly. We will come back to this "correspondence between contexts and environments" in Lemma 3.7.1. The second rule shows how to evaluate an identifier I: its value is simply looked up in the environment e.

2.4 The Principles in Developing Rules

The constructs and rules developed here in this thesis are not a mere collection of formalized intuitions about what a programming language should be capable of and what the constructs actually should "do" or "mean". Apart from some experimental investigations (mainly the rules for array indexing and record qualification) most constructs and rules follow well defined design strategies. We give a brief survey on the principles together before illustrating them with some examples in the subsections below; the following subsections are actually an excerpt of the presentation in (Schmidt, 1994), where a complete treatment and more advanced concepts can be found.

2.4.1 The Abstraction Principle

The abstraction principle is a technique for introducing naming devices ("declarations") into the language by binding identifiers to certain objects. Constants, variables, functions, procedures, types, modules and functors all arise from the abstraction principle. They differ in the domain they abstract and in the way of evaluation. Constants and functions, for instance, will (here in this thesis) both abstract the expression domain. Constants, however, are evaluated eagerly, i. e. at definition time, whereas functions are evaluated lazily, i. e. at each invocation again. An example for the definition and invocation rules for eager evaluation we gave already in the previous section (see the binding in the **let** construct).

The rules arising from the abstraction principle always come in pairs: one for defining an identifier and one for invoking this identifier. Troughout this thesis we will constantly introduce new forms of abstractions. The central Chapters 5 and 6 introducing modules, functors and signatures are mainly an application of this principle in various ways.

2.4.2 The Parameterization Principle

Abstractions gain utility if they can be applied in various situations. Parameters make this possible. The parameterization principle allows the various forms of abstractions to receive parameters. Functions, procedures and functors are such abstractions. The parameterization principle thus extends the rules introduced by the abstraction principle.

The effects on the rules are again pairwise: since the definition is parameterized, the corresponding invocation is extended by arguments. The central construct of this thesis, namely the functor construct parameterizing module definitions, is an application of the parameterization principle.

2.4.3 The Qualification Principle

The qualification principle introduces the notion of *scope*. Practice has proved that functions, procedures, modules and so on benefit from local definitions. The addition of local definitions and their handling is guided by the qualification principle.

A construct that admits local definitions is usually called a *block structure*. The **let** construct in our simple functional language introduces such a block structure. The crucial point is the extension of the context by the operator $\uplus$ that combines two type assignments but lets later definitions take precedence over previous ones.

2.4.4 Illustrating the Principles

It is hard to talk about the mentioned principles in more detail without looking at examples. Let us therefore extend our simple functional language by a declaration domain allowing also the definition of a function:

$$\begin{array}{lll} \mathrm{D} & ::= & \mathrm{I} = \mathrm{E} \mid \textbf{fun}\ \mathrm{I}_1(\mathrm{I}_2{:}\tau) = \mathrm{E} \\ \mathrm{E} & ::= & \ldots \mid \textbf{let}\ \mathrm{D}\ \textbf{in}\ \mathrm{E} \mid \mathrm{I}\,(\mathrm{E}) \end{array}$$

There are now different kinds of declarations possible in a **let** construct, namely constants and functions. For the invocation of a function an additional construct was added to the expression domain. Constants and functions arise from the abstraction principle. The function construct is additionally guided by the parameterization principle. The **let** construct is an application of the qualification principle.

The typing rules for declarations would look like the following:

$$\text{(D1)}\ \frac{\pi \vdash \mathrm{E}:\tau}{\pi \vdash \mathrm{I} = \mathrm{E}:\{\mathrm{I}:\tau\}} \qquad \text{(D2)}\ \frac{\pi \uplus \{\mathrm{I}_2:\tau\} \vdash \mathrm{E}_2:\tau_1}{\pi \vdash \mathbf{fun}\ \mathrm{I}_1(\mathrm{I}_2{:}\tau) = \mathrm{E}:\{\mathrm{I}_1:\tau \to \tau_1\}}$$

The type of a declaration is a type assignment. In case of a function declaration the type binding $\mathrm{I}:\tau$ of the parameter is added to the context for typing the function body E; this can be seen as an application of the qualification principle. Note also that the outer context π is also visible to the function body, thus allowing the body to use also "global" variables. This is not a necessity; functor bodies (see Chapter 5), for instance, will only be typed in the context of the corresponding parameter.

The resulting type assignment of a declaration is added to the local context in the following rule:

$$\text{(E3)}\ \frac{\pi \vdash \mathrm{D}:\pi_1 \qquad \pi \uplus \pi_1 \vdash \mathrm{E}:\tau}{\pi \vdash \mathbf{let}\ \mathrm{D}\ \mathbf{in}\ \mathrm{E}:\tau},$$

where the binding of D is visible to the body E of the **let** construct. This is the purest form of the application of the qualification principle. Now we can give the typing rules for "using" the declarations:

$$\text{(E4)}\ \frac{(\mathrm{I}:\tau) \in \pi}{\pi \vdash \mathrm{I}:\tau} \qquad \text{(E5)}\ \frac{(\mathrm{I}:\tau \to \tau_1) \in \pi \qquad \pi \vdash \mathrm{E}:\tau}{\pi \vdash \mathrm{I}\,(\mathrm{E}):\tau}$$

These rules are the counterparts to the corresponding declarations.

Now let us turn to the semantic rules. Declarations construct environments:

$$\begin{aligned}
&[\![\pi \vdash \mathrm{I} = \mathrm{E}:\{\mathrm{I}:\tau\}]\!]\ e = \{\mathrm{I} = [\![\vdash\!\dashv]\!]:\pi\mathrm{E}\tau\ e\} \\
&[\![\pi \vdash \mathbf{fun}\ \mathrm{I}_1(\mathrm{I}_2{:}\tau) = \mathrm{E}:\{\mathrm{I}_1:\tau \to \tau_1\}]\!]\ e = \{\mathrm{I}_1 = f\}, \\
&\qquad \text{where } f v' = [\![\pi \vdash \mathrm{E}:\tau_1]\!]\ \{\mathrm{I}_2 = v\}
\end{aligned}$$

These two rules show the difference between eager and lazy evaluation. Whereas the expression in the first semantic rule is already evaluated at definition time, the expression in the function declaration is encapsulated in a function f that is evaluated upon invocation. Here are the corresponding rules for invocation:

$$\begin{aligned}
&[\![\pi \vdash \mathrm{I}:\tau]\!]\ e = v, \quad \text{where } (\mathrm{I} = v) \in e \\
&[\![\pi \vdash \mathrm{I}\,(\mathrm{E}):\tau_1]\!]\ e = f([\![\pi \vdash \mathrm{E}:\tau]\!]\ e)
\end{aligned}$$

The semantic rule for the **let** construct is the following:

$$\begin{aligned}
&[\![\pi \vdash \mathbf{let}\ \mathrm{D}\ \mathbf{in}\ \mathrm{E}:\tau]\!]\ e = [\![\pi \uplus \pi_1 \vdash \mathrm{E}:\tau]\!]\ (e \uplus e_1), \\
&\qquad \text{where } e_1 = [\![\pi \vdash \mathrm{D}:\pi_1]\!]
\end{aligned}$$

Note that again the context and the environment are extended accordingly.

Most of the rules introduced in the following chapters can be viewed as variations on the just explained principles.

2.5 Auxiliary Functions

For constructing contexts and environments, we need an operator that combines sets and simultaneously overwrites possible previous bindings of identifiers by new ones. Contexts are used in the typing rules and contain bindings of identifiers to types. Environments are their analogon for the semantics: they contain bindings of identifiers to semantic objects like values, functions and so on. We will denote the corresponding operator in both cases by $\uplus$ and its behavior must be an analogous one in both cases.

Let us assume that an operator $\pm$ has been defined, that combines a context and a type assignment:

$\pi_1 = \pi \pm (\mathrm{I}_1 : \iota_1)$.

Then we can define the operator $\uplus$ combining two contexts using $\pm$:

$\pi_1 \uplus \pi_2 = (\ldots(\pi_1 \pm (\mathrm{I}_1 : \iota_1))\ldots) \pm (\mathrm{I}_n : \iota_n)$,

where $\pi_2 = \{\mathrm{I}_1 : \iota_1, \ldots, \mathrm{I}_n : \iota_n\}$.

The operator $\pm$ can for instance be defined like

$\pi \pm (\mathrm{I}_1 : \iota_1) = (\pi - \{\mathrm{I}_1 : \iota | (\mathrm{I}_1 : \iota) \in \pi\}) \cup \{\mathrm{I}_1 : \iota_1\}$.

Thus any previous binding(s) (in a well constructed context there should always be at most one binding to a specific identifier) to I_1 is/are replaced by the new one.

It is, however, also possible to allow several bindings to one identifier, if they are, for instance, in different name spaces or overloading of functions and procedures is wanted. What kind of name space a type assignment belongs to can be detected from the structure of ι. Suppose that ι has the following structure:

$\iota \quad ::= \quad \tau\, val \mid \sigma fun$

Then one could define the operator $\pm$ in the following way:

$\pi \pm (\mathrm{I}_1 : \tau_1 val) = (\pi - \{\mathrm{I}_1 : \tau val | (\mathrm{I}_1 : \tau val) \in \pi\}) \cup \{\mathrm{I}_1 : \tau_1 val\}$,

$\pi \pm (\mathrm{I}_1 : \sigma_1 fun) = \pi \cup \{\mathrm{I}_1 : \sigma_1 fun\}$.

With this definition an identifier I can be bound to at most one value and several functions. In an actual implementation of a type-checker the definition of $\pm$ should also contain checks for possible errors, like redeclaring a function with the same types of arguments.

It does not matter for the typing and semantic rules in this report, how the operator $\pm$ actually is defined, except for one subtle point: when allowing overloading of functions and/or procedures one has to make sure that two such declarations differ not only in the calling mode of its arguments; either the number of parameters has to be different or at least at one of the parameters the constructed type has to differ. This check is missing in the above exemplary definition of the operator $\pm$. Things will become obvious when looking at the typing rules for parameters and arguments and the corresponding comments in Section 3.4.

Chapter 3

An Imperative Core Language

An imperative (also called procedural) language gets its power from the store. The direct access to the store is abstracted from the programmer via variables. The following features are supported:

I/O and Exceptions: The syntax constructs contain **write** and **read** commands for simple I/O. If the type of the read in value does not match the expected type, a simple exception mechanism terminates the semantic evaluation with a corresponding error token.

Recursion: The typing and semantic rules allow functions and procedures to call themselves.

PASCAL style Argument Passing: The types of parameters together with their passing mechanism ("call by reference" or "call by value") are specified with the definition only and is transparent in the call. Whether a variable's location or value is passed in a function or procedure call depends on the typing of the corresponding parameter.

Variable Handling: Whether a variable denotes its location or its value is transparent in the syntax. This is resolved by the type-checker in natural and uniform way.

The main concept that is missing so far is that of type constructors such as records and arrays. Corresponding type expressions will be added to the language in the next chapter, together with a powerful type system that allows recursive type definitions. Another minor point is that forward declarations of functions and procedures are not supported in the given imperative language, but their lack does not impose severe restrictions.

Here is a brief overview over the following sections:

1. **Store:** the store, the central notion of imperative programming, viewed as an "abstract datatype",

2. **Syntax Rules:** the syntax rules together with explanations of the constructs and an example program,
3. **Preparations:** preparations for the typing and semantic rules including the definition of the type domains and semantic domains,
4. **Typing Rules:** the typing rules and comments,
5. **Unicity of Typing:** proving that the typing rules attach types to phrases in a programs in a unique way,
6. **Semantic Rules:** the semantic rules and comments,
7. **Soundness of Typing Rules:** proving that the semantic rules evaluate to domains as expected by the typing rules.

3.1 The Store

The store, denoted by the set *Store*, plays a central role in imperative programming. We will view the store as an "abstract datatype", i.e. instead of defining what the set *Store* really is (the "representation"), we define how the functions defined on the store interact (the "interface"). The following sections might appear somewhat cumbersome in contrast to just defining the store as a tuple of values (as done in (Schmidt, 1994)) or as a function from locations to values (as proposed in (Stoy, 1997)). But types such as pointers and arrays enforce complex memory management (especially the interplay of dynamic memory allocation and deallocation[1]). Our abstraction frees us from explicitly specifying how memory management is done. Still we give a model for the set *Store* and its access functions in Subsection 3.1.2 in order to show that the specified interface is satisfiable (see Theorem 3.1.1 and Lemma 3.1.1).

3.1.1 Store Interface

Let us assume that a set *Byte* has been defined that describes the values to be stored. The values can be stored and retrieved via locations:

$$Location = \{loc_1, loc_2, \ldots\}$$

The interface of the store consists of the access functions and their domains (see Figure 3.1) and the relationship among them (see figure 3.2).

The function *is_alloc* is actually a predicate that is true iff a location has been allocated. The function *lookup* retrieves the value at the specified location, but does not alter the store. The functions *allocate* and *free* alter the store with respect to the function *is_alloc*, whereas the function *update* alters the store with respect to *lookup*. The change is only at location l, whereas all other locations remain unchanged. This behavior is expressed by the predicate *good*. However, we only care for values at locations that have been allocated.

[1]The C functions `malloc`, `realloc` and `free` are highly non-trivial and even compiler dependent.

$is\_alloc : Location \times Store \rightarrow \mathbb{B}$
$allocate : Store \rightarrow (Location \times Store)$
$free : Location \times Store \rightarrow Store$
$update : Location \times Byte \times Store \rightarrow Store$
$lookup : Location \times Store \rightarrow Byte$

Figure 3.1: Domains of Access Functions to the Store

$$
\begin{array}{l}
\langle\rangle \in Store \text{ (the empty store)} \\
\forall l : \neg is\_alloc(l, \langle\rangle) \\
good(l, s, s') := \quad \forall l' \neq l : is\_alloc(l', s) \Rightarrow \\
\qquad\qquad\qquad [is\_alloc(l', s') \wedge lookup(l', s) = lookup(l', s')] \\
\forall l, s, s' : \quad allocate(s) = (l, s') \Rightarrow \\
\quad \neg is\_alloc(l, s) \wedge is\_alloc(l, s') \wedge good(l, s, s') \\
\forall l, s, s' : \quad free(l, s) = s' \Rightarrow \\
\quad \neg is\_alloc(l, s') \wedge good(l, s, s') \\
\forall l, v, s, s' : \quad update(l, v, s) = s' \Rightarrow \\
\quad (is\_alloc(l, s) \Rightarrow is\_alloc(l, s') \wedge lookup(l, s') = v) \wedge good(l, s, s')
\end{array}
$$

Figure 3.2: Interaction of Access Functions to the Store

That the specified interface is actually satisfiable is assured by the following theorem.

Theorem 3.1.1 (Store)
The interface of the store as described in Figures 3.1 and 3.2 is satisfiable.

Proof (Sketch): It suffices to give a model satisfying the interface. Such a model for the store and its access functions we give in the next subsection, which is proven by Lemma 3.1.1 to satisfy the interface. □

3.1.2 A Sample Model

We give here a simple model of the set *Store* fulfilling the specified interface, but actually not doing any intelligent memory management (see Figure 3.3). A store is modelled by a function from locations to the set of bytes extended by a special value *still_free* indicating that the location has not been allocated yet. The function *allocate* always returns the next unused location, not caring about previously allocated and in the meantime perhaps freed locations. The other functions are self-explaining.

Lemma 3.1.1 (Store Model)
The model given in Figure 3.3 fulfills the store interface defined in Figures 3.1 and 3.2.

Proof (Sketch): The proof consists of two stages: first we have to show that the access functions are over the domains as defined in Figure 3.1, before we can proof that they interact as defined in Figure 3.2. For both stages we will proof the most difficult parts.

$$
\begin{array}{l}
\mathit{Store} = \{f : \mathit{Location}^{\mathit{Byte} \cup \{\mathit{still\_free}\}} \mid \exists N : \forall n \geq N : f(\mathit{loc}_n) = \mathit{still\_free}\} \\
\mathit{store\_size} : \mathit{Store} \rightarrow \mathbb{N} \\
\quad \mathit{store\_size}(f) = \mu N : \forall n \geq N : f(\mathit{loc}_n) = \mathit{still\_free} \\
\mathit{still\_free} \notin \mathit{Byte}, b_0 \in \mathit{Byte} \text{ (a fixed element in } \mathit{Byte}\text{)} \\
\langle\rangle = f_0 : \forall l \in \mathit{Location} : f_0(l) = \mathit{still\_free} \\
\mathit{is\_alloc}(l, f) \Leftrightarrow \neg f(l) = \mathit{still\_free} \\
\mathit{allocate}(f) = (\mathit{loc}_N, f'), \\
\quad \text{where } N = \mathit{store\_size}(f) \\
\quad \text{and } f'(l) = \begin{cases} b_0, & l = \mathit{loc}_N \\ f(l), & \text{otherwise} \end{cases} \\
\mathit{free}(l, f) = f', \\
\quad \text{where } f'(l') = \begin{cases} \mathit{still\_free}, & l' = l \\ f(l), & \text{otherwise} \end{cases} \\
\mathit{update}(l, v, f) = f', \\
\quad \text{where } f'(l') = \begin{cases} v, & l' = l \\ f(l), & \text{otherwise} \end{cases} \\
\mathit{lookup}(l, f) = f(l) \circ p, \\
\quad \text{where } p(v) = \begin{cases} b_0, & v = \mathit{still\_free} \\ v, & \text{otherwise} \end{cases}
\end{array}
$$

Figure 3.3: A Simple Model for the Store

1. Let us show that *lookup* as defined in Figure 3.3 is indeed a function $\mathit{Location} \times \mathit{Store} \rightarrow \mathit{Byte}$; the domain check for the other access functions is also straight forward, however, it involves checking for a finite size. The domain of *lookup* is clearly $\mathit{Location} \times \mathit{Store}$ by definition. For the codomain we have to show that

$$\forall f \in \mathit{Store}, l \in \mathit{Location} : f(l) \circ p \in \mathit{Byte}. \tag{3.1}$$

   f is a store and therefore a function $\mathit{Location} \rightarrow (\mathit{Byte} \cup \{\mathit{still\_free}\})$, p is by definition a function $\mathit{Byte} \cup \{\mathit{still\_free}\} \rightarrow \mathit{Byte}$ (with the additional property $p|_{\mathit{Byte}} = id_{\mathit{Byte}}$) and hence (3.1).

2. Let us show here that the access functions *is_alloc* and *allocate* defined in Figure 3.3 behave as defined in Figure 3.2; the properties for the other access functions can be shown in a similar way.

   For *is_alloc* all we have to show is

$$\forall l : \neg \mathit{is\_alloc}(l, \langle\rangle). \tag{3.2}$$

   Specializing the definition of *is_alloc* to $f = \langle\rangle = f_0$ we get

$$\forall l : \mathit{is\_alloc}(l, f_0) \Leftrightarrow \neg f_0(l) = \mathit{still\_free},$$

   from which (3.2) follows by the definition of $\langle\rangle = f_0$.

   For *allocate* we have to show

$$\begin{gathered} \mathit{allocate}(s) = (l, s') \Rightarrow \\ \neg \mathit{is\_alloc}(l, s) \wedge \mathit{is\_alloc}(l, s') \wedge \mathit{good}(l, s, s') \end{gathered}$$

Let l, s, s' be arbitrarily fixed to $\bar{l}, \bar{f}, \bar{f}'$, respectively, and let us assume that $allocate(\bar{f}) = (\bar{l}, \bar{f}')$ holds. Then it remains to show that

$$\neg is\_alloc(\bar{l}, \bar{f}) \wedge is\_alloc(\bar{l}, \bar{f}') \wedge good(\bar{l}, \bar{f}, \bar{f}') \tag{3.3}$$

By definition of *allocate* we have $\bar{l} = loc_N$, where $N = store\_size(\bar{f})$. By definition of *store_size* we have $\bar{f}(\bar{l}) = still\_free$ and hence by definition of *is_alloc* we get

$$\neg is\_alloc(\bar{l}, \bar{f}).$$

Now $is\_alloc(\bar{l}, \bar{f}')$ and $good(\bar{l}, \bar{f}, \bar{f}')$ can be shown in a similar manner, from which we get (3.3). □

3.1.3 More Access Functions

For ease of use we will now define further access functions that handle blocks of locations. Therefore we need first an increment operation on locations:

$(loc_i)_{+j} = loc_{i+j}$

Now we can define access functions that handle a block of n locations together ($n \in \mathbb{N}$):

$allocate_n : Store \to (Location \times Store)$,
$\quad allocate_n(s) = (l_1, s_n)$,
$\qquad$ where $(l_{j+1}, s_{j+1}) = allocate(s_j), s_0 = s, 0 \le j < n$
$free_n : Location \times Store \to Store$,
$\quad free_n(l, s) = s_n$,
$\qquad$ where $s_{j+1} = free(l_{+j}, s_j), s_0 = s, 0 \le j < n$
$update_n : Location \times Byte^n \times Store \to Store$
$\quad update_n(l, \bar{v}, s) = s_n$,
$\quad$ where $s_{j+1} = update(l_{+j}, v_j, s_j), \bar{v} = (v_0, \ldots, v_{n-1}), s_0 = s, 0 \le j < n$
$lookup_n : Location \times Store \to Byte^n$,
$\quad lookup_n(l, s) = (lookup(l, s), \ldots, lookup(l_{+n-1}, s))$

3.2 Syntax Rules

For simplicity we make at first the type domain τ (defined in Section 3.3) part of the language. Corresponding syntax will be introduced together with an extended type system in the next chapter.

I	::=	i, $\quad i \in$ *Identifier*, the set of identifiers
D	::=	**const** I = E | **var** I : δ | **fun** I P : δ**=**E | **proc** I P**=**C | $\mathrm{D}_1; \mathrm{D}_2$
P	::=	$(\mathrm{I}_1 : \tau_1, \ldots, \mathrm{I}_n : \tau_n)$, $\quad n \ge 0$
A	::=	$(\mathrm{E}_1, \ldots, \mathrm{E}_n)$, $\quad n \ge 0$
E	::=	I | v | **eof** | I A | $*$ E | $\mathrm{E}_1 \circ \mathrm{E}_2$,
		$v \in V_\gamma$ for any predefined type γ,
		$* \in V_{\gamma_1} \to V_\gamma$ is any unary operator,
		$\circ \in V_{\gamma_1} \times V_{\gamma_2} \to V_\gamma$ any binary operator
C	::=	**skip** | I**:=**E | I A | **write** E | **read** I |
		if E **then** C_1 **else** C_2 **fi** | **while** E **do** C **od** |
		with D **do** C **end** | $\mathrm{C}_1; \mathrm{C}_2$
Prg	::=	C

3.2.1 Identifiers and Declarations: I, D

Identifiers abstract other syntax domains. Declarations introduce those identifiers. At the moment we allow abstractions for expressions, locations, parameterized expressions and parameterized command sequences by the **const**, **var**, **fun** and **proc** constructs, respectively. The syntax domain of declarations plays the central role and will be extended throughout this report.

3.2.2 Parameters and Arguments: P, A

For parameter and argument lists a short-hand notation was chosen that is not in BNF form, but the intention should be clear. Note that also an empty list () is allowed for both constructs. The type τ_i of a parameter determines also the passing mechanism. A *loc* flag indicates call by reference. With the parameter list
$(a : int, b : intloc)$
in a corresponding argument list
(x, y)
x and y must both be of type *int*, but x is passed by value, whereas y is passed by reference and must therefore be additionally a variable (x can also be a constant). In the next chapter the *loc* flag will be replaced by an explicit optional keyword, and the remaining type by a type expression.

3.2.3 Expressions: E

Expressions are made up from atomic expressions, function application and operators. The atomic expressions are identifiers, literals (v) and the **eof** ("end of file") construct. The latter is of boolean type and tells whether the input stream is at its end.

3.2.4 Command Sequences: C

The commands consist of the following constructs:

skip: This marks the empty command.

I:=E: That is the usual assignment; I must denote a variable.

I A: The procedure invocation is PASCAL-like. The distinction between "call by value" and "call by reference" is done in the declaration. Arguments that are passed by reference must be identifiers denoting a variable; constants or arbitrary expressions cannot only be passed by value.

write E**:** The value of the expression is written to the output stream.

read I**:** The next value of the input stream is assigned to the variable I. If the types of the value and the variable do not match or the input stream is at its end, an exception is raised and the program terminates abnormally with an error flag.

```
with
  proc fac( n:int, r: intloc ) =
    if n=0 then
      r := 1
    else
      fac( n-1, r );
      r := n*r;
  var in: int;
  var out: int
in
  while( not eof ) do
    read in;
    fac( in, out );
    write( out )
  od
end
```

Figure 3.4: Example Program for the Core Language

if, while constructs: These are the usual branching and looping constructs.

with D **do** C **end:** This construct allows local declarations.

$C_1; C_2$**:** A command can actually consist of several commands. Note that since we are only talking about abstract syntax, we do not get into any ambiguities; which commands belong to the else part of an **if** construct, and which commands follow the **if**-command is clear from the tree-structure of the abstract syntax.

3.2.5 Program

A program is just a command, usually a **with** construct. It is dealt with as a separate entity in order to initialize the store and the output stream.

3.2.6 Example Program

For simplicity the examplary code is given (see Figure 4.1) in text (ASCII) notation, not caring about different fonts as used elsewhere to distinguish between syntax constructs and meta-variables; the structure of the abstract syntax tree is reflected by the indentation.

Since we have not introduced a branching construct for the expression domain (which would actually be trivial), the recursive computation of factorials has to be implemented as a procedure. The procedure *fac* takes two arguments: n is passed by value and r is passed by reference. Note that n cannot be assigned a value; this is prohibited by the typing rules.

3.3 Preparations

3.3.1 Type Domains

The following type domains occur in the first version of the imperative language:

γ	$::=$	$bool \mid int$	predefined types
δ	$::=$	γ	constructed (user-defined) types
τ	$::=$	$\delta \mid \delta loc$	expression types
ρ	$::=$	τ^*	parameter/argument list types
ι	$::=$	$\tau \mid \rho \to \delta \mid \rho \to Cmd$	identifier types

The basic (or predefined) types γ consist in our example of the types *bool* and *int*, but can be easily extended (see Subsection 3.3.5). For basic types also a set of literals V_γ is defined and unary and binary operators. Since we have not introduced any type constructors yet, the set of constructed types δ (including the basic types) is here the same as the basic types; type constructors will be introduced in the next chapter.

The type domain τ covers all types expressions can have. These are the constructed types and references to them. Type domain ρ denotes type lists $(\tau_1, \ldots, \tau_n)$,$n \geq 0$, as occurring in parameters and arguments, and ι denotes the types identifiers can have: namely an expression type τ, a function type $\rho \to \delta$ or a procedure type $\rho \to Cmd$.

The types attached to phrases are the following ones:

$\mathrm{I} : \iota Idf$
$\mathrm{D} : \pi Dec$
$\mathrm{P} : \rho Par$
$\mathrm{A} : \rho Arg$
$\mathrm{E} : \tau Xpr$
$\mathrm{C} : Cmd$

3.3.2 Size of Types and Storing Values

When coming to semantics any type is mapped to its size, namely the number of bytes needed to represent objects of that type. The mapping is done by the function *size* taking an environment and a type (a tree of structure δ) as arguments. The type domain δ of constructed types consists up to now only of the predefined types, so it is sufficient to define

$size(c, bool) = size_{bool}$,
$size(c, int) = size_{int}$,

where $size_{bool}$ and $size_{int}$ are constants that tell how many bytes are needed to represent the types *bool* and *int*, respectively. The passing of the environment c to the function *size* is not necessary at this place, but it will be necessary from the next chapter on, when type definitions come in; then the size of a type can only be determined within the current environment containing previous type definitions.

In order to store values of different types they need to be mapped onto one common set *Byte* of "low level" values (see Section 3.1). The following set of functions is needed for constructing and representing values of type δ:

$constr_\delta : Byte^n \to V_\delta$,

$repr_\delta : V_\delta \to Byte^n$,

where n is the size of δ in the current context π.

Now we can extend the interface to the store by more convenient, namely type specific, access functions; let n denote $size(c, \delta)$:

$allocate_{c,\delta} = allocate_n$

$update_{c,\delta} : Location \times V_\delta \times Store \to Store$

$update_{c,\delta}(l, v, s) = update_n(l, repr_\delta(v), s)$

$lookup_{c,\delta} : Location \times Store \to V_\delta$

$lookup_{c,\delta} = lookup_n \circ constr_\delta$

3.3.3 Semantic Domains

Expression Values

Expressions are evaluated to objects of two different kinds: to values of any constructed type (type domain δ) or to locations (in case of an argument passed by reference). For the moment the type domain δ is equivalent to type domain γ, therefore the definition of *Value* is still simple:

$Value = V_{int} + V_{bool} + Location$, where

$V_{int} = \mathbb{Z}_{min,max} = \{min, \ldots, max\}$

$V_{bool} = \mathbb{B}$

Values of type δ can be stored and need therefore be converted to a representation in $Byte^n$ (see Subsection 3.3.2).

Identifier Values

Identifiers cover more kinds of values than expressions do. For the moment the extensions are function and procedure identifiers. The semantic domains of functions and procedures are the following ones:

$Function = Value^* \to RunInfo \to Value$,

$Procedure = Value^* \to RunInfo \to RunInfo$,

where the domain *RunInfo* is explained below. Both take a tuple of values (the arguments) and a run-time information to produce the domain specific result. The set of identifier values is defined as follows:

$IdValue = Value + Function + Procedure$

Streams

The I/O of our imperative language is modeled via streams. Streams are lists of values, i. e. represented as $Value^*$. They can be accessed via the following functions:

$read : Value^* \to Value^* \times Value$

$read((v_1, \ldots, v_n)) = ((v_2, \ldots, v_n), v_1), \quad n > 0, \quad$ else undefined

$write : Value^* \times Value \to Value^*$

$write((v_1, \ldots, v_n), v) = (v_1, \ldots, v_n, v), \quad n \geq 0$

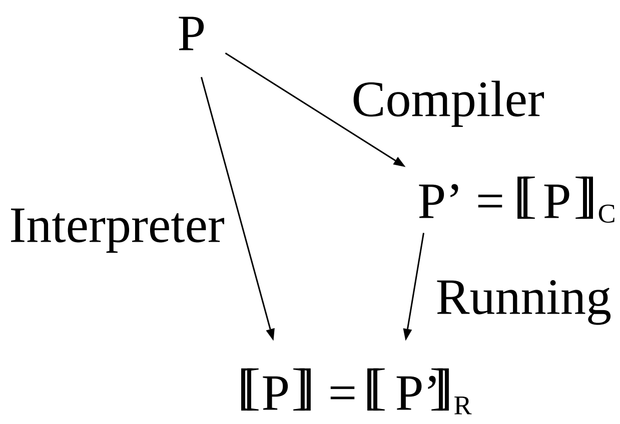

Figure 3.5: Interplay of Interpreter and Compiler

Compile-Time and Run-Time Information

As presented in (Schmidt, 1994) most meaning functions get two arguments: the environment and the store. We generalize this approach by extending the environment to the compile-time information and the store to the run-time information. See the following subsection on the deeper reasons for structuring the domains in this way.

For our imperative language the compile-time information is identical to the environment and does not contain any other components. It is, however, conceivable that other core languages might require additional compile-time information.

The run-time information is a true extension of the store; besides the store it contains streams for I/O, one for input and one for output, and a value for exception handling. The exception value can be on of:

$\quad Exception = \{ok, \diamond_1, \diamond_2, \ldots\}$.

If the exception value differs from ok the meaning function gives the exception value as the result. Here are now the definitions:

$\quad CompInfo = Env,$

$\quad\quad$ where for $e \in Env : e = \{\mathrm{I}_1 = v_1, \ldots, \mathrm{I}_n = v_n\}, n \geq 0, v_i \in IdValue$

$\quad RunInfo = Stream \times Stream \times Store^{\perp} \times Exception,$

$\quad\quad$ notation: $r = (r.i, r.o, r.s, r.e) \in RunInfo$

3.3.4 Interplay of Interpreter and Compiler

Structuring the arguments to the meaning functions into two domains, namely the compile-time and the run-time information, has a deeper reason. In (Stöcher, 1996) we experimented with generating translation rules for compilation based on the semantic rules. The situation can be depicted by Figure 3.5. The semantic rules are suitable for implementing an interpreter, where they can be used straight forwardly. But how to implement a compiler? We modeled the situation by defining a corresponding low-level language without identifiers and types. The main task of the compiler is then to replace identifiers by locations and types by sizes (in the figure denoted by the function $[\![\,]\!]_{\mathrm{C}}$). The low-level language is executed by the "running rules" (in the figure denoted by the function $[\![\,]\!]_{\mathrm{R}}$). It turns out that the compile-time information (the environment) is only needed for translating a program into the low-level language, and the run-time information only for executing the low-level language. See (Stöcher, 1996) for details.

3.3.5 How to Add a Predefined Type

The typing rules and semantic rules do not depend on the actual set of predefined types. Any predefined type γ can be added by defining the following:

- the set of values V_γ also denoting the literals,
- additional unary operators $*_{\gamma_1 \to \gamma_2} : V_{\gamma_1} \to V_{\gamma_2}$,
- additional binary operators $\circ_{\gamma_1 \times \gamma_2 \to \gamma_3} : V_{\gamma_1} \times V_{\gamma_2} \to V_{\gamma_3}$,
- the size $\mathit{size}(c, \gamma) = \mathit{size}_\gamma$
- and the conversion functions to and from bytes: constr_γ, repr_γ (see Subsection 3.3.2).

We regard the equality predicate as an ordinary binary operator with signature $\gamma \times \gamma \to \mathit{bool}$ that has to be defined. In computer science the equality predicate does not come by itself like in mathematics, where equality is assumed to be defined having given the definition of a domain. In programming languages like Lisp there are different notions of equality, in object oriented languages equality has to be defined for each new class. Testing for equality often requires a canonical representation. In mathematics this is usually given implicitly.

Additional Definitions

An additional function is needed to extract the type assignments of parameters, namely *type_attrs*:

$$\mathit{type\_attrs}(\mathrm{I}_1 : \tau_1, \ldots, \mathrm{I}_n : \tau_n) = \{\mathrm{I}_1 : \tau_1\} \uplus \cdots \uplus \{\mathrm{I}_n : \tau_n\}, \quad n \geq 0$$

3.4 Typing Rules

3.4.1 Identifiers

$$\text{(I1)}\ \ \pi \vdash \mathrm{I} : \iota\,\mathit{Idf}\,, \quad \text{if } (\mathrm{I} : \iota) \in \pi$$

An identifier I that has already been declared gets its typing from the set π of current valid type assignments. By the construction of π it is assured that there is always at most one such type assignment.

3.4.2 Declarations

$$\text{(D1)}\ \frac{\pi \vdash \mathrm{E} : \gamma Xpr}{\pi \vdash \textbf{const}\ \mathrm{I} = \mathrm{E} : \{\mathrm{I} : \gamma\} Dec} \qquad \text{(D2)}\ \pi \vdash \textbf{var}\ \mathrm{I} : \delta : \{\mathrm{I} : \delta loc\} Dec$$

A **const** declaration introduces a constant of a predefined type γ; A declaration **const** $a = b$, where b is a variable ($b : \gamma loc \in \pi$), binds the name a to the value of b (and not the location holding the value). In the next chapter this construct will be extended to also allow constants of constructed types. A **var** declaration introduces a variable.

$$\text{(D3)}\ \frac{\pi \vdash \mathrm{P} : \rho Par \qquad \pi \uplus \pi_1 \vdash \mathrm{E} : \delta Xpr}{\pi \vdash \textbf{fun}\ \mathrm{I}\ \mathrm{P} : \delta {=} \mathrm{E} : \{\mathrm{I} : \rho \to \delta\} Dec}, \quad \pi_1 = \{\mathrm{I} : \rho \to \delta\} \uplus type\_attrs(\mathrm{P})$$

Functions are expression abstractions. The return type δ of a function can be derived from the corresponding expression. It may not be a location and can therefore be uniquely determined.

reference to uniqueness

$$\text{(D4)}\ \frac{\pi \vdash \mathrm{P} : \rho Par \qquad \pi \uplus \pi_1 \vdash \mathrm{C} : Cmd}{\pi \vdash \textbf{proc}\ \mathrm{I}\ \mathrm{P} {=} \mathrm{C} : \{\mathrm{I} : \rho \to Cmd\} Dec},$$

$$\pi_1 = \{\mathrm{I} : \rho \to Cmd\} \uplus type\_attrs(\mathrm{P})$$

A **proc** declaration introduces a procedure identifier, which is a name for a parameterized command sequence. The typing of the parameters and the procedure identifier (collected in π_1) are needed for typing the command sequence C.

$$\text{(D5)}\ \frac{\pi \vdash \mathrm{D}_1 : \pi_1 Dec \qquad \pi \uplus \pi_1 \vdash \mathrm{D}_2 : \pi_2 Dec}{\pi \vdash \mathrm{D}_1 ; \mathrm{D}_2 : \pi_1 \uplus \pi_2 Dec}$$

In a sequence of declarations previous declarations are visible to following ones. For typing first of all the declaration sequence D_1 needs to be typed. The resulting type assignment π_1 is then combined with the current valid type assignment π, before the second declaration sequence D_2 can be typed.

3.4.3 Parameters

$$\text{(P1)}\ \pi \vdash (\mathrm{I}_1 : \tau_1, \ldots, \mathrm{I}_n : \tau_n) : (\tau_1, \ldots, \tau_n) Par$$

The typing $\rho = (\tau_1, \ldots, \tau_n)$ of a parameter list P is the list of types of the parameters, e. g. the parameter list in the procedure declaration

proc $s\ (a : \gamma_1, b : \gamma_2 loc) = \ldots$

is typed as

$(\gamma_1, \gamma_2 loc) Par.$

3.4.4 Arguments

$$\text{(A1)}\ \frac{\pi \vdash \mathrm{E}_1 : \tau_1 Xpr \quad \ldots \quad \pi \vdash \mathrm{E}_n : \tau_n Xpr}{\pi \vdash (\mathrm{E}_1, \ldots, \mathrm{E}_n) : (\tau_1, \ldots, \tau_n) Arg}, n \geq 0$$

Note: In an imperative implementation of a type-checker this rule must be applied bottom up, since the type list ρ is known from the context. Arguments only appear in a procedure call, where the type list ρ is inferred from a previous procedure declaration. For most expressions E the type is known by itself, hence the typing rule simply checks whether the type τ extracted from the type list of the argument list matches the type of the corresponding expression E.

Only for identifiers declared as variables the typing is not uniquely determined when appearing in an expression. Since we want to distinguish call by reference and call by value only at the definition of a procedure and not with each procedure call again, this information has to be put into the typing of arguments. Now in the case of a variable as argument the above rule forwards the typing extracted from a previous procedure declaration to the typing of the corresponding argument and thus resolves the ambiguity. Consider the type list

$\rho = (\gamma_1, \gamma_2 loc)$

and the procedure call

$s\, d, e.$

Then d can either be a constant $(d : \gamma_1 \in \pi)$ or a variable $(d : \gamma_1 loc \in \pi)$ that is called by value; e must be a variable $(e : \gamma_2 loc \in \pi)$ that is called by reference.

3.4.5 Expressions

$$\text{(E1)}\ \frac{\pi \vdash \mathrm{I} : \tau Idf}{\pi \vdash \mathrm{I} : \tau Xpr} \qquad \text{(E2)}\ \frac{\pi \vdash \mathrm{I} : \gamma locIdf}{\pi \vdash \mathrm{I} : \gamma Xpr}$$

These two rules introduce some ambiguity in typing: an identifier I declared as a variable can either denote the value assigned to it or the store location holding the value. This ambiguity corresponds to the two meanings a variable can have: either the location it refers to or the value this location holds. Consider the type assignment

$\pi = \{a : \gamma loc\}.$

Then from the typing rule for identifiers we know that

$\pi \vdash a : \gamma locIdf.$

Now according to the above typing rules a appearing in an expression could be typed as

$\pi \vdash a : \gamma locXpr$

or

$\pi \vdash a : \gamma Xpr.$

The ambiguity is resolved by the immediate context. There is only one place where a can denote both, the value and its store location, namely as an argument in a procedure call (see the above typing of arguments for how this ambiguity is resolved). In all other places expressions must be of type γ, forcing a variable to be typed as γXpr (see the definition of γ).

$$\text{(E3)}\ \pi \vdash v : \gamma Xpr\ , \quad v \in V_\gamma$$

The type of a literal v is inferred from the set V_γ it belongs to. In practice this is resolved by defining disjunctive representations for different types.

$$\text{(E4)}\quad \pi \vdash \mathbf{eof} : \mathit{boolXpr}$$

The typing process of a function call is analogously to the procedure call, except that that the phrase gets its type from the return type of the function. See the comments to rule (C3) below.

$$\text{(E5)}\quad \frac{\pi \vdash \mathrm{I} : \rho \to \delta\,\mathit{Idf} \qquad \pi \vdash \mathrm{A} : \rho\mathit{Arg}}{\pi \vdash \mathrm{I}\;\mathrm{A} : \delta\mathit{Xpr}}$$

$$\text{(E6)}\quad \frac{\pi \vdash \mathrm{E} : \gamma_1\mathit{Xpr}}{\pi \vdash *_\sigma\,\mathrm{E} : \gamma\mathit{Xpr}}, \quad \sigma = \gamma_1 \to \gamma$$

$$\text{(E7)}\quad \frac{\pi \vdash \mathrm{E}_1 : \gamma_1\mathit{Xpr} \qquad \pi \vdash \mathrm{E}_2 : \gamma_2\mathit{Xpr}}{\pi \vdash \mathrm{E}_1 \circ_\sigma \mathrm{E}_2 : \gamma\mathit{Xpr}}, \quad \sigma = \gamma_1 \times \gamma_2 \to \gamma$$

3.4.6 Commands

$$\text{(C1)}\quad \pi \vdash \mathbf{skip} : \mathit{Cmd} \qquad\qquad \text{(C2)}\quad \frac{\pi \vdash \mathrm{I} : \delta\,\mathit{locIdf} \qquad \pi \vdash \mathrm{E} : \delta\,\mathit{Xpr}}{\pi \vdash \mathrm{I} \mathbf{:=} \mathrm{E} : \mathit{Cmd}}$$

$$\text{(C3)}\quad \frac{\pi \vdash \mathrm{I} : (\rho \to \mathit{Cmd})\mathit{Idf} \qquad \pi \vdash \mathrm{A} : \rho\mathit{Arg}}{\pi \vdash \mathrm{I}\;\mathrm{A} : \mathit{Cmd}}$$

From the current type assignment π the type list ρ is inferred, which comes from a previous procedure declaration, and is passed to the typing of the argument list A.

$$\text{(C4)}\quad \frac{\pi \vdash \mathrm{E} : \gamma\mathit{Xpr}}{\pi \vdash \mathbf{write}\;\mathrm{E} : \mathit{Cmd}} \qquad\qquad \text{(C5)}\quad \frac{\pi \vdash \mathrm{I} : \gamma\mathit{locIdf}}{\pi \vdash \mathbf{read}\;\mathrm{I} : \mathit{Cmd}}$$

$$\text{(C6)}\quad \frac{\pi \vdash \mathrm{E} : \mathit{boolXpr} \qquad \pi \vdash \mathrm{C}_1 : \mathit{Cmd} \qquad \pi \vdash \mathrm{C}_2 : \mathit{Cmd}}{\pi \vdash \mathbf{if}\;\mathrm{E}\;\mathbf{then}\;\mathrm{C}_1\;\mathbf{else}\;\mathrm{C}_2\;\mathbf{fi} : \mathit{Cmd}}$$

$$\text{(C7)}\quad \frac{\pi \vdash \mathrm{E} : \mathit{boolXpr} \qquad \pi \vdash \mathrm{C} : \mathit{Cmd}}{\pi \vdash \mathbf{while}\;\mathrm{E}\;\mathbf{do}\;\mathrm{C}\;\mathbf{od} : \mathit{Cmd}}$$

$$\text{(C8)}\quad \frac{\pi \vdash \mathrm{D} : \pi_1\mathit{Dec} \qquad \pi \uplus \pi_1 \vdash \mathrm{C} : \mathit{Cmd}}{\pi \vdash \mathbf{with}\;\mathrm{D}\;\mathbf{do}\;\mathrm{C}\;\mathbf{end} : \mathit{Cmd}}$$

For typing the **with** construct first of all the declaration sequence D needs to be typed, since it contains definitions to which the following commands can refer. The resulting type assignment π_1 is then combined with the current valid type assignment π, before the command sequence C can be typed.

Sequencing commands is trivial:

$$\text{(C9)}\quad \frac{\pi \vdash \mathrm{C}_1 : \mathit{Cmd} \qquad \pi \vdash \mathrm{C}_2 : \mathit{Cmd}}{\pi \vdash \mathrm{C}_1 ; \mathrm{C}_2 : \mathit{Cmd}}$$

3.4.7 Programs

$$(\text{Prg1})\ \frac{\{\} \vdash \mathrm{C} : \mathit{Cmd}}{\mathrm{C} : \mathit{Prg}}$$

3.5 Unicity of Typing

Since the type of a phrase also influences its meaning (see the example for the typing of arguments) we want to make sure that the typing of a program is unique, that is:

Theorem 3.5.1 (Unicity of Typing)
For every phrase U in a well-typed program, if $U : \theta$ holds, then θ is unique.

There is one pair of typing rules corresponding to the same syntax construct and hence resulting in different typing under the same preconditions: namely the two rules (E1) and (E2) for embedding the identifiers into expressions. The deeper reason is that on the syntax level we do not distinguish between locations and the value contained, but somehow let the typing rules automatically choose the right option. Therefore the above theorem would definitely not be true if we considered the typing of expressions outside the context of a program. In the context of a program it is always clear whether a variable denotes its location or value.

Proof (Sketch): We will prove Theorem 3.5.1 in two steps. In the first step we introduce an additional syntax construct @I that is to be used whenever a variable shall denote its value. For this extended syntax we show that unicity of typing holds. In the second step we show that the places where the @I construct is to be inserted are well defined.

1. Let us first view the given syntax as a kind of abbreviation of the following full syntax
   E ::= I | @I | ... where the construct @I for a variable denoting its value is (re-)introduced. Now the typing rule (E2) turns out to be actually the corresponding rule for this "hidden" construct:

   $$(\text{E2})\ \frac{\pi \vdash \mathrm{I} : \delta\mathit{locIdf}}{\pi \vdash @\mathrm{I} : \delta\mathit{Xpr}}$$

   For this extended syntax and appropriate set of typing rules Theorem 3.5.1 becomes trivially true. The proof works by structural induction over all syntax constructs and goes through without any further considerations since each typing rule corresponds to exactly one syntax construct and vice versa.

2. It remains to show that for each appearance of a variable in any expression of a program it can be decided, whether the variable denotes its location or its value. In the latter case we can replace the corresponding expression

node in the abstract syntax tree of type I by the corresponding instance of the @I construct. After all such replacements we can apply Theorem 3.5.1.

Let us first locate the difficulty. A variable I, i. e. an identifier of type $\delta_1 loc$, appearing in an expression can either be typed as $\pi \vdash \mathrm{I} : \delta_1 locXpr$ (via rule (E1), denoting its location) or as $\pi \vdash \mathrm{I} : \delta_1 Xpr$ (via rule (E2), denoting its value). Now when looking at all syntax constructs containing expressions and their corresponding typing rules, it turns out that in almost all these typing rules the premise demands the expression to be explicitly typed as δ[2] (rules (D1), (D3), (E6), (E7), (C2), (C4), (C6), (C7)) or as δloc (rules (E2), (C5)). There is only one typing rule allowing both typings in its preconditions, namely the typing rule for arguments (A1). An expression E_i in an argument list is typed as τXpr. Note that the type domain τ is defined as $\delta \mid \delta loc$. Summarizing we can conclude that ambiguity in a well typed program can only arise for a variable being passed as an argument.

Now let us have a closer look at the typing rule (A1). The type of an expression E_i must exactly match the corresponding type τ_i. The argument type $\rho = (\tau_1, \ldots, \tau_n)$ itself is inferred via a call command I_1 A (rule (C3)), more precisely from the binding of the function or procedure identifier I_1. By the definition of$\uplus$ two such bindings must be distinguishable by just looking at the constructed types δ_i of the parameters, where $\tau_i = \delta_i \langle loc \rangle$. Hence, also in this case it is always well defined, whether a variable denotes its location (call by reference) or its value (call by value). This resolves the mentioned ambiguity and completes the proof. □

3.6 Semantic Rules

The meaning functions use the following typed variables as parameters:

- c, the compile time information defining the identifier bindings,
- r, the record $(r.i, r.o, r.s, r.e)$ of the runtime information, where the components denote the input and output streams, the store and the error flag, respectively,
- a, the evaluated arguments of a function or procedure call, i. e. a tuple of values $(v_0, \ldots, v_n) \in Value^*$.

3.6.1 Identifiers

$\mathrm{I} \quad ::= \quad i$
$[\![\pi \vdash . : \tau Idf]\!] : CompInfo \rightarrow IdValue$

The denotation of an identifier is simply the value of its binding in the environment.
$[\![\pi \vdash \mathrm{I} : \iota Idf]\!] c = v$, if $(\mathrm{I} = v) \in c.e$

[2]Since γ is a sub-domain of δ, it is also included here.

By the construction of the environment it is assured that there is always at most one such binding. By the correspondence between environment and type assignment there is one such correctly typed binding (see the following subsection 3.7 on the soundness of typing).

3.6.2 Declarations

D ::= **const** I = E | **var** I:δ | **fun** I P:δ=E | **proc** I P=C | $\mathrm{D}_1;\mathrm{D}_2$

$[\![\pi \vdash . : \pi_1 Dec]\!] : \mathit{CompInfo} \rightarrow \mathit{RunInfo} \rightarrow (\mathit{CompInfo} \times \mathit{RunInfo})$

$[\![\pi \vdash \textbf{const}\ \mathrm{I} = \mathrm{E} : \{\mathrm{I}:\gamma\} Dec]\!] c\ r = (\{\mathrm{I} = [\![\pi \vdash \mathrm{E}:\gamma Xpr]\!] c\ r\}, r)$

The **const** construct generates a binding of the new identifier with the denotation of the expression.

$[\![\pi \vdash \textbf{var}\ \mathrm{I}:\delta : \{\mathrm{I}:\delta loc\} Dec]\!] c\ r = (\{\mathrm{I} = l\}, r|_s[s_1])$,
where $(l, s_1) = \mathit{allocate}_{c,\delta}(r.s)$

The **new** construct allocates memory for the corresponding type and generates a binding of the new identifier with the new location.

$[\![\pi \vdash \textbf{fun}\ \mathrm{I}\ \mathrm{P}:\delta{=}\mathrm{E} : \{\mathrm{I}:\rho \rightarrow \delta\} Dec]\!] c\ r = (\{\mathrm{I} = f\}, r)$,
where $f\ a' = [\![\pi \uplus \pi_1 \vdash \mathrm{E}:\delta Xpr]\!](c \uplus \{\mathrm{I} = f\} \uplus ([\![\pi \vdash \mathrm{P}:\rho Par]\!] a'))$

$[\![\pi \vdash \textbf{proc}\ \mathrm{I}\ \mathrm{P}{=}\mathrm{C} : \{\mathrm{I}:\rho \rightarrow \mathit{Cmd}\} Dec]\!] c\ r = (\{\mathrm{I} = f\}, r)$,
where $f\ a' = [\![\pi \uplus \pi_1 \vdash \mathrm{C} : \mathit{Cmd}]\!](c \uplus \{\mathrm{I} = f\} \uplus ([\![\pi \vdash \mathrm{P}:\rho Par]\!] a'))$

The expression and command abstractions generate a binding of the new identifier with the denotation function f parameterized by the argument values. The denotation function is evaluated upon a function or procedure call. The abstractions are *statically scoped*, i. e. they use the compile-time information c at definition time and do not care about the compile-time information at the time when the function or procedure is invoked.

$[\![\pi \vdash \mathrm{D}_1;\mathrm{D}_2 : \pi_1 \uplus \pi_2 Dec]\!] c\ r = (c_1 \uplus c_2, r_2)$,
where $(c_1, r_1) = [\![\pi \vdash \mathrm{D}_1 : \pi_1 Dec]\!] c\ r$
and $(c_2, r_2) = [\![\pi \uplus \pi_1 \vdash \mathrm{D}_2 : \pi_2 Dec]\!] c \uplus c_1\ r_1$

The bindings of a sequence of declarations are combined such that previous declarations can be referred to and a new binding of an identifier overwrites a previous one.

3.6.3 Parameters

P ::= $(\mathrm{I}_1:\tau_1, \ldots, \mathrm{I}_n:\tau_n), \quad n \geq 0$

$[\![\pi \vdash . : \rho Par]\!] : \mathit{Value}^* \rightarrow \mathit{CompInfo}$

The denotation of parameters is a set of bindings. The list of evaluated arguments is bound to the corresponding identifiers.

$[\![\pi \vdash (\mathrm{I}_1:\tau_1, \ldots, \mathrm{I}_n:\tau_n) : (\tau_1, \ldots, \tau_n) Par]\!] a = \{\mathrm{I}_1 = v_1\} \uplus \cdots \uplus \{\mathrm{I}_n = v_n\}$,
where $a = (v_1, \ldots, v_n), \quad n \geq 0$

3.6.4 Arguments

A ::= $(\mathrm{E}_1, \ldots, \mathrm{E}_n), \quad n \geq 0$

$[\![\pi \vdash . : \rho Arg]\!] : \mathit{CompInfo} \rightarrow \mathit{RunInfo} \rightarrow \mathit{Value}^*$

$[\![\pi \vdash (\mathrm{E}_1, \ldots, \mathrm{E}_n) : (\tau_1, \ldots, \tau_n) Arg]\!] c\ r =$
$\quad ([\![\pi \vdash \mathrm{E}_1 : \tau_1 Xpr]\!] c\ r, \ldots, [\![\pi \vdash \mathrm{E}_n : \tau_n Xpr]\!] c\ r)$
The tuple of expressions is mapped to the tuple of corresponding denotations (meanings) of the expressions.

3.6.5 Expressions

$\mathrm{E} \ ::= \ \mathrm{I} \mid v \mid \mathbf{eof} \mid \mathrm{I\,A} \mid \ast\ \mathrm{E} \mid \mathrm{E}_1 \circ \mathrm{E}_2$
$[\![\pi \vdash .: \tau Xpr]\!] : \mathit{CompInfo} \to \mathit{RunInfo} \to \mathit{Value}$

For identifiers appearing in expressions there are two possible denotation functions. The ambiguity is resolved by the annotated types.
$[\![\pi \vdash \mathrm{I} : \tau Xpr]\!] c\ r = [\![\pi \vdash \mathrm{I} : \tau Idf]\!] c$
If the type of the expression and the identifier are the same, the denotation of the expression is simply the denotation of the identifier.
$[\![\pi \vdash \mathrm{I} : \delta Xpr]\!] c\ r = \mathit{lookup}([\![\pi \vdash \mathrm{I} : \delta locIdf]\!] c, s)$
If the type of the expression is a user accessible one and the type of the identifier is the one of its corresponding location, then the denotation of the identifier must be a location, which is looked up on the store (right hand value of an identifier).

$[\![\pi \vdash v : \gamma Xpr]\!] c\ r = v,$
$\quad$ if $v \in V_\gamma$
$[\![\pi \vdash \mathbf{eof} : boolXpr]\!] c\ r =$
- true, if $r.i = [\,]$,
- false, otherwise

$[\![\pi \vdash \mathrm{I\,A} : \delta Xpr]\!] c\ r = f\ a\ r,$
$\quad$ where $f = [\![\pi \vdash \mathrm{I} : \rho \to \delta Idf]\!] c$
$\quad$ and $a = [\![\pi \vdash \mathrm{A} : \rho Arg]\!] c\ r$
$[\![\pi \vdash \ast_\sigma \mathrm{E} : \gamma Xpr]\!] c\ r = \ast_\sigma([\![\pi \vdash \mathrm{E} : \gamma_1 Xpr]\!] c\ r),$
$\quad$ if $\sigma = \gamma_1 \to \gamma$
$[\![\pi \vdash \mathrm{E}_1 \circ_\sigma \mathrm{E}_2 : \gamma Xpr]\!] c\ r = ([\![\pi \vdash \mathrm{E}_1 : \gamma_1 Xpr]\!] c\ r) \circ_\sigma ([\![\pi \vdash \mathrm{E}_2 : \gamma_2 Xpr]\!] c\ r),$
$\quad$ if $\sigma = \gamma_1 \times \gamma_2 \to \gamma$

3.6.6 Commands

C ::= **skip** | I**:=**E | I A | **write** E | **read** I |
if E **then** C_1 **else** C_2 **fi** | **while** E **do** C **od** |
with D **do** C **end** | $\mathrm{C}_1; \mathrm{C}_2$

$[\![\pi \vdash .: \mathit{Cmd}]\!] : \mathit{CompInfo} \to \mathit{RunInfo} \to \mathit{RunInfo}$
$[\![\pi \vdash .: \mathit{Cmd}]\!] c\ r = r$, if $r.s = \bot$ or $r.e \neq \mathit{ok}$

$[\![\pi \vdash \mathbf{skip} : \mathit{Cmd}]\!] c\ r = r$
$[\![\pi \vdash \mathrm{I} \mathbf{:=} \mathrm{E} : \mathit{Cmd}]\!] c\ r = r|_s[\mathit{update}_{c,\delta}([\![\pi \vdash \mathrm{I} : \delta locIdf]\!] c, [\![\pi \vdash \mathrm{E} : \delta Xpr]\!] c\ r, r.s)]$
In order to choose the appropriate update function $\mathit{update}_{c,\delta}$ the type δ is extracted from the typing of the expression E.

$[\![\pi \vdash \mathrm{I\,A} : \mathit{Cmd}]\!] c\ r = f\ a\ r,$
$\quad$ where $f = [\![\pi \vdash \mathrm{I} : (\rho \to \mathit{Cmd}) Idf]\!] c$
$\quad$ and $a = [\![\pi \vdash \mathrm{A} : \rho Arg]\!] c\ r$

In a procedure call first the arguments are evaluated into a list, which is passed to the denotation function P of the parameters. The resulting bindings c_1 are combined with the environment c_p at the time of the procedure definition, which is passed to the denotation function C of the command sequence. As a procedure may call itself, termination is not assured here.

$[\![\pi \vdash \textbf{write}\ \mathrm{E} : \mathit{Cmd}]\!] c\ r = r|_o[\mathit{write}(r.o, [\![\pi \vdash \mathrm{E} : \gamma \mathit{Xpr}]\!] c\ r)]$
$[\![\pi \vdash \textbf{read}\ \mathrm{I} : \mathit{Cmd}]\!] c\ r =$

- $r|_e[\diamond_1]$,
  if $r.i = ()$
- $(r|_i[i_1])|_s[\mathit{update}_{c,\gamma}(l, v_1, r.s)]$,
  if $(i_1, v) = \mathit{read}(r.i)$ and $v = \mathrm{in}\, V_\gamma(v_1)$,
- $r|_e[\diamond_2]$, otherwise,
  where $l = [\![\pi \vdash \mathrm{I} : \gamma \mathit{locIdf}]\!] c$

A **read** command can cause a (run-time) exception that stops evaluation of the program. An exception $\diamond_1$ means reading at the end of the input stream, an exception $\diamond_2$ means a type error in reading: the type of the read token does not match the expected type of the variable.

$[\![\pi \vdash \textbf{if}\ \mathrm{E}\ \textbf{then}\ \mathrm{C}_1\ \textbf{else}\ \mathrm{C}_2\ \textbf{fi} : \mathit{Cmd}]\!] c\ r =$
$\quad \mathit{if}([\![\pi \vdash \mathrm{E} : \mathit{boolXpr}]\!] c\ r, [\![\pi \vdash \mathrm{C}_1 : \mathit{Cmd}]\!] c\ r, [\![\pi \vdash \mathrm{C}_2 : \mathit{Cmd}]\!] c\ r)$
$[\![\pi \vdash \textbf{while}\ \mathrm{E}\ \textbf{do}\ \mathrm{C}\ \textbf{od} : \mathit{Cmd}]\!] c\ r = w\ r$
$\quad$ where $w : \mathit{RunInfo} \to \mathit{RunInfo}$
$\qquad w\ r' = \mathit{if}([\![\pi \vdash \mathrm{E} : \mathit{boolXpr}]\!] c\ r', w([\![\pi \vdash \mathrm{C} : \mathit{Cmd}]\!] c\ r'), r')$

This the second place after the procedure call where non-termination may come in.

$[\![\pi \vdash \textbf{with}\ \mathrm{D}\ \textbf{do}\ \mathrm{C}\ \textbf{end} : \mathit{Cmd}]\!] c\ r = [\![\pi \uplus \pi_1 \vdash \mathrm{C} : \mathit{Cmd}]\!](c \uplus c_1)\ r_1$,
$\quad$ where $(c_1, r_1) = [\![\pi \vdash \mathrm{D} : \pi_1 \mathit{Dec}]\!] c\ r$

The **with** construct introduces new bindings and possibly more locations in the store for the corresponding command sequence C.

$[\![\pi \vdash \mathrm{C}_1 ; \mathrm{C}_2 : \mathit{Cmd}]\!] c\ r = [\![\pi \vdash \mathrm{C}_2 : \mathit{Cmd}]\!] c\ r_1$,
$\quad$ where $r_1 = [\![\pi \vdash \mathrm{C}_1 : \mathit{Cmd}]\!] c\ r$

3.6.7 Programs

$\mathrm{Prg} \ ::= \ \mathrm{C}$
$[\![\pi \vdash . : \mathit{Prg}]\!] : \mathit{Value}^* \to (\mathit{Value}^* \times \mathit{Exception})^\perp$

$[\![\mathrm{C} : \mathit{Prg}]\!] i =$

- $\perp$, if $r.s = \perp$,
- $(r.o, r.e)$, otherwise,

where $r = [\![\{\} \vdash \mathrm{C} : \mathit{Cmd}]\!] \{\}\ (i, [\,], \langle\,\rangle, \mathit{ok})$

The meaning of a program is basically the meaning of its command sequence C with the output stream, the store and the error flag initially set to the empty stream, the empty store and *ok*, respectively. The meaning of a program, however, only contains the output stream and the exception flag. If the program does not terminate, the store is undefined ($\perp$) and so is the meaning of the program.

3.7 Soundness of Typing Rules

The semantic rules only apply to well-typed syntax trees. Now we want to make sure that the semantic rules apply to all well-typed syntax trees, and that all well-formed programs have a meaning.

The meaning functions for the different syntax domains have certain signatures, as already given with the semantic rules. We give a definition $[\![\ ]\!]$ that maps each type attribute to its intended set of meanings:

$$\begin{array}{lcl}
[\![\tau \mathit{Idf}]\!] & = & \mathit{CompInfo} \rightarrow \mathit{Value} \\
[\![\rho \mathit{Par}]\!] & = & \mathit{Value}^* \rightarrow \mathit{CompInfo} \\
[\![\rho \mathit{Arg}]\!] & = & \mathit{CompInfo} \rightarrow \mathit{RunInfo} \rightarrow \mathit{Value}^* \\
[\![\pi \mathit{Dec}]\!] & = & \mathit{CompInfo} \rightarrow \mathit{RunInfo} \rightarrow (\mathit{CompInfo} \times \mathit{RunInfo}) \\
[\![\tau \mathit{Xpr}]\!] & = & \mathit{CompInfo} \rightarrow \mathit{RunInfo} \rightarrow \mathit{Value} \\
[\![\mathit{Cmd}]\!] & = & \mathit{CompInfo} \rightarrow \mathit{RunInfo} \rightarrow \mathit{RunInfo} \\
[\![\mathit{Prg}]\!] & = & \mathit{Value}^* \rightarrow (\mathit{Value}^* \times \mathit{Exception})^{\perp}
\end{array}$$

Our goal is that the typing rules are sound, that is:

Theorem 3.7.1 (Soundness of Typing)
For all well typed phrases $U\!:\!\theta$: $[\![U\!:\!\theta]\!] \in [\![\theta]\!]$.

By structural induction one can check for each semantic rule that it corresponds to the specified signature. For the semantic rule for identifiers we need a further property, namely the correspondence between type assignment and environment.

Lemma 3.7.1 (Correspondence of Context and Environment)
In each call of a semantic rule, where an environment is passed, the current type assignment and the passed environment hold bindings for the same identifiers and the types in the type assignment correspond to the values in the environment.

This again can be checked by structural induction when going through all calls of semantic rules and comparing the changes of the environment with the changes of the context in the corresponding typing rule.

Chapter 4

Extending the Type System

We describe here the type system for the imperative core language by introducing

- type expressions,
- type definitions (including recursion),
- type constructors and destructors and
- the possibility of hiding the representation of a type.

The type system we are about to describe contains most of the features of common imperative programming languages like C or PASCAL and goes in some aspects far beyond. Some features will show their full power when the module system comes in. Here a brief description of the features:

Type Expressions: The type expressions replace and extend the previously embedded type domains in the abstract syntax. This affects quite a few typing rules, but not all of them essentially; we rewrite here only those rules with major changes. The full set of consistent typing rules can be found in the appendix. The expression domain has to be extended accordingly in order to allow handling of expressions of all kinds of types.

Type Definitions: The declarations are extended by constructs for type definitions, allowing also forward and recursive definitions.

Type Constructors and Destructors: For ease of programming and for handling abstract data types, later in the module language, type constructors are introduced. Type constructors are more or less procedures that are bound to a type expression instead of a new identifier.

Representation Hiding: New types can be declared such that they do not match their representation. This will show useful when introducing modules, where the representation of certain types should not be visible outside the defining module.

Here is a brief overview over the following sections:

1. **Syntax Rules:** the syntax rules together with detailed explanations of the constructs and an example program,
2. **Preparations:** preparations for the typing and semantic rules including necessary extensions of type domains and introducing relations on them,
3. **Typing Rules:** the typing rules and comments,
4. **Unicity of Typing:** updating the considerations about uniqueness in the typing process,
5. **Semantic Rules:** the semantic rules and comments.

4.1 Syntax Rules

First we give the definition of the abstract syntax, before explaining the intentions of the constructs in detail.

T	::=	**int** | **bool** | I | **Array**[T] | **Pointer**[T] | **Record**[CL]
CL	::=	$(\mathrm{I}_1 : \mathrm{T}_1, \ldots, \mathrm{I}_n : \mathrm{T}_n)$, $n \geq 1$
D	::=	**const** I:T = IE | **var** I:T | **fun** I P:T=E | ... |
	::=	**type** I = T | **newtype** I | **newtype** I = T |
		construct I(P)= C | **destruct** I = C | **var** $\mathrm{I}_1 : \mathrm{I}_2$(A)
IE	::=	E | **Array**[A] | **Record**[A]
P	::=	(⟨**ref** ⟩$\mathrm{I}_1 : \mathrm{T}_1, \ldots,$ ⟨**ref** ⟩$\mathrm{I}_n : \mathrm{T}_n$), $n \geq 0$
E	::=	... | T′**nil** | **!**E | $\mathrm{E}_1[\mathrm{E}_2]$ | E.I
C	::=	... | E_1**:=**E_2 | ... | **read** E |
		... | **new** E | **new** $\mathrm{E}_1[\mathrm{E}_2]$

4.1.1 Type Expressions: T, CL

The type expressions contain the following constructs:

int, bool: There are only two predefined types, namely integers and booleans.

I: Previously defined type names can be referenced. Also within a type definition the type to be defined can be referenced recursively.

Array[T], **Pointer**[T], **Record**[CL]**:** These are the constructors for introducing new types. For records the syntax domain CL of component lists was introduced.

4.1.2 Declarations and Initializing Expressions: D, IE

The declarations are extended by the following constructs:

const I:T = IE**:** In order to enable constants of array and record type the expression domain has to be extended by corresponding constructs. This is done by the syntax domain IE, the initializing expressions. Since both, arrays and records, can be initialized by simple lists of expressions, the type has to be explicitly specified by the type expression T. This syntax construct replaces the corresponding syntax construct given in the previous chapter.

var I:T, **fun** I P:T=E**:** These constructs are only affected by a minor change: the type of the variable or the return value, respectively, are now given by a type expression instead of the corresponding δ type. The changed rules for typing and semantics can be found in the complete listing in the appendix.

type I = T**:** This construct allows aliasing of type expression, in the sense of abbreviation; it does not introduce a new type! Type identifiers introduced this way are considered equal to all structural equivalent type expressions. After such a type definition the type expressions I, T or any other equivalent type expression can be used interchangeably. (See subsection 4.2.2 on the definition of type equivalence.)

newtype I,**newtype** I = T**:** These are the constructs for defining new types; also recursive definitions are allowed. Type names may be declared without a type definition; this is referred to as a *forward definition*. Type identifiers declared in a forward definition **newtype** I may be used in other type definitions and can be bound to a type expression later on. A type I defined via **newtype** I = T is not equivalent to its representation, i. e. it makes a difference in the following code whether you use I or T as a type of a parameter or argument. Within the expression domain, however, an expression of type I is implicitly cast to its representation whenever needed, i. e. for dereferencing, indexing or qualification. With the module system, more specifically with signatures introduced in Chapter 6, this implicit cast will be restricted to the defining module.

construct I(P)= C**:** A type constructor is actually a procedure bound to a type identifier; within the body of this procedure an additional identifier *this* is visible, referring to the location of the constructed object. Argument handling is analogous to that of procedures. Type constructors are invoked by the new form of variable declaration below. There can be several type constructors for one type identifier.

destruct I = C**:** A type destructor is a parameterless command sequence bound to a type identifier. Whenever a variable of an equivalent type, that was declared after the destructor, falls out of the scope, the type destructor is invoked.

var I_1:I_2(A)**:** With this variable declaration the new object is also initialized by the corresponding type constructor. The type constructor is picked according to the type I_2 and the types of the arguments. This syntax construct does not replace the corresponding construct given in the previous chapter; it is an additional way of declaring a variable where the value can be initialized at the same time.

The constructs for the syntax domain IE, the initializing expressions, can only be used for constant definitions.

4.1.3 Parameters: P

The types of the parameters can now be specified by type expressions T_i instead of directly stating the type domain τ_i. However, the distinction between "call by value" and "call by reference" can not be done by type expressions themselves. By default we assume "call by value". The optional keyword **ref** changes this to "call by reference". (Before a call by reference could be caused by choosing directly δloc as the type of a parameter.)

4.1.4 Expressions: E

The expression domain has to be extended by the following constructs in order to handle the newly introduced type constructors:

T'**nil:** This denotes an empty pointer of type T. For strong type checking the type expression T has to be explicitly stated.

$!\mathrm{E}, \mathrm{E}_1[\mathrm{E}_2], \mathrm{E.I}$**:** These are the access functions for the newly introduced types: dereferencing a pointer, indexing an array and qualifying a record, respectively.

4.1.5 Commands: C

For initializing pointer and array structures usually usually a **new** construct is added to the expression domain. Since this expression alters the store, but here in our semantics expressions are supposed not to change it, we put the construct containing also the assignment into the command domain.

new E, **new** $\mathrm{E}_1[\mathrm{E}_2]$**:** Both commands are to be viewed as one construct initializing a variable of type pointer and array, respectively.

4.1.6 Example Program

Again for simplicity the examplary code, actually just a list of declarations, is given (see Figure 4.1) in text (ASCII) notation.

The declarations first define a recursive type *List*. Since it is a pointer type a constant *empty* for the corresponding nil pointer can be declared. A constructor for the type *List* with no arguments simply initializes a variable to the empty list. The following procedure and function declarations define the basic operations on lists.

```
...
newtype List = Pointer[ Record[ elem:int,
                                next:List ] ];
const empty:List = List'nil;
constructor List() =
  this = empty;
var list = List();
proc addHead( h:int, l:List, ref r:List ) =
  new r;
  r.elem = h; r.next = l;
fun head( l:List ):int  = !l.elem;
fun rest( l:List ):List = !l.next;
...
```

Figure 4.1: Example Program for the Type System

4.2 Preparations

4.2.1 Type Domains

Types can now have the following structure:

γ	$::=$	$bool \mid int$	predefined
δ	$::=$	$\gamma \mid \mathrm{I} \mid \delta ptr \mid \delta arr \mid \lambda rec$	constructed
λ	$::=$	$(\mathrm{I}_1 : \delta_1, \ldots, \mathrm{I}_n : \delta_n), \quad n \geq 1$	component list
τ	$::=$	$\delta \mid \delta loc$	expression
ρ	$::=$	τ^*	list
ι	$::=$	$\tau \mid \rho \rightarrow \delta \mid \rho \rightarrow Cmd \mid (\delta, \phi) type \mid cons \mid dst$	identifier
ϕ	$::=$	$alias \mid forward \mid new$	type flag

The type domain δ for constructed types is now extended by alternatives corresponding to the type constructors. The type domain ι is extended by alternatives for type definitions, type constructors and type destructors. The flag ϕ reflects the different roles type identifiers can play: they can be

- just an *alias* for a type expression,
- an intermediate *forward* declaration that must be overwritten by an explicit type definition before it can be used
- or the name of a new type that is regarded different from its representing type expression.

The types attached to syntax domains remain unchanged:

$\mathrm{T} : \delta\, Type$
$\mathrm{CL} : \lambda\, Comp$
$\mathrm{D} : \pi\, Dec$
$\mathrm{IE} : \delta\, Init$
$\mathrm{E} : \tau\, Xpr$
$\mathrm{C} : Cmd$

4.2.2 Type Equivalence

For matching arguments against the types of parameters we need to define when to regard two types as equal. Therefore we introduce the following equivalence relation.

$$\text{(TE1)}\ \pi \vdash \delta \sim \delta \qquad \text{(TE2)}\ \frac{\pi \vdash \delta_1 \sim \delta_2}{\pi \vdash \delta_2 \sim \delta_1} \qquad \text{(TE3)}\ \frac{\pi \vdash \delta_1 \sim \delta_2 \quad \pi \vdash \delta_2 \sim \delta_3}{\pi \vdash \delta_1 \sim \delta_3}$$

$$\text{(TE4)}\ \frac{\mathrm{I}\!:\!(\delta, \mathit{alias})\mathit{type} \in \pi}{\pi \vdash \mathrm{I} \sim \delta}$$

$$\text{(TE5)}\ \frac{\pi \vdash \delta_1 \sim \delta_2}{\pi \vdash \delta_1 \mathit{ptr} \sim \delta_2 \mathit{ptr}} \qquad \text{(TE6)}\ \frac{\pi \vdash \delta_1 \sim \delta_2}{\pi \vdash \delta_1 \mathit{arr} \sim \delta_2 \mathit{arr}}$$

$$\text{(TE7)}\ \frac{\pi \vdash \delta_1 \sim \delta_1' \quad \ldots \quad \pi \vdash \delta_n \sim \delta_n'}{\pi \vdash (\mathrm{I}_1\!:\!\delta_1 \ldots \mathrm{I}_n\!:\!\delta_n)\mathit{rec} \sim (\mathrm{I}_1\!:\!\delta_1' \ldots \mathrm{I}_n\!:\!\delta_n')\mathit{rec}}$$

The first three rules make the relation an equivalence relation. The other four rules correspond to the four alternatives for constructing types.

Note that the above rules require that recursive types are not equivalent to their representation. Otherwise they would not define all pairs of types equivalent that one would intuitively maybe view as the same types. Suppose that the **type** construct for type aliasing would also allow recursive definitions. Then two analogously defined recursive types ("isomorphic (type-)trees") are not equivalent according to the above defined relation. The following two type definitions
type *list* = **Pointer[Record[***elem* : *int*, *next* : *list***]]**
type *list2* = **Pointer[Record[***elem* : *int*, *next* : *list2***]]**
would introduce two different types. Detecting these two types as isomorphic requires a more powerful type handling system. Therefore we decided to allow recursive type definitions only in the context of the **newtype** construct.

4.2.3 Type Structure

For some typing rules we need to know the top-level structure of a type. The just defined type equivalence will not suffice, since types defined as a **newtype** are not equivalent to their representing type. Therefore we introduce another predicate $\pi \vdash \delta_1 \succeq \delta_2$ with the following rules:

$$\text{(TS1)}\ \pi \vdash \delta \succeq \delta$$

$$\text{(TS2)}\ \frac{(\mathrm{I}\!:\!(\delta, \mathit{alias})\mathit{type}) \in \pi}{\pi \vdash \mathrm{I} \succeq \delta} \qquad \text{(TS3)}\ (\mathrm{I}\!:\!(\delta, \mathit{new})\mathit{type}) \in \pi \ \ \pi \vdash \mathrm{I} \succeq \delta$$

Literally $\pi \vdash \delta_1 \succeq \delta_2$ means that type δ_1 has structure δ_2.

4.2.4 Size of Constructed Types

The size function *size* giving the size of a constructed type δ needs to be extended. The size of pointer and array types is constant, regardless of the type structure below, since they are represented by references to store locations. The size of a record type is only known, when the size of all of its components is known. In this case the total size is the sum of the sizes of the components. Since now also forward declarations are allowed, type identifiers can be unbound (represented by actually binding them to themselves) and their size is therefore not known. A size of 0 shall mean that the size is unknown. If the size of one record component is unknown (0) the total size of the record is unknown (0); this is modeled by the special operator $+_0$:

$$+_0 : V_{int} \to V_{int}, s +_0 t = \begin{cases} s+t, & s \neq 0, t \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

Now we can extend the function *size*:

$$size(c, \mathrm{I}) = \begin{cases} 0, & \delta = \mathrm{I} \\ size(c, \delta), & \text{otherwise} \end{cases}$$

where $(\mathrm{I} = \delta\, Type \in c$

$size(c, \delta\, ptr) = size(c, \delta\, arr) = size_{loc}$

$size(c, (\mathrm{I}_1 : \delta_1, \ldots, \mathrm{I}_n : \delta_n) rec) = size(c, \delta_1) +_0 \cdots +_0 size(c, \delta_n)$

By allowing forward definitions types need not be fully specified anymore and their representation size may therefore be unknown. This was not an issue up to now, since the types in domain δ were just the predefined ones and their size is trivially known. For the typing rules we do not need the actual size, but just whether the size is known. Therefore we introduce the predicate $\pi \vdash is\_sized(\delta)$ defined by the following rules:

$$\text{(TS1)}\ \pi \vdash is\_sized(bool) \qquad \text{(TS2)}\ \pi \vdash is\_sized(int)$$

$$\text{(TS3)}\ \frac{(\mathrm{I} : (\delta, \phi) type) \in \pi \qquad \pi \vdash is\_sized(\delta)}{\pi \vdash is\_sized(\mathrm{I})}$$

$$\text{(TS4)}\ \pi \vdash is\_sized(\delta\, arr) \qquad \text{(TS5)}\ \pi \vdash is\_sized(\delta\, ptr)$$

$$\text{(TS6)}\ \frac{\pi \vdash is\_sized(\delta_1) \qquad \ldots \qquad \pi \vdash is\_sized(\delta_n)}{\pi \vdash is\_sized((\mathrm{I}_1 : \delta_1, \ldots, \mathrm{I}_n : \delta_n) rec)}$$

$\pi \vdash is\_sized(\delta)$ in a typing rule ensures that in the corresponding semantic rule $size(c, \delta) > 0$ holds (c is the environment corresponding to context π).

4.2.5 Semantic Domains

Expression Values

Since the domain of constructed types δ has been extended, we have to define the corresponding value sets:

$V_{\mathrm{I}} = Identifier$

$V_{\delta ptr} = Location_0$

$V_{\delta arr} = Location$

$V_{\lambda rec} = V_{\delta_1} \times \cdots \times V_{\delta_n} \subset Value^*$,

where $\lambda = (\mathrm{I}_1 : \delta_1, \ldots, \mathrm{I}_n : \delta_n)$

For representing the nil-pointer we need the set $Location_0 = \{loc_0\} \cup Location$. Note that the set *Location* starts with loc_1.

The set *Value* of expression values now has the following *recursive structure*:

$Value = V_{int} + V_{bool} + Location_0 + Value^*$.

See Subsection 2.2.2 on the understanding of recursive definitions.

For storing constructed types we need to extend the functions *constr* and *repr* (see Subsection 3.3.2). Since we represent pointers and arrays in a C-like way, namely as references to the actual value(s), the representation does not depend on the underlying type δ:

$constr_{\delta ptr} : Location_0 \to Byte^n$
$repr_{\delta ptr} : Byte^n \to Location_0$
$constr_{\delta arr} : Location \to Byte^n$
$repr_{\delta arr} : Byte^n \to Location$,

where $n = size(c, \delta ptr) = size(c, \delta arr)$. The corresponding functions for records can be reduced to simpler domains:

$constr_{\lambda rec} : Value^* \to Byte^n$
$constr_{\lambda rec}((v_1, ldots, v_n)) = (constr_{\delta_1}(v_1), \ldots, constr_{\delta_n}(v_n))$,

where $n = size(c, \delta rec)$ and $\lambda = (\mathrm{I}_1 : \delta_1, \ldots, \mathrm{I}_n : \delta_n)$.

4.3 Typing Rules

Here we give the typing rules for the different constructs and detailed explanation of the intuition behind them.

4.3.1 Type Constructors

$$\text{(T1)}\ \frac{\pi \vdash \mathrm{I} : (\delta, \phi)\, typeIdf}{\pi \vdash \mathrm{I} : \delta\, Type}$$

$$\text{(T2)}\ \frac{\pi \vdash \mathrm{T} : \delta\, Type}{\pi \vdash \textbf{Array}[\mathrm{T}] : \delta\, arr\, Type} \qquad \text{(T3)}\ \frac{\pi \vdash \mathrm{T} : \delta\, Type}{\pi \vdash \textbf{Pointer}[\mathrm{T}] : \delta\, ptr\, Type}$$

$$\text{(T4)}\ \frac{\pi \vdash \mathrm{CL} : \lambda\, Comp}{\pi \vdash \textbf{Record}[\mathrm{CL}] : \lambda\, rec\, Type}$$

4.3.2 Component Lists

$$\text{(CL1)}\ \frac{\pi \vdash \mathrm{T}_1 : \delta_1\, Type \quad \ldots \quad \pi \vdash \mathrm{T}_n : \delta_n\, Type}{\pi \vdash (\mathrm{I}_1 : \mathrm{T}_1, \ldots, \mathrm{I}_n : \mathrm{T}_n) : (\delta_1, \ldots, \delta_n)\, Comp}, \quad n \geq 1$$

4.3.3 Declarations

The new rule for a constant definition is more general and replaces the rule given in the previous chapter.

$$\text{(D1)}\ \frac{\pi \vdash \mathrm{IE} : \delta\, Init \quad \pi \vdash \mathrm{T} : \delta_1\, Type \quad \pi \vdash \delta \sim \delta_1}{\pi \vdash \textbf{const}\ \mathrm{I} : \mathrm{T} = \mathrm{IE} : \{\mathrm{I} : \delta\}\, Dec}$$

The following rule allows type aliasing: a type expression may be abbreviated by a name.

$$\text{(D6)}\ \frac{\pi \vdash \mathrm{T} : \delta\, \mathit{Type}}{\pi \vdash \textbf{type}\ \mathrm{I} = \mathrm{T} : \{\mathrm{I} : (\delta, \mathit{alias})\mathit{type}\}\mathit{Dec}}\ ,\quad (\mathrm{I} : (\delta', \phi)\mathit{type}) \notin \pi$$

In general type identifiers may not be overwritten, since types interact on a global level. Consider, for instance, the following case: type *int* gets an alias

type *alias* = *int*,

that is used in a function definition:

fun *succ* (*i*:*alias*) : *alias*=*i*+1.

When allowing the overwriting of the alias

type *alias* = *bool*,

the type system is not consistent any more, since function applications will be typed wrongly:

succ (*true*)

will be admitted by the type checker, whereas an intended application of the function will be refused:

succ (3).

Whereas for type aliases this problem could also be solved by internally always replacing aliases with the actual type expression, this approach does not work for recursive type definitions.

$$\text{(D7)}\ \pi \vdash \textbf{newtype}\ \mathrm{I} : \{\mathrm{I} : (\mathrm{I}, \mathit{forward})\mathit{type}\}\mathit{Dec}\ ,\quad (\mathrm{I} : (\delta', \phi)\mathit{type}) \notin \pi$$

The following rule allows recursive type definitions. A type assignment for I is added to π for typing type expression T.

$$\text{(D8)}\ \frac{\pi \uplus \{\mathrm{I} : (\mathrm{I}, \mathit{forward})\mathit{type}\} \vdash \mathrm{T} : \delta\, \mathit{Type}}{\pi \vdash \textbf{newtype}\ \mathrm{I} = \mathrm{T} : \{\mathrm{I} : (\delta, \mathit{new})\mathit{type}\}\mathit{Dec}}\ ,$$

$$\text{if } (\mathrm{I} : (\delta, \phi)\mathit{type}) \in \pi, \text{ then } \phi = \mathit{forward}$$

A definition of a new type may only overwrite a previous forward declaration.

The following two rules introduce type constructors and destructors. Both type constructors and destructors are command sequences bound to type identifiers, where a special variable *this* points to the corresponding object. A type constructor is additionally parameterized analogously to a procedure and is therefore invoked with arguments. A type destructor is invoked implicitly whenever a corresponding variable falls out of scope.

$$\text{(D9)}\ \frac{\pi \vdash \mathrm{I} : (\delta, \phi)\mathit{typeIdf} \qquad \pi \vdash \mathrm{P} : \rho\mathit{Par} \qquad \pi \uplus \mathit{type\_attrs}(\rho) \uplus \{\mathit{this} : \delta\mathit{loc}\} \vdash \mathrm{C} : \mathit{Cmd}}{\pi \vdash \textbf{construct}\ \mathrm{I}(\mathrm{P}) = \mathrm{C} : \{\mathrm{I} : (\rho \to \mathit{Cmd})\mathit{cons}\}\mathit{Dec}}\ ,$$

$$\pi \vdash \mathit{is\_sized}(\delta) \text{ and } \phi = \begin{cases} \mathit{alias} \\ \mathit{new} \end{cases}$$

$$\text{(D10)}\ \frac{\pi \vdash \mathrm{I} : (\delta, \phi)\mathit{typeIdf} \qquad \pi \uplus \{\mathit{this} : \delta\mathit{loc}\} \vdash \mathrm{C} : \mathit{Cmd}}{\pi \vdash \textbf{destruct}\ \mathrm{I} = \mathrm{C} : \{\mathrm{I} : \mathit{dst}\}\mathit{Dec}}\ ,\quad \phi = \begin{cases} \mathit{alias} \\ \mathit{new} \end{cases}$$

Now we can introduce an additional rule for a variable declaration, where the value is at the same time initialized by a corresponding type constructor.

$$(\mathrm{D11})\ \frac{\pi \vdash \mathrm{I}_2 : (\rho \to \mathit{Cmd})\mathit{consIdf} \qquad \pi \vdash \mathrm{A} : \rho \mathit{Arg}}{\pi \vdash \mathbf{var}\ \mathrm{I}_1 : \mathrm{I}_2(\mathrm{A}) : \{\mathrm{I} : \delta\, \mathit{loc}\}\mathit{Dec}}$$

4.3.4 Initializing Expressions

$$(\mathrm{IE1})\ \frac{\pi \vdash \mathrm{E} : \delta\, \mathit{Xpr}}{\pi \vdash \mathrm{E} : \delta\, \mathit{Init}}$$

$$(\mathrm{IE2})\ \frac{\pi \vdash \mathrm{A} : \rho \mathit{Arg} \qquad \pi \vdash \rho \sim (\delta, \ldots, \delta)}{\pi \vdash \mathbf{Array}[\mathrm{A}] : \delta\, \mathit{arrInit}}$$

$$(\mathrm{IE3})\ \frac{\pi \vdash \mathrm{A} : \rho \mathit{Arg} \qquad \pi \vdash \rho \sim (\delta_1, \ldots, \delta_n)}{\pi \vdash \mathbf{Record}[\mathrm{A}] : (\mathrm{I}_1 : \delta_1, \ldots, \mathrm{I}_n : \delta_n)\mathit{recInit}}$$

4.3.5 Parameters

$$(\mathrm{P1})\ \frac{\pi \vdash \mathrm{T}_1 : \delta_1\, \mathit{Type} \qquad \ldots \qquad \pi \vdash \mathrm{T}_n : \delta_n\, \mathit{Type}}{\pi \vdash (\langle \mathbf{ref} \rangle \mathrm{I}_1 : \mathrm{T}_1, \ldots, \langle\langle \mathbf{ref} \rangle\rangle \mathrm{I}_n : \mathrm{T}_n) : (\delta_1 \langle \mathit{loc} \rangle, \ldots, \delta_n \langle\langle \mathit{loc} \rangle\rangle)\mathit{Par}}, \quad n \geq 0$$

The optional keyword **ref** triggers the flag *loc* in the corresponding type, which in turn triggers "call by reference". Note that the semantics do not change!

4.3.6 Expressions

Before it was only identifiers that could denote both, a location or its value; now this ambiguity also holds for certain expressions, like **!**E or $\mathrm{E}_1[\mathrm{E}_2]$. This ambiguity can only be resolved by the context. Therefore we will type such expressions as locations, whenever this is allowed, and introduce the following *automatic dereferencing rule*:

$$(\mathrm{E2})\ \frac{\pi \vdash \mathrm{E} : \delta\, \mathit{locXpr}}{\pi \vdash \mathrm{E} : \delta\, \mathit{Xpr}}$$

This rule replaces the corresponding rule of the previous chapter, that allowed only identifiers in the premise. Rule (E2) will be applied whenever the context needs it. In Subsection 4.4 we will show that this is deterministic.

$$(\mathrm{E8})\ \frac{\pi \vdash \mathrm{T} : \delta_1\, \mathit{Type} \qquad \pi \vdash \delta_1 \succeq \_\mathit{ptr}}{\pi \vdash \mathrm{T}'\mathbf{nil} : \delta_1 \mathit{Xpr}}$$

$$(\mathrm{E9})\ \frac{\pi \vdash \mathrm{E} : \delta_1 \mathit{locXpr} \qquad \pi \vdash \delta_1 \succeq \delta_2 \mathit{ptr}}{\pi \vdash \mathbf{!}\mathrm{E} : \delta_2 \mathit{locXpr}}$$

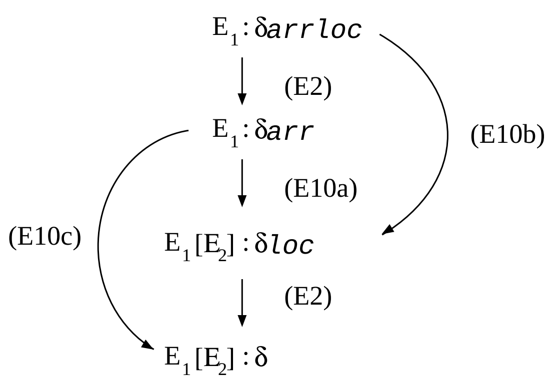

Figure 4.2: Typing Rules for Array Indexing

The typing rule for indexing arrays actually is the following:

$$\text{(E10a)}\ \frac{\pi \vdash \mathrm{E}_1 : \delta_1 Xpr \qquad \pi \vdash \mathrm{E}_2 : int Xpr \qquad \pi \vdash \delta_1 \succeq \delta_2 arr}{\pi \vdash \mathrm{E}_1[\mathrm{E}_2] : \delta_2 loc Xpr}$$

This rule works also fine together with the semantics, however it allows assignment to the cells of constant arrays. In order to disable updating (the cells of) constant arrays we demand the following: the above rule for indexing arrays may only be applied together with the typing rule (E2), the "automatic dereferencing rule" for implicitly looking up locations. In case of E_1 actually meaning a location this rule must be applied anyway before the index rule can be applied. In the case of E_1 meaning a constant array the rule (E2) must be applied afterwards. Figure 4.2 depicts the situation. These additional restriction can also be expressed by the following two typing rules instead of the above rule for indexing:

$$\text{(E10b)}\ \frac{\pi \vdash \mathrm{E}_1 : \delta_1 loc Xpr \qquad \pi \vdash \mathrm{E}_2 : int Xpr \qquad \pi \vdash \delta_1 \succeq \delta_2 arr}{\pi \vdash \mathrm{E}_1[\mathrm{E}_2] : \delta_2 loc Xpr}$$

$$\text{(E10c)}\ \frac{\pi \vdash \mathrm{E}_1 : \delta_1 Xpr \qquad \pi \vdash \mathrm{E}_2 : int Xpr \qquad \pi \vdash \delta_1 \succeq \delta_2 arr}{\pi \vdash \mathrm{E}_1[\mathrm{E}_2] : \delta_2 Xpr}$$

These two rules can be written in one by making the type flag *loc* optional. It either has to appear both in the premise and the conclusion, or not at all.

$$\text{(E10)}\ \frac{\pi \vdash \mathrm{E}_1 : \delta_1 \langle loc \rangle Xpr \qquad \pi \vdash \mathrm{E}_2 : int Xpr \qquad \pi \vdash \delta_1 \succeq \delta_2 arr}{\pi \vdash \mathrm{E}_1[\mathrm{E}_2] : \delta_2 \langle loc \rangle Xpr}$$

The typing rule below is to be understood analogously as actually combining two separate typing rules. However, there is no "core" typing rule like (E10a) (see the comments along with the corresponding semantic rules for an explanation). The interplay with rule (E2) is shown in Figure 4.3.

$$\text{(E11)}\ \frac{\pi \vdash \mathrm{E} : \delta_1 \langle loc \rangle Xpr \qquad \pi \vdash \delta_1 \succeq \lambda rec \qquad \lambda = (\ldots, \mathrm{I} : \delta_2, \ldots)}{\pi \vdash \mathrm{E.I} : \delta_2 \langle loc \rangle Xpr}$$

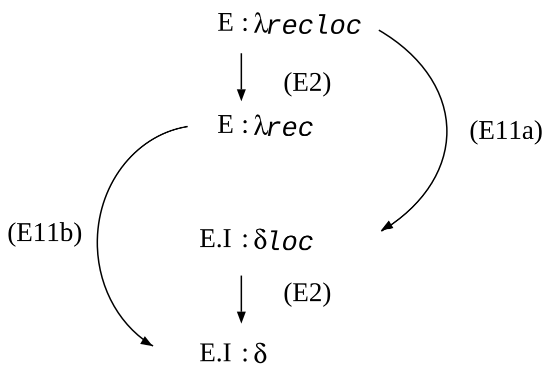

Figure 4.3: Typing Rules for Record Qualification

4.3.7 Commands

The syntax constructs for assignment and reading now allow an arbitrary expression (as long as it denotes a location on the store) instead of an identifier. The corresponding typing rules change in an obvious way. We give here only the typing rule for the assignment; the typing rule for the reading construct can be found in the complete listing of rules in the appendix.

$$\text{(C2)}\ \frac{\pi \vdash \mathrm{E}_1 : \delta\, locXpr \qquad \pi \vdash \mathrm{E}_2 : \delta\, Xpr}{\pi \vdash \mathrm{E}_1 \mathbf{:=} \mathrm{E}_2 : Cmd}$$

The **new** constructs allocate memory for pointer and array expressions, which must denote a location on the store. In the first case we have to make sure that the type δ_1 has the structure of a pointer type and that the underlying type has a known size, since that gives the number of bytes that are allocated. The second construct is handled in an analogous way; here additionally the second expression has to be of type *int*.

$$\text{(C10)}\ \frac{\pi \vdash \mathrm{E} : \delta_1 locXpr \qquad \pi \vdash \delta_1 \succeq \delta_2 ptr}{\pi \vdash \mathbf{new}\ \mathrm{E} : Cmd}\ , \quad \pi \vdash is\_sized(\delta_2)$$

$$\text{(C11)}\ \frac{\pi \vdash \mathrm{E}_1 : \delta_1 locXpr \qquad \pi \vdash \delta_1 \succeq \delta_2 arr \qquad \pi \vdash \mathrm{E}_2 : intXpr}{\pi \vdash \mathbf{new}\ \mathrm{E}_1[\mathrm{E}_2] : Cmd}\ ,$$
$$\pi \vdash is\_sized(\delta_2)$$

4.4 Unicity of Typing

The claim remains the same (see Section 3.5), the proof, however, becomes a little more complicated. Before only identifiers could denote both, a location or the contained value, now this is true for (almost) arbitrary expressions.

We again divide the proof into two steps. First we introduce a "full" syntax by adding the @ construct for referencing a location:

E ::= I | @E | ...

Again rule (E2) turns out to be actually the corresponding typing rule:

$$(\mathrm{E2})\ \frac{\pi \vdash \mathrm{E}:\delta\, locXpr}{\pi \vdash @\mathrm{E}:\delta\, Xpr}$$

Rule (E10) is actually composed out of rule (E2) and rule (E10a) (see the comments, when rule (E10) was introduced). For the following proof we will split this rule again and take (E10a) instead of (E10). Rule (E11) has to be handled in an analogous way; a rule (E11a) was not explicitly stated, but it is totally analogous to rule (E10a):

$$(\mathrm{E11a})\ \frac{\pi \vdash \mathrm{E}:\delta_1 locXpr \qquad \pi \vdash \delta_1 \succeq \lambda rec \qquad \lambda = (\ldots, \mathrm{I}:\delta_2, \ldots)}{\pi \vdash \mathrm{E.I}:\delta_2 Xpr}$$

Now the proof for the unicity of typing for the full syntax is straight forward. Each phrase can be typed uniquely by itself, without needing to know where this phrase appears in a program. The proof is simply by structural induction on the (full) syntax constructs, which are in one-one correspondence to the typing rules.

4.5 Semantic Rules

4.5.1 Type Constructors

The meaning of a type is actually its (representation) size, so one could define it the following way:
$[\![\pi \vdash \mathrm{T}:\delta\, Type]\!] c\ r = size(c, \delta)$
But we will always use the *size* function directly and never refer to the meaning function for types.

4.5.2 Declarations

D ::= **const** I : T = IE | ... | **type** I = T | **newtype** I | **newtype** I = T |
construct I(P)= C | **destruct** I = C | **var** $\mathrm{I}_1 : \mathrm{I}_2$(A)

$[\![\pi \vdash . : \pi_1 Dec]\!] : CompInfo \rightarrow RunInfo \rightarrow$
$(CompInfo \times (RunInfo \rightarrow RunInfo) \times RunInfo)$

The signature of the meaning function for declarations changes, since the meaning must now also contain a destructor. Most rules remain unaffected since their destructor is the identity.

$[\![\pi \vdash \mathbf{const}\ \mathrm{I}:\mathrm{T} = \mathrm{IE}:\{\mathrm{I}:\delta\} Dec]\!] c\ r = (\{\mathrm{I} = v\}, id, r_1)$,
where $(v, r_1) = [\![\pi \vdash \mathrm{IE}:\delta\, Init]\!] c\ r$

For computing the size of types we need to know the bindings of type identifiers. Thus we simply take over the representation of a type identifier from the context. Note that a forward definition does not contain any information about the type

structure and therefore it is ignored in the semantics. $[\![\pi \vdash \textbf{newtype}\ \mathrm{I} : \{\mathrm{I} : (\mathrm{I}, \mathit{forward})\mathit{type}\}\mathit{Dec}]\!] c\ r = (\{\}, \mathit{id}, r)$
$[\![\pi \vdash \textbf{type}\ \mathrm{I} = \mathrm{T} : \{\mathrm{I} : (\delta, \mathit{alias})\mathit{type}\}\mathit{Dec}]\!] c\ r = (\{\mathrm{I} = \delta\}, \mathit{id}, r)$
$[\![\pi \vdash \textbf{newtype}\ \mathrm{I} = \mathrm{T} : \{\mathrm{I} : (\delta, \mathit{new})\mathit{type}\}\mathit{Dec}]\!] c\ r = (\{\mathrm{I} = \delta\}, \mathit{id}, r)$

For the following two rules we need to introduce separate binding domains, since a type identifier may get several constructors and maybe also a destructor assigned. These separate domains are marked by *cons* and *dst* constructs, respectively.
$[\![\pi \vdash \textbf{construct}\ \mathrm{I}(\mathrm{P}) = \mathrm{C} : \{\mathrm{I} : (\rho \to \mathit{Cmd})\mathit{cons}\}\mathit{Dec}]\!] c\ r =$
$(\{\mathrm{I} = f\mathit{cons}\}, \mathit{id}, r)$,
where $f(a', t', r') = [\![\pi \uplus \mathit{type\_attrs}(\rho) \uplus \{\mathit{this} : \delta \mathit{loc}\} \vdash \mathrm{C} : \mathit{Cmd}]\!] g(a', t')\ r'$,
where $g(a', t') = c \uplus ([\![\pi \vdash \mathrm{P} : \rho \mathit{Par}]\!] a') \uplus \{\mathit{this} = t'\}$
$[\![\pi \vdash \textbf{destruct}\ \mathrm{I} = \mathrm{C} : \{\mathrm{I} : \mathit{dst}\}\mathit{Dec}]\!] c\ r = (\{\mathrm{I} = f\mathit{dst}\}, \mathit{id}, r)$,
where $f\ t\ r' = [\![\pi \uplus \mathit{type\_attrs}(\rho) \uplus \{\mathit{this} : \delta \mathit{loc}\} \vdash \mathrm{C} : \mathit{Cmd}]\!](c \uplus \{\mathit{this} = t\})r'$
$[\![\pi \vdash \textbf{var}\ \mathrm{I}_1 : \mathrm{I}_2(\mathrm{A}) : \{\mathrm{I} : \delta \mathit{loc}\}\mathit{Dec}]\!] c\ r = (\{\mathrm{I} = l\}, (f_2\ l) \circ d, r_2)$,
where $(l, r_1) = \mathit{allocate}_{c,\delta}(r.s)$
and $r_2 = f_1([\![\pi \vdash \mathrm{A} : \rho \mathit{Arg}]\!] c\ r, l, r_1)$,
where $f_1 = [\![\pi \vdash \mathrm{I}_2 : (\rho \to \mathit{Cmd})\mathit{cons}\,\mathit{Idf}]\!]$
and $(\mathrm{I} = f_2 \mathit{dst}) \in c$ or $f_2 = \mathit{id}$
and $d\ r' = r'|_s[\mathit{free}_\delta(l, r.s)]$

For two constructs the semantics change: in a simple variable declaration the destructor d like the one just above has to be created (see the Appendix A.2 for the complete rule set), and the semantics for concatenating declarations must now also concatenate the destructors:
$[\![\pi \vdash \mathrm{D}_1; \mathrm{D}_2 : \pi_1 \uplus \pi_2 \mathit{Dec}]\!] c\ r = (c_1 \uplus c_2, d_2 \circ d_1, r_2)$,
where $(c_1, d_1, r_1) = [\![\pi \vdash \mathrm{D}_1 : \pi_1 \mathit{Dec}]\!] c\ r$
and $(c_2, d_2, r_2) = [\![\pi \uplus \pi_1 \vdash \mathrm{D}_2 : \pi_2 \mathit{Dec}]\!] c \uplus c_1\ r_1$

4.5.3 Initializing Expressions

IE ::= E | **Array**[A] | **Record**[A] $[\![\pi \vdash . : \delta \mathit{Init}]\!] : \mathit{CompInfo} \to \mathit{RunInfo} \to (\mathit{Value} \times \mathit{RunInfo})$

The meaning of an initializing expression is a pair (v, r), where v denotes the initializing value and r the possibly altered runtime information.
$[\![\pi \vdash \mathrm{E} : \delta \mathit{Init}]\!] c\ r = ([\![\pi \vdash \mathrm{E} : \delta \mathit{Xpr}]\!] c\ r, r)$

$[\![\pi \vdash \textbf{Array}[\mathrm{A}] : \delta\, \mathit{arrInit}]\!] c\ r = (\mathit{loc}_i, r|_s[s_n])$,
where $(a_1, \ldots, a_k) = [\![\pi \vdash \mathrm{A} : \rho \mathit{Arg}]\!]$
and $n = \mathit{size}(c, \delta)$
and $(\mathit{loc}_i, s_0) = \mathit{allocate}(n * k, r.s)$
and $s_{j+1} = \mathit{update}_\delta(\mathit{loc}_{i+j*d}, a_{j+1}, s_j), 0 \leq j < k$

$[\![\pi \vdash \textbf{Record}[\mathrm{A}] : (\mathrm{I}_1 : \delta_1, \ldots, \mathrm{I}_n : \delta_n)\mathit{recInit}]\!] c\ r = (a, r)$,
where $a = [\![\pi \vdash \mathrm{A} : \rho \mathit{Arg}]\!]$

4.5.4 Expressions

E ::= ... | T'**nil** | **!**E | $\mathrm{E}_1[\mathrm{E}_2]$ | E.I

$[\![\pi \vdash \mathrm{E}:\delta Xpr]\!] c\ r = lookup_{c,\delta}([\![\pi \vdash \mathrm{E}:\delta loc Xpr]\!] c\ r, r)$ $[\![\pi \vdash \mathrm{T}'\mathbf{nil}:\delta_1 Xpr]\!] c\ r = loc_0$
$[\![\pi \vdash !\mathrm{E}:\delta_2 loc Xpr]\!] c\ r = lookup_{c,\delta_1}([\![\pi \vdash \mathrm{E}:\delta_1 loc Xpr]\!] c\ r, r)$
Dereferencing is essentially the same as looking up a variable's value (see the first rule for expressions above). However, here an additional problem comes in: dereferencing the nil pointer (i. e. loc_0) is not defined by the function *lookup*. One way to resolve this would be to extend the meaning of the function *lookup* to return a suitable value. What we would prefer, ofcourse, is raising an exception. In our set of rules we only handle exceptions raised in commands (by the **read** construct); other syntax domains are not touched by this. Raising an exception here, within the rules for expressions, would impose exception handling on most other syntax domains. We will not do this here, but just mention that the extension of a domain by exception values can be thought of as a domain construction by itself (like lifting, see Subsection 2.2.1). The affected functions must then be extended to just pass through an exception value; this can be defined analogously to strict extensions of functions.

The semantic rule for arrays below corresponds to the typing rule (E10a). Recall that rule (E10) is actually an abbreviation writing rules (E10b) and (E10c) in one, which in turn are combinations of rules (E2) and (E10a) in different order. The semantic rule corresponding to typing rule (E2) was already given above.
$[\![\pi \vdash \mathrm{E}_1[\mathrm{E}_2]:\delta_2 loc Xpr]\!] c\ r = loc_{i+os}$,
  where $loc_i = [\![\pi \vdash \mathrm{E}_1:\delta_1 Xpr]\!] c\ r$
  and $o = [\![\pi \vdash \mathrm{E}_2: int Xpr]\!] c\ r$
  and $s = size(c, \delta_2)$

The typing rule (E11) for record qualification is also an abbreviation writing actually two rules in one. However, these rules cannot be further decomposed and there is no analogon to rule (E10a). The reason is that a record E : λrec denotes the tuple of component values, from which the location of a component E.I : $\delta_i loc$ on the store cannot be determined anymore (in contrast to arrays denoting a store reference to the contained values, where the locations of the components can be computed as shown in the above semantic rule). For computing the location of a record component we need the location of the record - this is done in the first rule below. The second rule we need for qualifying constant records.
$[\![\pi \vdash \mathrm{E.I}:\delta_2 Xpr]\!] c\ r = p_i$,
  where $(p_1, \ldots, p_n) = [\![\pi \vdash \mathrm{E}:\delta_1 Xpr]\!] c\ r$
  and $i = index(\mathrm{I}, \delta_1)$
$[\![\pi \vdash \mathrm{E.I}:\delta_2 loc Xpr]\!] c\ r = loc_{i+o}$,
  where $loc_i = [\![\pi \vdash \mathrm{E}:\delta_1 loc Xpr]\!] c\ r$
  and $o = rec\_off(c, \mathrm{I}, \delta_1)$
Here we made us of two auxiliary functions for record types and their components. The function *index* determines the index of a component I in a record type $\delta = \lambda rec$, i.e. it gives 1 if I is the first component, and so on. The function *rec_off* computes the offset of a component in a record by adding up the (representation) sizes of the previous components, i.e. *rec_off* is 0 for the first component, it gives the size of the first component for the second component, and so on. This can be formalized in the following way:
  $index(\mathrm{I}_i, (\mathrm{I}_1:\delta_1, \ldots, \mathrm{I}_i:\delta_i, \ldots, \mathrm{I}_n:\delta_n) rec) = i$
  $rec\_off(c, \mathrm{I}_i, (\mathrm{I}_1:\delta_1, \ldots, \mathrm{I}_i:\delta_i, \ldots, \mathrm{I}_n:\delta_n) rec) = size(c, \delta_1) + \cdots + size(c, \delta_{i-1})$

4.5.5 Commands

$$\mathrm{C} \quad ::= \quad \ldots \mid \mathrm{E}_1\mathbf{:=}\mathrm{E}_2 \mid \ldots \mid \ \mathbf{new}\ \mathrm{E} \mid \mathbf{new}\ \mathrm{E}_1[\mathrm{E}_2]$$

$[\![\pi \vdash \mathrm{E}_1\mathbf{:=}\mathrm{E}_2 : Cmd]\!] c\ r = r|_s[update_\delta([\![\pi \vdash \mathrm{E}_1 : \delta locXpr]\!] c\ r, [\![\pi \vdash \mathrm{E}_2 : \delta Xpr]\!] c\ r, r.s)]$
The left-hand side of the assignment is now an expression E_1, from which we get the location to store the meaning of E_2; the rest remains unchanged. Again, in order to choose the appropriate update function $update_\delta$ the type δ is extracted from the typing of the expression E_2.

$[\![\pi \vdash \mathbf{new}\ \mathrm{E} : Cmd]\!] c\ r = r|_s[update(l, v, s)]$,
  where $l = [\![\pi \vdash \mathrm{E} : \delta_1 locXpr]\!] c\ r$
  and $(v, r_1) = allocate_{c,\delta_1}(r.s)$

$[\![\pi \vdash \mathbf{new}\ \mathrm{E}_1[\mathrm{E}_2] : Cmd]\!] c\ r = r|_s[update(l, v, s)]$,
  where $l = [\![\pi \vdash \mathrm{E}_1 : \delta_1 locXpr]\!] c\ r$
  and $(v, s) = allocate(size(c, \delta_1) * ([\![\pi \vdash \mathrm{E}_2 : intXpr]\!] c\ r), r.s)$

The semantics for the **with** construct must now also include the destructor calls after the command sequence has been evaluated:

$[\![\pi \vdash \mathbf{with}\ \mathrm{D}\ \mathbf{do}\ \mathrm{C}\ \mathbf{end} : Cmd]\!] c\ r = r_3$,
  where $(c_1, d, r_1) = [\![\pi \vdash \mathrm{D} : \pi_1 Dec]\!] c\ r$
  and $r_2 = [\![\pi \uplus \pi_1 \vdash \mathrm{C} : Cmd]\!] (c \uplus c_1)\ r_1$
  and $r_3 = d\ r_2$

Chapter 5

Introducing Modules and Functors

Now, after having introduced the core language and its type system, we are ready to develop the module system. The module system is a stand-alone unit, essentially independent of the core language. The typing and semantic rules interfere with the given imperative core language, but can be easily adopted to any other core language. The design principles are the already known ones: abstraction and parameterization, this time applied to the domain of abstractions and parameterizations itself, namely the declarations. The basic ideas are taken from SML; we go, however, further by attaching a module with an initializer and terminator and preparing the rules for later on introducing *axioms* (see Section 6.4 for an outlook on this extension).

The concepts and syntax constructs for allowing abstract types have already been introduced in the previous chapter (except for the **sharing** construct introduced in the next chapter). Here we only deal with the abstraction and parameterization of declarations yielding modules and functors, respectively. Let us explain the features in more detail:

Modules: Modules encapsulate declarations. We also allow the specification of two special procedures attached to a module, namely an initializer and a terminator. The initializer is invoked when declaring a module, directly after the declarations contained in the module have been "executed"[1]. The terminator is invoked when the module falls out of scope. As a consequence the initializers of a program built of a hierarchy of modules are executed in the order in which they occur in the program, while the terminators are executed in the corresponding reverse order. The type-checker ensures that the order of occurrence is a linear representation of the topological dependence of the modules. A module can be declared by explicitly giving the declarations and the commands for initializing and terminating or via a functor instantiation. The components of a module can be accessed

[1] Declarations add bindings to the environment (usually implemented by a stack) and can have side-effects on the store; therefore we use the notion of "execution" also for declarations.

via qualification or by importing all of the components into the current environment.

Functors: Functors are modules parameterized by other modules. This construct allows the implementation of the common structure of modules in one place, namely as a functor. Such a functor can then be instantiated with the actually needed details. For instance, the list operations for construction and selection do not depend on the underlying type at all; the matrix operations for addition and multiplication can be defined simply in terms of the corresponding operations of the underlying type[2]. A functor for lists would, for instance, be parameterized by (a module containing) the underlying type, a functor for matrices could be parameterized by (a module containing) the type of the elements together with the operations needed on them. One can then construct modules for matrices over integers or any other suitable type by simply instantiating the matrix functor with a suitable module implementing a basic type. What "suitable" means in this context, we will describe just below in the next item. The restriction to one parameter is not a severe one: several components can always be grouped together into one component. (This is the same approach as taken in the design of SML. There, additionally, an extended syntax allows more parameters, that are transparently grouped into a single module by the parser.)

Classifying Modules: Since we now have modules as parameters, we need a way of classifying them like parameters of functions and procedures that are classified by types. Such a classification allows separate type checking of the functor body and an application, i. e. a functor instantiation. The appropriate domain is that of type assignments together with additional operations on them. The corresponding syntax construct is called *signature* to be introduced in the next chapter. In this chapter we restrict our investigations to the type assignments corresponding to modules and signatures and allow type assignments for the moment also to appear in the abstract syntax (like we did for type expressions and the corresponding type domains δ and τ in Chapter 3).

Compile-Time Instantiation and Information Passing: In SML functors are compiled by themselves and abstract types all have equal size in the representation. Both strategies allow separation of modules on a high level but at the same time introduce at least one level of indirection. Instantiating functors at compile-time allows direct access to the components in the compiled code and gives furthermore a lot of opportunities for code optimization that would otherwise be lost. Allowing abstract types to have arbitrary size enables more efficient representations where the components can also be accessed more directly. Both issues involve the passing of low-level information between modules. Here we show how to handle the size of types as an example for such information passing; another example would be the inlining of function calls that is, however, not treated in this report.

[2]The *properties* of the matrix operations of course also depend on the properties of the underlying type and its operations. But this is an issue not covered by this report; it is sketched, however, in more detail in Section 6.4.

Here is a brief overview over the following sections:

1. **Syntax Rules:** the syntax rules together with detailed explanations of the constructs and an example program,
2. **Preparations:** preparations for the typing and semantic rules including necessary extensions of type domains and new operators,
3. **Typing Rules:** the typing rules and comments,
4. **Semantic Rules:** the semantic rules and comments.

5.1 Syntax Rules

$$\begin{array}{lll} \mathrm{X} & ::= & \mathrm{I} \mid \mathrm{X}::\mathrm{I} \\ \mathrm{D} & ::= & \ldots \mid \textbf{functor}\ \mathrm{I}_1[\mathrm{I}_2:\pi] = \mathrm{M} \mid \textbf{module}\ \mathrm{I} = \mathrm{M} \mid \textbf{import}\ \mathrm{X} \\ \mathrm{M} & ::= & \textbf{mod}\ \mathrm{D}\ \textbf{init}\ \mathrm{C}_1\ \textbf{term}\ \mathrm{C}_2\ \textbf{end} \mid \mathrm{X}_1[\mathrm{X}_2] \end{array}$$

5.1.1 Long Identifiers: X

Long identifiers are the construct for qualifying components of a module. A function *eval* in a module *Polynom* is accessed by *Polynom* :: *eval*. Since modules can be nested, also the qualification can be nested. Long identifiers X can now be used in many places where simple identifiers I where used up to now. The exceptions are the constructs in the syntax domains of declarations and parameters (newly introduced identifiers must still be simple ones) and the record qualification E.I. The syntax rule for a procedure call, for instance, is now X A. We will not discuss the affected constructs and rules here, since the changes are trivial. The complete listing of all constructs and rules in the appendix, however, contains these changes.

5.1.2 Declarations: D

The concepts have already been explained quite detailed in the introduction to this chapter, therefore here only briefly the intentions:

module I = M**:** The module body M consists of declarations, an initializing command and a terminating command. These items can be specified directly or come from a functor instantiation (see below). The declarations are executed and grouped together under the identifier I. The initializing command is executed immediately after the declarations, whereas the terminating command is viewed as an additional destructor (besides the destructors of the components); it is put on the stack of destructors and will be executed when the module falls out of scope.

functor $\mathrm{I}_1[\mathrm{I}_2:\pi] = \mathrm{M}$**:** The functor body M has the same structure as the module body, however, the items are not executed yet. Instead they are parameterized by another module. In order to know which components with what types this module will have, the parameter is attached by a type assignment π.

import X**:** This is a construct for mere convenience; the components of a module I are imported into the current environment and need no longer be qualified by I.

5.1.3 Module Body: M

The module body defines "the contents" of a module or, when parameterized, a functor. As mentioned before, a module body consists of three items:

Declarations: The declarations are the components of a module or functor.

Initializer: This is a command executed upon the definition of a module.

Terminator: This is a command executed upon termination of the module, i. e. when it falls out of scope. It is invoked transparently, without any direct correspondence in the program.

These three items can either be specified directly by the

mod D **init** C_1 **term** C_2 **end**

construct, or by instantiating a functor, i. e. by the

$X_1[X_2]$

construct. In the latter case these three items have already been declared with a previous functor definition

functor $I_1[I_2 : \pi] = M$.

5.1.4 Example Program

For simplicity the program is given (see Figure 5.1) in text (ASCII) notation again.

The program defines a functor *List* parameterized by a module that must contain (at least) a type component *Type*; of course, it can contain more components. The initializer and terminator of the functor are empty (the empty command **skip**). Here we pack the predefined type *int* into a module *Int* and then instantiate the list functor with it yielding a module *IntList*, which is imported into the current environment. Thus we now have the type *Type*, a list over integers, together with its access functions available. The main part of the program reads in integers and writes them onto the output stream in reverse order by using the list type.

5.2 Preparations

5.2.1 Type Domains

The constructed types now allow long identifiers (instead of just simple ones) and the type domain ι for identifiers is extended by alternatives for the new kinds of identifiers, namely module and functor identifiers:

```
with
  functor List[ Elem: {Type:(Type,new)type} ] =
    mod
      newtype Type = Pointer[ Record[ elem: Elem::Type,
                                      next: Type ] ];
      const Nil:Type = Type'nil;
      proc addHead( h: Elem::Type, l: Type, ref r: Type ) =
        with var r1: Type do
        r1.elem = h; r1.next = l;
        r = r1;
      fun head( l:Type ):Elem::Type = !l.elem;
      fun rest( l:Type ):Type       = !l.next
    init skip  term skip;

  module Int = mod type Type = int  init skip  term skip;
  module IntList = List[Int]; import IntList;

  var l:Type; var n:int; var e:int;
do
  l := Nil;
  while( not eof ) do
    read e; addHead( e, l, l );
  while( not l = Nil )
    write head l; l := rest l
```

Figure 5.1: Example Program for the Module System

$$
\begin{array}{lcll}
\delta & ::= & \gamma \mid \mathrm{X} \mid \delta\, ptr \mid \delta\, arr \mid \lambda rec & \text{constructed} \\
\iota & ::= & \tau \mid \rho \to \delta \mid \rho \to Cmd \mid (\delta, \phi) type \mid cons \mid dst \mid & \\
 & & \pi\, mod \mid (\pi_1, \pi_2) func & \text{identifier}
\end{array}
$$

The first argument π_1 in the type alternatives for module and functor identifiers, is the (unconstrained) type assignment corresponding to the components; in the notion of the next chapter it will be referred to as the *inherent signature*. The functor identifier contains a second argument π_2 that describes the minimal contents of an argument module to the functor (in the notion of the next chapter: "the signature an argument module must satisfy").

The types attached to phrases of the new syntax domains are the following ones:

$\mathrm{X} : \iota Long$

$\mathrm{M} : \pi Mod$

5.2.2 Type Equivalence

For the long identifiers in the domain of constructed types we need one additional rule for type equivalence:

$$
(\mathrm{TE8}) \quad \frac{\pi \vdash \mathrm{X} : \pi_1 mod Long \qquad \pi_1 \vdash \mathrm{I} \sim \delta}{\pi \vdash \mathrm{X} :: \mathrm{I} \sim \delta}
$$

It basically says, that if X is a module in context π, and within this module a type equivalence for type I can be proven, then the type equivalence holds also for $\mathrm{X} :: \mathrm{I}$ in the outer context.

5.2.3 Operations on Type Assignments

Passing Type Information

As mentioned in the introduction to this chapter, we will have to pass the sizes of types between different modules. We will do this by passing the representation of types between type assignments. Therefore we introduce the operator $+$ on type assignments: $\pi_1 + \pi_2$ is essentially the type assignment π, where possible undefined types $(\mathrm{I}\!:\!(\mathrm{I}, \mathit{hidden})\mathit{type}) \in \pi_1$ get their representation from a definition $(\mathrm{I}\!:\!(\delta, \phi)\mathit{type}) \in \pi_2$:

$$\pi_1 + \pi_2 = \pi_1 \uplus \{\mathrm{I}\!:\!(\delta, \mathit{hidden})\mathit{type} \mid \mathrm{I}\!:\!(\mathrm{I}, \mathit{hidden})\mathit{type} \in \pi_1 \wedge \mathrm{I}\!:\!(\delta, \phi)\mathit{type} \in \pi_2\}$$

Module Matching

Signature matching is to type assignments what type equivalence is to types. When instantiating a functor we have to check whether the argument module, more precisely its type assignment π_1, matches the type assignment π_2 of the parameter; formally this is expressed by $\pi_1 \Rightarrow \pi_2$. The matching basically comes down to checking whether each component in π_2 is also defined in π_1 and the corresponding types are equivalent. See Subsection 6.2.4 for the complete definition.

Type Sizing

Another important issue is the size of types when type checking a functor body. Types declared in the type assignment for the parameter need not be fully defined (and actually, as we will see in the next chapter, they will never be defined, as the syntax of signatures does not allow this), and therefore their size is not known. Still they might be used in constructs where the size of the type must exist, like in a variable declaration or a **new** construct. Furthermore, when eventually instantiating the functor by a module, the actual type definitions can have arbitrary size.

Luckily the actual size of a type is not important for type checking. It suffices that the type has a size greater than 0. Therefore we will assume that the types in the type assignment of the parameter have a known size; this is the purpose of the function *sized*. See Subsection 6.2.5 for the complete definition.

5.3 Typing Rules

5.3.1 Long Identifiers

$$(\mathrm{X1})\ \frac{\pi \vdash \mathrm{I}\!:\!\tau\mathit{Idf}}{\pi \vdash \mathrm{I}\!:\!\tau\mathit{Long}} \qquad (\mathrm{X2})\ \frac{\pi \vdash \mathrm{X}\!:\!\pi_1 \mathit{modLong} \qquad \pi_1 \vdash \mathrm{I}\!:\!\tau\mathit{Idf}}{\pi \vdash \mathrm{X}\!::\!\mathrm{I}\!:\!\tau\mathit{Long}}$$

A long identifier is either a simple identifier, in which case the type is just passed on, or it is a module identifier qualified by a simple identifier, in which case the module must contain a corresponding component. *List* :: *Elem* :: *Type* is a valid long identifier, if *List* denotes a module, that contains another module *Elem* containing a component *Type*. The type of the whole qualified identifier is in this case the type of component *Type* in the type assignment corresponding to module *Elem*.

5.3.2 Declarations

$$\text{(D12)}\ \frac{\pi_{par} \vdash \mathrm{M} : \pi_1 \mathit{Mod}}{\pi \vdash \textbf{functor}\ \mathrm{I}_1[\mathrm{I}_2 : \pi_2] = \mathrm{M} : \{\mathrm{I}_1 : (\pi_1, \pi_2)\mathit{func}\}\mathit{Dec}},$$

$$\pi_{par} = \{\mathrm{I}_2 : \mathit{sized}(\pi_2)\}$$

The functor body shall, essentially, only depend on its argument module (which is a design decision for gaining maximal modularity). In the next chapter we will, however, extend the definition of π_{par} to also contain the signatures of the current context π. Therefore the typing of the functor body M is for the moment only based on the type assignments π_2 of the parameter signature with its types assumed to be sized. Since for some typing rules the sizes of the types need to be defined, the types in π_2 are assumed to have a known (dummy) size (this is the purpose of the function *sized*). Later, when instantiating the functor, these types are checked for their sizes (see Subsection 5.3.3).

$$\text{(D13)}\ \frac{\pi \vdash \mathrm{M} : \pi_1 \mathit{Mod}}{\pi \vdash \textbf{module}\ \mathrm{I} = \mathrm{M} : \{\mathrm{I} : \pi_1 \mathit{mod}\}\mathit{Dec}}$$

A module declaration simply introduces a module identifier that is bound to the corresponding type assignment.

$$\text{(D14)}\ \frac{\pi \vdash \mathrm{X} : \pi_1 \mathit{modLong}}{\pi \vdash \textbf{import}\ \mathrm{X} : \pi_1 \mathit{Dec}}$$

An **import** statement makes the components of a module accessible without qualification; its corresponding type assignment π_1 is embedded ("lifted up") into the current type assignment.

5.3.3 Module Body

$$\text{(M1)}\ \frac{\pi \vdash \mathrm{D} : \pi_1 \mathit{Dec} \qquad \pi_1 \vdash \mathrm{C}_1 : \mathit{Cmd} \qquad \pi_1 \vdash \mathrm{C}_2 : \mathit{Cmd}}{\pi \vdash \textbf{mod}\ \mathrm{D}\ \textbf{init}\ \mathrm{C}_1\ \textbf{term}\ \mathrm{C}_2\ \textbf{end} : \pi_1 \mathit{Mod}}$$

The type of a module body is the type of its declarations. The initializer C_1 and terminator C_2 are typed in the context of the module.

$$\text{(M2)}\ \frac{\pi \vdash \mathrm{X}_1 : (\pi_1, \pi_2)\mathit{funcLong} \qquad \pi \vdash \mathrm{X}_2 : \pi_3 \mathit{modLong} \qquad \pi_3 \Rightarrow \pi_2}{\pi \vdash \mathrm{X}_1[\mathrm{X}_2] : \pi_1 \mathit{Mod}}$$

For instantiating a functor the main check is the matching of the type assignment π_3 of the argument module X_2 against the type assignment π_2 of the parameter. This match includes the check for the sizes of the types defined in the argument module (see Subsection 6.2.4).

5.4 Semantic Rules

5.4.1 Long Identifiers

$$\mathrm{X} \quad ::= \quad \mathrm{I} \mid \mathrm{X}::\mathrm{I}$$

$[\![\pi \vdash . : \tau Long]\!] : CompInfo \rightarrow IdValue$

$[\![\pi \vdash \mathrm{I} : \tau Long]\!] c = [\![\pi \vdash \mathrm{I} : \tau Idf]\!] c$
$[\![\pi \vdash \mathrm{X}::\mathrm{I} : \tau Long]\!] c = [\![\pi_1 \vdash \mathrm{I} : \tau Idf]\!] c_1$,
where $c_1 = [\![\pi \vdash \mathrm{X} : \pi_1 modLong]\!] c$

5.4.2 Declarations

$$\mathrm{D} \quad ::= \quad \ldots \mid \textbf{functor}\ \mathrm{I}_1[\mathrm{I}_2 : \pi_2] = \mathrm{M} \mid \textbf{module}\ \mathrm{I} = \mathrm{M} \mid \textbf{import}\ \mathrm{X}$$

$[\![\pi \vdash \textbf{functor}\ \mathrm{I}_1[\mathrm{I}_2 : \pi_2] = \mathrm{M} : \{\mathrm{I}_1 : (\pi_1, \pi_2) func\} Dec]\!] c\ r = (\{\mathrm{I}_1 = f\}, id, r)$,
where $f\ c' = [\![\pi_{par} \vdash \mathrm{M} : \pi_1 Mod]\!] \{\mathrm{I}_2 = c'\}$
A functor identifier is bound to the meaning of the module body, which can only be evaluated upon instantiation. The function f is then evaluated with the environment resulting from the parameter (see below).

$[\![\pi \vdash \textbf{module}\ \mathrm{I} = \mathrm{M} : \{\mathrm{I} : \pi_1 mod\} Dec]\!] c\ r = (\{\mathrm{I} = c_1\}, d \circ d_1, r_2)$,
where $(dec, init, term) = [\![\pi \vdash \mathrm{M} : \pi_1 Mod]\!] c$
and $(c_1, d_1, r_1) = dec\ r$
and $r_2 = init\ c_1\ r_1$
and $d = term\ c_1$
A module identifier is bound to an environment. The environment is constructed from the declarations *dec* of the module body. Afterwards the initializer *init* is executed. The terminator *term* gets the environment of the module attached and is concatenated with the destructors of the module components.

$[\![\pi \vdash \textbf{import}\ \mathrm{X} : \pi_1 Dec]\!] c\ r = (c_1, id, r)$,
where $c_1 = [\![\pi \vdash \mathrm{X} : \pi_1 modLong]\!]$

5.4.3 Module Body

$$\mathrm{M} \quad ::= \quad \textbf{mod}\ \mathrm{D}\ \textbf{init}\ \mathrm{C}_1\ \textbf{term}\ \mathrm{C}_2\ \textbf{end} \mid \mathrm{X}_1[\mathrm{X}_2]$$

$$\begin{aligned}[\![\pi \vdash . : \pi_1 Mod]\!] : CompInfo \rightarrow\ & ((RunInfo \rightarrow (CompInfo \times RunInfo)) \times \\ & (CompInfo \rightarrow RunInfo \rightarrow RunInfo) \times \\ & (CompInfo \rightarrow RunInfo \rightarrow RunInfo))\end{aligned}$$

A module body consists of three items: a declaration, an initializer and a terminator. These items can be declared directly or via a functor instantiation.
$[\![\pi \vdash \textbf{mod}\ \mathrm{D}\ \textbf{init}\ \mathrm{C}_1\ \textbf{term}\ \mathrm{C}_2\ \textbf{end} : \pi_1 Mod]\!] c = (dec, init, term)$,
where $dec = [\![\pi \vdash \mathrm{D} : \pi_1 Dec]\!] c$
and $init = [\![\pi_1 \vdash \mathrm{C}_1 : Cmd]\!]$
and $term = [\![\pi_1 \vdash \mathrm{C}_2 : Cmd]\!]$
This is the central semantic rule for modules and functors. The three items of

the body are functions parameterized by an environment. In case of the declaration, the first item, this argument is provided for the semantic value of a functor argument; when defining a module the function *dec* will be called with the empty environment (see the corresponding rule above). The parameter to the initializer and the terminator is for the semantic value of the declaration, such that they can access the components of the body.

$[\![\pi \vdash \mathrm{X}_1[\mathrm{X}_2] : \pi_1 \mathit{Mod}\,]\!] c = f\ c_1$,
 where $f = [\![\pi \vdash \mathrm{X}_1 : (\pi_1, \pi_2) \mathit{funcLong}\,]\!] c$
 and $c_1 = [\![\pi \vdash \mathrm{X}_2 : \pi_3 \mathit{modLong}\,]\!] c$

Instantiating a functor is now straight forward: first the argument module X_2 is evaluated and then plugged in into the meaning f of the functor X_1; this actually comes down to evaluate the corresponding module body within the context of the parameter.

Chapter 6

Signatures - Typing Modules

We now replace the type assignments π in the syntax by a new syntax domain S, namely *signatures*. What type expressions are to values, signatures are to modules and functors: they classify the objects and control parameterization. Signature matching, however, is more complex than type matching since several different, non-equivalent, signatures can be attached to one module or functor. Signatures can hide information in two ways:

- Only a subset of the components of a module or functor are specified in the signature and therefore visible from the outside.
- The representation of types is not visible from the outside; a **share** declaration, however, can impose type equivalence.

Another difference to types is that signatures do not have any semantics; their only purpose is to ensure existence of certain components in modules and functors and to control their visibility.

"Thinking big", signatures can be even more expressive and powerful. Signatures describe modules. In this thesis we are only concerned with type checking. However, when instantiating a functor, for instance, we actually do not only expect that the argument module has components with certain types, we usually also assume a certain functionality and certain properties of the components. This cannot be expressed by the signatures described here. In Section 6.4 we will discuss possible extensions to signatures to also specify the functionality of modules.

Here is a brief overview over the following sections:

1. **Syntax Rules:** the syntax rules together with detailed explanations of the constructs and an example program,
2. **Preparations:** preparations for the typing and semantic rules including necessary extensions of type domains and a detailed description of handling signatures,

3. **Typing Rules:** the typing rules and comments,

4. **Axioms - Specifying Functionality** possible extensions to signatures allowing to specify not only the types but also the functionality of module components.

6.1 Syntax Rules

D	::=	... | **functor** $\mathrm{I}_1[\mathrm{I}_2:\mathrm{S}_2]\langle:\mathrm{S}_1\rangle = \mathrm{M}$ | **module** $\mathrm{I}\langle:\mathrm{S}\rangle = \mathrm{M}$ |
	::=	... | **signature** I = S
S	::=	X | **sig** SD **end**
SD	::=	**const** I:T | **var** I:T | **fun** I P:T | **proc** I P |
		type I | **constructor** I P | **destructor** I |
		module I:S | **share type** $\mathrm{X}_1=\mathrm{X}_2$ | **include** X | $\mathrm{SD}_1;\mathrm{SD}_2$

6.1.1 Declarations

The definitions of functors and modules are extended by an optional signature. This allows to constrain the definitions and write thus safer programs. Note that this extension does not have any affect on the semantics!

Furthermore also signatures can be given names. Again this does not have any influences on the semantics; a definition of a signature is only used for type-checking, but ignored by the semantic rules. Giving a name to a signature allows include the components in other signatures without having to retype them.

6.1.2 Signatures

Signatures can be either specified directly by listing the components, i. e. signature declarations, or via a name.

6.1.3 Signature Declarations

Signature declarations correspond to declarations. For each kind of declaration there is a corresponding kind of signature declaration. The only exception are functors and the sharing construct. Though we have some evidence that allowing functors in signatures and thus in arguments to a functor immediately introduces the possibility of declaring higher-order functors in a well-defined way, this topic needs further investigations. The sharing construct is needed to tell the type-checker that types occurring in several places (in different submodules, for instance) are actually the same type.

Let us have a look at the constructs with peculiarities:

type I: Types specified in a signature are always abstract. With the sharing construct it is, however, possible to introduce further equivalence on types.

```
...
signature ELEM = sig type Type end;
signature LIST = sig
  module Elem: ELEM;
  type Type;
  const Nil:Type;
  proc addHead( h:Elem::Type, l:Type, ref r:Type );
  fun head( l:Type ):Elem::Type;
  fun rest( l:Type ):Type
end;

functor List[Elem: ELEM]: LIST = ... ;

module Int: ELEM = ... ;
...
```

Figure 6.1: Example Program for Signatures

share type X_1=X_2**:** This construct comes into play when building complex hierarchies of modules and functors. The same (abstract) type may be needed in several modules, which in turn may be used together in another module. Then it is necessary to have a way of telling the type-checker, that certain types in different sub-modules are actually the same type in order to let interact corresponding functions.

6.1.4 Example Program

Let us give here signatures corresponding to the modules and the list functor given in the previous chapter Figure 5.1. See Figure 6.1 for the exemplary signatures.

The signature *ELEM* replaces the type assignment for the functor parameter. The signature *LIST* is not actually needed, but it is good (because safe) programming style to attach signatures to functors and modules. Since signatures do not have parameters, the parameter module *Elem* has to be explicitly stated in the signature. The rest is straight forward.

6.2 Preparations

6.2.1 Type Domains

The alternatives for module and functor identifiers in type domain ι are extended by an optional component containing (the type assignment of) the optional signature attached (see the corresponding syntax constructs for declaring functors and modules in the previous section). One more alternative is added to the domain ι for signature identifiers. The type flag ϕ is extended by two more alternatives: *hidden* for denoting an abstract type, whose structure may not be used in the typing rules, and *sized* for denoting that the type is assumed to have a size (see also the added rule for the predicate *is_sized* below). The alternative *sized* will only occur together with abstract types in the signatures of functor parameters.

$$\begin{array}{lll} \iota & ::= & \tau \mid \rho \to \delta \mid \rho \to Cmd \mid (\delta,\phi)type \mid cons \mid dst \mid \\ & & (\pi_1\langle,\pi_2\rangle)mod \mid (\pi_1,\pi_2\langle,\pi_3\rangle)func \mid \pi Sig \qquad \text{identifier} \\ \phi & ::= & alias \mid forward \mid new \mid hidden \mid sized \qquad \text{type flag} \end{array}$$

6.2.2 Handling Optional Type Assignments

Here we introduce a simple operator that will help us to handle type assignments corresponding to optionally given signatures (as occurring in the declarations of modules and functors):

$$\pi_2 : \pi_1 = \pi_2$$

The purpose of this operator ':' becomes clear, when using it together with optional syntax:

$$\langle \pi_2 :\rangle \pi_1 = \begin{cases} \pi_2, & \pi_2 \text{ given} \\ \pi_1, & \text{otherwise} \end{cases}$$

Optional syntax always arises when combining two very similar rules into one for ease and brevity of notation.

6.2.3 Extracting Signatures

In some places in the typing rules we only want the signatures of a context to be visible: namely for typing a functor body and for typing signatures (see the corresponding remarks). This is achieved by the function *sigs*:

$$sigs(\pi) = \{\mathrm{I} : \pi_1 sig | \mathrm{I} : \pi_1 sig \in \pi\}$$

Since signatures are only needed for type checking and do not have any influence on the semantic rules, they can be safely added to another context without violating the correspondence between context and environment.

6.2.4 Signature Matching

Modules and functors have an "inherent type", namely the type assignment corresponding to their components; let us call this the *inherent signature* of a module or functor. Having, for instance,

$\pi \vdash$ **module** $\mathrm{I}\langle\mathbf{:}\mathrm{S}\rangle = \mathrm{M} : \pi_2 Mod$,

where

$\pi \vdash \mathrm{M} : \pi_1 Mod, \quad \pi \vdash \mathrm{S} : \pi_2 Sig,$

π_1 is the inherent signature of the module I. π_2 is an optional *explicitly defined signature*, which must be "matched" by the inherent signature π_1 (see below for an exact definition). Explicitly defined signatures serve two purposes:

Visibility Control: By attaching an optional signature to a module or functor one can constrain the visibility of components. They can be hidden to the outside at all, or for types only the representation can be hidden. On the other hand one can assure that the components specified in the signature are really declared inside the module and can be accessed from the outside.

Parameterization: In order to type-check a parameterized module without needing to know the module(s) this functor is going to be instantiated with, the components of the parameter that may be accessed and their types need to be known. This is achieved by specifying the parameter with a signature. When instantiating the functor the argument module must match the parameter signature.

In both cases the inherent signature of a module or functor has to be matched against a maybe more general explicitly defined signature. We write
$\pi_1 \Rightarrow \pi_2$
iff a specific signature π_1 matches the defined signature π_2.

Signature matching (i. e. checking for $\pi_1 \Rightarrow \pi_2$) consists of three parts:

- Components declared in π_2 must also be declared in π_1; loosely speaking, the set of identifiers bound by π_2 is a subset of set of identifiers bound by π_1 ($\pi_1 \supset \pi_2$). Components appearing in both signatures that are not types themselves must have the same type assigned in both signatures.
- Types declared in π_2 must be defined in π_1. A type I declared in a signature is always hidden and unspecified ($(\mathrm{I}, \mathit{hidden})\mathit{type}$). A corresponding binding $(\delta, \phi)\mathit{type}$ in the inherent signature must exist and be fully specified ($\pi \vdash \mathit{is\_sized}(\delta)$).
- Types $\mathrm{X}_1, \mathrm{X}_2$ involved in a sharing constraint of signature π_2 ($\mathit{share}(\mathrm{X}_1, \mathrm{X}_2) \in \pi_2$) must be defined in π_1 and must be equivalent: $\pi_1 \vdash \mathrm{X}_1 \sim \mathrm{X}_2$.

These rules can be formalized in the following way:

$$(\mathrm{SM1})\ \frac{\pi_1 \supseteq \pi_2}{\pi_1 \Rightarrow \pi_2} \qquad (\mathrm{SM2})\ \frac{\pi \vdash \mathrm{I} : (\delta, \phi)\mathit{typeIdf} \qquad \pi \vdash \mathit{is\_sized}(\delta)}{\pi \Rightarrow \{\mathrm{I} : (\mathrm{I}, \mathit{hidden})\mathit{type}\}}$$

$$(\mathrm{SM3})\ \frac{\pi \vdash \mathrm{X}_1 \sim \mathrm{X}_2}{\pi \Rightarrow \{\mathit{share}(\mathrm{X}_1, \mathrm{X}_2)\}}$$

We need one more rule to combine the above implications:

$$(\mathrm{SM4})\ \frac{\pi_1 \Rightarrow \pi_2 \qquad \pi_1 \Rightarrow \pi_2'}{\pi_1 \Rightarrow \pi_2 \uplus \pi_2'}$$

Note that rule (SM1) could be relaxed. Here we demand by the superset operator that the types of corresponding components are not only equivalent, but literally the same. It would, of course, suffice having the types equivalent; but we will not go into details here how to specify this.

In order to let the sharing construct take affect we need one more rule for type equivalence:

$$(\mathrm{TE9})\ \frac{\mathit{share}(\mathrm{X}_1, \mathrm{X}_2) \in \pi}{\pi \vdash \mathrm{X}_1 \sim \mathrm{X}_2}$$

6.2.5 Type Sizing

Types in an explicitly defined signature do not have a representation attached and therefore also their size is not known. Hence when typing a functor body there may be types with an unknown size in the current context (coming in via the signature of the parameter). However, in a variable declaration or a **new** command, the size of the involved types must be known. We solved this problem by assuming that the sizes of all types in the parameter signature are known. This is achieved by the function *sized* that attaches a dummy size to all types in the signature (analogously to recursive type definitions):

$$\mathit{sized}(\{\mathrm{I}:\iota\}) = \begin{cases} \{\mathrm{I}:(\mathrm{I},\mathit{sized})\mathit{type}\}, & \iota = (\mathrm{I},\mathit{hidden})\mathit{type} \\ \{\mathrm{I}:\mathit{sized}(\pi_1)\}, & \iota = \pi_1 \\ \{\mathrm{I}:\iota\}, & \text{otherwise} \end{cases},$$

In order to let this operation take affect, we add one more rule for the predicate *is_sized*:

$$\text{(TS7)}\ \frac{(\mathrm{I}:(\mathrm{I},\mathit{sized})\mathit{type}) \in \pi}{\pi \vdash \mathit{is\_sized}(\mathrm{I})}$$

6.3 Typing Rules

6.3.1 Declarations

The following two rules replace the corresponding rules of the previous chapter. Now the signatures represent a syntax domain by itself and the type assignments need no longer be declared directly. Note that the semantics do not change!

$$\text{(D12)}\frac{\pi_{par} \vdash \mathrm{M}:\pi_1 \mathit{Mod} \quad \pi \vdash \mathrm{S}_2:\pi_2 \mathit{Sig} \quad \langle \pi \vdash \mathrm{S}_1:\pi_3 \mathit{Sig}\rangle \quad \langle \pi_1 \Rightarrow \pi_3\rangle}{\pi \vdash \textbf{functor}\ \mathrm{I}_1[\mathrm{I}_2:\mathrm{S}_2]\langle\mathbf{:}\mathrm{S}_1\rangle\ =\ \mathrm{M}:\{\mathrm{I}_1:(\pi_1,\pi_2\langle,\pi_3\rangle)\mathit{func}\}\mathit{Dec}},$$

$$\pi_{par} = \mathit{sigs}(\pi) \uplus \{\mathrm{I}_2:\mathit{sized}(\pi_2)\}$$

Here also π_{par} was changed in a subtle detail: by adding the signatures $\mathit{sigs}(\pi)$ of the current context they can now be used inside the functor body. This is a matter of convenience and does not violate the concept of modularity.

$$\text{(D13)}\ \frac{\pi \vdash \mathrm{M}:\pi_1 \mathit{Mod} \quad \langle \pi \vdash \mathrm{S}:\pi_2 \mathit{Sig}\rangle \quad \langle \pi_2 \Rightarrow \pi_1\rangle}{\pi \vdash \textbf{module}\ \mathrm{I}\langle\mathbf{:}\mathrm{S}\rangle\ =\ \mathrm{M}:\{\mathrm{I}:(\pi_1\langle,\pi_2\rangle)\mathit{mod}\}\mathit{Dec}}$$

$$\text{(D15)}\ \frac{\pi \vdash \mathrm{S}:\pi_1 \mathit{Sig}}{\pi \vdash \textbf{signature}\ \mathrm{I} = \mathrm{S}:\{\mathrm{I}:\pi_1 \mathit{sig}\}\mathit{Dec}}$$

A signature declaration simply introduces a signature identifier that is bound to the corresponding type assignment.

6.3.2 Signatures

Signatures can be either a previously defined identifier or specified directly.

$$\text{(S1)}\ \frac{\pi \vdash \mathrm{X} : \pi_1 \mathit{sigIdf}}{\pi \vdash \mathrm{X} : \pi_1 \mathit{Sig}}$$

$$\text{(S2)}\ \frac{\mathit{sigs}(\pi) \vdash \mathrm{SD} : \pi_1 \mathit{SDec}}{\pi \vdash \textbf{sig}\ \mathrm{SD}\ \textbf{end} : \pi_1 \mathit{Sig}}$$

The function *sigs* extracts the signature identifiers from a type assignment. This is a design decision and enforces signatures to only depend on other signatures, but not on arbitrary declarations.

6.3.3 Signature Declarations

The following four rules correspond to the respective declarations (D1) – (D4):

$$\text{(SD1)}\ \frac{\pi \vdash \mathrm{T} : \delta\, \mathit{Type}}{\pi \vdash \textbf{const}\ \mathrm{I} : \mathrm{T} : \{\mathrm{I} : \delta\} \mathit{SDec}} \qquad \text{(SD2)}\ \frac{\pi \vdash \mathrm{T} : \delta\, \mathit{Type}}{\pi \vdash \textbf{var}\ \mathrm{I} : \mathrm{T} : \{\mathrm{I} : \delta\, \mathit{loc}\} \mathit{SDec}}$$

$$\text{(SD3)}\ \frac{\pi \vdash \mathrm{P} : \rho \mathit{Par} \qquad \pi \vdash \mathrm{T} : \delta\, \mathit{Type}}{\pi \vdash \textbf{fun}\ \mathrm{I}\ \mathrm{P} : \mathrm{T} : \{\mathrm{I} : \rho \to \delta\} \mathit{SDec}}$$

$$\text{(SD4)}\ \frac{\pi \vdash \mathrm{P} : \rho \mathit{Par}}{\pi \vdash \textbf{proc}\ \mathrm{I}\ \mathrm{P} : \{\mathrm{I} : \rho \to \mathit{Cmd}\} \mathit{SDec}}$$

Types in a signature are always abstract: their representation is hidden.

$$\text{(SD5)}\ \pi \vdash \textbf{type}\ \mathrm{I} : \{\mathrm{I} : (\mathrm{I}, \mathit{hidden})\mathit{type}\} \mathit{SDec}$$

Type constructors and destructors are treated analogously to procedures:

$$\text{(SD6)}\ \frac{\pi \vdash \mathrm{I} : (\delta, \phi)\mathit{typeIdf} \qquad \pi \vdash \mathrm{P} : \rho \mathit{Par}}{\pi \vdash \textbf{constructor}\ \mathrm{I}\ \mathrm{P} : \{\mathrm{I} : (\rho \to \mathit{Cmd})\mathit{cons}\} \mathit{SDec}}$$

$$\text{(SD7)}\ \frac{\pi \vdash \mathrm{I} : (\delta, \phi)\mathit{typeIdf}}{\pi \vdash \textbf{destructor}\ \mathrm{I} : \{\mathrm{I} : \mathit{dst}\} \mathit{SDec}}$$

The following rule corresponds to a module declaration:

$$\text{(SD8)}\ \frac{\pi \vdash \mathrm{S} : \pi_1 \mathit{Sig}}{\pi \vdash \textbf{module}\ \mathrm{I} : \mathrm{S} : \{\mathrm{I} : \pi_1\} \mathit{SDec}}$$

$$\text{(SD9)}\ \frac{\pi \vdash \mathrm{X}_1 : (\mathrm{I}_1, \mathit{hidden})\mathit{typeLong} \qquad \pi \vdash \mathrm{X}_2 : (\mathrm{I}_2, \mathit{hidden})\mathit{typeLong}}{\pi \vdash \textbf{share type}\ \mathrm{X}_1 = \mathrm{X}_2 : \{\mathit{share}(\mathrm{X}_1, \mathrm{X}_2)\} \mathit{SDec}}$$

The typing rule for the sharing construct is rather simple; its importance and consequences come through its influence on type equivalence and signature matching. More specifically the sharing construct has the following two effects:

- On the one hand it influences the type equivalence (see rule (TE9) in Subsection 6.2.4). Under a type assignment π containing $share(\mathrm{X}_1, \mathrm{X}_2)$ the type equivalence $\pi \vdash \mathrm{X}_1 \sim \mathrm{X}_2$ holds, which maybe could not be derived otherwise. This is needed for typing a functor body, where a sharing construct in the signature of the parameter module can specify two type names (usually the same type appearing in different submodules of the parameter) as equivalent and therefore allow interaction of the corresponding functions and procedures within the functor body.
- On the other hand the sharing construct further restricts signature matching in the sense that it imposes type equivalence on types in an inherent signature (see rule (SM3) in Subsection 6.2.4). Given an inherent signature π_1 and an explicitly defined signature π_2 containing a sharing condition $share(\mathrm{X}_1, \mathrm{X}_2)$, the matching condition $\pi_1 \Rightarrow \pi_2$ is only fulfilled if the types X_1 and X_2 in π_1, respectively in the corresponding module, are equivalent. This is needed to make sure that the argument module in a functor instantiation is suitable.

Including another signature avoids retyping the declarations: the corresponding type assignment is just taken over.

$$\text{(SD10)}\ \frac{\pi \vdash \mathrm{S} : \pi_1 Sig}{\pi \vdash \textbf{include}\ \mathrm{X} : \pi_1 SDec}$$

Finally, sequential signature declarations are combined like actual declarations.

$$\text{(SD11)}\ \frac{\pi \vdash \mathrm{SD}_1 : \pi_1 SDec \qquad \pi \vdash \mathrm{SD}_2 : \pi_2 SDec}{\pi \vdash \mathrm{SD}_1;\mathrm{SD}_2 : \pi_1 \uplus \pi_2 SDec}$$

6.4 Axioms - Specifying Functionality

Here in this report, and also in SML, signatures specify the types of module (and functor) components. However, when we think of a module, we immediately think actually more about the functionality than the types. It would also make sense to restrict the arguments to a functor not only by type information but also by information about the functionality. This could help in eliminating programming errors (see the remarks on formally specified functionality below).

There are already approaches to develop programs entirely based on rigid specifications. The language OBJ (Goguen et al., 1992), for instance, was developed for expressing and executing algebraic specifications. OBJ's specifications can be viewed as "executable signatures" that are evaluated by rewriting. Another approach is that of Extended ML (Kahrs et al., 1993). SML signatures are extended to specifications by adding axioms (logic formulae with universal and existential quantification) that any structure matching this specification must fulfill. A formal verification model has been developed that allows the programmer to derive by a sequence of proven transformation steps a correct SML program from an specification in Extended ML. However, these approaches are still experimental and not practical yet.

Yet another approach is to add assertions to programs that are checked at run-time like the `assert` macro in C. This approach can be a valuable aid in debugging.

What we propose (and actually are investigating in course of the current HPGP project at RISC-Linz) is an extension of signatures by axioms. Axioms specify in a formalized way the behavior and interplay of functions, procedures and types. Whether these axioms are checked at compile-time or translated to run-time assertions is for now not a primary concern. Simply the fact that the functionality of components of a module is formally specified has important impacts:

- Specifying the behavior of functions and procedures by axioms forces one to think more rigorously about the implementation and helps structuring programs.
- The axioms can serve as a contract between the user of a module and its implementor. Errors can be mapped to single pieces of code by checking which axioms are violated.

Eventually the axioms could be used for automatic code verification by future, more powerful systems that can go over the code and verify the functionality of the implementation.

Let us give here an example for the signature of the list functor (see Figure 5.1 for the functor definition). The code is again in text (ASCII) notation.

```
signature ListS = sig
  type Type;
  const Nil:Type;

  let
    module Elem: sig type Type end
  in
    fun  head( l: Type ): Elem::Type
      axiom input: not l = Nil;
    fun  rest( l: Type ): Type
      axiom input: not l = Nil;
    proc addHead( h: Elem::Type, l: Type, ref r: Type )
      axiom output: head( r ) = h and rest( r ) = l
  end
end
```

Instead of just specifying the types of the functions and procedures, we also give input and output axioms. The selecting functions *head* and *rest* shall only be called on inputs different from the empty list *Nil*. The functionality of the procedure *addHead* shall be in some sense inverse to the selecting functions.

The axioms can be seen from two different views:

- From a computational point of view they are executable functions with a boolean return value that could be translated to run-time assertions.

- From a logical point of view axioms are predicates where free variables are universally quantified. The predicates could be incorporated into the corresponding type assignment for further restricting signature matching: $\pi_1 \Rightarrow \pi_2$ would only hold if the module corresponding to the inherent signature π_1 implements the specification corresponding to the explicitly defined signature π_2 also with respect to those predicates.

Chapter 7

Making the Rules Executable

Bibliography

Aichinger, E. (1997). A linear system solver for Mathematica that uses SACLIB (Version 1.0). Technical Report 5, RISC-Linz, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz (Austria).

Becker, T. and Weispfenning, V. (1993). *Gröbner Bases — A Computational Approach to Computer Algebra.* Springer, Graduate Texts in Mathematics.

Buchberger, B. (1996). Symbolic Computation: Computer Algebra and Logic. *Frontiers of Combining Systems (F. Baader, K.U. Schulz eds.)*, pages 193–220.

Buchberger, B., Collins, G. E., and Loos, R. (1982). *Computer Algebra — Symbolic and Algebraic Computation.* Springer, Wien.

Buchberger, B. and Collins, G. e. a. (1993). SACLIB 1.1 User's Guide. Technical Report 19, RISC-Linz, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz (Austria).

Buchberger, B. and Windsteiger, W. (1993). GRÖBNER: A Library for Computing Gröbner Bases based on SACLIB. Technical Report 72, RISC-Linz, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz (Austria).

Buchberger, B. e. a. (1997). An Overview on the *Theorema* Project. *Accepted at ISSAC'97.*

Collings, G. E. (1975). Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. *Automata Theory and Formal Languages, 2^{nd} GI Conference, Kaiserslautern, May 20-23.*

Goguen, J. A., Coleman, D., and Gallimore, R. e. (1992). *Applications of Algebraic Specifications using OBJ.* Cambridge University Press.

Hong, H. (1989). Partial CAD Construction in Quantifier Elimination. Technical Report OSU-CISRC-10/89 TR45, Ohio State University, 2036 Neil Avenue Mall, Columbus, Ohio 43210.

Hong, H., Neubacher, A., and Stahl, V. (1994). The Sturm Library Manual. Technical Report 30, RISC-Linz, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz (Austria).

Hong, H. and Schreiner, W. (1996). HPGP — High-Performance Generic Programming for Computational Mathematics by Compile-Time Instantiation of Higher-Order Functors. Technical Report 96-10, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria. World Wide Web http://www.risc.uni-linz.ac.at:81/projects/basic/hpgp/reports/96-1.

Kahrs, S., Sannella, D., and A., T. (1993). The Semantics of Extended ML: A Gentle Introduction. *International Workshop on Semantics of Specification Languages, Springer Workshops in Computing, pages 186-215.*

Milner, R. and Tofte, M. (1991). *Commentary on Standard ML.* MIT Press.

Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML.* MIT Press.

Paule, P. and Schorn, M. (1995). A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities . Technical Report 10, RISC-Linz, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz (Austria).

Schmidt, D. A. (1986). *Denotational Semantics - A Methodology for Language Development.* Allyn and Bacon, Boston, MA.

Schmidt, D. A. (1994). *The Structure of Typed Programming Languages.* MIT Press.

Stöcher, W. (1995). Denotational Semantics of a Procedural and a Functional Core Language. Technical report, RISC-Linz, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz (Austria).

Stöcher, W. (1996). Denotational Semantics of a Procedural and a Functional Core Language. Project Report HPGP-96-5, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria. World Wide Web http://www.risc.uni-linz.ac.at:81/projects/basic/hpgp/reports/96-5.

Stoy, J. (1997). *The Scott-Strachey Approach to Programming Language Theory.* MIT Press.

Appendix A

Summary of Semantics

A.1 Syntax Rules

Syntax Domain	typed variable	Chapter(s)
Identifier	I	3
Long Identifier	X	5
Type Expressions	T	4
Declarations	D	3,4,5,6
Initializing Expressions	IE	4
Module Body	M	5
Signatures	S	6
Signature Declarations	SD	6
Parameters	P	3,4
Arguments	A	3
Expressions	E	3,4
Commands	C	3,4
Program	Prg	3

I	::=	i, $\quad i \in$ *Identifier*, the set of identifiers
X	::=	I | X :: I
T	::=	**int** | **bool** | X | **Array**[T] | **Pointer**[T] | **Record**[CL]
CL	::=	$(\mathrm{I}_1 : \mathrm{T}_1, \ldots, \mathrm{I}_n : \mathrm{T}_n), \quad n \geq 1$
D	::=	**const** I : T = IE | **var** I : T | **fun** I P : T=E | **proc** I P=C | $\mathrm{D}_1 ; \mathrm{D}_2$
		type I = T | **newtype** I | **newtype** I = T |
		construct I(P)= C | **destruct** I = C | **var** $\mathrm{I}_1 : \mathrm{I}_2(\mathrm{A})$ |
		functor $\mathrm{I}_1[\mathrm{I}_2 : \mathrm{S}_2]\langle : \mathrm{S}_1\rangle$ = M | **module** I⟨:S⟩ = M | **import** X |
		signature I = S
IE	::=	E | **Array**[A] | **Record**[A]
M	::=	**mod** D **init** C_1 **term** C_2 **end** | $\mathrm{X}_1[\mathrm{X}_2]$
S	::=	X | **sig** SD **end**
SD	::=	**const** I : T | **var** I : T | **fun** I P : T | **proc** I P |
		type I | **constructor** I P | **destructor** I |
		module I : S | **share type** $\mathrm{X}_1 = \mathrm{X}_2$ | **include** X | $\mathrm{SD}_1 ; \mathrm{SD}_2$
P	::=	(⟨**ref** ⟩$\mathrm{I}_1 : \mathrm{T}_1, \ldots,$ ⟨**ref** ⟩$\mathrm{I}_n : \mathrm{T}_n), \quad n \geq 0$
A	::=	$(\mathrm{E}_1, \ldots, \mathrm{E}_n), \quad n \geq 0$
E	::=	X | v | **eof** | X A | $*$ E | $\mathrm{E}_1 \circ \mathrm{E}_2$ |
		T'**nil** | **!**E | $\mathrm{E}_1[\mathrm{E}_2]$ | E.I
C	::=	**skip** | E_1**:=**E_2 | X A | **write** E | **read** E |
		if E **then** C_1 **else** C_2 **fi** | **while** E **do** C **od** |
		with D **do** C **end** | $\mathrm{C}_1 ; \mathrm{C}_2$ |
		new E | **new** $\mathrm{E}_1[\mathrm{E}_2]$
Prg	::=	C

A.2 Typing Rules

Identifiers

$$(\mathrm{I1}) \quad \pi \vdash \mathrm{I} : \iota\, \mathit{Idf}\,, \quad \text{if } (\mathrm{I} : \iota) \in \pi$$

Long Identifiers

$$(\mathrm{X1}) \ \frac{\pi \vdash \mathrm{I} : \tau\, \mathit{Idf}}{\pi \vdash \mathrm{I} : \tau\, \mathit{Long}} \qquad (\mathrm{X2}) \ \frac{\pi \vdash \mathrm{X} : (\pi_1 \langle, \pi_2 \rangle)\, \mathit{modLong} \qquad \langle \pi_2 : \rangle \pi_1 \vdash \mathrm{I} : \tau\, \mathit{Idf}}{\pi \vdash \mathrm{X} :: \mathrm{I} : \tau\, \mathit{Long}}$$

Type Expressions

$$(\mathrm{T1}) \ \frac{\pi \vdash \mathrm{X} : (\delta, \phi)\, \mathit{typeIdf}}{\pi \vdash \mathrm{X} : \delta\, \mathit{Type}}$$

$$\text{(T2)}\ \frac{\pi \vdash \mathrm{T} : \delta\,\mathit{Type}}{\pi \vdash \mathbf{Array}[\mathrm{T}] : \delta\,\mathit{arrType}} \qquad \text{(T3)}\ \frac{\pi \vdash \mathrm{T} : \delta\,\mathit{Type}}{\pi \vdash \mathbf{Pointer}[\mathrm{T}] : \delta\,\mathit{ptrType}}$$

$$\text{(T4)}\ \frac{\pi \vdash \mathrm{CL} : \lambda\,\mathit{Comp}}{\pi \vdash \mathbf{Record}[\mathrm{CL}] : \lambda\,\mathit{recType}}$$

Component Lists

$$\text{(CL1)}\ \frac{\pi \vdash \mathrm{T}_1 : \delta_1\,\mathit{Type} \quad \ldots \quad \pi \vdash \mathrm{T}_n : \delta_n\,\mathit{Type}}{\pi \vdash (\mathrm{I}_1 : \mathrm{T}_1, \ldots, \mathrm{I}_n : \mathrm{T}_n) : (\delta_1, \ldots, \delta_n)\,\mathit{Comp}}, \quad n \geq 1$$

Declarations

$$\text{(D1)}\ \frac{\pi \vdash \mathrm{IE} : \delta\,\mathit{Init} \quad \pi \vdash \mathrm{T} : \delta_1\,\mathit{Type} \quad \pi \vdash \delta \sim \delta_1}{\pi \vdash \mathbf{const}\ \mathrm{I} : \mathrm{T} = \mathrm{IE} : \{\mathrm{I} : \delta\}\,\mathit{Dec}}$$

$$\text{(D2)}\ \frac{\pi \vdash \mathrm{T} : \delta\,\mathit{Type}}{\pi \vdash \mathbf{var}\ \mathrm{I} : \mathrm{T} : \{\mathrm{I} : \delta\,\mathit{loc}\}\,\mathit{Dec}}, \quad \pi \vdash \mathit{is\_sized}(\delta)$$

$$\text{(D3)}\ \frac{\pi \vdash \mathrm{P} : \rho\,\mathit{Par} \quad \pi \vdash \mathrm{T} : \delta_1\,\mathit{Type} \quad \pi \uplus \pi_1 \vdash \mathrm{E} : \delta_2\,\mathit{Xpr} \quad \pi \vdash \delta_1 \succeq \delta_2}{\pi \vdash \mathbf{fun}\ \mathrm{I}\ \mathrm{P} : \mathrm{T} = \mathrm{E} : \{\mathrm{I} : \rho \to \delta\}\,\mathit{Dec}},$$

$$\pi_1 = \{\mathrm{I} : \rho \to \delta\} \uplus \mathit{type\_attrs}(\mathrm{P})$$

$$\text{(D4)}\ \frac{\pi \vdash \mathrm{P} : \rho\,\mathit{Par} \qquad \pi \uplus \pi_1 \vdash \mathrm{C} : \mathit{Cmd}}{\pi \vdash \mathbf{proc}\ \mathrm{I}\ \mathrm{P} = \mathrm{C} : \{\mathrm{I} : \rho \to \mathit{Cmd}\}\,\mathit{Dec}},$$

$$\pi_1 = \{\mathrm{I} : \rho \to \mathit{Cmd}\} \uplus \mathit{type\_attrs}(\mathrm{P})$$

$$\text{(D5)}\ \frac{\pi \vdash \mathrm{D}_1 : \pi_1\,\mathit{Dec} \qquad \pi \uplus \pi_1 \vdash \mathrm{D}_2 : \pi_2\,\mathit{Dec}}{\pi \vdash \mathrm{D}_1 ; \mathrm{D}_2 : \pi_1 \uplus \pi_2\,\mathit{Dec}}$$

$$\text{(D6)}\ \frac{\pi \vdash \mathrm{T} : \delta\,\mathit{Type}}{\pi \vdash \mathbf{type}\ \mathrm{I} = \mathrm{T} : \{\mathrm{I} : (\delta, \mathit{alias})\mathit{type}\}\,\mathit{Dec}}, \quad (\mathrm{I} : (\delta', \phi)\mathit{type}) \notin \pi$$

$$\text{(D7)}\ \pi \vdash \mathbf{newtype}\ \mathrm{I} : \{\mathrm{I} : (\mathrm{I}, \mathit{forward})\mathit{type}\}\,\mathit{Dec}, \quad (\mathrm{I} : (\delta', \phi)\mathit{type}) \notin \pi$$

$$\text{(D8)}\ \frac{\pi \uplus \{\mathrm{I} : (\mathrm{I}, \mathit{forward})\mathit{type}\} \vdash \mathrm{T} : \delta\,\mathit{Type}}{\pi \vdash \mathbf{newtype}\ \mathrm{I} = \mathrm{T} : \{\mathrm{I} : (\delta, \mathit{new})\mathit{type}\}\,\mathit{Dec}},$$

$$\text{if } (\mathrm{I} : (\delta, \phi)\mathit{type}) \in \pi, \text{ then } \phi = \mathit{forward}$$

$$\text{(D9)}\ \frac{\pi \vdash \mathrm{I} : (\delta, \phi)\mathit{typeIdf} \quad \pi \vdash \mathrm{P} : \rho\,\mathit{Par} \quad \pi \uplus \mathit{type\_attrs}(\rho) \uplus \{\mathit{this} : \delta\,\mathit{loc}\} \vdash \mathrm{C} : \mathit{Cmd}}{\pi \vdash \mathbf{construct}\ \mathrm{I}(\mathrm{P}) = \mathrm{C} : \{\mathrm{I} : (\rho \to \mathit{Cmd})\mathit{cons}\}\,\mathit{Dec}},$$

$$\pi \vdash \mathit{is\_sized}(\delta) \text{ and } \phi = \begin{cases} \mathit{alias} \\ \mathit{new} \end{cases}$$

$$\text{(D10)}\ \frac{\pi \vdash \mathrm{I}:(\delta,\phi)\,\mathit{typeIdf} \qquad \pi \uplus \{\mathit{this}:\delta\,\mathit{loc}\} \vdash \mathrm{C}:\mathit{Cmd}}{\pi \vdash \textbf{destruct}\ \mathrm{I} = \mathrm{C}:\{\mathrm{I}:\mathit{dst}\}\mathit{Dec}}\ , \quad \phi = \begin{cases} \mathit{alias} \\ \mathit{new} \end{cases}$$

$$\text{(D11)}\ \frac{\pi \vdash \mathrm{I}_2:(\rho \to \mathit{Cmd})\,\mathit{consIdf} \qquad \pi \vdash \mathrm{A}:\rho\,\mathit{Arg}}{\pi \vdash \textbf{var}\ \mathrm{I}_1:\mathrm{I}_2(\mathrm{A}):\{\mathrm{I}:\delta\,\mathit{loc}\}\mathit{Dec}}$$

$$\text{(D12)}\frac{\pi_{par} \vdash \mathrm{M}:\pi_1\mathit{Mod} \qquad \pi \vdash \mathrm{S}_2:\pi_2\mathit{Sig} \qquad \langle \pi \vdash \mathrm{S}_1:\pi_3\mathit{Sig}\rangle \qquad \langle \pi_1 \Rightarrow \pi_3\rangle}{\pi \vdash \textbf{functor}\ \mathrm{I}_1[\mathrm{I}_2:\mathrm{S}_2]\langle\textbf{:}\mathrm{S}_1\rangle = \mathrm{M}:\{\mathrm{I}_1:(\pi_1,\pi_2\langle,\pi_3\rangle)\mathit{func}\}\mathit{Dec}},$$

$$\pi_{par} = \mathit{sigs}(\pi) \uplus \{\mathrm{I}_2:\mathit{sized}(\pi_2)\}$$

$$\text{(D13)}\ \frac{\pi \vdash \mathrm{M}:\pi_1\mathit{Mod} \qquad \langle \pi \vdash \mathrm{S}:\pi_2\mathit{Sig}\rangle \qquad \langle \pi_2 \Rightarrow \pi_1\rangle}{\pi \vdash \textbf{module}\ \mathrm{I}\langle\textbf{:}\mathrm{S}\rangle = \mathrm{M}:\{\mathrm{I}:(\pi_1\langle,\pi_2\rangle)\mathit{mod}\}\mathit{Dec}}$$

$$\text{(D14)}\ \frac{\pi \vdash \mathrm{X}:(\pi_1\langle,\pi_2\rangle)\,\mathit{modLong}}{\pi \vdash \textbf{import}\ \mathrm{X}:\langle \pi_2 :\rangle \pi_1\mathit{Dec}}$$

$$\text{(D15)}\ \frac{\pi \vdash \mathrm{S}:\pi_1\mathit{Sig}}{\pi \vdash \textbf{signature}\ \mathrm{I} = \mathrm{S}:\{\mathrm{I}:\pi_1\mathit{sig}\}\mathit{Dec}}$$

Initializing Expressions

$$\text{(IE1)}\ \frac{\pi \vdash \mathrm{E}:\delta\,\mathit{Xpr}}{\pi \vdash \mathrm{E}:\delta\,\mathit{Init}}$$

$$\text{(IE2)}\ \frac{\pi \vdash \mathrm{A}:\rho\,\mathit{Arg} \qquad \pi \vdash \rho \sim (\delta,\ldots,\delta)}{\pi \vdash \textbf{Array}[\mathrm{A}]:\delta\,\mathit{arrInit}}$$

$$\text{(IE3)}\ \frac{\pi \vdash \mathrm{A}:\rho\,\mathit{Arg} \qquad \pi \vdash \rho \sim (\delta_1,\ldots,\delta_n)}{\pi \vdash \textbf{Record}[\mathrm{A}]:(\mathrm{I}_1:\delta_1,\ldots,\mathrm{I}_n:\delta_n)\mathit{recInit}}$$

Module Body

$$\text{(M1)}\ \frac{\pi \vdash \mathrm{D}:\pi_1\mathit{Dec} \qquad \pi_1 \vdash \mathrm{C}_1:\mathit{Cmd} \qquad \pi_1 \vdash \mathrm{C}_2:\mathit{Cmd}}{\pi \vdash \textbf{mod}\ \mathrm{D}\ \textbf{init}\ \mathrm{C}_1\ \textbf{term}\ \mathrm{C}_2\ \textbf{end}:\pi_1\mathit{Mod}}$$

$$\text{(M2)}\ \frac{\pi \vdash \mathrm{X}_1:(\pi_1,\pi_2\langle,\pi_4\rangle)\mathit{funcLong} \qquad \pi \vdash \mathrm{X}_2:(\pi_3\langle,\pi_5\rangle)\mathit{modLong} \qquad \langle \pi_5 :\rangle \pi_3 \Rightarrow \pi_2}{\pi \vdash \mathrm{X}_1[\mathrm{X}_2]:\langle \pi_4 + \rangle \pi_1\mathit{Mod}}$$

Signatures

$$\text{(S1)}\ \frac{\pi \vdash \mathrm{X} : \pi_1 \mathit{sigIdf}}{\pi \vdash \mathrm{X} : \pi_1 \mathit{Sig}}$$

$$\text{(S2)}\ \frac{\mathit{sigs}(\pi) \vdash \mathrm{SD} : \pi_1 \mathit{SDec}}{\pi \vdash \textbf{sig}\ \mathrm{SD}\ \textbf{end} : \pi_1 \mathit{Sig}}$$

Signature Declarations

$$\text{(SD1)}\ \frac{\pi \vdash \mathrm{T} : \delta\, \mathit{Type}}{\pi \vdash \textbf{const}\ \mathrm{I} : \mathrm{T} : \{\mathrm{I} : \delta\} \mathit{SDec}}$$

$$\text{(SD2)}\ \frac{\pi \vdash \mathrm{T} : \delta\, \mathit{Type}}{\pi \vdash \textbf{var}\ \mathrm{I} : \mathrm{T} : \{\mathrm{I} : \delta\, \mathit{loc}\} \mathit{SDec}}$$

$$\text{(SD3)}\ \frac{\pi \vdash \mathrm{P} : \rho \mathit{Par} \qquad \pi \vdash \mathrm{T} : \delta\, \mathit{Type}}{\pi \vdash \textbf{fun}\ \mathrm{I}\ \mathrm{P} : \mathrm{T} : \{\mathrm{I} : \rho \to \delta\} \mathit{SDec}}$$

$$\text{(SD4)}\ \frac{\pi \vdash \mathrm{P} : \rho \mathit{Par}}{\pi \vdash \textbf{proc}\ \mathrm{I}\ \mathrm{P} : \{\mathrm{I} : \rho \to \mathit{Cmd}\} \mathit{SDec}}$$

$$\text{(SD5)}\ \pi \vdash \textbf{type}\ \mathrm{I} : \{\mathrm{I} : (\mathrm{I}, \mathit{hidden}) \mathit{type}\} \mathit{SDec}$$

$$\text{(SD6)}\ \frac{\pi \vdash \mathrm{I} : (\delta, \phi) \mathit{typeIdf} \qquad \pi \vdash \mathrm{P} : \rho \mathit{Par}}{\pi \vdash \textbf{constructor}\ \mathrm{I}\ \mathrm{P} : \{\mathrm{I} : (\rho \to \mathit{Cmd}) \mathit{cons}\} \mathit{SDec}}$$

$$\text{(SD7)}\ \frac{\pi \vdash \mathrm{I} : (\delta, \phi) \mathit{typeIdf}}{\pi \vdash \textbf{destructor}\ \mathrm{I} : \{\mathrm{I} : \mathit{dst}\} \mathit{SDec}}$$

$$\text{(SD8)}\ \frac{\pi \vdash \mathrm{S} : \pi_1 \mathit{Sig}}{\pi \vdash \textbf{module}\ \mathrm{I} : \mathrm{S} : \{\mathrm{I} : \pi_1\} \mathit{SDec}}$$

$$\text{(SD9)}\ \frac{\pi \vdash \mathrm{X}_1 : (\mathrm{I}_1, \mathit{hidden}) \mathit{typeLong} \qquad \pi \vdash \mathrm{X}_2 : (\mathrm{I}_2, \mathit{hidden}) \mathit{typeLong}}{\pi \vdash \textbf{share type}\ \mathrm{X}_1 \boldsymbol{=} \mathrm{X}_2 : \{\mathit{share}(\mathrm{X}_1, \mathrm{X}_2)\} \mathit{SDec}}$$

$$\text{(SD10)}\ \frac{\pi \vdash \mathrm{S} : \pi_1 \mathit{Sig}}{\pi \vdash \textbf{include}\ \mathrm{X} : \pi_1 \mathit{SDec}}$$

$$\text{(SD11)}\ \frac{\pi \vdash \mathrm{SD}_1 : \pi_1 \mathit{SDec} \qquad \pi \vdash \mathrm{SD}_2 : \pi_2 \mathit{SDec}}{\pi \vdash \mathrm{SD}_1 \boldsymbol{;} \mathrm{SD}_2 : \pi_1 \uplus \pi_2 \mathit{SDec}}$$

Parameters

$$\text{(P1)}\ \frac{\pi \vdash \mathrm{T}_1 : \delta_1\, \mathit{Type} \qquad \ldots \qquad \pi \vdash \mathrm{T}_n : \delta_n\, \mathit{Type}}{\pi \vdash (\langle \textbf{ref} \rangle \mathrm{I}_1 : \mathrm{T}_1, \ldots, \langle\langle \textbf{ref} \rangle\rangle \mathrm{I}_n : \mathrm{T}_n) : (\delta_1 \langle \mathit{loc} \rangle, \ldots, \delta_n \langle\langle \mathit{loc} \rangle\rangle) \mathit{Par}}, \quad n \geq 0$$

Arguments

$$\text{(A1)}\ \frac{\pi \vdash \mathrm{E}_1 : \tau_1 Xpr \quad \ldots \quad \pi \vdash \mathrm{E}_n : \tau_n Xpr}{\pi \vdash (\mathrm{E}_1, \ldots, \mathrm{E}_n) : (\tau_1, \ldots, \tau_n) Arg}\ , n \geq 0$$

Expressions

$$\text{(E1)}\ \frac{\pi \vdash \mathrm{X} : \tau Long}{\pi \vdash \mathrm{X} : \tau Xpr}$$

$$\text{(E2)}\ \frac{\pi \vdash \mathrm{E} : \delta loc Xpr}{\pi \vdash \mathrm{E} : \delta Xpr}$$

$$\text{(E3)}\ \pi \vdash v : \gamma Xpr\ , \quad v \in V_\gamma$$

$$\text{(E4)}\ \pi \vdash \mathbf{eof} : bool Xpr$$

$$\text{(E5)}\ \frac{\pi \vdash \mathrm{X} : \rho \to \delta Long \qquad \pi \vdash \mathrm{A} : \rho Arg}{\pi \vdash \mathrm{X}\ \mathrm{A} : \delta Xpr}$$

$$\text{(E6)}\ \frac{\pi \vdash \mathrm{E} : \gamma_1 Xpr}{\pi \vdash *_\sigma\ \mathrm{E} : \gamma Xpr}\ , \quad \sigma = \gamma_1 \to \gamma$$

$$\text{(E7)}\ \frac{\pi \vdash \mathrm{E}_1 : \gamma_1 Xpr \qquad \pi \vdash \mathrm{E}_2 : \gamma_2 Xpr}{\pi \vdash \mathrm{E}_1\ \circ_\sigma\ \mathrm{E}_2 : \gamma Xpr}\ , \quad \sigma = \gamma_1 \times \gamma_2 \to \gamma$$

$$\text{(E8)}\ \frac{\pi \vdash \mathrm{T} : \delta_1\, Type \qquad \pi \vdash \delta_1 \succeq \_ptr}{\pi \vdash \mathrm{T}'\mathbf{nil} : \delta_1 Xpr}$$

$$\text{(E9)}\ \frac{\pi \vdash \mathrm{E} : \delta_1 loc Xpr \qquad \pi \vdash \delta_1 \succeq \delta_2 ptr}{\pi \vdash \mathbf{!}\mathrm{E} : \delta_2 loc Xpr}$$

$$\text{(E10)}\ \frac{\pi \vdash \mathrm{E}_1 : \delta_1 \langle loc \rangle Xpr \qquad \pi \vdash \mathrm{E}_2 : int Xpr \qquad \pi \vdash \delta_1 \succeq \delta_2 arr}{\pi \vdash \mathrm{E}_1[\mathrm{E}_2] : \delta_2 \langle loc \rangle Xpr}$$

$$\text{(E11)}\ \frac{\pi \vdash \mathrm{E} : \delta_1 \langle loc \rangle Xpr \qquad \pi \vdash \delta_1 \succeq \lambda rec \qquad \lambda = (\ldots, \mathrm{I} : \delta_2, \ldots)}{\pi \vdash \mathrm{E}.\mathrm{I} : \delta_2 \langle loc \rangle Xpr}$$

Commands

$$\text{(C1)}\ \ \pi \vdash \mathbf{skip} : \mathit{Cmd} \qquad \text{(C2)}\ \ \frac{\pi \vdash \mathrm{E}_1 : \delta\,\mathit{locXpr} \qquad \pi \vdash \mathrm{E}_2 : \delta\,\mathit{Xpr}}{\pi \vdash \mathrm{E}_1 \mathbf{:=} \mathrm{E}_2 : \mathit{Cmd}}$$

$$\text{(C3)}\ \ \frac{\pi \vdash \mathrm{X} : (\rho \to \mathit{Cmd})\mathit{Long} \qquad \pi \vdash \mathrm{A} : \rho\,\mathit{Arg}}{\pi \vdash \mathrm{X}\ \mathrm{A} : \mathit{Cmd}}$$

$$\text{(C4)}\ \ \frac{\pi \vdash \mathrm{E} : \gamma\,\mathit{Xpr}}{\pi \vdash \mathbf{write}\ \mathrm{E} : \mathit{Cmd}} \qquad \text{(C5)}\ \ \frac{\pi \vdash \mathrm{E} : \gamma\,\mathit{locXpr}}{\pi \vdash \mathbf{read}\ \mathrm{E} : \mathit{Cmd}}$$

$$\text{(C6)}\ \ \frac{\pi \vdash \mathrm{E} : \mathit{boolXpr} \qquad \pi \vdash \mathrm{C}_1 : \mathit{Cmd} \qquad \pi \vdash \mathrm{C}_2 : \mathit{Cmd}}{\pi \vdash \mathbf{if}\ \mathrm{E}\ \mathbf{then}\ \mathrm{C}_1\ \mathbf{else}\ \mathrm{C}_2\ \mathbf{fi} : \mathit{Cmd}}$$

$$\text{(C7)}\ \ \frac{\pi \vdash \mathrm{E} : \mathit{boolXpr} \qquad \pi \vdash \mathrm{C} : \mathit{Cmd}}{\pi \vdash \mathbf{while}\ \mathrm{E}\ \mathbf{do}\ \mathrm{C}\ \mathbf{od} : \mathit{Cmd}}$$

$$\text{(C8)}\ \ \frac{\pi \vdash \mathrm{D} : \pi_1 \mathit{Dec} \qquad \pi \uplus \pi_1 \vdash \mathrm{C} : \mathit{Cmd}}{\pi \vdash \mathbf{with}\ \mathrm{D}\ \mathbf{do}\ \mathrm{C}\ \mathbf{end} : \mathit{Cmd}}$$

$$\text{(C9)}\ \ \frac{\pi \vdash \mathrm{C}_1 : \mathit{Cmd} \qquad \pi \vdash \mathrm{C}_2 : \mathit{Cmd}}{\pi \vdash \mathrm{C}_1 ; \mathrm{C}_2 : \mathit{Cmd}}$$

$$\text{(C10)}\ \ \frac{\pi \vdash \mathrm{E} : \delta_1 \mathit{locXpr} \qquad \pi \vdash \delta_1 \succeq \delta_2 \mathit{ptr}}{\pi \vdash \mathbf{new}\ \mathrm{E} : \mathit{Cmd}}, \quad \pi \vdash \mathit{is\_sized}(\delta_2)$$

$$\text{(C11)}\ \ \frac{\pi \vdash \mathrm{E}_1 : \delta_1 \mathit{locXpr} \qquad \pi \vdash \delta_1 \succeq \delta_2 \mathit{arr} \qquad \pi \vdash \mathrm{E}_2 : \mathit{intXpr}}{\pi \vdash \mathbf{new}\ \mathrm{E}_1[\mathrm{E}_2] : \mathit{Cmd}},$$
$$\pi \vdash \mathit{is\_sized}(\delta_2)$$

Programs

$$\text{(Prg1)}\ \ \frac{\{\} \vdash \mathrm{C} : \mathit{Cmd}}{\mathrm{C} : \mathit{Prg}}$$

A.3 Semantic Rules

Identifiers

$[\![\pi \vdash . : \tau\,\mathit{Idf}]\!] : \mathit{CompInfo} \to \mathit{IdValue}$

$[\![\pi \vdash \mathrm{I} : \iota\,\mathit{Idf}]\!]c = v$, if $(\mathrm{I} = v) \in c.e$

Long Identifiers

$[\![\pi \vdash . : \tau\,\mathit{Long}]\!] : \mathit{CompInfo} \to \mathit{IdValue}$

$[\![\pi \vdash \mathrm{I} : \tau\,\mathit{Long}]\!]c = [\![\pi \vdash \mathrm{I} : \tau\,\mathit{Idf}]\!]c$
$[\![\pi \vdash \mathrm{X} :: \mathrm{I} : \tau\,\mathit{Long}]\!]c = [\![\langle \pi_2 :\rangle \pi_1 \vdash \mathrm{I} : \tau\,\mathit{Idf}]\!]c_1$,
where $c_1 = [\![\pi \vdash \mathrm{X} : (\pi_1 \langle, \pi_2\rangle)\,\mathit{modLong}\,]\!]c$

Declarations

$[\![\pi \vdash . : \pi_1 Dec]\!] : CompInfo \rightarrow RunInfo \rightarrow$
$\quad (CompInfo \times (RunInfo \rightarrow RunInfo) \times RunInfo)$

$[\![\pi \vdash \textbf{const}\ \mathrm{I}:\mathrm{T} = \mathrm{IE} : \{\mathrm{I}:\delta\} Dec]\!] c\ r = (\{\mathrm{I} = v\}, id, r_1),$
where $(v, r_1) = [\![\pi \vdash \mathrm{IE} : \delta Init]\!] c\ r$
$[\![\pi \vdash \textbf{var}\ \mathrm{I}:\mathrm{T} : \{\mathrm{I}:\delta loc\} Dec]\!] c\ r = (\{\mathrm{I} = l\}, d, r|_s[s_1]),$
where $(l, s_1) = allocate_{c,\delta}(r.s)$
and $d\ r' = r'|_s[free_\delta(l, r.s)]$
$[\![\pi \vdash \textbf{fun}\ \mathrm{I\ P}:\mathrm{T}{=}\mathrm{E} : \{\mathrm{I}:\rho \rightarrow \delta\} Dec]\!] c\ r = (\{\mathrm{I} = f\}, id, r),$
where $f\ a' = [\![\pi \uplus \pi_1 \vdash \mathrm{E} : \delta_2 Xpr]\!] (c \uplus \{\mathrm{I} = f\} \uplus ([\![\pi \vdash \mathrm{P} : \rho Par]\!] a'))$
$[\![\pi \vdash \textbf{proc}\ \mathrm{I\ P}{=}\mathrm{C} : \{\mathrm{I}:\rho \rightarrow Cmd\} Dec]\!] c\ r = (\{\mathrm{I} = f\}, id, r),$
where $f\ a' = [\![\pi \uplus \pi_1 \vdash \mathrm{C} : Cmd]\!] (c \uplus \{\mathrm{I} = f\} \uplus ([\![\pi \vdash \mathrm{P} : \rho Par]\!] a'))$
$[\![\pi \vdash \mathrm{D}_1; \mathrm{D}_2 : \pi_1 \uplus \pi_2 Dec]\!] c\ r = (c_1 \uplus c_2, d_1 \circ d_2, r_2),$
where $(c_1, d_1, r_1) = [\![\pi \vdash \mathrm{D}_1 : \pi_1 Dec]\!] c\ r$
and $(c_2, d_2, r_2) = [\![\pi \uplus \pi_1 \vdash \mathrm{D}_2 : \pi_2 Dec]\!] c \uplus c_1\ r_1$

$[\![\pi \vdash \textbf{newtype}\ \mathrm{I} : \{\mathrm{I} : (\mathrm{I}, forward) type\} Dec]\!] c\ r = (\{\}, id, r)$
$[\![\pi \vdash \textbf{type}\ \mathrm{I} = \mathrm{T} : \{\mathrm{I} : (\delta, alias) type\} Dec]\!] c\ r = (\{\mathrm{I} = \delta\}, id, r)$
$[\![\pi \vdash \textbf{newtype}\ \mathrm{I} = \mathrm{T} : \{\mathrm{I} : (\delta, new) type\} Dec]\!] c\ r = (\{\mathrm{I} = \delta\}, id, r)$
$[\![\pi \vdash \textbf{construct}\ \mathrm{I(P)} = \mathrm{C} : \{\mathrm{I} : (\rho \rightarrow Cmd) cons\} Dec]\!] c\ r =$
$\quad (\{\mathrm{I} = f cons\}, id, r),$
where $f(a', t', r') = [\![\pi \uplus type\_attrs(\rho) \uplus \{this : \delta loc\} \vdash \mathrm{C} : Cmd]\!] g(a', t')\ r',$
where $g(a', t') = c \uplus ([\![\pi \vdash \mathrm{P} : \rho Par]\!] a') \uplus \{this = t'\}$
$[\![\pi \vdash \textbf{destruct}\ \mathrm{I} = \mathrm{C} : \{\mathrm{I} : dst\} Dec]\!] c\ r = (\{\mathrm{I} = f dst\}, id, r),$
where $f\ t\ r' = [\![\pi \uplus type\_attrs(\rho) \uplus \{this : \delta loc\} \vdash \mathrm{C} : Cmd]\!] (c \uplus \{this = t\}) r'$
$[\![\pi \vdash \textbf{var}\ \mathrm{I}_1 : \mathrm{I}_2(\mathrm{A}) : \{\mathrm{I} : \delta loc\} Dec]\!] c\ r = (\{\mathrm{I} = l\}, (f_2\ l) \circ d, r_2),$
where $(l, r_1) = allocate_{c,\delta}(r.s)$
and $r_2 = f_1([\![\pi \vdash \mathrm{A} : \rho Arg]\!] c\ r, l, r_1),$
where $f_1 = [\![\pi \vdash \mathrm{I}_2 : (\rho \rightarrow Cmd) cons Idf]\!]$
and $(\mathrm{I} = f_2 dst) \in c$ or $f_2 = id$
and $d\ r' = r'|_s[free_\delta(l, r.s)]$

$[\![\pi \vdash \textbf{functor}\ \mathrm{I}_1[\mathrm{I}_2 : \mathrm{S}_2]\langle{:}\mathrm{S}_1\rangle = \mathrm{M} : \{\mathrm{I}_1 : (\pi_1, \pi_2\langle, \pi_3\rangle) func\} Dec]\!] c\ r = (\{\mathrm{I}_1 = f\}, id, r),$
where $f\ c' = [\![\pi_{par} \vdash \mathrm{M} : \pi_1 Mod]\!] \{\mathrm{I}_2 = c'\}$
$[\![\pi \vdash \textbf{module}\ \mathrm{I}\langle{:}\mathrm{S}\rangle = \mathrm{M} : \{\mathrm{I} : (\pi_1\langle, \pi_2\rangle) mod\} Dec]\!] c\ r = (\{\mathrm{I} = c_1\}, d \circ d_1, r_2),$
where $(dec, init, term) = [\![\pi \vdash \mathrm{M} : \pi_1 Mod]\!] c$
and $(c_1, d_1, r_1) = dec\ r$
and $r_2 = init\ c_1\ r_1$
and $d = term\ c_1$
$[\![\pi \vdash \textbf{import}\ \mathrm{X} : \langle \pi_2 : \rangle \pi_1 Dec]\!] c\ r = (c_1, id, r),$
where $c_1 = [\![\pi \vdash \mathrm{X} : (\pi_1\langle, \pi_2\rangle) mod Long]\!]$

Initializing Expressions

$[\![\pi \vdash . : \delta Init]\!] : CompInfo \rightarrow RunInfo \rightarrow (Value \times RunInfo)$

$[\![\pi \vdash \mathrm{E} : \delta Init]\!] c\ r = ([\![\pi \vdash \mathrm{E} : \delta Xpr]\!] c\ r, r)$

$[\![\pi \vdash \mathbf{Array}[\mathrm{A}] : \delta\, arrInit]\!] c\ r = (loc_i, r|_s[s_n])$,
  where $(a_1, \ldots, a_k) = [\![\pi \vdash \mathrm{A} : \rho Arg]\!]$
  and $n = size(c, \delta)$
  and $(loc_i, s_0) = allocate(n * k, r.s)$
  and $s_{j+1} = update_\delta(loc_{i+j*d}, a_{j+1}, s_j), 0 \leq j < k$
$[\![\pi \vdash \mathbf{Record}[\mathrm{A}] : (\mathrm{I}_1 : \delta_1, \ldots, \mathrm{I}_n : \delta_n) recInit]\!] c\ r = (a, r)$,
  where $a = [\![\pi \vdash \mathrm{A} : \rho Arg]\!]$

Module Body

$$[\![\pi \vdash . : \pi_1 Mod]\!] : CompInfo \to ((RunInfo \to (CompInfo \times RunInfo)) \times (CompInfo \to RunInfo \to RunInfo) \times (CompInfo \to RunInfo \to RunInfo))$$

$[\![\pi \vdash \mathbf{mod}\ \mathrm{D}\ \mathbf{init}\ \mathrm{C}_1\ \mathbf{term}\ \mathrm{C}_2\ \mathbf{end} : \pi_1 Mod]\!] c = (dec, init, term)$,
  where $dec = [\![\pi \vdash \mathrm{D} : \pi_1 Dec]\!] c$
  and $init = [\![\pi_1 \vdash \mathrm{C}_1 : Cmd]\!]$
  and $term = [\![\pi_1 \vdash \mathrm{C}_2 : Cmd]\!]$
$[\![\pi \vdash \mathrm{X}_1[\mathrm{X}_2] : \langle \pi_4 + \rangle \pi_1 Mod]\!] c = f\ c_1$,
  where $f = [\![\pi \vdash \mathrm{X}_1 : (\pi_1, \pi_2 \langle, \pi_4 \rangle) funcLong]\!] c$
  and $c_1 = [\![\pi \vdash \mathrm{X}_2 : (\pi_3 \langle, \pi_5 \rangle) modLong]\!] c$

Parameters

$$[\![\pi \vdash . : \rho Par]\!] : Value^* \to CompInfo$$

$[\![\pi \vdash (\langle \mathbf{ref} \rangle \mathrm{I}_1 : \mathrm{T}_1, \ldots, \langle \langle \mathbf{ref} \rangle \rangle \mathrm{I}_n : \mathrm{T}_n) : (\delta_1 \langle loc \rangle, \ldots, \delta_n \langle \langle loc \rangle \rangle) Par]\!] a =$
  $\{\mathrm{I}_1 = v_1\} \cup \cdots \cup \{\mathrm{I}_n = v_n\}$,
  where $a = (v_1, \ldots, v_n), \quad n \geq 0$

Arguments

$$[\![\pi \vdash . : \rho Arg]\!] : CompInfo \to RunInfo \to Value^*$$

$[\![\pi \vdash (\mathrm{E}_1, \ldots, \mathrm{E}_n) : (\tau_1, \ldots, \tau_n) Arg]\!] c\ r =$
  $([\![\pi \vdash \mathrm{E}_1 : \tau_1 Xpr]\!] c\ r, \ldots, [\![\pi \vdash \mathrm{E}_n : \tau_n Xpr]\!] c\ r)$

Expressions

$$[\![\pi \vdash . : \tau Xpr]\!] : CompInfo \to RunInfo \to Value$$

$[\![\pi \vdash \mathrm{X} : \tau Xpr]\!] c\ r = [\![\pi \vdash \mathrm{X} : \tau Long]\!] c$
$[\![\pi \vdash \mathrm{E} : \delta Xpr]\!] c\ r = lookup([\![\pi \vdash \mathrm{E} : \delta locLong]\!] c, s)$
$[\![\pi \vdash v : \gamma Xpr]\!] c\ r = v$,
  if $v \in V_\gamma$
$[\![\pi \vdash \mathbf{eof} : boolXpr]\!] c\ r =$
- $true$, if $r.i = [\,]$,
- $false$, otherwise

$[\![\pi \vdash \mathrm{X}\,\mathrm{A} : \delta Xpr]\!] c\ r = f\ a\ r$,
 where $f = [\![\pi \vdash \mathrm{X} : \rho \rightarrow \delta Long]\!] c$
 and $a = [\![\pi \vdash \mathrm{A} : \rho Arg]\!] c\ r$
$[\![\pi \vdash *_\sigma\ \mathrm{E} : \gamma Xpr]\!] c\ r = *_\sigma([\![\pi \vdash \mathrm{E} : \gamma_1 Xpr]\!] c\ r)$,
 if $\sigma = \gamma_1 \rightarrow \gamma$
$[\![\pi \vdash \mathrm{E}_1\ \circ_\sigma\ \mathrm{E}_2 : \gamma Xpr]\!] c\ r = ([\![\pi \vdash \mathrm{E}_1 : \gamma_1 Xpr]\!] c\ r) \circ_\sigma ([\![\pi \vdash \mathrm{E}_2 : \gamma_2 Xpr]\!] c\ r)$,
 if $\sigma = \gamma_1 \times \gamma_2 \rightarrow \gamma$

$[\![\pi \vdash \mathrm{E} : \delta Xpr]\!] c\ r = lookup_{c,\delta}([\![\pi \vdash \mathrm{E} : \delta locXpr]\!] c\ r, r)$ $[\![\pi \vdash \mathrm{T}'\mathbf{nil} : \delta_1 Xpr]\!] c\ r = loc_0$
$[\![\pi \vdash \mathbf{!}\mathrm{E} : \delta_2 locXpr]\!] c\ r = lookup_{c,\delta_1}([\![\pi \vdash \mathrm{E} : \delta_1 locXpr]\!] c\ r, r)$
$[\![\pi \vdash \mathrm{E}_1[\mathrm{E}_2] : \delta_2 locXpr]\!] c\ r = loc_{i+os}$,
 where $loc_i = [\![\pi \vdash \mathrm{E}_1 : \delta_1 Xpr]\!] c\ r$
 and $o = [\![\pi \vdash \mathrm{E}_2 : intXpr]\!] c\ r$
 and $s = size(c, \delta_2)$
$[\![\pi \vdash \mathrm{E.I} : \delta_2 Xpr]\!] c\ r = p_i$,
 where $(p_1, \ldots, p_n) = [\![\pi \vdash \mathrm{E} : \delta_1 Xpr]\!] c\ r$
 and $i = index(\mathrm{I}, \delta_1)$
$[\![\pi \vdash \mathrm{E.I} : \delta_2 locXpr]\!] c\ r = loc_{i+o}$,
 where $loc_i = [\![\pi \vdash \mathrm{E} : \delta_1 locXpr]\!] c\ r$
 and $o = rec\_off(c, \mathrm{I}, \delta_1)$

Commands

$[\![\pi \vdash . : Cmd]\!] : CompInfo \rightarrow RunInfo \rightarrow RunInfo$
$[\![\pi \vdash . : Cmd]\!] c\ r = r$, if $r.s = \bot$ or $r.e \neq ok$

$[\![\pi \vdash \mathbf{skip} : Cmd]\!] c\ r = r$
$[\![\pi \vdash \mathrm{E}_1 \mathbf{:=} \mathrm{E}_2 : Cmd]\!] c\ r =$
  $r|_s[update_{c,\delta}([\![\pi \vdash \mathrm{E}_1 : \delta locXpr]\!] c\ r, [\![\pi \vdash \mathrm{E}_2 : \delta Xpr]\!] c\ r, r.s)]$
$[\![\pi \vdash \mathrm{X}\,\mathrm{A} : Cmd]\!] c\ r = f\ a\ r$,
 where $f = [\![\pi \vdash \mathrm{X} : (\rho \rightarrow Cmd) Long]\!] c$
 and $a = [\![\pi \vdash \mathrm{A} : \rho Arg]\!] c\ r$
$[\![\pi \vdash \mathbf{write}\ \mathrm{E} : Cmd]\!] c\ r = r|_o[write(r.o, [\![\pi \vdash \mathrm{E} : \gamma Xpr]\!] c\ r)]$
$[\![\pi \vdash \mathbf{read}\ \mathrm{E} : Cmd]\!] c\ r =$
- $r|_e[\diamond_1]$,
  if $r.i = ()$
- $(r|_i[i_1])|_s[update_{c,\gamma}(l, v_1, r.s)]$,
  if $(i_1, v) = read(r.i)$ and $v = \mathrm{in}\, V_\gamma(v_1)$,
- $r|_e[\diamond_2]$, otherwise,
  where $l = [\![\pi \vdash \mathrm{E} : \gamma locXpr]\!] c$

$[\![\pi \vdash \mathbf{if}\ \mathrm{E}\ \mathbf{then}\ \mathrm{C}_1\ \mathbf{else}\ \mathrm{C}_2\ \mathbf{fi} : Cmd]\!] c\ r =$
 $if([\![\pi \vdash \mathrm{E} : boolXpr]\!] c\ r, [\![\pi \vdash \mathrm{C}_1 : Cmd]\!] c\ r, [\![\pi \vdash \mathrm{C}_2 : Cmd]\!] c\ r)$
$[\![\pi \vdash \mathbf{while}\ \mathrm{E}\ \mathbf{do}\ \mathrm{C}\ \mathbf{od} : Cmd]\!] c\ r = w\ r$
 where $w : RunInfo \rightarrow RunInfo$
  $w\ r' = if([\![\pi \vdash \mathrm{E} : boolXpr]\!] c\ r', w([\![\pi \vdash \mathrm{C} : Cmd]\!] c\ r'), r')$
$[\![\pi \vdash \mathbf{with}\ \mathrm{D}\ \mathbf{do}\ \mathrm{C}\ \mathbf{end} : Cmd]\!] c\ r = r_3$,
 where $(c_1, d, r_1) = [\![\pi \vdash \mathrm{D} : \pi_1 Dec]\!] c\ r$
 and $r_2 = [\![\pi \uplus \pi_1 \vdash \mathrm{C} : Cmd]\!](c \uplus c_1)\ r_1$
 and $r_3 = d\ r_2$

$[\![\pi \vdash \mathrm{C}_1; \mathrm{C}_2 : \mathit{Cmd}]\!] c\ r = [\![\pi \vdash \mathrm{C}_2 : \mathit{Cmd}]\!] c\ r_1,$
    where $r_1 = [\![\pi \vdash \mathrm{C}_1 : \mathit{Cmd}]\!] c\ r$

$[\![\pi \vdash \mathbf{new}\ \mathrm{E} : \mathit{Cmd}]\!] c\ r = r|_s[\mathit{update}(l, v, s)],$
    where $l = [\![\pi \vdash \mathrm{E} : \delta_1 \mathit{locXpr}]\!] c\ r$
    and $(v, r_1) = \mathit{allocate}_{c,\delta_1}(r.s)$
$[\![\pi \vdash \mathbf{new}\ \mathrm{E}_1[\mathrm{E}_2] : \mathit{Cmd}]\!] c\ r = r|_s[\mathit{update}(l, v, s)],$
    where $l = [\![\pi \vdash \mathrm{E}_1 : \delta_1 \mathit{locXpr}]\!] c\ r$
    and $(v, s) = \mathit{allocate}(\mathit{size}(c, \delta_1) * ([\![\pi \vdash \mathrm{E}_2 : \mathit{intXpr}]\!] c\ r), r.s)$

Programs

$[\![\pi \vdash . : \mathit{Prg}]\!] : \mathit{Value}^* \to (\mathit{Value}^* \times \mathit{Exception})^\perp$

$[\![\mathrm{C} : \mathit{Prg}]\!] i =$
- $\perp$, if $r.s = \perp$,
- $(r.o, r.e)$, otherwise,

    where $r = [\![\{\} \vdash \mathrm{C} : \mathit{Cmd}]\!] \{\}\ (i, [\,], \langle\,\rangle, ok)$