A Methodology for the Definition of Programming Languages

NIKOLAOS S. PAPASPYROU
Department of Electrical and Computer Engineering
National Technical University of Athens
Polytechnioupoli, 15780 Zografou, Athens
GREECE
E-mail: nickie@softlab.ntua.gr

Abstract: The formal definition of a programming language is a valuable tool for the study, design, evaluation and even for the implementation of the language. A methodology for developing formal definitions of programming languages is proposed in this paper. We follow the denotational approach and use monads, in order to improve the modularity and elegance of the result. The definition of semantics is divided in three distinct consecutive phases: static, typing and dynamic semantics, each using the results of the previous two. Emphasis has been given on the straightforward translation of the developed definitions into abstract interpreters, in order to obtain verifiably correct prototype implementations for the studied languages. The methodology is described through a complete example, defining a small imperative programming language. Its use for the definition of ANSI C has resulted in considerable success.

Key-Words: Programming languages, formal definition, denotational semantics, monads.

1 Introduction

The study of programming languages invariably distinguishes between two fundamental features: *syntax* and *semantics*. Although the two cannot always be clearly separated, syntax refers to the appearance and structure of the well-formed sentences of the language, whereas semantics refers to the meanings of these sentences. Every person who uses a programming language must understand both its syntax and semantics at some level of abstraction. Programmers usually understand a programming language by means of examples, intuition and descriptions in natural language. Such descriptions are informal, typically based on a set of assumptions about the reader's knowledge and understanding, and therefore inherently ambiguous.

In the definition of programming languages, the syntax usually formally specified. On the other hand, semantics is most commonly specified in an informal way. Research in the area of formal programming language semantics started in the 1960s. Since then, the product of more than three decades of research has been the development and thorough study of numerous methods and formalisms, usually classified for historical reasons as following one of three main approaches: *operational*, *denotational* or *axiomatic* semantics.

Denotational semantics is a formalism introduced by Scott and Strachey in the late 1960s [1, 2, 3]. According to the denotational approach, semantics is described by attributing mathematical *denotations* to programs and program segments. Denotations are elements of appropriate mathematical structures called *domains*. Variations of the λ-calculus over domains are commonly used as metalanguages for expressing denotations.

The main objective of our research is to develop and evaluate a methodology for the formal definition of programming languages, with the main effort lying on the definition of semantics. Our methodology is described in this paper through a simple, yet not trivial example of a programming language called $\mathcal{L}$. Three distinct consecutive phases for the definition of semantics are proposed, each phase using the results of the previous ones. The methodology has been used with considerable success for the definition of ANSI C [4].

The rest of the paper is structured as follows. In Section 2, an overview of the proposed methodology

is given and the various stages in the formal definition of a programming language are briefly presented. Section 3 contains a short introduction to monads and their correspondence with computations. In the following sections 4 through 7 the various stages of the proposed methodology are presented in detail, using a small imperative programming language called $\mathcal{L}$ as an example. Section 8 discusses the implementation of programming language definitions obtained by the proposed methodology, in the form of prototypical interpreters. Section 9 attempts an evaluation of the proposed methodology, discussing the results of an ambitious project concerning the definition of ANSI C. Finally, Section 10 concludes with a few remarks and directions for future research.

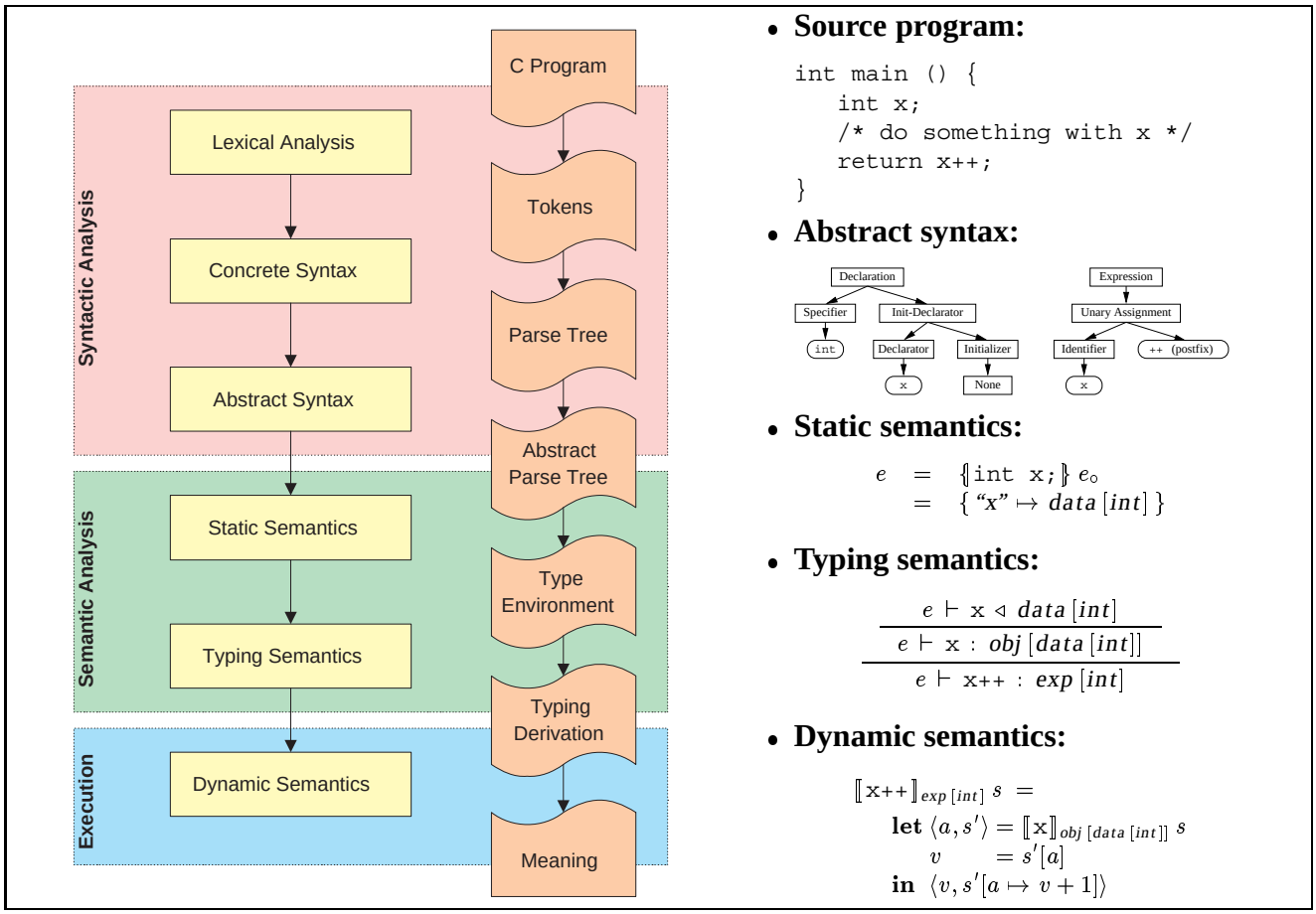
2 Overview of the methodology

According to the proposed methodology, the formal definition of a given programming language $\mathcal{L}$ can be best understood as part of an *abstract interpreter* for programs written in $\mathcal{L}$. Such an interpreter is depicted in Fig. 1 as a data process. The left part of the figure is a module diagram of the interpreter. It shows the chain of actions performed by the interpreter as well as the data that is processed by these actions. Each action uses the results of the previous actions. The initial piece of data is a *source program* and the final result is a representation of this program's *meaning*, i.e. a description of what the program does when executed. The right part of the figure gives a small example illustrating the function of the interpreter, where the source language is taken to be C.

The interpreter consists of the following three layers, represented in Fig. 1 as big boxes. Each layer contains a series of actions.

- *Syntactic analysis*: aims at checking the syntactic validity of the source program and signalling syntax errors. Syntactically correct programs are transformed to abstract parse trees, which represent their syntactic structure in detail. This layer consists of three actions:
  - *Lexical analysis*: transforms the source program to a sequence of lexical units, also called *tokens*.
  - *Concrete syntax*: checks the syntactic validity of the source program by grouping tokens together in more complex syntax entities.
  - *Abstract syntax*: simplifies the parse tree by extracting superfluous information and results in an abstract parse tree.
- *Semantic analysis*: aims at checking the semantic validity of the program, as represented by the abstract parse tree, and signalling semantic errors, e.g. use of undeclared identifiers or type mismatches. It contains two actions:
  - *Static semantics*: aims at detecting static semantic errors, e.g. redefinition of an identifier in the same scope, as well as associating identifiers with appropriate types or values. Static semantics is based solely on the abstract syntax. For each syntactically well-formed program phrase P, its static semantic meaning is denoted by $\{\!|P|\!\}$. Static semantic meanings are mathematical objects, typically type environments associating identifiers to types.
  - *Typing Semantics*: aims at detecting type mismatch errors, e.g. assignment to a constant value, and at associating syntactically well-formed phrases with appropriate phrase types. Such associations are given by means of typing derivations, that is, formal proofs that phrases are well-typed. Typing semantics is based not only on the abstract syntax of the program, but also on static semantic environments.
- *Execution*: the main part of the interpreter. It contains only one action:
  - *Dynamic Semantics*: aims at defining the execution behaviour of well-typed programs. As a useful side effect, run-time errors and other sources of undefined behaviour are detected at the same time. Dynamic semantics is based on the abstract syntax, the static semantic environments and the typing derivations. For each well-typed program phrase P of type θ, its dynamic semantic meaning is denoted by $[\![P]\!]_\theta$. Such meanings are also mathematical objects, typically functions describing aspects of the execution of the corresponding program phrases. The typing derivation for P is important, as the same phrase is allowed to have different meanings when attributed different types.

Fig. 1: An abstract interpreter for $\mathcal{L}$.



3 Computations and monads

The concept of monads comes from category theory. Monads have been proposed by Moggi as a useful structuring tool for denotational semantics [5, 6]. Moggi demonstrated their use for representing different aspects of computations and defined monads for programming language features such as state, exceptions and continuations. In a short time, the idea of monads became very popular in the functional programming community as a way of structuring functional programs and simulating non-functional features. The work of Wadler [7] played a very important role in this direction.

In the last few years, research related to the application of monads in denotational semantics has focused on the combination of monads to structure semantic interpreters. Monad transformers, which were also first proposed by Moggi, have attracted the attention of many researchers. In the work of Liang, Hudak and Jones [8], monad transformers are demonstrated to successfully modularize semantic interpreters and the lifting of monad operations is investigated.

The notion of computation is very significant in the semantics of programming languages. If D is a domain of values, then $M(D)$ can be taken to denote a domain of computations which return values from D. In this sense, M can be thought of as a domain constructor which specifies the characteristics of computations, e.g. whether they are deterministic, whether they require access to the program state, or whether they can cause run-time errors. This is the principal concept behind the use of monads in denotational semantics. In brief, a monad is a triple $\langle \mathsf{M}, unit_{\mathsf{M}}, *_{\mathsf{M}} \rangle$, where M is a domain constructor, $unit_{\mathsf{M}} : A \to \mathsf{M}(A)$ and $\cdot *_{\mathsf{M}} \cdot : \mathsf{M}(A) \times (A \to \mathsf{M}(B)) \to \mathsf{M}(B)$ are polymorphic functions for arbitrary domains A and B, satisfying three monad laws.

The definition of M reflects our notion of computa-

tion. The computation denoted by $unit_{\mathsf{M}}\ v$ is usually the simplest computation returning the value v. If m is a computation returning a value v in D, then the computation denoted by $m *_{\mathsf{M}} f$ is the combined computation of m followed by computation $f\ v$. Many different types of computations are implicit in the semantics of programs and a number of monads is required to represent them. Typical examples of such computations and related monads are:

- *Error monad*: Computations allowing errors.
- *Value monad*: Constant computations.
- *State monad*: Computations accessing the state
- *Continuation monad*: Computations accessing the state and allowing unrestricted jumps.
- *Powerdomain monad*: Non-deterministic computations.

4 Syntax

The formal definition of a programming language begins with the definition of its syntax. There are various standard formalisms for this purpose, the most widely known and used being context-free grammars, usually expressed in the Backus Naur Form (BNF) and its variations. This formalism is used in the proposed methodology. As a convention, non-terminal symbols are written in italics. The symbol ::= separates the two sides of production rules, whereas the symbol | separates alternative productions. Finally, the symbol ϵ denotes the empty string.

Since the area of formal syntax specification has been thoroughly studied and is not so interesting from a researcher's point of view, our definition of the syntax of the example language $\mathcal{L}$ will be slightly informal. We will only focus on the abstract syntax of $\mathcal{L}$, which is given in Fig. 2. The intended semantics of $\mathcal{L}$ can be easily deduced from the abstract syntax. The language supports variables of just one data type (integer), functions and procedures to which parameters can be passed by value or by reference, a variety of statements including a loop statement and a variety of expression operators.

5 Static semantics

Static semantics can be thought of as the symbol table in our abstract interpreter. It calculates the environments containing type information for all identifiers defined in the source program and, for this reason, it mainly deals with the program's declarations. At the same time static semantic errors are detected. The definition of static semantics can be very complicated in the case of programming languages with rich type systems. It consists of two steps. At the first step, the static semantic domains must be defined, together with the basic operations that are allowed on the elements of each domain. For this reason, a careful analysis of the type system of $\mathcal{L}$ is necessary.

Computations performed during this step are compile-time computations and can typically be represented by a simple error monad E, defined below. An additional operation on the monad, used for the generation of compile-time errors is given and it is easy to prove that errors are correctly propagated through the use of the standard monad operations.

$$\blacksquare\ \mathsf{E}(D) = D \oplus \mathbf{U}$$

$$\blacktriangleright\ unit_{\mathsf{E}} : D \to \mathsf{E}(D)$$
$$unit_{\mathsf{E}} = inl$$

$$\blacktriangleright\ \cdot *_{\mathsf{E}} \cdot : \mathsf{E}(A) \times (A \to \mathsf{E}(B)) \to \mathsf{E}(B)$$
$$m *_{\mathsf{E}} f = \mathbf{case}\ m\ \mathbf{of}$$
$$inl\ v \Rightarrow f\ v$$
$$inr\ u \Rightarrow inr\ u$$

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

From the abstract syntax of $\mathcal{L}$, it is apparent that at least two classes of types exist. A *data type* is a basic type of the language: one can assign values of such a type to variables, pass them as parameters to functions, use them in expressions, etc. The only data type supported by $\mathcal{L}$ is $integer$. On the other hand, a *return-type* is a type that can be returned from a function. Apart from the $integer$ data type, the empty type $nothing$ can also be returned from a function.

At a closer look, one can identify more classes of types. Another important class in the study of programming languages is the class of *denotable types*, that is, types that can be denoted by identifiers but whose values cannot always be assigned to variables or passed to functions as parameters. Functions in $\mathcal{L}$, are typical such values. They can only be used in expressions and statements in a limited way, i.e. in function calls. Identifiers in $\mathcal{L}$ can denote three kinds of things: data objects (variables or parameters passed by-value), reference objects (parameters passed by-reference) and function names.

Fig. 2: Abstract syntax of $\mathcal{L}$.

program ::= *body*
body ::= *definition-list statement-list*
definition-list ::= ϵ | *definition definition-list*
definition ::= `var` *I* `as` *data-type* | `fun` *I* `(` *formal-parameter-list* `)` `as` *return-type body* `end`
data-type ::= `integer`
return-type ::= *data-type* | `nothing`
formal-parameter-list ::= ϵ | *formal-parameter formal-parameter-list*
formal-parameter ::= *I* `as` *data-type* `byval` | *I* `as` *data-type* `byref`
statement-list ::= ϵ | *statement statement-list*
statement ::= `skip` | *I* `:=` *expression* | `begin` *statement-list* `end` | `if` *condition* `then` *statement* `else` *statement*
| `while` *condition* `do` *statement* | `call` *I* `(` *expression-list* `)` | `return` | `return` *expression*
expression ::= *n* | *I* | `call` *I* `(` *expression-list* `)` | *un-op expression* | *expression bin-op expression*
condition ::= `true` | `false` | `not` *condition* | *expression rel-op expression* | *condition log-op condition*
expression-list ::= ϵ | *expression expression-list*
un-op ::= `+` | `-`
bin-op ::= `+` | `-` | `*` | `/` | `mod`
rel-op ::= `=` | `<>` | `>` | `<` | `>=` | `<=`
log-op ::= `and` | `or`

Expression types form yet another type class. Apart from data types, expressions can also be of *boolean* type which can only be used in conditions. Finally, the class of *phrase types* contains one type for each kind of program phrase. All the type classes that were mentioned are defined below in the form of static semantic domains. The values of the domain for phrase types are further explained in Fig. 3.

- $I : \mathbf{Ide}$ *(domain of identifiers)*
- $\tau : \mathbf{Type}_{dat} = integer$
- $r : \mathbf{Type}_{ret} = \tau \mid nothing$
- $\phi : \mathbf{Type}_{den} = data[\tau] \mid ref[\tau] \mid func[r,p]$
- $v : \mathbf{Type}_{exp} = \tau \mid boolean$
- $\theta : \mathbf{Type}_{phr} = prog \mid body[r] \mid stmt[r] \mid decl \mid exp[v] \mid obj[\phi] \mid arg[p] \mid prot[p] \mid par[\tau, m]$
- $m : \mathbf{Mode} = byval \mid byref$

As a particular case of static domains, *static environments* are mappings from identifiers to appropriate static type domains. In the case of $\mathcal{L}$, static type environments are elements of domain $\mathbf{Env}$ and map identifiers of a program scope to denotable types. The domain $\mathbf{Env}$ and the required operations on its elements are shown below as an abstract data type. Its implementation is slightly perplexed because of nested scopes and is not included in this paper.

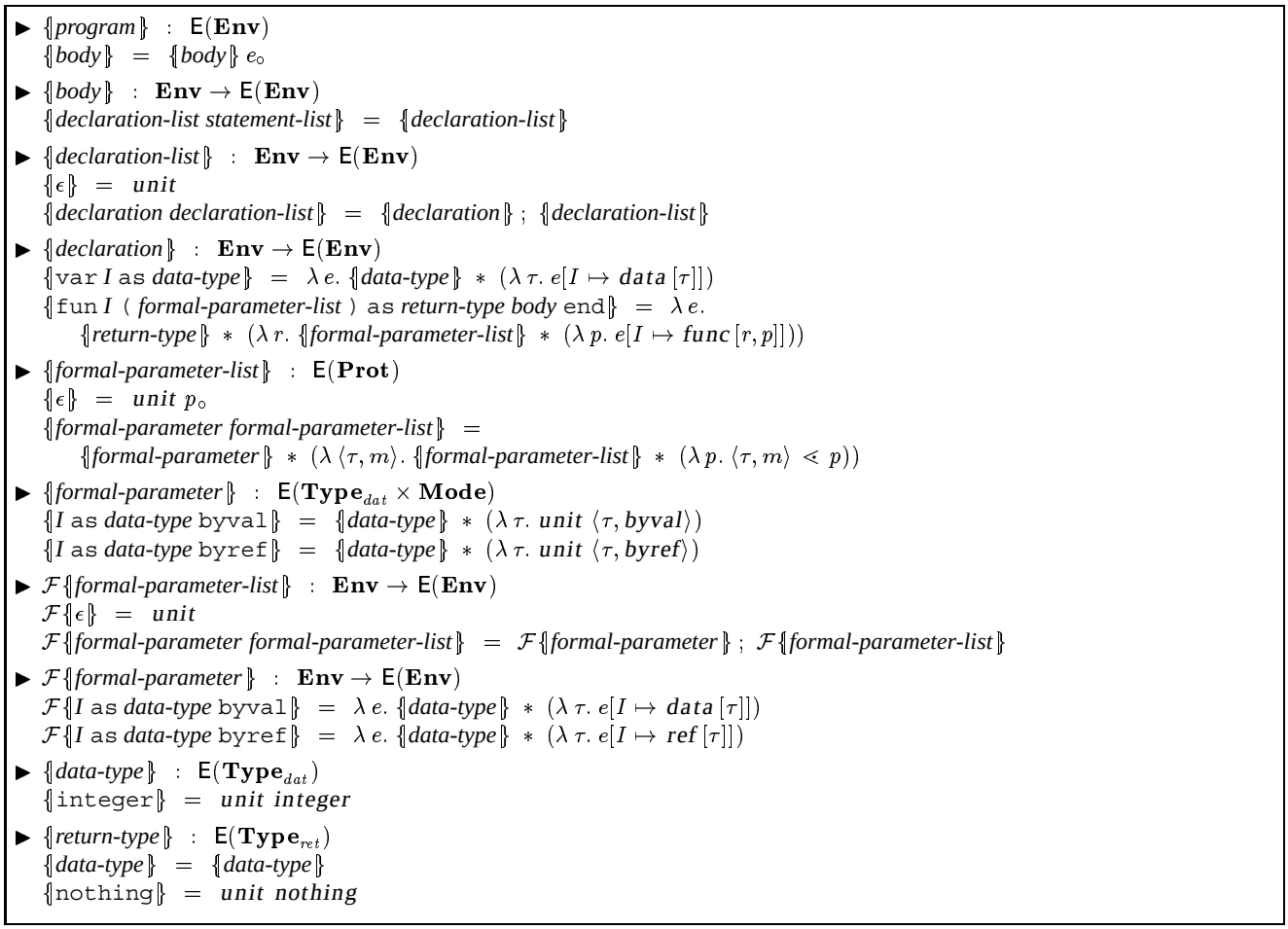
Fig. 3: Static semantics, domain of phrase types.

Type	**Description**
$prog$	Complete program
$body[r]$	Body of function returning a result of type r
$stmt[\tau]$	Statement in a function returning a result of type r.
$decl$	Declaration.
$exp[v]$	Expression, whose result is an r-value of type v.
$obj[\phi]$	Expression, whose result is an l-value of type ϕ.
$arg[p]$	Actual parameters to a function with prototype p.
$prot[p]$	Formal parameters to a function with prototype p.
$par[\tau, m]$	Formal parameter of type τ and passing mode m.

- $e : \mathbf{Env}$ *(domain of type environments)*
- ▶ $e_\circ : \mathbf{Env}$ *(empty environment)*
- ▶ $\cdot[\cdot] : \mathbf{Env} \times \mathbf{Ide} \to \mathsf{E}(\mathbf{Type}_{den})$ *(lookup function)*
- ▶ $\cdot[\cdot \mapsto \cdot] : \mathbf{Env} \times \mathbf{Ide} \times \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Env})$ *(update function)*
- ▶ $\uparrow \cdot : \mathbf{Env} \to \mathbf{Env}$ *(new enclosed scope)*
- ▶ $\downarrow \cdot : \mathbf{Env} \to \mathbf{Env}$ *(enclosing scope)*

A second kind of static environment is also shown below as an abstract data type. It represents function

Fig. 4: Static semantics for $\mathcal{L}$, functions and equations.



prototypes, i.e. mappings from the formal parameters of a function to their type and passing mode. The order of formal parameters is important and this is why they are represented by integer numbers instead of names.

■ p : $\mathbf{Prot}$ *(domain of function prototypes)*

▶ $p_\circ$: $\mathbf{Prot}$ *(empty prototype)*

▶ $\cdot[\cdot]$: $\mathbf{Prot} \times \mathbf{N} \to \mathsf{E}(\mathbf{Type}_{dat} \times \mathbf{Mode})$ *(lookup function)*

▶ $\cdot \lessdot \cdot$: $(\mathbf{Type}_{dat} \times \mathbf{Mode}) \times \mathbf{Prot} \to \mathsf{E}(\mathbf{Prot})$ *(prepend parameter)*

▶ $\Downarrow \cdot$: $\mathbf{Prot} \to \mathsf{E}(\mathbf{Type}_{dat} \times \mathbf{Mode} \times \mathbf{Prot})$ *(extract 1st parameter)*

At the second step, for each non-terminal symbol involved in the static semantics, a *semantic function* must be defined. This function represents the static meaning of phrases generated by this non-terminal symbol and its definition is given by induction on the set of production rules. For the case of $\mathcal{L}$, the complete set of static semantic functions is defined in Fig. 4. There is one equation for each production rule. The type of each function reflects the static meaning of the corresponding program phrase, e.g. the meaning of declarations is typically a function from some initial type environment to a compile-time computation of an updated type environment, which contains the newly declared identifiers.

In some cases, program phrases may have more than one static meaning, depending on their use. For example, formal parameters determine the prototype of a defined function, but also modified the static type environment when this function is called. The multiple meanings are distinguished by prefixed calligraphic letters. Furthermore, most programming languages support recursively defined types, requiring the use of the least fixed point operator in their static semantics.

6 Typing semantics

The primary aim of typing semantics is the association of program phrases with phrase types. The relation between program phrases and phrase types is formally established by means of *typing judgements*. The most common form of judgement is the main typing relation $e \vdash \mathit{phrase} : \theta$. However, in specifying the typing semantics of a complex programming language, additional forms of typing judgements may be necessary. Fig. 5 shows the forms of judgements that are needed for the typing semantics of $\mathcal{L}$.

Fig. 5: Typing semantics, judgements.

Judgement	**Description**
$e \vdash \mathit{phrase} : \theta$	The given *phrase* can be attributed phrase type θ in type environment e.
$e \vdash I \triangleleft \delta$	Identifier I is associated with the denotable type ϕ in type environment e.
$v := z$	The compile-time computation $z : \mathsf{E}(D)$ produces the (non-error) value $v : D$.

Typing rules are inference rules whose premises and conclusion are typing judgements. The proof of a typing judgement using a set of typing rules is called *typing derivation*. A typing derivation combines a number of typing rules in a tree-like structure, in such a way that the conclusion of one rule becomes a premise of some other rule.

A large subset of the rules that define the typing semantics of $\mathcal{L}$ is given in Fig. 6. Rules I1 and I2 specify conversions that take place implicitly in the evaluation of expressions, stating that l-values are implicitly converted to the values stored in the designated objects. Such conversions are called *implicit coercions*.

7 Dynamic semantics

Dynamic semantics specifies the execution behaviour of well-typed programs. As a useful side-effect, run-time errors and other sources of undefined behaviour are detected. As in the case of static semantics, dynamic semantics is specified in two steps. The first step aims at the definition of dynamic semantic domains, whereas the second aims at the definition of dynamic semantic functions.

The definition of dynamic semantic domains begins with a mapping from static types, i.e. elements of static type domains, to dynamic domains representing the values of these types. In the case of $\mathcal{L}$, this mapping is done below.

- $[\![\mathit{integer}]\!]_{dat} = \mathbf{N}$
- $[\![\tau]\!]_{ret} = [\![\tau]\!]_{dat}$
- $[\![\mathit{nothing}]\!]_{ret} = \mathbf{U}$
- $[\![\mathit{data}\,[\tau]]\!]_{den} = \mathbf{Obj}$
- $[\![\mathit{ref}\,[\tau]]\!]_{den} = \mathbf{Obj}$
- $[\![\mathit{func}\,[r,p]]\!]_{den} = [\![p]\!]_{Prot} \to \mathsf{G}([\![r]\!]_{ret})$
- $[\![\tau]\!]_{exp} = [\![\tau]\!]_{dat}$
- $[\![\mathit{boolean}]\!]_{exp} = \mathbf{T}$

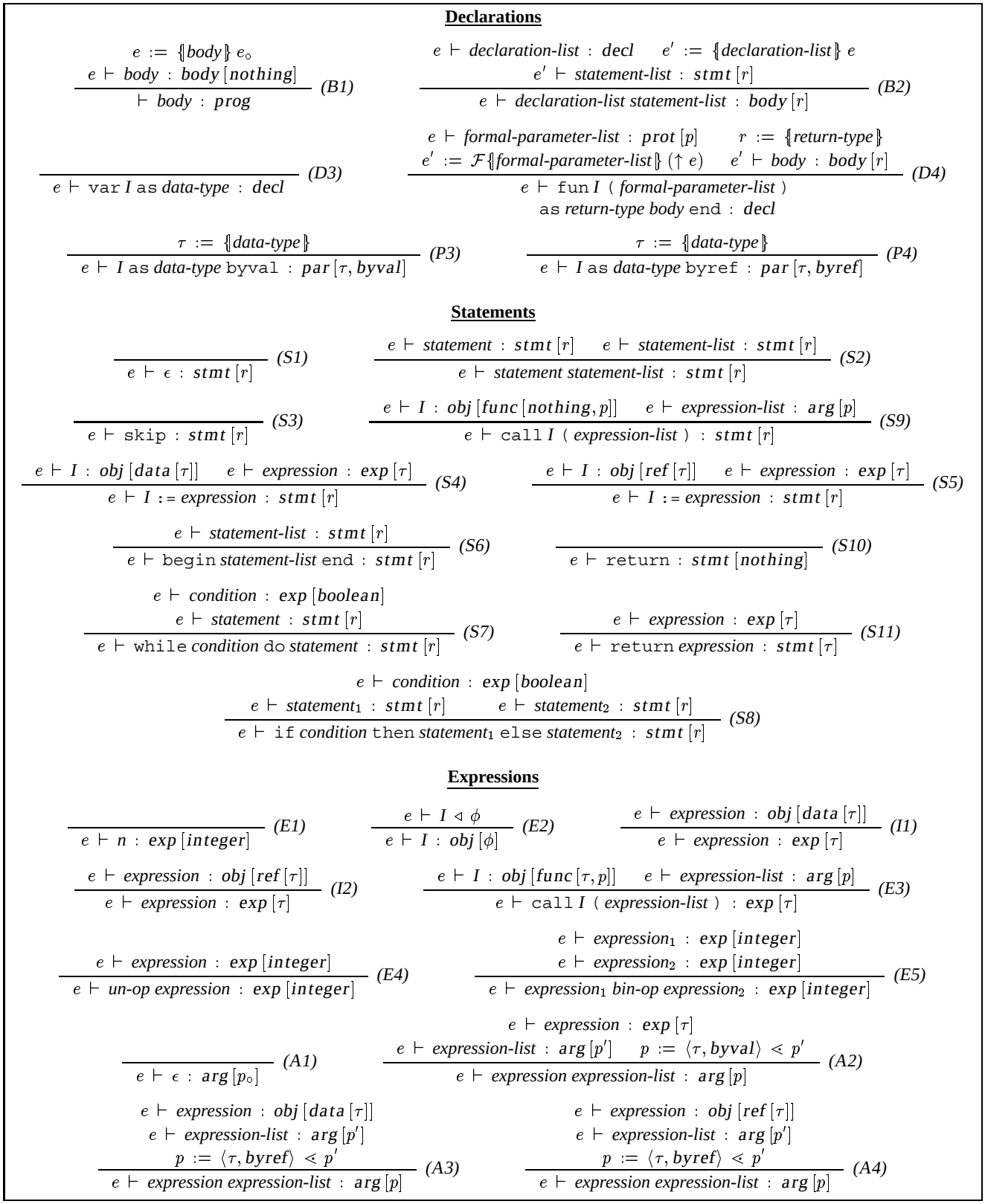
A few additional domains are necessary, such as the domain $\mathbf{Obj}$ of object descriptors, i.e. object locations in memory, $\mathbf{S}$ of program states and the domain $\mathbf{C}$ of program continuations. Complete definitions are omitted for brevity.

- $o : \mathbf{Obj}$ *(domain of object descriptors)*
- $s : \mathbf{S}$ *(domain of program states)*
- ▶ $s_\circ : \mathbf{S}$ *(initial program state)*
- ▶ $\mathit{write} : \mathbf{Obj} \to \mathbf{N} \to \mathbf{S} \to \mathbf{S}$
- ▶ $\mathit{read} : \mathbf{Obj} \to \mathbf{S} \to \mathbf{N}$
- $\mathbf{A}$ *(domain of program answers)*
- $\mathbf{C} = \mathbf{S} \to \mathbf{A}$ *(domain of continuations)*
- ▶ $\mathit{error}_C : \mathbf{C}$

The domain ordering relation in dynamic semantic domains is usually very important. Bottom elements can model non-termination and top elements, if they exist, can model the occurrence of run-time errors. Intermediate values represent results of computations which produce at least some non-erroneous results. Non-termination and errors must be appropriately propagated by various operations of the dynamic domains and by monad operations.

The dynamic semantics of programming languages can be greatly simplified by proper use of monads and monad transformers. A monad must be defined for each kind of computation that is inherent in the given programming language. In the case of $\mathcal{L}$, there are two kinds of computations that may be expressed by monads, in addition to monad E representing compile-time (i.e. constant) computations. *Expression computations* are represented by monad G, the standard continuation monad; they may affect the program state, i.e. read and

Fig. 6: Typing semantics, rules.



alter the values of variables. However, they may not terminate the execution of the program or the current function.

- $\mathsf{G}(D) \;=\; (D \to \mathbf{C}) \to \mathbf{C}$
- $unit_{\mathsf{G}} \;:\; D \to \mathsf{G}(D)$
  $unit_{\mathsf{G}} \;=\; \lambda\, v.\, \lambda\, \kappa.\, \kappa\; v$
- $\cdot\; *_{\mathsf{G}}\; \cdot \;:\; \mathsf{G}(A) \times (A \to \mathsf{G}(B)) \to \mathsf{G}(B)$
  $m \;*_{\mathsf{G}}\; f \;=\; \lambda\, \kappa.\, m(\lambda\, v.\, f\; v\; \kappa)$
- $error_{\mathsf{G}} \;:\; \mathsf{G}(D)$
  $error_{\mathsf{G}} \;=\; \lambda\, \kappa.\, error_{\mathsf{C}}$
- $lift_{\mathsf{E}\to\mathsf{G}} \;:\; \mathsf{E}(D) \to \mathsf{G}(D)$
  $lift_{\mathsf{E}\to\mathsf{G}} \;=\; \lambda\, m.\, \mathbf{case}\; m\; \mathbf{of}$
  $\quad inl\; v \;\Rightarrow\; unit_{\mathsf{G}}\; v$
  $\quad inr\; u \;\Rightarrow\; error_{\mathsf{G}}$
- $setState \;:\; (\mathbf{S} \to \mathbf{S}) \to \mathsf{G}(\mathbf{S})$
  $setState \;=\; \lambda\, f.\, \lambda\, \kappa.\, \lambda\, s.\, \kappa\; s\; (f\; s)$
- $getState \;:\; \mathsf{G}(\mathbf{S})$
  $getState \;=\; setState\; id$

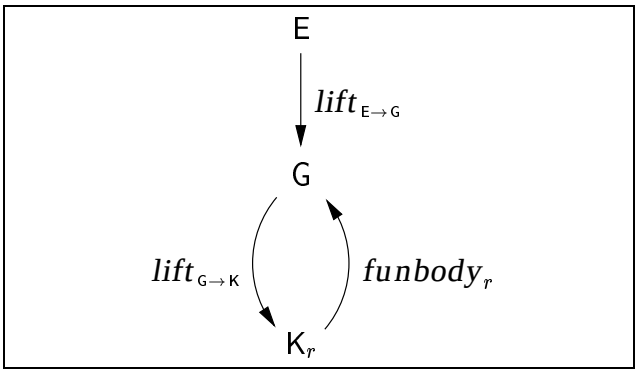
On the other hand, *statement* computations are represented by monad K_r; they may affect the program state but may also terminate the program or the current function by returning a result of type r.

- $\mathsf{K}_r(D) \;=\; (\llbracket r \rrbracket_{ret} \to \mathbf{C}) \to \mathsf{G}(D)$
- $unit_{\mathsf{K}} \;:\; D \to \mathsf{K}_r(D)$
  $unit_{\mathsf{K}} \;=\; \lambda\, v.\, \lambda\, \kappa_r.\; unit_{\mathsf{G}}\; v$
- $\cdot\; *_{\mathsf{K}}\; \cdot \;:\; \mathsf{K}_r(A) \times (A \to \mathsf{K}_r(B)) \to \mathsf{K}_r(B)$
  $m \;*_{\mathsf{K}}\; f \;=\; \lambda\, \kappa_r.\; m\; \kappa_r \;*_{\mathsf{G}}\; (\lambda\, v.\, f\; v\; \kappa_r)$
- $error_{\mathsf{K}} \;:\; \mathsf{K}_r(D)$
  $error_{\mathsf{K}} \;=\; \lambda\, \kappa_r.\; error_{\mathsf{G}}$
- $lift_{\mathsf{G}\to\mathsf{K}} \;:\; \mathsf{G}(D) \to \mathsf{K}_r(D)$
  $lift_{\mathsf{G}\to\mathsf{K}} \;=\; \lambda\, m.\, \lambda\, \kappa_r.\; m$
- $funbody_r \;:\; \mathsf{K}_r(\mathbf{U}) \to \mathsf{G}(\llbracket r \rrbracket_{ret})$
  $funbody_{nothing} \;=\; \lambda\, k.\, \lambda\, \kappa.\; k\; \kappa\; \kappa$
  $funbody_{\tau} \;=\; \lambda\, k.\, \lambda\, \kappa.\; k\; \kappa\; (\lambda\, u.\; error_{\mathsf{C}})$
- $result_r \;:\; \llbracket r \rrbracket_{ret} \to \mathsf{K}_r(\mathbf{U})$
  $result_r \;=\; \lambda\, d.\, \lambda\, \kappa_r.\, \lambda\, \kappa.\; \kappa_r\; d$

The relation between the three monads and the conversion functions are shown in Fig. 7.

Dynamic environments are a special case of dynamic semantic domains. They correspond directly to static environments and associate static program elements, such as identifiers or formal parameters, with the dynamic entities that these elements represent during the program's execution. In the case of $\mathcal{L}$, two kinds of dynamic environments are needed: the domain $\llbracket e \rrbracket_{Env}$ of dynamic type environments corresponding to the static type environment e, and the domain $\llbracket p \rrbracket_{Prot}$ of dynamic function prototypes corresponding to the static function prototype p. They are

Fig. 7: Relations between the defined monads.



both defined below as abstract data types. In brief, the first associates variables with objects in memory, whereas the second associates formal with actual parameters.[1]

- $\rho \;:\; \llbracket e \rrbracket_{Env}$ *(domain of dynamic type environments)*
- $\llbracket e \Downarrow I \rrbracket \;=\; \begin{cases} \llbracket \phi \rrbracket_{den} & , \text{if } e[I] = unit_{\mathsf{E}}\; \phi \\ \mathbf{U} & , \text{if } e[I] = error_{\mathsf{E}} \end{cases}$
- $\rho_{\circ} \;:\; \llbracket e_{\circ} \rrbracket_{Env}$ *(empty dynamic environment)*
- $\cdot \uparrow \cdot \;:\; e : \mathbf{Env} \ltimes \llbracket \downarrow e \rrbracket_{Env} \to \llbracket e \rrbracket_{Env}$
- $lookup \;:\; e : \mathbf{Env} \twoheadrightarrow \llbracket e \rrbracket_{Env} \to \mathbf{Ide} \to \mathsf{E}(\llbracket e \Downarrow I \rrbracket)$
- $create \;:\; e : \mathbf{Env} \twoheadrightarrow \llbracket e \rrbracket_{Env} \to \mathbf{Ide} \to \mathsf{E}(\llbracket e \rrbracket_{Env})$
- $assign \;:\; e : \mathbf{Env} \twoheadrightarrow \llbracket e \rrbracket_{Env} \to$
  $\qquad I : \mathbf{Ide} \twoheadrightarrow \llbracket e \Downarrow I \rrbracket \to \mathsf{E}(\llbracket e \rrbracket_{Env})$

- $\llbracket p \rrbracket_{Prot}$ *(domain of dynamic function prototypes)*
- $\llbracket p \Downarrow i \rrbracket \;=\; \begin{cases} \llbracket \tau \rrbracket_{dat} & , \text{if } p[i] = unit_{\mathsf{E}}\; \langle \tau, byval \rangle \\ \llbracket ref\,[\tau] \rrbracket_{den} & , \text{if } p[i] = unit_{\mathsf{E}}\; \langle \tau, byref \rangle \\ \mathbf{U} & , \text{if } p[i] = error_{\mathsf{E}} \end{cases}$
- $\llbracket \tau, byval \rrbracket_{Par} \;=\; \llbracket \tau \rrbracket_{dat}$
- $\llbracket \tau, byref \rrbracket_{Par} \;=\; \mathbf{Obj}$
- $noargs \;:\; \llbracket p_{\circ} \rrbracket_{Prot}$
- $prepend_{p,\tau,m,p'} \;:\; \llbracket \tau, m \rrbracket_{Par} \to \llbracket p' \rrbracket_{Prot} \to \mathsf{E}(\llbracket p \rrbracket_{Prot})$
- $split_{p,\tau,m,p'} \;:\; \llbracket p \rrbracket_{Prot} \to \mathsf{E}(\llbracket \tau, m \rrbracket_{Par} \times \llbracket p' \rrbracket_{Prot})$

During the second step in the definition of dynamic semantics, dynamic functions and equations must be defined. In general, a dynamic semantic function maps well-typed program phrases to dynamic semantic domains. A program phrase is well-typed if there exists a typing derivation for it, and the conclusion of this typing derivation specifies the phrase type. At least one

[1] In the notation that we use, the domains $x : A \twoheadrightarrow B(x)$ and $x : A \ltimes B(x)$ are respectively dependent function and dependent product domains.

Fig. 8: Dynamic semantics for $\mathcal{L}$, functions and equations for declarations.

▶ $[\![prog]\!] \ : \ \mathsf{K}_{nothing}(\mathbf{U})$

(B1) $[\![body]\!]_{prog} \ = \ lift_{\mathsf{E}\to\mathsf{K}} \ (\{\!| body |\!\}\ e_\circ) \ * \ (\lambda\, e.\ [\![body]\!]_{body\ [r]}\ e\ (e \uparrow \rho_\circ))$

▶ $[\![body\ [r]]\!] \ : \ e : \mathbf{Env} \twoheadrightarrow [\![e]\!]_{Env} \to \mathsf{K}_r(\mathbf{U})$

(B2) $[\![$ *declaration-list statement-list* $]\!]_{body\ [r]} \ = \ \lambda\, e.\ \lambda\, \rho.$
lift$_{\mathsf{E}\to\mathsf{K}}$ ($\{\!|$*declaration-list*$|\!\}$ e) $*$ ($\lambda\, e'$.
lift$_{\mathsf{E}\to\mathsf{K}}$ (*mclo* ($[\![$*declaration-list*$]\!]_{decl}$ e' ρ) ($\mathcal{F}[\![$*declaration-list*$]\!]_{decl}$ e')) $*$ ($\lambda\, \rho'$.
$[\![$*statement-list*$]\!]_{stmt\ [r]}$ e' ρ'))

▶ $[\![decl]\!] \ : \ e : \mathbf{Env} \twoheadrightarrow [\![e]\!]_{Env} \to \mathsf{E}([\![e]\!]_{Env})$

(D3) $[\![$ `var` I `as` *data-type* $]\!]_{decl} \ = \ \lambda\, e.\ \lambda\, \rho.\ create\ e\ \rho\ I$

(D4) $[\![$ `fun` I `(` *formal-parameter-list* `)` `as` *return-type body* `end` $]\!]_{decl} \ = \ \lambda\, e.\ unit$

▶ $\mathcal{F}[\![decl]\!] \ : \ e : \mathbf{Env} \twoheadrightarrow [\![e]\!]_{Env} \to \mathsf{E}([\![e]\!]_{Env})$

(D3) $\mathcal{F}[\![$ `var` I `as` *data-type* $]\!]_{decl} \ = \ \lambda\, e.\ unit$

(D4) $\mathcal{F}[\![$ `fun` I `(` *formal-parameter-list* `)` `as` *return-type body* `end` $]\!]_{decl} \ = \ \lambda\, e.\ \lambda\, \rho.$
$\mathcal{F}\{\!|$*formal-parameter-list*$|\!\}$ ($\uparrow$ e) $*$ ($\lambda\, e'$. $\{\!|$*return-type*$|\!\}$ $*$ ($\lambda\, r$.
let $f = \lambda\, d_p.\ [\![$*formal-parameter-list*$]\!]_{prot\ [p]}\ d_p\ e'\ (e' \uparrow \rho)\ *\ (\lambda\, \rho'.\ funbody_r\ ([\![body]\!]_{body\ [r]}\ e'\ \rho'))$
in *assign* $e\ \rho\ I\ f$))

▶ $[\![prot\ [p]]\!] \ : \ [\![p]\!]_{Prot} \to e : \mathbf{Env} \twoheadrightarrow [\![e]\!]_{Env} \to \mathsf{G}([\![e]\!]_{Env})$

▶ $[\![par\ [\tau, m]]\!] \ : \ [\![\tau, m]\!]_{Par} \to e : \mathbf{Env} \twoheadrightarrow [\![e]\!]_{Env} \to \mathsf{G}([\![e]\!]_{Env})$

(P3) $[\![I$ `as` *data-type* `byval` $]\!]_{par\ [\tau, byval]} \ = \ \lambda\, d_a.\ \lambda\, e.\ \lambda\, \rho.$
lift$_{\mathsf{E}\to\mathsf{G}}$ (*create* $e\ \rho\ I$) $*$ ($\lambda\, \rho'$.
lift$_{\mathsf{E}\to\mathsf{G}}$ (*lookup* $e\ \rho'\ I$) $*$ ($\lambda\, o$.
setState (*write* $o\ d_a$) $*$ ($\lambda\, s$.
unit ρ')))

(P4) $[\![I$ `as` *data-type* `byref` $]\!]_{par\ [\tau, byref]} \ = \ \lambda\, d_a.\ \lambda\, e.\ \lambda\, \rho.$
lift$_{\mathsf{E}\to\mathsf{G}}$ (*assign* $e\ \rho\ I\ d_a$)

function is required for each phrase type, representing the dynamic meaning of phrases that can be attributed this type. More than one functions may be necessary if there are more than one possible meanings for a program phrase. Subsequently, one equation is required for each typing rule whose conclusion is a main typing relation. A large subset of the dynamic semantic functions and equations for $\mathcal{L}$, corresponding to the typing rules given in Fig. 6 is given in Fig. 8, 9 and 10.

8 Implementation

Easily produced and verifiably correct implementations, directly based on formal definitions, are valuable tools for the semantic analysis, design and evaluation of programming languages. Although not in the spirit of operational semantics, a denotational description of a programming language defines directly an execution model in the form of the abstract interpreter discussed in Section 2. This execution model can be implemented by translating the denotational semantics to a program, written in some target language.

Several languages have been suggested and used for this purpose, with more or less success. It seems that typed functional programming languages, such as ML or Haskell, are more suitable for this purpose. The latter is a better choice for implementing the semantics obtained by the proposed methodology, mainly because of its richer type system, more flexible syntax, elegant support for monads and also because lazy evaluation avoids a number of non-termination problems.

A direct implementation of the definition of $\mathcal{L}$ using Haskell as the target language consists of approximately 700 lines of code: 15 lines were needed for the implementation of the abstract syntax, 120 for the static semantics, 200 for the typing semantics and 275 for the dynamic semantics. The translation from the source programs in $\mathcal{L}$ to the abstract syntax was implemented as 700 additional lines of Flex/Bison code[2]

[2]For the complete implementation the reader is referred to [9].

Fig. 9: Dynamic semantics for $\mathcal{L}$, functions and equations for statements.

▶	$[\![stmt\ [r]]\!] \ : \ e : \mathbf{Env} \twoheadrightarrow [\![e]\!]_{Env} \to \mathsf{K}_r(\mathbf{U})$
(S3)	$[\![\texttt{skip}]\!]_{stmt\ [r]} = \lambda e.\, \lambda \rho.\, unit\ u$
(S4, 5)	$[\![I := expression]\!]_{stmt\ [r]} = \lambda e.\, \lambda \rho.\, [\![I]\!]_{obj\ [\phi]}\ e\ \rho\ *\ (\lambda o.$ $lift_{\mathsf{G}\to\mathsf{K}}\ ([\![expression]\!]_{exp\ [\tau]}\ e\ \rho)\ *\ (\lambda v.\ lift_{\mathsf{G}\to\mathsf{K}}\ (setState\ (write\ o\ v))\ *\ (\lambda s.\ unit\ u)))$
(S6)	$[\![\texttt{begin}\ statement\text{-}list\ \texttt{end}]\!]_{stmt\ [r]} = [\![statement\text{-}list]\!]_{stmt\ [r]}$
(S7)	$[\![\texttt{if}\ condition\ \texttt{then}\ statement_1\ \texttt{else}\ statement_2]\!]_{stmt\ [r]} = \lambda e.\, \lambda \rho.\, lift_{\mathsf{G}\to\mathsf{K}}([\![condition]\!]_{exp\ [boolean]}\ e\ \rho)\ *\ (\lambda v.$ $\mathbf{if}\ v\ \mathbf{then}\ [\![statement_1]\!]_{exp\ [stmt\ [r]]}\ e\ \rho\ \mathbf{else}\ [\![statement_2]\!]_{exp\ [stmt\ [r]]}\ e\ \rho)$
(S8)	$[\![\texttt{while}\ condition\ \texttt{do}\ statement]\!]_{stmt\ [r]} = \lambda e.\, \lambda \rho.\, mfix\ (\lambda k.\ lift_{\mathsf{G}\to\mathsf{K}}([\![condition]\!]_{exp\ [boolean]}\ e\ \rho)\ *\ (\lambda v.$ $\mathbf{if}\ v\ \mathbf{then}\ [\![statement]\!]_{exp\ [stmt\ [r]]}\ e\ \rho\ *\ (\lambda u.\ k)\ \mathbf{else}\ unit\ u))$
(S9)	$[\![\texttt{call}\ I\ \texttt{(}\ expression\text{-}list\ \texttt{)}]\!]_{stmt\ [r]} = \lambda e.\, \lambda \rho.\, [\![I]\!]_{obj\ [func\ [nothing,p]]}\ e\ \rho\ *\ (\lambda f.$ $lift_{\mathsf{G}\to\mathsf{K}}\ ([\![expression\text{-}list]\!]_{arg\ [p]}\ e\ \rho)\ *\ (\lambda d_p.\ lift_{\mathsf{G}\to\mathsf{K}}\ (f\ d_p)))$
(S10)	$[\![\texttt{return}]\!]_{stmt\ [r]} = \lambda e.\, \lambda \rho.\, result\ u$
(S11)	$[\![\texttt{return}\ expression]\!]_{stmt\ [r]} = \lambda e.\, \lambda \rho.\, lift_{\mathsf{G}\to\mathsf{K}}\ ([\![expression]\!]_{exp\ [\tau]}\ e\ \rho)\ *\ (\lambda v.\ result\ v)$

Fig. 10: Dynamic semantics for $\mathcal{L}$, functions and equations for expressions.

▶	$[\![exp\ [v]]\!] \ : \ e : \mathbf{Env} \twoheadrightarrow [\![e]\!]_{Env} \to \mathsf{G}([\![v]\!]_{exp})$
▶	$[\![obj\ [\phi]]\!] \ : \ e : \mathbf{Env} \twoheadrightarrow [\![e]\!]_{Env} \to \mathsf{G}([\![\phi]\!]_{den})$
(E1)	$[\![n]\!]_{exp\ [integer]} = \lambda e.\, \lambda \rho.\, unit\ \mathcal{N}[\![n]\!]$
(E2)	$[\![I]\!]_{obj\ [\phi]} = \lambda e.\, \lambda \rho.\, lift_{\mathsf{E}\to\mathsf{G}}\ (lookup\ e\ \rho\ I)$
(E3)	$[\![\texttt{call}\ I\ \texttt{(}\ expression\text{-}list\ \texttt{)}]\!]_{exp\ [\tau]} = \lambda e.\, \lambda \rho.\, [\![I]\!]_{obj\ [func\ [\tau,r]]}\ e\ \rho\ *\ (\lambda f.$ $[\![expression\text{-}list]\!]_{arg\ [p]}\ e\ \rho\ *\ (\lambda d_p.\ f\ d_p))$
(E4)	$[\![un\text{-}op\ expression]\!]_{exp\ [integer]} = \lambda e.\, \lambda \rho.\, [\![expression]\!]_{exp\ [integer]}\ e\ \rho\ *\ (\lambda n.\ lift_{\mathsf{E}\to\mathsf{G}}\ (\mathcal{UO}[\![un\text{-}op]\!]\ n))$
(E5)	$[\![expression_1\ bin\text{-}op\ expression_2]\!]_{exp\ [integer]} = \lambda e.\, \lambda \rho.\, [\![expression_1]\!]_{exp\ [integer]}\ e\ \rho\ *\ (\lambda n_1.$ $[\![expression_2]\!]_{exp\ [integer]}\ e\ \rho\ *\ (\lambda n_2.\ lift_{\mathsf{E}\to\mathsf{G}}\ (\mathcal{BO}[\![bin\text{-}op]\!]\ \langle n_1, n_2 \rangle)))$
(I1, 2)	$[\![expression]\!]_{exp\ [\tau]} = \lambda e.\, \lambda \rho.\, [\![expression]\!]_{obj\ [\phi]}\ e\ \rho\ *\ (\lambda \phi.\ getState\ *\ (\lambda s.\ unit\ (read\ s\ o)))$
▶	$[\![arg\ [p]]\!] \ : \ e : \mathbf{Env} \twoheadrightarrow [\![e]\!]_{Env} \to \mathsf{G}([\![p]\!]_{Prot})$
(A1)	$[\![\epsilon]\!]_{arg\ [p_o]} = \lambda e.\, \lambda \rho.\, unit\ noargs$
(A2)	$[\![expression\ expression\text{-}list]\!]_{arg\ [p]} = \lambda e.\, \lambda \rho.\, lift_{\mathsf{E}\to\mathsf{G}}\ (\Downarrow p)\ *\ (\lambda \langle \tau, byval, p' \rangle.$ $[\![expression]\!]_{exp\ [\tau]}\ e\ \rho\ *\ (\lambda d.\ [\![expression\text{-}list]\!]_{arg\ [p']}\ e\ \rho\ *\ (\lambda d_{p'}.\ prepend_{p,\tau,byval,p'}\ d\ d_{p'})))$
(A3, 4)	$[\![expression\ expression\text{-}list]\!]_{arg\ [p]} = \lambda e.\, \lambda \rho.\, lift_{\mathsf{E}\to\mathsf{G}}\ (\Downarrow p)\ *\ (\lambda \langle \tau, byref, p' \rangle.$ $[\![expression]\!]_{obj\ [\phi]}\ e\ \rho\ *\ (\lambda d.\ [\![expression\text{-}list]\!]_{arg\ [p']}\ e\ \rho\ *\ (\lambda d_{p'}.\ prepend_{p,\tau,byref,p'}\ d\ d_{p'})))$
▶	$\mathcal{UO}[\![un\text{-}op]\!] \ : \ \mathbf{N} \to \mathsf{E}(\mathbf{N})$
▶	$\mathcal{BO}[\![bin\text{-}op]\!] \ : \ \mathbf{N} \times \mathbf{N} \to \mathsf{E}(\mathbf{N})$

9 Evaluation

The proposed methodology has been used for the formal definition of the ANSI C programming language, as described in [4]. The definition was based on the standard for the language and is as complete and accurate as possible. The definition of abstract syntax consists of 39 production rules, whereas the definition of static semantics involves 17 domains, one monad, 33 semantic functions and 93 semantic equations. Furthermore, approximately 200 typing rules are used for the typing semantics. The dynamic semantics is the most complicated part, because of C's inherent complexity: 41 domains are used, 6 monads, one monad transformer, 45 semantic functions and more than 250 dynamic equations.

A structured implementation of the semantics of ANSI C in Haskell has also been developed. It consists of approximately 15,000 lines of Haskell code, which are distributed roughly as follows: 3,000 lines for the static semantics, 3,000 lines for the typing semantics, 5,000 lines for the dynamic semantics, 3,000 lines for parsing and pretty-printing and 1,000 more lines of general code and code related to testing. The implementation was used for the evaluation of the completeness and accuracy of the developed semantics, with very positive results.

10 Conclusion and future work

In this paper, we have presented a methodology for the formal definition of programming languages, following the denotational approach and using monads to improve modularity and elegance of notation. The main contribution of the proposed methodology are the three distinct consecutive phases that it proposes, for the definition of the semantics. The obtained programming language definitions are well structured and modular and can be implemented in the form of abstract interpreters in a relatively straightforward way.

Our research in the near future will focus on the process of evaluating and improving the proposed methodology, by applying it to programming languages of different nature and complexity. A secondary aim is to study the practical applications that formal definitions of programming languages may have in the software industry, especially in tools for program transformation, debugging and understanding, and how our methodology can be enriched to support these applications.

References:

[1] R. D. Tennent, "The denotational semantics of programming languages," *Communications of the ACM*, vol. 19, pp. 437–453, Aug. 1976.

[2] P. D. Mosses, "Denotational semantics," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. B, ch. 11, pp. 577–631, Elsevier Science Publishers B.V., 1990.

[3] R. D. Tennent, *Semantics of Programming Languages*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[4] N. S. Papaspyrou, *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, Software Engineering Laboratory, Feb. 1998.

[5] E. Moggi, "Computational lambda calculus and monads," in *IEEE Symposium on Logic in Computer Science*, pp. 14–23, 1989.

[6] E. Moggi, "An abstract view of programming languages," Tech. Rep. ECS-LFCS-90-113, University of Edinburgh, Laboratory for Foundations of Computer Science, 1990.

[7] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL'92)*, Jan. 1992.

[8] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, (San Francisco, CA), Jan. 1995.

[9] N. Papaspyrou, "A methodology for the definition of programming languages," Tech. Rep. CSD-SW-TR-1-99, National Technical University of Athens, Software Engineering Laboratory, Apr. 1999.