ELSEVIER

Computer Standards & Interfaces 23 (2001) 169–185

COMPUTER STANDARDS & INTERFACES

www.elsevier.com/locate/csi

Denotational semantics of ANSI C

Nikolaos S. Papaspyrou *

Department of Electrical and Computer Engineering, Software Engineering Laboratory, National Technical University of Athens, Polytechnioupoli, 15780 Zografou, Athens, Greece

Received 20 September 2000; received in revised form 28 December 2000; accepted 5 January 2001

Abstract

The semantics of C is described in the ANSI/ISO standard using natural language. This paper contains a brief summary, more descriptive than technical, of our research in specifying a complete and accurate formal semantics for ANSI C. We follow the denotational approach and divide the specification in three distinct phases: static, typing and dynamic semantics. Moreover, we have developed a direct implementation of the semantics, using the programming language Haskell. We argue that our formal specification results in a better understanding of the semantics of ANSI C and comment on its readability, precision, abstraction and applications.

Keywords: ANSI C programming language; ISO/IEC 9899:1999 standard; Formal definition; Denotational semantics; Monads

1. Introduction

C is a well-known and very popular general purpose programming language which represents, together with its descendants, a strong and indisputable status quo in the current software industry. It is a medium-level language, mainly characterized by its economy of expression, its large set of operators and data types, and its concern for source code portability. The general feeling towards C can probably be best summarized in a statement made by its inventor, Dennis Ritchie: "C is quirky, flawed and an enormous success" [32]. In 1990, ISO/IEC 9899:1990 was adopted as the first standard for the ANSI C programming language [14]. It was later amended by a few complementary documents. A long review process completed in 1999, resulting in the revised standard ISO/IEC 9899:1999, nicknamed "C9X", which is currently considered as the official language documentation [15]. The standard is nowadays accepted as a common basis by the developers and the users of C implementations and other tools.

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

* Tel.: +30-1-772-2486; fax: +30-1-772-2519.
E-mail address: nickie@softlab.ntua.gr (N.S. Papaspyrou).

0920-5489/01/$ - see front matter
PII: S0920-5489(01)00059-9

The semantics of ANSI C is informally defined in the standard [15] using natural language. This causes a number of ambiguities and problems of interpretation, clearly manifested in numerous discussions which often take place in the news-group `comp.std.c`. It is worth noticing that members of the standardization committee and other distinguished researchers participating in the discussions often give contradictory answers when asked about the intended semantics of surprisingly small programs, and that their answers are usually based on different possible interpretations of the standard. With all this in mind, the necessity for a formal description of the semantics of C becomes apparent. Such a description would serve as a rigid mathematical model for the language, without restricting the techniques used in implementations. Moreover, it would provide a basis for reasoning about properties of C programs and would be a valuable tool for the analysis, evaluation and possible redesign of the language.

The semantics of many popular programming languages have been formally specified in literature using various formalisms. However, in most cases the specifications are incomplete, inaccurate or both, to some extent. By *incomplete* we mean that they do not specify the semantics of the whole language but that of a subset, often leaving out the most complicated features. By *inaccurate* we mean that the formal descriptions are not entirely correct, either because of intended simplifications or by mistake. Few real programming languages, i.e. high-level languages that are widely used in industry for software development, have been given formal semantics as part of their definition. Among them one should mention Scheme [1,12,13] and Standard ML [22,41]. Other languages that have been at least partly formalized include: Ada [25,30]; Algol 60 [2,10]; C++ [39,40]; Cobol [37]; Modula-2 [9,24]; PL/1 [21,31,42]; Pascal [3,11,36]; Prolog [5]; and Smalltalk 80 [4,43].

Significant research has been conducted recently concerning the semantics of C. In what seems to be the earliest formal approach, Sethi [33,34] addresses mainly the semantics of pre-ANSI C declarations and control structures, using the denotational approach and making several simplifications. In the work of Gurevich and Huggins [8], a formal semantics for C is given as an evolving algebra. Again, a number of simplifications are made, e.g. no interleaving is possible in expression evaluation and side effects are assumed to take place at the same time that they are generated. In the work of Cook and Subramanian [7], an incomplete semantics for C is developed in the theorem prover Nqthm. Cook et al. [6] have also developed a denotational semantics for C based on temporal logic, which again makes a number of simplifying assumptions mainly concerning evaluation order. Finally, in the work of Norrish [27], a complete operational semantics for C is given using small-step reductions. To the best of our knowledge, this is the only approach that formalizes correctly C's unspecified order of evaluation and sequence points. No similar denotational approach is known to us.

In the present paper, we summarize the results of our research [28], aiming at the development of a complete and accurate formal description for the semantics of ANSI C. For this purpose, we have chosen the denotational approach [26,35] and employ monads and monad transformers [20,23,38] in order to improve the modularity and elegance of the developed semantics. Our formalization is based on the 1990 version of the standard and all references to paragraphs and pages are with respect to that version [14].

The rest of the paper is structured as follows. In Section 2, we identify a few common misunderstandings concerning the semantics of C. We discuss their causes, consequences and to what extent they can be attributed to faults and weeknesses in the language's standard. Section 3 contains a (more descriptive than technical) summary of our approach, dividing the specification of C's formal semantics in three distinct phases (static, typing and dynamic semantics) and illustrating their collaboration. In Section 4, we present an evaluation of our semantics and comment on its possible applications. Finally, in Section 5, we summarize the contribution of our research and show directions for future work.

2. Misinterpretations in the semantics of C

C is probably the most widely spread programming language in today's software industry. We

believe, however, that there is a large number of programmers who are confident of their understanding of C, but whose understanding is unfortunately subjective and incorrect, i.e. they do not understand the language in the way that is intended in the standard. To support this opinion, we present four simple program segments that are sources of common misinterpretations among C programmers. In the first three cases, taken respectively from the areas of static, typing and dynamic semantics as distinguished in our research, all doubts vanish when one reads the standard carefully.

2.1. Case 1

Consider the two small program segments shown in Fig. 1. The two segments differ only in the presence of identifier `dummy` in line 3. This identifier is never used within function `f` and therefore, one might assume that the two segments are equivalent. A question that may seem easy at first is: *What are the members of the structure pointed by* `x` *?*

Possible candidates are obviously `a` and `b`, depending on which of the declarations for structure `s` is in effect at line 4. But which one is it? The correct answer is that the two programs are not equivalent and the only member of the structure is `a` in the case of segment (A), and `b` in the case of segment (B). The rationale behind this answer can be found in Section 6.5.2.3 of the standard. In brief, line 3 of segment (B) defines a new incomplete structure type and overrides the definition of `s` in the enclosing scope, whereas line 3 of segment (A) does not have the same effect.

2.2. Case 2

Next, consider the following program segment, which is intended to increase the contents of a variable representing the number of counted men or women, depending on the value of a boolean flag.

```
int countMen = 0, countWomen = 0;

(sexFlag ? countMen : countWomen)++;
```

The question now is: *Is this program segment legal?*

The answer depends on whether a conditional expression can be an *l*-value or not, an *l*-value being roughly an expression designating an object whose value can be accessed and modified. A footnote in Section 6.3.15 of the standard states that it cannot, thus invalidating the above segment. However, popular C compilers (e.g. GNU C) support conditional *l*-values as an extension to the standard and by default allow such constructs.

2.3. Case 3

As a third example, consider one of the most infamous C expressions:

```
x = x++
```

together with the question: *Is this expression legal and, if yes, what are the contents of variable* `x` *after its execution?*

Before attempting an answer, one should be familiar with some key notions regarding the semantics of C. According to the standard, evaluation order in C is unspecified and so is the order in which side effects take place. Moreover, the standard defines the notion of *sequence points*, which are points in the execution sequence when "all previous side effects shall be complete and no side effects of subsequent evaluations shall have taken place". Sequence points occur as the result of specific C constructs, among which one should mention: complete evaluation of an expression in a statement, function call, operators `&&`, `||`, `?:` and `,` (comma).

The correct answer to the previous question is that this expression leads to undefined behaviour since it violates the restriction in Section 6.3 of the standard, according to which "between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression". The purpose of this restriction is exactly to rule out expressions with ambiguous results, like the one presented above.

```
1: struct s { int a; };
2: void f () {
3:   struct s dummy;
4:   struct s * x;
5:   struct s { int b; } y;
6: }
```

Segment (A)

```
struct s { int a; };
void f () {
  struct s;
  struct s * x;
  struct s { int b; } y;
}
```

Segment (B)

Fig. 1. Example of misinterpretation in static semantics.

Although these three exact program segments will not occur very often in practice, it is very probable that any given C programmer will eventually run into a similar case. One might argue that, since the informal standard dictates the correct answers, the real reason behind misinterpretations is the programmers' incompetence. It is true that not many C programmers have ever read the standard, and this fact is totally in agreement with the current requirements of the software industry. But, before we declare the standard innocent, let us consider one interesting fourth case.

2.4. Case 4

Consider the following simple C program. It presents a situation that will eventually arise in practice, although probably not in this exact form. In this case, however, the standard itself is not so clear and many possible interpretations exist.

```
int r = 0;

int f (int x)
{
   return (r = x);
}

int main ()
{
   return (f(1) + f(2), r);
}
```

In this program, function `f` assigns the value of its parameter to the global variable `r`. The last stored value in `r` is also the result of the program. There are two calls to `f`, having `1` and `2` as parameters, but unspecified evaluation order prevents us from knowing which one will be the last to execute.

In the light of all this, a natural question is: *Is this program legal and, if yes, what is the value returned by* `main`*?*

Similar programs and questions are often discussed in `comp.std.c`, invariably leading to the expression of numerous contradictory opinions and no conclusions reached. Although a technical discussion will be avoided here, two sound answers corresponding to different possible interpretations of the standard are the following.

- The program is legal and its result may be 1 or 2, but it is unspecified which one.
- The program is not legal because `r` is modified twice between successive sequence points.

The reason behind the two different answers is that the standard does not clearly specify whether sequence points occurring in the body of `f` should "count" as sequence points in the body of `main`. If they should, the first answer is correct, otherwise the second. The approach taken by our dynamic semantics corresponds to the first answer, which allows C programs to be non-deterministic.

As a conclusion, we believe that programmers' incompetence, although a significant problem on its own right, is not solely responsible for misunderstandings. Responsibility lies in the standard as well. C is inherently complicated and it is reasonable that an informal standard may fail to define it precisely

and beyond misinterpretation. Informal texts are valuable as introductions to the language and for educational purposes. However, we believe that the definition of the language must be more formal, in order to preclude ambiguities. After all, C is very often used to program applications of a very delicate nature, where software failure may have disastrous results. In this context, misunderstandings about the programming language cannot be allowed.

3. A formal semantics for C

Our denotational description of the semantics of C can be best understood as part of an *abstract interpreter*, illustrated in Fig. 2. The left part of the figure is a module diagram of the interpreter, showing the chain of steps that are required, whereas the right part shows the chain of data that is processed. Each step takes as input the results of previous steps. The initial piece of data is a *source program*, written in C, and the final result is a representation of this program's *meaning*, i.e. a description of the program's behaviour when it is executed.

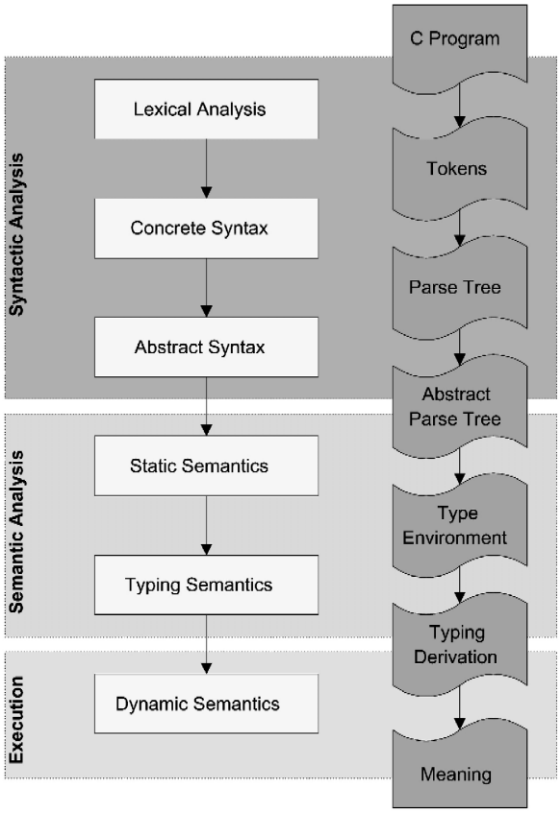

Fig. 2. An abstract interpreter for ANSI C.

The interpreter consists of three layers, each containing a series of steps. *Syntactic analysis* aims at checking the syntactic validity of the source program. Syntactically correct programs are transformed to abstract parse trees, which represent their structure in detail. *Semantic analysis* aims at checking the semantic validity of the program. Finally, *execution* aims at describing the meaning of programs. Our research focuses on the last two layers, containing a total of three steps: *static semantics*, *typing semantics* and *dynamic semantics*. An overview of these steps and their collaboration is given in the rest of this section. The notion of computation and the use of monads for representing computations are discussed first, since they are common to all three steps.

3.1. Computations and monads

The notion of computation is very significant in the semantics of programming languages. Following the denotational approach, in which the meanings of programs and program phrases are elements of mathematical spaces called *domains*, computations are represented by elements of appropriate such spaces. If D is a domain of values, then $M(D)$ can be taken to denote a domain of computations that return values from D. In this sense, M can be thought of as a domain constructor that 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. For a proper introduction to monads and their relation with computations, the reader is referred to Refs. [23,38]. The following brief description, although naive, is sufficient for the purpose of this paper. A monad is a triple $\langle M, \mathrm{unit}_M, *_M \rangle$, where:

- M is a domain constructor,
- $\mathrm{unit}_M\colon A \to M(A)$ is a polymorphic function for *inserting* values in computations, and

- $*_M : M(A) \times (A \to M(B)) \to M(B)$ is a polymorphic binary operator for *extracting* values from computations,

for arbitrary domains A and B. Furthermore, three monad laws (omitted in this paper) must be satisfied by the definition of every monad M.

The value of $\mathrm{unit}_M v : M(A)$ is a trivial computation that immediately returns the result $v : A$. The behaviour of $*_M$ is more complex. Assume that $m : M(A)$ is a computation that returns the result $v : A$. Also assume that $f : A \to M(B)$. Then, $m *_M f : M(B)$ is the combined computation of m followed by $f(v)$. The subscripts in unit_M and $*_M$ may be omitted if the corresponding monad can be easily deduced from the context.

Many different types of computations are implicit in the semantics of C programs and a number of monads is required to represent them. Fig. 3 shows a brief description of these monads and their interconnection. The powerdomain monad and the resumption monad transformer model non-deterministic and interleaved computations, respectively. Both are required as a consequence of the unspecified evaluation order and the presence of side effects in C expressions. The other monads shown in the figure represent various aspects of computations that are related to the execution of C programs.

3.2. Static semantics

The static semantics of C 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, such as the redefinition of an identifier in the same scope. Apart from the complicated syntax of declarations that is characteristic of C, the static semantics is further complicated by the presence of incomplete types. Forward declarations of tags used in the recursive definition of structures and unions are also sources of complexity.

For each syntactically well-formed program phrase P, its static semantic meaning is denoted by $\{\!|P|\!\}$. Such meanings are typically types, type environments, i.e. associations of identifiers to types, or functions involving these two.

The domains that we use in the static semantics of C are summarized in Fig. 4. Most of them are defined by simple enumeration of their elements.

Auxiliary monads and monad transformers.

$\mathsf{P}(D)$	Powerdomain monad. — Non-deterministic computations.
$\mathsf{R}(M)(D)$	Resumption monad transformer. — Interleaved computations of type $M(D)$.

Monads for computations.

$\mathsf{E}(D)$	Error monad. — Static computations that may result in compile-time errors.
$\mathsf{V}(D)$	Value monad. — Constant computations.
$\mathsf{C}(D)$	Continuation monad. — Computations accessing the state.
$\mathsf{G}(D)$	Expression monad. — Interleaved computations of expressions.
$\mathsf{K}_\tau(D)$	Statement monad. — Computations of statements in a function returning τ.

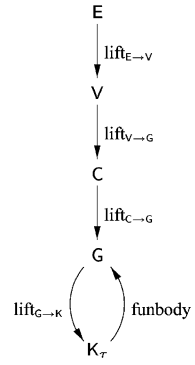

Fig. 3. Monads used in the semantics of ANSI C.

Auxiliary domains.

$I : \mathbf{Ide}$		*(identifiers)*
$t : \mathbf{Tag}$		*(tags)*

Domains for types.

$\tau : \mathbf{Type}_{dat}$	= void | char | signed-char | unsigned-char | short-int | unsigned-short-int | int | unsigned-int | long-int | unsigned-long-int | float | double | long-double | ptr $[\phi]$ | enum $[\epsilon]$ | struct $[t, \pi]$ | union $[t, \pi]$	*(data types)*
$q : \mathbf{Qual}$	= noqual | const | volatile | const-volatile	*(type qualifiers)*
$\alpha : \mathbf{Type}_{obj}$	= obj $[\tau, q]$ | array $[\alpha, n]$	*(object types)*
$f : \mathbf{Type}_{fun}$	= func $[\tau, p]$	*(function types)*
$\phi : \mathbf{Type}_{den}$	= α | f	*(denotable types)*
$m : \mathbf{Type}_{mem}$	= α | bitfield $[\beta, q, n]$	*(member types)*
$\beta : \mathbf{Type}_{bit}$	= int | signed-int | unsigned-int	*(bit-field types)*
$v : \mathbf{Type}_{val}$	= τ | f	*(value types)*
$\delta : \mathbf{Type}_{ide}$	= normal $[\phi]$ | typedef $[\phi]$ | enum-const $[n]$	*(identifier types)*
$\theta : \mathbf{Type}_{phr}$	*(Defined in Fig. 5, page 9)*	*(phrase types)*

Domains for environments.

$e : \mathbf{Ent}$	*(type environments)*
$\epsilon : \mathbf{Enum}$	*(enumeration environments)*
$\pi : \mathbf{Memb}$	*(member environments)*
$p : \mathbf{Prot}$	*(function prototypes)*

Fig. 4. Static semantic domains.

The domain ordering relation $\sqsubseteq$ is crucial in the treatment of incomplete types: $x \sqsubseteq y$ denotes that y is a better approximation of a possibly incomplete element x. Notice the number of different types that are dictated by C's type system. Among them, *data types* provide the basis for the type system, representing types that C programs can manipulate as first class elements. *Object types* are associated with objects in memory and consist of qualified versions of data types and array types. *Denotable types* are associated with identifiers in type environments, while *identifier types* are used for the classification of these identifiers. *Member types* are associated with identifiers defined as members of structures or unions. Finally, *phrase types* are associated with program phrases by the typing semantics of Section 3.3. Most domains for environments can be taken as functions from identifiers to types; their full definition is omitted in this paper.

Computations related to this phase are static computations, usually performed at compile time. Monad `E`, implemented as a simple error monad, represents computations of this kind. The complete static semantics of ANSI C is too long to be included in this paper. It consists of 17 domains, 1 monad, 33 semantic functions, 93 semantic equations and more than 100 auxiliary functions and operators on the basic static domains. We proceed with two short and illustrative excerpts.

Using monad `E`, the static meaning of a declaration can be defined as a function mapping the current type environment e to the computation of an updated

type environment, which will also contain mappings for the newly declared identifiers.

▶ $\{\!|\textit{declaration}|\!\}$: **Ent** → E(**Ent**)

As an example of how this function can be used, let us assume that e_0:**Ent** is the empty type environment. Then, the effect of the simple declaration `int x;` on this environment is the static computation $\{\!|$`int x;`$|\!\}\ e_0$:E(**Ent**), which should result in the type environment {"x" ↦ normal [obj [int, noqual]]}. In this environment, the only declared identifier is `x`, which is a normal object (i.e. a variable or function parameter) of the unqualified type `int`. If the function $\{\!|$`int x;`$|\!\}$ is applied to an environment *e* that already contains a different definition for `x`, the obtained computation will result in a static (compile-time) error.

Let us now consider the case of C declarators, the abstract syntax of which is defined in the standard as follows.

declarator ::= *I*
| *declarator* [*constant-expression*]
| *declarator* []
| ∗ *type-qualifier declarator*
| *declarator* (*parameter-type-list*)

Informally, a declarator is what follows the type specifier in a declaration. It can be a simple identifier, an array declarator (of a given or unspecified size), a pointer declarator (optionally qualified) or a function declarator.

The static meaning of a declarator can be defined as a function that takes as parameters the current type environment *e*:**Ent** and the type ϕ:$\mathbf{Type}_{den}$ that is provided by the type specifier. The result of the function is a pair, consisting of the declarator's identifier and the type that should be associated with it. The following equations correspond to the first four cases of declarators; function declarators are more complicated and have been omitted from this paper. Notice the use of the monadic operations unit and ∗ that insert and extract values to and from static computations.

$$\blacktriangleright\ \{\!|\textit{declarator}|\!\} \ :\ \mathbf{Ent} \to \mathbf{Type}_{den} \to \mathsf{E}(\mathbf{Ide} \times \mathbf{Type}_{den})$$

$$\{\!|I|\!\} \ =\ \lambda e.\, \lambda \phi.\, \mathrm{unit}\, \langle I, \phi \rangle$$

$$\{\!|\textit{declarator}\ \texttt{[}\ \textit{constant-expression}\ \texttt{]}|\!\} \ =\ \lambda e.\, \lambda \phi.\, \mathbf{case}\ \phi\ \mathbf{of}$$
$$\alpha \Rightarrow \mathcal{IC}[\![\textit{constant-expression}]\!]\ e\ *\ (\lambda n.\ \mathbf{if}\ (n > 0)\ \mathbf{then}\ \{\!|\textit{declarator}|\!\}\ e\ \mathrm{array}\,[\alpha, n]\ \mathbf{else}\ \mathrm{error})$$
$$\mathbf{otherwise} \Rightarrow \mathrm{error}$$

$$\{\!|\textit{declarator}\ \texttt{[ ]}|\!\} \ =\ \lambda e.\, \lambda \phi.\, \mathbf{case}\ \phi\ \mathbf{of}$$
$$\alpha \Rightarrow \{\!|\textit{declarator}|\!\}\ e\ \mathrm{array}\,[\alpha, \bot]$$
$$\mathbf{otherwise} \Rightarrow \mathrm{error}$$

$$\{\!|\texttt{*}\ \textit{type-qualifier declarator}|\!\} \ =\ \lambda e.\, \lambda \phi.$$
$$\{\!|\textit{type-qualifier}|\!\}\ \mathrm{obj}\,[\mathrm{ptr}\,[\phi], \mathrm{noqual}]\ *\ (\lambda \phi'.\ \{\!|\textit{declarator}|\!\}\ e\ \phi')$$

The case of a simple declarator is trivial. In the two cases of array declarators, there are a few things to be noticed:

- The type ϕ of the array's element must be an object type α (arrays of functions are not allowed).
- If the size of the array is specified as a constant expression of integer type, it is statically evaluated (by function $\mathcal{IC}$) and the result *n* must be a positive number. If it is not specified, the choice of $n = \bot$ produces an incomplete array type.
- The produced array type is passed as the second parameter to the recursive call of $\{\!|\textit{declarator}|\!\}$; recursion will eventually end upon a simple declarator.

Phrase type	Description
exp $[v]$	Expression, whose result is a non-constant r-value of type v.
lvalue $[m]$	Expression, whose result is an l-value of type m.
val $[\tau]$	Expression, whose result is a constant r-value of type τ.
arg $[p]$	Actual arguments of a function with prototype p.
stmt $[\tau]$	Statement in a function returning a result of type τ.
tunit	Translation unit.
xdecl	External declaration.
decl	Declaration.
prot $[p]$	Description of a function prototype specified by p.
par $[\tau]$	Description of parameter of type τ.
idtor	Declarator with initializer.
dtor $[\phi]$	Declarator for identifier of type ϕ.
init $[m]$	Initializer for a member of type m.
init-a $[\alpha]$	Initializer for an array object with elements of type α.
init-s $[\pi]$	Initializer for a structure object with members included in π.
init-u $[\pi]$	Initializer for a union object with members included in π.

Fig. 5. Phrase types.

Finally, in the case of pointer declarators, ϕ' is the type obtained by applying the qualifier to obj[ptr[ϕ], noqual], the result being a possibly qualified pointer to an object or function of type ϕ.

3.3. Typing semantics

Typing semantics focuses on program phrases, aiming at the detection of type-mismatch errors, e.g. dereferencing an expression that is not a pointer, assigning to something that is not an *l*-value, etc. In the absence of type errors, syntactically well-formed phrases are associated with appropriate phrase types. Such associations are given by means of *typing derivations*, i.e. formal proofs whose phrases are well-typed. Typing derivations use inference rules to prove *typing judgements* such as $e \vdash P{:}\theta$, which states that phrase P has type θ in environment e.

The primary aim of typing semantics is the association of program phrases with phrase types. Fig. 5 shows all the phrase types that we use. Typing rules are inference rules whose premises and conclusion are typing judgements. Several forms of typing judgements are necessary in the typing rules for ANSI C. A summary of the most important ones is given in Fig. 6. Approximately 200 typing rules are used in our approach. Seventy percent of those deal with expressions, 10% with statements and the remaining 20% with declarations. Some non-trivial examples of typing rules are presented next, followed by a discussion of our typing semantics.

The following rules specify the typing semantics of four types of expressions, in accordance with the ANSI C standard. Rule (E1) states that decimal constants with no suffix are attributed type val[τ], where τ is the first type from the list [int, long-int, unsigned-long-int] that can represent their constant value. According to Rule (E2), identifiers that have been defined as normal variables in e are *l*-values of the appropriate type. Similarly, Rules (E3) and (E4) specify the typing semantics of the pointer dereference operator and function calls, respectively.

$$\frac{\text{suffix}(n) = \emptyset \qquad \text{isDecimal}(n) \qquad \tau := \text{firstToRepresent}(n, [\text{int}, \text{long-int}, \text{unsigned-long-int}])}{e \vdash n{:}\text{val}[\tau]} \tag{E1}$$

$$\frac{e \vdash I \lhd \text{normal}[\alpha]}{e \vdash I{:}\text{lvalue}[\alpha]} \tag{E2}$$

$$\frac{e \vdash E : \exp[\mathrm{ptr}[\alpha]]}{e \vdash *E : \mathrm{lvalue}[\alpha]} \tag{E3}$$

$$\frac{e \vdash E : \exp[\mathrm{ptr}[\mathrm{func}[\tau, p]]] \quad e \vdash \mathit{arguments} : \arg[p]}{e \vdash E(\mathit{arguments}) : \exp[\tau]} \tag{E4}$$

The following two rules specify in part the semantics of assignments. According to Rule (E5), the expression on the left side of the simple assignment operator must be a modifiable *l*-value, while the expression on the right side must be assignable to the corresponding data type. Rule (A1) states that an expression of arithmetic type τ can be assigned to an object of another arithmetic type τ'.

$$\frac{e \vdash E_1 : \mathrm{lvalue}[m] \quad \mathrm{isModifiable}(m) \quad \tau := \mathrm{datify}\ m \quad e \vdash E_2 \gg \tau}{e \vdash E_1 = E_2 : \exp[\tau]} \tag{E5}$$

$$\frac{e \vdash E : \exp[\tau] \quad \mathrm{isArithmetic}(\tau) \quad \mathrm{isArithmetic}(\tau')}{e \vdash E \gg \tau'} \tag{A1}$$

A small number of typing rules specify conversions that take place implicitly in the evaluation of expressions. Such conversions are called implicit coercions. For example, Rule (C1) states that *l*-values may be implicitly converted to the values stored in the designated objects. This rule can be applied

Main typing relation.	
$e \vdash \mathit{phrase} : \theta$	The given *phrase* can be attributed phrase type θ in type environment e.
Predicates as judgements.	
P	Predicate P is true, where P can be any valid predicate over truth values.
$v := z$	The static semantic valuation $z : \mathsf{E}(D)$ produces the value $v : D$ without generating a static (compile-time) error.
Judgements related to environments.	
$e \vdash I \triangleleft \delta$	Identifier I is associated with identifier type δ in type environment e.
$\pi \vdash I \triangleleft m$	Identifier I is associated with member type m in member environment π.
Judgements related to expressions.	
$e \vdash E \gg \tau$	Expression E can be assigned to an object of data type τ in type environment e.
$e \vdash E = \mathtt{NULL}$	Expression E is a null pointer constant in type environment e.
$e \vdash T \equiv \phi$	Type name T denotes type ϕ in type environment e.

Fig. 6. Typing judgements.

whenever it is the value of the object that is needed and not the object itself. According to Rule (C2), function designators can be converted to function pointers. Both kinds of conversions are explicitly mentioned in the standard.

$$\frac{e \vdash E{:}\mathrm{lvalue}[\mathrm{obj}[\tau, q]] \quad \mathrm{isComplete}(\tau)}{e \vdash E{:}\mathrm{exp}[\tau]} \qquad \text{(C1)}$$

$$\frac{e \vdash E{:}\mathrm{exp}[f]}{e \vdash E{:}\mathrm{exp}[\mathrm{ptr}[f]]} \qquad \text{(C2)}$$

The typing semantics of statements is specified in a similar way. Rule (S1) deals with compound statements. Notice that a compound statement defines a new scope. This is achieved by using the semantic operator $\uparrow : \mathbf{Ent} \to \mathbf{Ent}$. To this new scope the declarations occurring in the block statement must be added, allowing for recursively defined structures or unions. This is achieved by the semantic function $\mathbf{rec}\,\{\!|\textit{declaration-list}|\!\} : \mathbf{Ent} \to \mathtt{E}(\mathbf{Ent})$. The resulting environment e' is used for the typing of the compound statement's body.

$$\frac{\begin{array}{c} e' := \mathbf{rec}\{\!|\textit{declaration-list}|\!\}(\uparrow e) \\ e' \vdash \textit{declaration-list}{:}\mathrm{decl} \quad e' \vdash \textit{statement-list}{:}\mathrm{stmt}[\tau] \end{array}}{e \vdash \{\,\textit{declaration-list}\;\textit{statement-list}\,\}{:}\mathrm{stmt}[\tau]} \qquad \text{(S1)}$$

Rules (S2) and (S3), specifying the semantics of `while` and `return` statements respectively, are relatively easier. In Rule (S2), the condition must be of scalar type, i.e. arithmetic or pointer. In Rule (S3), the type of the returned expression must be assignable to the function's returned type.

$$\frac{e \vdash \textit{expression}{:}\mathrm{exp}[\tau'] \quad \mathrm{isScalar}(\tau') \quad e \vdash \textit{statement}{:}\mathrm{stmt}[\tau]}{e \vdash \mathtt{while}\ (\textit{expression})\ \textit{statement}{:}\mathrm{stmt}[\tau]} \qquad \text{(S2)}$$

$$\frac{e \vdash \textit{expression} \gg \tau}{e \vdash \mathtt{return}\ \textit{expression};{:}\mathrm{stmt}[\tau]} \qquad \text{(S3)}$$

The suggested typing semantics for C leads to two forms of ambiguity problems. The first concerns the uniqueness of typing results: it should be clear that the main typing relation does not always provide a unique phrase type for a given program phrase (implicit coercion rules such as Rules (C1) and (C2) are one source of such ambiguities). The second concerns the uniqueness of typing derivations for a given typing judgement. For example, there are two different derivations concluding with the fact that the sum of two integer constants is an integer expression: the first adds the constants and coerces the constant sum to an (possibly non-constant) integer expression, while the second coerces the summands separately and adds the resulting expressions.

The first form of ambiguity is not only harmless but in fact useful in our typing semantics. A given program phrase can be attributed different phrase types, depending on its role in the program, and a different dynamic semantic meaning will be calculated for each phrase type. The second form, however, is potentially harmful. To obtain a sound semantics for C, we require that all different possible derivations for a given typing judgement will result in the same dynamic semantics. This property is satisfied by the developed dynamic semantics for C, described in Section 3.4.

3.4. Dynamic semantics

The dynamic semantics of C specifies the execution behaviour of well-typed programs and program phrases. At the same time, run-time errors and other sources of undefined behaviour are detected. The dynamic meaning of a well-typed program for which the typing judgement $e \vdash P{:}\theta$ can be derived is denoted by $[\![e \vdash P{:}\theta]\!]$. Such a meaning is typically a function describing some aspect of the execution of P. The typing derivation for P is important since it determines the way in which the dynamic semantics will be calculated.

The most important source of complexity in an accurate definition of C's dynamic semantics is the unspecified evaluation order, combined with the fact that expressions generate side effects. An investigation of related issues, isolating the causes and proposing a general semantic framework, appears in Ref. [29]. In order to disallow undesired ambiguities, the ANSI C standard has introduced restrictions imposed on expression evaluation with the mechanism of sequence points. Additional restrictions are imposed on the access of objects between consecutive sequence points; however, according to our interpretation of the standard, this mechanism does not always prevent non-determinism, as was discussed in Section 2. The dynamic semantics is further complicated by pointer arithmetic, complex control statements like `for` and `switch`, and the presence of `goto` in combination with nested block scopes containing variable declarations.

For each static type, a dynamic semantic domain is defined, in order to represent the dynamic meaning of values of this type. The definitions of some dynamic semantic domains are shown in Fig. 7. Among other things, the domain of integer numbers **N** is used to represent values of type `int`, pointers to objects are represented by the objects' address or a special null value, and addresses of objects are offsets in the biggest (possibly aggregate) object containing them, in order to correctly model pointer arithmetic. Elements of the dynamic domain for type environments $[\![e]\!]_{Ent}$ map all identifiers defined in e to their dynamic meanings. Similarly, elements of the dynamic domain for function prototypes $[\![p]\!]_{prot}$ map a function's parameters to their dynamic meanings. Domain **Cod** contains mappings of all functions defined in a program to the dynamic meaning of their bodies. Moreover, $\mathbf{Lab}_\tau$ contains mappings of `goto` labels present in the body of a function returning a result of type τ to the dynamic meanings of the corresponding statements.

The notion of execution *interleaving* is a well-known one in the theory of concurrency. An interleaved evaluation of an expression consists of an arbitrary merging of the *atomic steps* that constitute the evaluation of its subparts. In the case of C, interleaving can naturally model the unspecified evaluation order of expressions. Side effects, i.e. read and write accesses to the state, and sequence points are considered to be the atomic steps in the evaluation of expressions. In our approach, the semantics of interleaving are hidden in the expression monad `G`. The interested reader is referred to our related paper [29].

The definition of dynamic semantics for program phrases is similar to that of static semantics. An important difference, however, is that the typing derivations provide useful information about the types associated with a well-typed program phrase, as well as its components. Thus, typing derivations control the definition of dynamic semantics, instead of ab-

Auxiliary domains and environments.

$\mathbf{U}$ *(domain with two elements: $\perp$ and $\mathbf{u}$)*
$\mathbf{N}$ *(flat domain of integer numbers)*

$\mathbf{Obj}$, $\mathbf{Fun}$, $\mathbf{Offset}$, $\mathbf{BitOfs}$, $\mathbf{Addr} = \mathbf{Obj} \times \mathbf{Offset}$
$[\![e]\!]_{Ent}$, $[\![p]\!]_{Prot}$, $\mathbf{Cod}$, $\mathbf{Lab}_\tau$

Domains for types.

$[\![\text{void}]\!]_{dat} = \mathbf{U}$, $[\![\text{int}]\!]_{dat} = \mathbf{N}$ *(data types)*
$[\![\text{ptr}\,[\alpha]]\!]_{dat} = \mathbf{Addr} \oplus \mathbf{U}$, $[\![\text{ptr}\,[f]]\!]_{dat} = \mathbf{Fun} \oplus \mathbf{U}$

$[\![\text{obj}\,[\tau, q]]\!]_{obj} = \mathbf{Addr}$, $[\![\text{array}\,[\alpha, n]]\!]_{obj} = \mathbf{N} \to [\![\alpha]\!]_{obj}$ *(object types)*

$[\![\text{func}\,[\tau, p]]\!]_{fun} = [\![p]\!]_{Prot} \to \mathsf{G}([\![\tau]\!]_{dat})$ *(function types)*

$[\![\alpha]\!]_{mem} = [\![\alpha]\!]_{obj}$, $[\![\text{bitfield}\,[\tau, q, n]]\!]_{mem} = \mathbf{Addr} \times \mathbf{BitOfs}$ *(member types)*

Fig. 7. Dynamic semantic domains.

stract syntax alone, and there is one dynamic semantic equation for each typing rule. In the rest of this section, we illustrate the definition of dynamic semantics by presenting some small examples.

Let us consider the simple case of typing Rule (E3) in Section 3.3. Well-typed phrases of types exp $[\upsilon]$ and lvalue $[m]$ participate in this rule, so we begin by describing what the dynamic meanings of such phrases are.

- $[\![e \vdash E\text{:exp}\,[\upsilon]]\!] : [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathtt{G}([\![\upsilon]\!]_{val})$
- $[\![e \vdash E\text{:lvalue}\,[m]]\!] : [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathtt{G}([\![m]\!]_{mem})$

The first line states that the dynamic semantic meaning for an expression E which has type exp$[\upsilon]$ in environment e is a function taking as arguments the dynamic environment and the code environment and returning an interleaved expression computation with a result of type $[\![\upsilon]\!]_{val}$. In a similar way, the type of dynamic semantic meanings for expressions of type lvalue $[m]$ is defined in the second line.

We now proceed to define the dynamic semantics for an expression of the form $\mathtt{*E}$ (pointer dereference) under the typing given in Rule (E3). The equation that follows defines $[\![e \vdash {*}E : \text{lvalue}\,[\alpha]]\!]$ in terms of $[\![e \vdash E\text{:exp}[\text{ptr}[\alpha]]]\!]$.

$$
\begin{aligned}
[\![e \vdash {*}E : \text{lvalue}\,[\alpha]]\!] \;=\; & \lambda\rho.\,\lambda\xi.\,[\![e \vdash E : \text{exp}\,[\text{ptr}\,[\alpha]]]\!]\,\rho\,\xi \;*\; (\lambda d_e.\\
& \quad \mathbf{case}\ d_e\ \mathbf{of}\\
& \qquad \mathbf{inl}\ d \;\Rightarrow\; \text{unit}\ (\text{fromAddr}_\alpha\ d)\\
& \qquad \mathbf{otherwise} \;\Rightarrow\; \text{error})
\end{aligned}
$$

Informally, it can be described as follows. First, compute the value d_e of the pointer expression. If the pointer contains an object's address d, this address (trivially converted to an *l*-value) is the dynamic meaning of the *l*-value. An error occurs if the pointer is null.

The following equations define the dynamic semantics that correspond to typing Rules (E4) and (C1) of Section 3.3, respectively.

$$
\begin{aligned}
& [\![e \vdash E\,(\textit{arguments}) : \text{exp}\,[\tau]]\!] \;=\; \lambda\rho.\,\lambda\xi.\\
& \quad \mathbf{let}\ g_e = [\![e \vdash E : \text{exp}\,[\text{ptr}\,[\text{func}\,[\tau, p]]]]\!]\,\rho\,\xi\\
& \qquad\ g_p = [\![e \vdash \textit{arguments} : \text{arg}\,[p]]\!]\,\rho\,\xi\\
& \quad \mathbf{in}\ \ g_e \bowtie g_p \;*\; (\lambda\langle d_e, d_p\rangle.\\
& \qquad\quad \text{seqpt} \;*\; (\lambda u.\ \mathbf{case}\ d_e\ \mathbf{of}\\
& \qquad\qquad \mathbf{inl}\ d_f \;\Rightarrow\; \mathbf{let}\ \langle f, b_f\rangle = \xi[d_f]\\
& \qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathbf{if}\ \text{isCompatible}(f, \text{func}\,[\tau, p])\ \mathbf{then}\ b_f\ d_p\ \mathbf{else}\ \text{error}\\
& \qquad\qquad \mathbf{otherwise} \;\Rightarrow\; \text{error}))\\
& [\![e \vdash E : \text{exp}\,[\tau]]\!] \;=\; \lambda\rho.\,\lambda\xi.\\
& \quad [\![e \vdash E : \text{lvalue}\,[\text{obj}\,[\tau, q]]]\!]\,\rho\,\xi \;*\; \text{getValue}_{\text{obj}\,[\tau,q]\mapsto\tau}
\end{aligned}
$$

In the first equation, there are two things to notice. The first is the use of operator $\bowtie : \mathtt{G}(A) \times \mathtt{G}(B) \to \mathtt{G}(A \times B)$ which allows the interleaving of the two expression computations g_e and g_p, reflecting the fact that the order in which the function's designator and the function's arguments are evaluated is unspecified. The second is the use of seqpt:$\mathtt{G}(\mathbf{U})$ to introduce a sequence point just before the function is actually called. Assuming the function pointer is not null, the function's dynamic meaning is looked up in the code environment. Furthermore, the function's actual type is required to be compatible with the type of the designator that is used. In the second equation, function $\text{getValue}_{m\mapsto\tau} : [\![m]\!]_{mem} \to \mathtt{G}([\![\tau]\!]_{dat})$ retrieves a stored value from memory using one atomic step.

The dynamic semantics of statements and declarations is defined in a similar way. For the whole definition of the dynamic semantics of ANSI C, we used 41 domains, 6 monads, 1 monad transformer, 45 semantic functions, more than 250 dynamic equations and more than 100 auxiliary functions and operators on the basic dynamic domains.

4. Evaluation

A significant effort has been made to evaluate our developed semantics for the C programming language. In this task, the major issue was to assess how complete and accurate the developed semantics is, with respect to the standard. Unfortunately, there is no systematic way to evaluate our approach and be absolutely certain that the results are valid: there is simply no way to compare a formal system of this complexity against an informal specification. For this reason, we have resorted in testing an interpreter that directly implements our semantics, by using test suites for C implementations that are publicly available. Moreover, the consistency of our formulae can be mechanically validated to some extent by a direct implementation in an appropriate strongly typed higher-order functional programming language.

An earlier version of our semantics was implemented in Standard ML. Subsequently, Haskell[1] was adopted as the implementation language, mainly because it has a richer type system, more flexible syntax, elegant support for monads and also because lazy evaluation avoids a number of non-termination problems. The current implementation consists of approximately 15,000 lines of Haskell code, roughly distributed as follows: 3000 lines for the static semantics, 3000 lines for the typing semantics, 5000 lines for the dynamic semantics, 3000 lines for parsing and pretty-printing and 1000 more lines of general code and code related to testing. As it was expected, the implementation is very slow and this presents a serious handicap in our yet unfinished evaluation process.

[1] The reader is referred to `http://haskell.org/` for the definition and a complete reference to the Haskell programming language.

Although the evaluation of our semantics is still under way and minor bugs are waiting to be fixed, both in the semantic description and the implementation, the results indicate that the developed semantics is complete and accurate to a great extent, with respect to the ANSI C standard. The most important deviations are the following:

- The developed semantics requires function prototypes to exist for all called functions. This is a step towards a more strongly typed C, which will probably be taken in a future revised standard.
- Storage specifiers other than `typedef` are currently ignored. Static variables may of course be preprocessed out, but a solution integrated in the semantics is currently investigated.
- Fully bracketed initializations are required when initializing aggregate objects.
- Declarations of identifiers, other than statement labels, are forbidden in expressions or statements. This is rather regarded as an improvement than as an omission.
- Our work does not cover the features introduced in the language by the “C9X” standard [15]. However, we do not believe that adding the new features will require excessive changes in the semantics.

A formal semantics for a widely used programming language such as ANSI C is expected to have numerous applications. As a complement to the informal standard, it improves our understanding of the language by specifying a rigid mathematical model but without restricting the techniques used in implementations. It is also a valuable tool for language analysis, design and evaluation; semantic descriptions can be tuned to suggest efficient implementations and often pinpoint weaknesses in language definitions. Furthermore, a formal semantics provides a basis for reasoning about the correctness of programs, based on formal specifications. It can also be used as input to compiler generators, for the automatic construction of provably correct compilers. Many experimental systems based on semantics-driven compilation have been developed. A quite recent approach, based on the use of monads, is described in Refs. [17–19].

5. Conclusion

Standards for programming languages pursue precision and abstraction, two properties that unfortunately are often in conflict. Precision is required to guarantee that the compliance of an implementation to a given standard can be certified without room for doubt. Abstraction is required so that no excessive restrictions are imposed upon implementations, since unnecessary restrictions often compromise efficiency. The development of precise and abstract standards for programming languages is an important application area for formal description techniques [16]. The formalism of denotational semantics can be used to provide precise and sufficiently abstract language definitions. Combined with the use of monads, it also satisfies two additional desired properties: modularity and readability.

In this paper, we have presented a summary of our work in developing a formal semantics for the ANSI C programming language, following the denotational approach. The developed semantics is satisfactorily complete and accurate, with respect to the standard, and is a demonstration that a programming language as useful in practice and as inherently complicated as C can nonetheless be given a formal semantics. Furthermore, although it is based on a rigid and rather complex theoretical background, the essence of the developed semantics can be captured without assuming a detailed knowledge of the theory, as soon as the reader becomes familiar with the formalism. A further significant contribution of our research is the application of monads and monad transformers for the specification of a real programming language. Interesting results have also been achieved in our attempt to model the interleaving of computations and non-determinism, which may be useful in specifying the semantics of programming languages supporting parallelism.

Our research in the near future will focus on the process of evaluating and improving the developed semantics. Given the current state of the art in formal methods, the task of validating the semantics is unfortunately very similar to software testing, both in terms of methodology and of complexity. Beyond that, we would like to study the practical applications that a formal semantics for C may have in the software industry, especially in tools for program transformation, debugging and understanding. The implementation of the developed semantics also gave rise to an interesting question: what are the characteristics of a programming language that make it suitable for implementing denotational specifications, especially using monadic notation? Finally, another direction for future research aims at studying and specifying the semantics of C's object-oriented descendants, C++ and Java.

References

[1] H. Abelson et al., Revised 4 report on the algorithmic language Scheme, Lisp Pointers 4 (3) (1991) 1–55, July.

[2] D. Bjørner, C.B. Jones, Algol 60,Formal Specification and Software Development, Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 141–173, Chap. 6.

[3] D. Bjørner, C.B. Jones, Pascal,Formal Specification and Software Development, Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 175–251, Chap. 7.

[4] B. Blakley, A Smalltalk Evolving Algebra and its Uses. PhD thesis, University of Michigan, Ann Arbor, MI, 1992.

[5] E. Börger, D. Rosenzweig, A mathematical definition of full Prolog, Science of Computer Programming 24 (3) (1995) 249–286, June.

[6] J. Cook, E. Cohen, T. Redmond, A formal denotational semantics for C. Technical Report 409D, Trusted Information Systems, September 1994.

[7] J. Cook, S. Subramanian, A formal semantics for C in Nqthm. Technical Report 517D, Trusted Information Systems, October 1994.

[8] Y. Gurevich, J.K. Huggins, The semantics of the C programming language, in: E. Börger (Ed.), Selected Papers from CSL'92 (Computer Science Logic), Lecture Notes in Computer Science, vol. 702, Springer-Verlag, New York, NY, 1993, pp. 274–308.

[9] Y. Gurevich, J. Morris et al., Algebraic operational semantics and Modula-2, in: E. Börger, H. Kleine Büning, M.M. Richter (Eds.), Proceedings of the 1st Workshop on Computer Science Logic (CSL'87), Lecture Notes in Computer Science, vol. 329, Springer-Verlag, New York, NY, 1988, pp. 81–101.

[10] W. Henhapl, C.B. Jones, A formal definition of Algol 60 as described in the 1975 modified report, in: D. Bjørner, C.B. Jones (Eds.), The Vienna Development Method: The Meta-language, Lecture Notes in Computer Science, vol. 61, Springer-Verlag, New York, NY, 1978, pp. 305–336.

[11] C.A.R. Hoare, N. Wirth, An axiomatic definition of the programming language Pascal, Acta Informatica 2 (1973) 335–355.

[12] F. Honsell, A. Pravato, S. Ronchi della Rocca, Structured operational semantics of the language Scheme. Technical

report, University of Torino, Department of Informatics, 1995.
[13] IEEE Standard for the Scheme Programming Language, IEEE Standard 1178–1990, Institute of Electrical and Electronics Engineers, New York, NY, 1991.
[14] International Organization for Standardization, New York, NY. ISO/IEC 9899-1990, Programming Languages: C, 1990. Technical Committee: JTC 1/SC 22/WG 14. Revision and redesignation of ANSI X3.159-1989.
[15] International Organization for Standardization, New York, NY. ISO/IEC 9899-1999, Programming Languages: C, 1999. Technical Committee: JTC 1/SC 22/WG 14.
[16] H. Kilov, The formal way, Computer Standards and Interfaces 17 (5–6) (1995) 409–412, Guest editorial to a Special Issue on Formal Description Techniques.
[17] S. Liang, A modular semantics for compiler generation. Technical Report YALEU/DCS/TR-1067, Yale University, Department of Computer Science, February 1995.
[18] S. Liang, Modular Monadic Semantics and Compilation. PhD thesis, Yale University, Department of Computer Science, May 1998.
[19] S. Liang, P. Hudak, Modular denotational semantics for compiler construction, Proceedings of the 6th European Symposium on Programming, 1996, pp. 219–234, April.
[20] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), San Francisco, CA, January, 1995, pp. 333–343.
[21] P. Lucas, K. Walk, On the formal description of PL/1, Annual Review in Automatic Programming 6 (3) (1969) 105–182.
[22] R. Milner, M. Tofte, R. Harper, The Definition of Standard ML, MIT Press, Cambridge, MA, 1990.
[23] E. Moggi, An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, Laboratory for Foundations of Computer Science, 1990.
[24] J. Morris, Algebraic Operational Semantics for Modula-2. PhD thesis, University of Michigan, Ann Arbor, MI, 1988.
[25] J. Morris, G. Pottinger, Ada-Ariel semantics. Technical report, Odyssey Research Associates, July 1990.
[26] P.D. Mosses, Denotational semantics, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, Amsterdam, The Netherlands, 1990, pp. 577–631, Chap. 11.
[27] M. Norrish, C Formalized in HOL. PhD thesis, University of Cambridge, Computer Laboratory, December 1998.
[28] N.S. Papaspyrou, A Formal Semantics for the C Programming Language. PhD thesis, National Technical University of Athens, Software Engineering Laboratory, February 1998.
[29] N.S. Papaspyrou, D. Maćoš, A study of evaluation order semantics in expressions with side effects, Journal of Functional Programming 10 (3) (2000) 227–244, May.
[30] J.S. Pedersen, A formal semantics definition of sequential Ada, in: D. Bjørner, O.N. Oest (Eds.), Towards a Formal Description of Ada, Lecture Notes in Computer Science, vol. 98, Springer-Verlag, New York, NY, 1980, pp. 213–308.
[31] PL/1 Definition Group. Formal definition of PL/1–ULD version III. Technical Report TR 25.095 bis 099, IBM Laboratory Vienna, June 1969.
[32] D.M. Ritchie, The development of the C language, ACM SIGPLAN Notices 28 (3) (1993) 201–208, March, Preprints of the Second ACM SIGPLAN History of Programming Language (HOPL II).
[33] R. Sethi, A case study in specifying the semantics of a programming language, Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages, 1980, pp. 117–130, January.
[34] R. Sethi, Control flow aspects of semantics-directed compiling, ACM Transactions on Programming Languages and Systems 5 (4) (1983) 554–595, October.
[35] J.E. Stoy, Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory, MIT Press, Cambridge, MA, 1977.
[36] R.D. Tennent, A denotational definition of the programming language Pascal. Technical report, Oxford University, Programming Research Group, April 1978.
[37] M. Vale, The evolving algebra semantics of COBOL, Part 1: Programs and control. Technical Report CSE-TR-162-93, University of Michigan, EECS Department, Ann Arbor, MI, 1993.
[38] P. Wadler, The essence of functional programming, Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL'92), 1992, pp. 1–14, January.
[39] C. Wallace, The semantics of the C++ programming language. Technical Report CSE-TR-190-93, University of Michigan, Department of Electrical Engineering and Computer Science, December 1993.
[40] C. Wallace, The semantics of the C++ programming language, in: E. Börger (Ed.), Specification and Validation Methods, Oxford Univ. Press, Oxford, England, 1995, pp. 131–164.
[41] D.A. Watt, An action semantics of Standard ML, Proceedings of the 3rd Workshop on the Mathematical Foundations of Programming Language Semantics, Lecture Notes in Computer Science, vol. 298, Springer-Verlag, New York, NY, 1987, pp. 572–598.
[42] P. Wegner, The Vienna definition language, Computing Surveys 4 (1) (1972) 5–63.
[43] M. Wolczko, Semantics of Smalltalk-80, European Conference on Object-Oriented Programming (ECOOP'87), Lecture Notes in Computer Science, vol. 276, Springer-Verlag, New York, NY, 1987, pp. 108–120.

Dr. Nikolaos S. Papaspyrou was born in Athens in 1971. He received a BSc in Electrical and Computer Engineering from the National Technical University of Athens (NTUA), Greece (1993), an MSc in Computer Science from Cornell University, Ithaca, NY (1995), and a PhD in Computer Science from NTUA (1998). His doctoral research focused on the denotational semantics of programming languages and its relation with the software development process. Other research interests include functional, logic and intensional programming, compilers, man–machine interface, educational software and learning environments. Dr. Papaspyrou has worked as a primary researcher in several R&D projects and as a short-term lecturer at the Department of Electronic Engineering and Computer Science of the Technical University of Crete. He is currently a visiting postdoctoral researcher at Yale University, New Haven, CT.