A CASE STUDY IN SPECIFYING THE DENOTATIONAL SEMANTICS OF C

N. S. PAPASPYROU
National Technical University of Athens
Department of Electrical and Computer Engineering
Division of Computer Science, Software Engineering Laboratory
Polytechnioupoli, 15780 Zografou, Athens, Greece.
Tel. +30-1-7722486, Fax. +30-1-7722519.

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. Its semantics is informally defined in the ISO/IEC 9899:1990 standard [1] using natural language. This causes a number of ambiguities and problems of interpretation, clearly manifested in numerous discussions taking place in the newsgroup `comp.std.c`. It is worthwhile 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 precise standard for compiler implementation and would provide a basis for reasoning about properties of C programs.

The semantics of many popular programming languages have been formally specified in literature using various formalisms. However, in most cases these specifications are incomplete, inaccurate or both, in varying degrees. 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.

Significant research has been conducted recently concerning semantic aspects of C. In what seems to be the earliest formal approach, Sethi addresses mainly the semantics of pre-ANSI C declarations, using the denotational approach and making several simplifications, e.g. requiring left-to-right evaluation [2]. In a different paper, Sethi addresses the semantics of C's control structures using again a denotational framework [3]. In the work of Gurevich and Higgins a formal semantics for C is given using the formalism of evolving algebras [4]. 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 an incomplete semantics for C is developed in the theorem prover Nqthm [5]. Cook et al. have also developed a denotational semantics for C based on temporal logic, which again makes a number of simplifying assumptions, mainly concerning evaluation order [6]. An operational semantics for C has been sketched, in terms of a random access machine, as a part of the MATHS project in California State University. Finally, in the work of Norrish a complete operational semantics for C is given using small-step reductions [7]. 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.

This chapter summarizes the results of our research, aiming at the development of an accurate and complete formal description for the semantics of the C programming language [8]. For this purpose, we have chosen the denotational approach.[1] As a remedy for the most important drawback of classic denotational semantics, its lack of modularity, we have used a number of monads which represent different aspects of computations.[2] In this way, the developed semantics was significantly improved in terms of modularity and elegance and its development was greatly facilitated.

2. Overview of the semantics

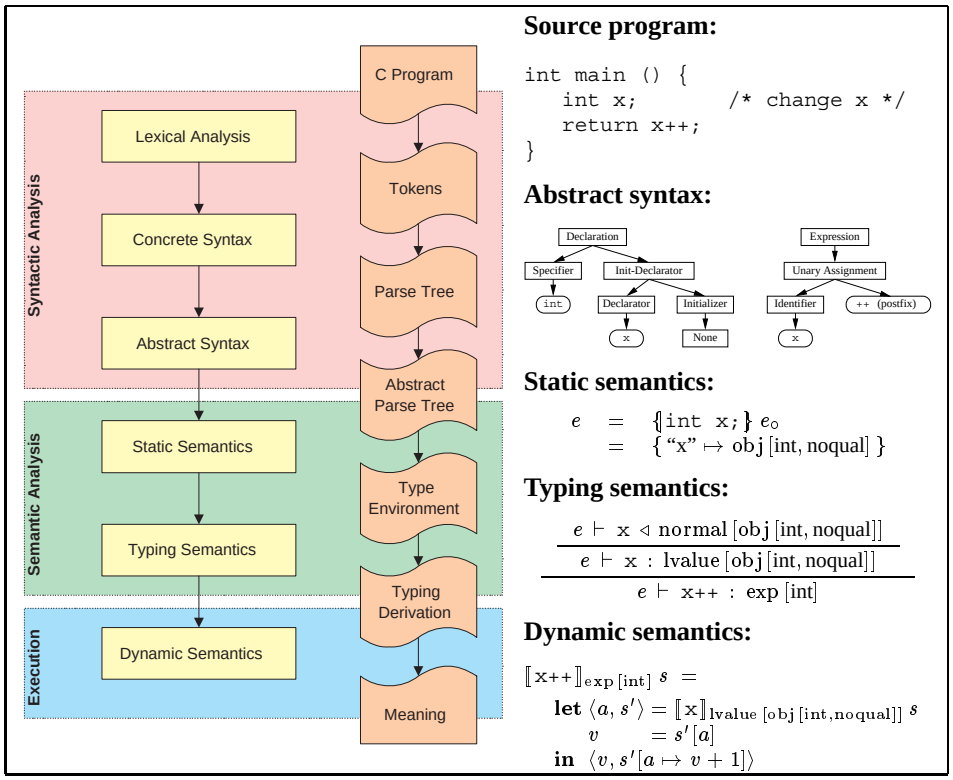
Our denotational specification for the semantics of C can be best understood as part of the *abstract interpreter* depicted in Fig. 1. The left part of the figure is a module diagram of the interpreter, showing the chain of actions performed and the processed data. Each action takes as input the result of 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 the program's behaviour when it is executed. The right part of the figure presents parts of a small example that will be discussed gradually until the end of this section. It should be noted that the example is simplified and does not correctly specify the semantics of C.

The interpreter consists of three layers, each containing 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 structure in detail. *Semantic analysis* is used to define aspects of C that cannot be defined by a context-free grammar. It aims at checking the semantic validity of the program and signalling semantic errors, e.g. use of undeclared identifiers or type mismatches. Finally, *execution* aims at describing the meaning of programs. Our research mainly focuses on the last two layers, containing a total of three actions: *static semantics*, *typing semantics* and *dynamic semantics*. A brief overview of these actions and their collaboration is given in this section. The following three sections present the basics of each action in more detail.

Static semantics keeps track of identifiers that are defined in the source program. It aims at detecting static semantic errors, such as the redefinition of an identifier in the same scope, as well as associating identifiers with appropriate types or values. For each

[1] Introductions to denotational semantics, including useful bibliography, can be found in [9] and [10].
[2] Introductions to monads and their use in denotational semantics can be found in [11] and [12].

Figure 1: An abstract interpreter for C.



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 handling these two. Considering the simple C program that is used as an example in Fig. 1, let us isolate the declaration "`int x;`". The static semantic meaning of this declaration is a function that updates the type environment by declaring an integer variable "`x`". If $e_\circ$ is the empty type environment, containing no declarations, then the result of $\{\!|$`int x;`$|\!\}$ applied to $e_\circ$ is the updated environment e shown in Fig. 1, which contains a declaration for "`x`".

Typing semantics focuses on program phrases. It aims at the detection of type-mismatch errors, such as assignment to a constant value, and at associating syntactically well-formed phrases with appropriate phrase types. Such associations are given by means of *typing derivations*, i.e. formal proofs that 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. In the same example program of Fig. 1, let us now consider the expression "`x++`". Assuming that the static semantic analysis has resulted in type environment e, it is possible to derive that "`x++`" is an expression that computes an integer value, in other words "$e \vdash$ `x++` $:$ exp [int]". A sketch of such a typing derivation is shown in Fig. 1. The derivation makes use of two inference rules, stating in short that

variables are l-values and how l-values can be used as operands of postfix "++". The initial assumption means that "x" is declared to be a variable of type int in e, and is not further analyzed here.

Finally, dynamic semantics aims primarily at defining the execution behaviour of well-typed programs. For each well-typed program phrase P of type θ, its dynamic semantic meaning is denoted by $[\![P]\!]_\theta$. 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. Going back once more to the same example, let us consider again the expression "x++", together with its typing derivation. Assuming a simple direct semantics for the dynamic meaning of expressions, the dynamic semantics for "x++" is given in Fig. 1. It is a function, taking the initial program state s and returning the result of the expression's evaluation and the final program state. Notice that the typing derivation dictates the types that are used for the dynamic semantics of phrases on both sides of the equation. For the sake of clarity, a represents the address of the object designated by the l-value "x", s' is the program state after evaluating "x" and v is the value stored in a at the program state s'. The result of the evaluation is v, and the final program state is the same as s', with the value stored in a incremented by one.

3. 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. Apart from the complicated syntax of declarations that is characteristic of C, the static semantics is further perplexed by the presence of incomplete types in C's type system. Forward declarations of tags used in the recursive definition of structures and unions are also sources of complexity.

The domains that we use in the static semantics of C are summarized in Fig. 2. Most of them are defined as coalesced sums, by enumeration of their elements. 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 §4. Most domains for environments can be taken as functions from identifiers to types; their full definition is omitted in this chapter.

The use of a simple error monad in the definition of the static semantics provides an elegant way of generating and propagating errors. In brief, a monad is a tuple $\langle \mathsf{M}, \mathrm{unit}, * \rangle$, where M is a domain constructor, $\mathrm{unit} : A \to \mathsf{M}(A)$ and $* : \mathsf{M}(A) \times (A \to \mathsf{M}(B)) \to \mathsf{M}(B)$ are polymorphic functions for arbitrary domains A and B, satisfying three monad

Figure 2: Static semantic domains.

Auxiliary domains.

$I : \mathbf{Ide}$ *(identifiers)*, $t : \mathbf{Tag}$ *(tags)*, $\sigma : \mathbf{TagType}$ *(tag types)*.

Domains for types.

$\tau : \mathbf{Type}_{dat}$	$=$	void $\mid$ char $\mid$ signed-char $\mid$ unsigned-char $\mid$ short-int $\mid$ unsigned-short-int $\mid$ int $\mid$ unsigned-int $\mid$ long-int $\mid$ unsigned-long-int $\mid$ float $\mid$ double $\mid$ long-double $\mid$ $\mathrm{ptr}\,[\phi]$ $\mid$ $\mathrm{enum}\,[\epsilon]$ $\mid$ $\mathrm{struct}\,[t, \pi]$ $\mid$ $\mathrm{union}\,[t, \pi]$	*(data types)*
$q : \mathbf{Qual}$	$=$	noqual $\mid$ const $\mid$ volatile $\mid$ const-volatile	*(type qualifiers)*
$\alpha : \mathbf{Type}_{obj}$	$=$	$\mathrm{obj}\,[\tau, q]$ $\mid$ $\mathrm{array}\,[\alpha, n]$	*(object types)*
$f : \mathbf{Type}_{fun}$	$=$	$\mathrm{func}\,[\tau, p]$	*(function types)*
$\phi : \mathbf{Type}_{den}$	$=$	$\alpha \mid f$	*(denotable types)*
$m : \mathbf{Type}_{mem}$	$=$	$\alpha \mid \mathrm{bitfield}\,[\tau, q, n]$	*(member types)*
$\beta : \mathbf{Type}_{bit}$	$=$	int $\mid$ signed-int $\mid$ unsigned-int	*(bit-field types)*
$v : \mathbf{Type}_{val}$	$=$	$\tau \mid f$	*(value types)*
$\delta : \mathbf{Type}_{ide}$	$=$	$\mathrm{normal}\,[\phi]$ $\mid$ $\mathrm{typedef}\,[\phi]$ $\mid$ enum-const $[n]$	*(identifier types)*
$\theta : \mathbf{Type}_{phr}$		*(See Fig. 3)*	*(phrase types)*

Domains for environments.

$e : \mathbf{Ent}$ *(types)*, $\epsilon : \mathbf{Enum}$ *(enumerations)*, $\pi : \mathbf{Memb}$ *(members)*, $p : \mathbf{Prot}$ *(function prototypes)*.

laws. Given a domain D, the domain $\mathsf{M}(D)$ is a domain of *computations* resulting in values of type D. Function unit converts values to (trivial) computations and the binary operator $*$ is used to extract the results of computations.[3] Monad E can be defined by taking $\mathsf{E}(D) = D \oplus \mathbf{U}$ where $\mathbf{U}$ is a singleton domain, whose element represents errors.

Using monad E to allow static errors, the static meaning of declarations can be defined as a function of the domain:

$$\blacktriangleright \quad \{\!|\mathit{declaration}|\!\} \;:\; \mathbf{Ent} \to \mathsf{E}(\mathbf{Ent})$$

that is, a function which takes as argument an initial type environment and returns an updated type environment, which contains information about the declared identifiers. The equations defining such functions are often very complex but can be somewhat simplified by the use of several auxiliary operations, defined separately.

The meaning of recursively defined types requires special treatment. In a way similar to the one suggested in [2], a monadic closure operator is used, defined as:

$$\mathrm{mclo}\; z\; f \;=\; \mathbf{clo}\; z\; (\lambda\, x.\; x \;*\; f) \;=\; \bigsqcup_{n=0}^{\infty} (\lambda\, x.\; x \;*\; f)^n\; z$$

on condition that f is continuous and $z \sqsubseteq z \;*\; f$. This operator is applied to the initial type environment and the static meaning of a list of declarations, in order to obtain a least upper bound for recursively defined types.

[3] This description, although naïve, is sufficient for the purpose of this chapter.

Figure 3: Some of the phrase types.

Phrase type	**Description**
$\mathrm{exp}\,[v]$	Expression, whose result is a non-constant r-value of type v.
$\mathrm{lvalue}\,[m]$	Expression, whose result is an l-value of type m.
$\mathrm{val}\,[\tau]$	Expression, whose result is a constant r-value of type τ.
$\mathrm{arg}\,[p]$	Actual arguments of a function with prototype p.
$\mathrm{stmt}\,[\tau]$	Statement in a function returning a result of type τ.
decl	Declaration.

Figure 4: Typing judgements.

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 $\mathbf{T}$.
$v := z$	The static semantic valuation $z : \mathsf{E}(D)$ produces the (non-error) value $v : D$.

Judgements related to environments.

$e \vdash I \lhd \delta$	Identifier I is associated with identifier type δ in type environment e.
$\pi \vdash I \lhd m$	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.

4. Typing semantics

The primary aim of typing semantics is the association of program phrases with phrase types. Fig. 3 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 order to simplify the typing rules. A summary of the most important ones is given in Fig. 4. Approximately 200 typing rules are used in our approach, in order to specify the typing semantics of C. 70% of those deal with expressions, 10% with statements and the remaining 20% with declarations. In the rest of this section we will present some non-trivial examples of typing rules and we will conclude with a discussion on 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 $\mathrm{val}\,[\tau]$, where τ is the first integer type that can represent their value. According to E2, identifiers that have been defined as normal variables in an e are l-values of the appropriate type. Similarly, rules E3 and E4 specify the typing semantics of function calls and the indirection operator.

$$\frac{\begin{array}{c}\text{suffix(n)} = \varnothing \qquad \text{isDecimal(n)} \\ \tau := \text{repType}(n, [\,\text{int}, \text{long-int}, \text{unsigned-long-int}\,])\end{array}}{e \vdash n : \text{val}[\tau]}\ (E1) \qquad \frac{e \vdash I \lhd \text{normal}[\alpha]}{e \vdash I : \text{lvalue}[\alpha]}\ (E2)$$

$$\frac{e \vdash E : \text{exp}[\text{ptr}[\text{func}[\tau, p]]] \qquad e \vdash \textit{args} : \text{arg}[p]}{e \vdash E\,(\textit{args}) : \text{exp}[\tau]}\ (E3) \qquad \frac{e \vdash E : \text{exp}[\text{ptr}[\alpha]]}{e \vdash *E : \text{lvalue}[\alpha]}\ (E4)$$

The following two rules specify in part the semantics of assignments. According to 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'.

$$\frac{\begin{array}{c}e \vdash E_1 : \text{lvalue}[m] \qquad \text{isModifiable(m)} \\ \tau := \text{datify } m \qquad e \vdash E_2 \gg \tau\end{array}}{e \vdash E_1 = E_2 : \text{exp}[\tau]}\ (E5) \qquad \frac{\begin{array}{c}e \vdash E : \text{exp}[\tau] \\ \text{isArithmetic}(\tau) \qquad \text{isArithmetic}(\tau')\end{array}}{e \vdash E \gg \tau'}\ (A1)$$

A small number of typing rules specifies conversions that take place implicitly in the evaluation of expressions. Such conversions are called implicit coercions. For example, rule C1 states that l-values are implicitly converted to the values stored in the designated objects. Even more obviously, according to rule C2 constant values may be treated as normal non-constant expressions.

$$\frac{e \vdash E : \text{lvalue}[\text{obj}[\tau, q]] \qquad \text{isComplete}(\tau)}{e \vdash E : \text{exp}[\tau]}\ (C1) \qquad \frac{e \vdash E : \text{val}[\tau]}{e \vdash E : \text{exp}[\tau]}\ (C2)$$

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, containing its declarations, and therefore a new environment e' must be calculated, allowing for recursively defined structures or unions. This new environment e' is used for the typing of the compound statement's body. Rules S2 and S3, specifying the semantics of `while` and `return` statements respectively, are relatively easier. In S3, the type of the returned expression must be assignable to the function's returned type.

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

$$\frac{\begin{array}{c}e \vdash \textit{expr} : \text{exp}[\tau'] \qquad \text{isScalar}(\tau') \\ e \vdash \textit{stmt} : \text{stmt}[\tau]\end{array}}{e \vdash \texttt{while (}\ \textit{expr}\ \texttt{)}\ \textit{stmt} : \text{stmt}[\tau]}\ (S2) \qquad \frac{e \vdash \textit{expr} \gg \tau}{e \vdash \texttt{return}\ \textit{expr}\ \texttt{;} : \text{stmt}[\tau]}\ (S3)$$

The suggested typing semantics for C leads to two forms of ambiguity problems. The first concerns the uniqueness of typing results: the main typing relation does not always provide a unique phrase type for a given program phrase. (Implicit coercion rules such as C1 and C2 are one source of such ambiguities.) This form of ambiguity is in fact useful. A given program phrase can be attributed different phrase types, depending on its role in the program, and a different dynamic semantic meaning may exist for each phrase type. The second form concerns the uniqueness of typing derivations for a given typing judgement.

Figure 5: Dynamic semantic domains.

Auxiliary domains and environments.

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

Domains for types.

$[\![\mathrm{void}]\!]_{dat} = \mathbf{U}$, $[\![\mathrm{int}]\!]_{dat} = \mathbf{N}$, $[\![\mathrm{ptr}\,[\alpha]]\!]_{dat} = \mathbf{Addr} \oplus \mathbf{U}$, $[\![\mathrm{ptr}\,[f]]\!]_{dat} = \mathbf{Fun} \oplus \mathbf{U}$

$[\![\mathrm{obj}\,[\tau, q]]\!]_{obj} = \mathbf{Addr}$, $[\![\mathrm{array}\,[\alpha, n]]\!]_{obj} = \mathbf{N} \to [\![\alpha]\!]_{obj}$

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

$[\![\alpha]\!]_{mem} = [\![\alpha]\!]_{obj}$, $[\![\mathrm{bitfield}\,[\tau, q, n]]\!]_{mem} = \mathbf{Addr} \times \mathbf{BitOfs}$

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 using C3, while the second coerces the summands separately using C3 and adds the resulting expressions. It is required that all different possible derivations for a given typing judgement result in the same dynamic semantics for the program phrase.

5. Dynamic semantics

The dynamic semantics of C specify the execution behaviour of well-typed programs. As a useful side-effect, run-time errors and other sources of undefined behaviour are detected. 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. 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.[4] The dynamic semantics is further perplexed by pointer arithmetic, complex control statements like `for` and `switch`, and the presence of `goto` in combination with block scopes containing variable declarations.

For each static type, a dynamic semantic domain is defined for representing the dynamic meaning of values of this type. The definitions of some dynamic semantic domains are shown in Fig. 5. Among other things, the domain of integer numbers $\mathbf{N}$ 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) objects containing them, in order to correctly model pointer arithmetic. The domain for type environments $[\![e]\!]_{Ent}$ contains the dynamic meanings of all identifiers defined in e, while that for function prototypes contains the values of all parameters. Domain $\mathbf{Cod}$ contains the code of all defined functions and $\mathbf{Lab}_\tau$ contains the meanings of labeled statements in a function returning a result of type τ and is used in the semantics of jumps.

A number of monads is used in order to represent various aspects of the computations that are related to the execution of C programs. Brief descriptions of these monads are

[4] This issue has been discussed a lot in `comp.std.c`. Several opinions have been expressed but no conclusion has been reached.

Figure 6: Monads used in the dynamic semantics.

Auxiliary monads and monad transformers.

$\mathsf{P}(D)$	Powerdomain monad, allowing for non-determinism in values of domain D.
$\mathsf{R}(M)(T)$	Resumption monad transformer, allowing interleaving in computations of type $M(T)$. Defined as a solution of the equation $\mathsf{R}(M)(T) = T \oplus M(\mathsf{R}(M)(T))$.

Monads for computations.

$\mathsf{V}(T)$	Computation of a constant value of type T, not accessing the memory (value monad).
$\mathsf{C}(T)$	Computation of a non-constant value of type T, possibly accessing the memory (continuation monad). Defined as $\mathsf{C}(T) = (T \to \mathbf{C}) \to \mathbf{C}$, where $\mathbf{C} = \mathbf{S} \to \mathsf{P}(\mathbf{A})$ is the domain of non-deterministic continuations, $\mathbf{S}$ is the domain of program states and $\mathbf{A}$ the domain of final program answers.
$\mathsf{G}(T)$	Computation of a non-constant valye of type T in the evaluation of an expression, possibly accessing the memory and allowing for interleaving. Defined as $\mathsf{G}(T) = \mathsf{R}(\mathsf{C})(T)$.
$\mathsf{K}_\tau(T)$	Computation of a non-constant value of type T in the execution of a statement, possibly accessing the memory or terminating the function by returning a result of type τ.

shown in Fig. 6. The powerdomain monad, based on the convex powerdomain, is used to model non-determinism. The resumption monad transformer is used to model the interleaving in the evaluation of expressions, that is required by C's unspecified order of evaluation.[5]

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 semantic meanings of both the defined phrase and its components. Thus, typing derivations control the definition of dynamic semantics, instead of abstract syntax, and there is one dynamic equation for each typing rule. In the rest of this section we will illustrate the definition of dynamic semantics by presenting some small examples.

Let us consider the simple case of typing rule E4 in §4. The dynamic semantics for an expression of the form "`*`E" under the typing given in E4 can be defined as follows:

▶ $[\![\exp[v]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{G}([\![v]\!]_{val})$

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

The first line states that the dynamic semantic meaning for phrases of type $\exp[\tau]$ is a function taking as arguments the static and dynamic environments and the code environment and returning an interleaved computation with a result of type $[\![\tau]\!]_{dat}$. Notice the dependent function type, denoted here as $x : A \twoheadrightarrow B(x)$, which dictates a connection between the static and the dynamic environments. The equation that follows defines $[\![*E]\!]_{\mathrm{lvalue}\,[\alpha]}$ in terms of $[\![E]\!]_{\exp[\mathrm{ptr}\,[\alpha]]}$. If the pointer contains an object's address, this address is used in the resulting l-value. An error occurs if the pointer is null. Dynamic semantic meanings for phrases of type $\mathrm{lvalue}\,[m]$ are functions of the form:

▶ $[\![\mathrm{lvalue}\,[m]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathsf{G}([\![m]\!]_{mem})$

[5] The resumption monad transformer is defined in detail in a paper that will be submitted for publication in the near future. For an introduction to monad transformers, the reader is referred to [13].

In a similar way, the following equations define the dynamic semantics that correspond to typing rules E3 and C1 of §4 respectively.

$$
\begin{array}{l}
[\![E\,(\mathit{args})\,]\!]_{\mathrm{exp}\,[\tau]} \;=\; \lambda\, e.\, \lambda\, \rho.\, \lambda\, \xi. \\
\quad [\![E]\!]_{\mathrm{exp}\,[\mathrm{ptr}\,[\mathrm{func}\,[\tau,p]]]}\; e\; \rho\; \xi \;\bowtie\; [\![\mathit{args}]\!]_{\mathrm{arg}\,[p]}\; e\; \rho\; \xi \;*\; (\lambda\, \langle d_e, d_p \rangle. \\
\quad \mathrm{seqpt} \;*\; (\lambda\, u.\; \mathbf{case}\; d_e\; \mathbf{of} \\
\qquad \mathbf{inl}\; d_f \quad\;\; \Rightarrow\; \mathbf{let}\; \langle f, b_f \rangle = \xi[d_f]\; \mathbf{in}\; \mathrm{isCompatible}(f, \mathrm{func}\,[\tau,p]) \to b_f\; d_p\;,\; \mathrm{error} \\
\qquad \mathbf{otherwise}\; \Rightarrow\; \mathrm{error}))
\end{array}
$$

$$
[\![E]\!]_{\mathrm{exp}\,[\tau]} \;=\; \lambda\, e.\, \lambda\, \rho.\, \lambda\, \xi.\; [\![E]\!]_{\mathrm{lvalue}\,[\mathrm{obj}\,[\tau,q]]}\; e\; \rho\; \xi \;*\; \mathrm{getValue}_{\mathrm{obj}\,[\tau,q] \mapsto \tau}
$$

The first thing to notice is the use of operator $\cdot \bowtie \cdot : \mathsf{G}(A) \times \mathsf{G}(B) \to \mathsf{G}(A \times B)$ for the interleaving of two computations, in order to model the unspecified order in which the function's designator and its arguments are evaluated. The second is the use of $\mathrm{seqpt} : \mathsf{G}(\mathbf{U})$ to introduce a sequence point just before the function is actually called. The function's dynamic meaning is looked up in the code environment. Furthermore, the function's actual type must be compatible with the type of the designator that is used. In the second equation, the important point is the use of $\mathrm{getValue}_{m \mapsto \tau} : [\![m]\!]_{mem} \to \mathsf{G}([\![\tau]\!]_{dat})$ which retrieves a stored value from memory.

The dynamic semantics of statements is defined in a similar way. We give here two relatively simple examples, corresponding to the typing rules S2 and S3 of §4. Notice that the dynamic meaning of statements uses also a label environment of type $\mathbf{Lab}_\tau$. The calculation of this environment requires a least fixed point, to allow infinite loops implemented by `goto` statements.

▶ $[\![\mathrm{stmt}\,[\tau]]\!] \;:\; e : \mathbf{Ent} \twoheadrightarrow [\![e]\!]_{Ent} \to \mathbf{Cod} \to \mathbf{Lab}_\tau \to \mathsf{K}_\tau(\mathbf{U})$

$$
\begin{array}{l}
[\![\texttt{while (}\mathit{expression}\texttt{)}\ \mathit{statement}]\!]_{\mathrm{stmt}\,[\tau]} \;=\; \lambda\, e.\, \lambda\, \rho.\, \lambda\, \xi.\, \lambda\, \ell.\; \mathbf{fix}\; (\lambda\, g. \\
\quad \mathrm{lift}_{\mathsf{G} \to \mathsf{K}}\; ([\![\mathit{expression}]\!]_{\mathrm{exp}\,[\tau']}\; e\; \rho\; \xi) \;*\; (\lambda\, d. \\
\quad \mathrm{checkBoolean}_{\tau'}(d) \to \\
\qquad \mathrm{setBreakContinue}\; \langle \mathrm{unit}\; \mathbf{u}, g \rangle\; ([\![\mathit{statement}]\!]_{\mathrm{stmt}\,[\tau]}\; e\; \rho\; \xi\; \ell) \;*\; (\lambda\, u.\, g)\;,\; \mathrm{unit}\; \mathbf{u}))
\end{array}
$$

$$
[\![\texttt{return}\ \mathit{expression}\texttt{;}]\!]_{\mathrm{stmt}\,[\tau]} \;=\; \lambda\, e.\, \lambda\, \rho.\, \lambda\, \xi.\, \lambda\, \ell.\; \mathrm{lift}_{\mathsf{G} \to \mathsf{K}}\; (\mathcal{A}[\![\mathit{expression}]\!]_{\mathrm{exp}\,[\tau']}\; e\; \rho\; \xi) \;*\; \mathrm{result}
$$

The first equation uses the least fixed point operator **fix** to model the semantics of the `while` statement. The correct continuations that will be used in case of a `break` or `continue` statement are passed to the body of the loop by using $\mathrm{setBreakContinue}$. The second equation defines the semantics of the `return` statement. The expression is evaluated and converted as if by assignment to the function's return type. Notice the use of $\mathrm{result} : [\![\tau]\!]_{dat} \to \mathsf{K}_\tau(\mathbf{U})$ which signals the termination of this function and specifies the returned result.

6. Evaluation

A significant effort has been made to evaluate our approach in defining a denotational semantics for the C programming language. In this task, the major issue was to assess how complete and accurate the developed semantics is. Unfortunately, there is no systematic way to evaluate our approach and be absolutely certain that the results are correct: 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 some test suites for C implementations that were available.

An earlier version of our semantics was first implemented using SML as the implementation language. Later, SML was abandoned and Haskell was used instead, mainly because it has a richer type system, more flexible syntax, elegant support for monads and also because lazy evaluation avoids a number of non-termination problems. The current implementation consists of approximately 15,000 lines of Haskell code, which are distributed roughly as follows: 3,000 lines for the static semantics, 3,000 lines for the typing semantics, 5,000 lines for the dynamic semantics, 3,000 lines for parsing and pretty-printing and 1,000 more lines of general code and code related to testing. As it was expected, the implementation is very slow and this presents a serious handicap in our yet unfinished evaluation process, significantly limiting the size of test programs.

Although the evaluation of our semantics is still under way and minor bugs are waiting to be fixed, the results indicate that the developed semantics is complete and accurate to a great extent, with respect to the ANSI C standard. The most important deviations from the standard are that the developed semantics requires function prototypes to exist for all called functions, something already favoured by the current standard, and that storage specifiers other than `typedef` are currently ignored. Static variables may be preprocessed out, but a solution integrated in the semantics is currently investigated. As less important deviations, the developed semantics requires fully bracketed initializations and forbids the declaration of identifiers, other than labels, in expressions or statements.

7. Conclusion and future work

In this chapter, 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. A significant contribution of our research, besides the developed semantics itself, is the application of monads and monad transformers for the specification of a real programming language. Furthermore, interesting results have been achieved in our attempt to model the interleaving of computations and non-determinism using monads and monad transformers, 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. 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] American National Standards Institute, New York, NY. *ANSI/ISO 9899-1990, American National Standard for Programming Languages: C*, 1990. Revision and redesignation of ANSI X3.159-1989.

[2] R. Sethi. A case study in specifying the semantics of a programming language. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 117–130, January 1980.

[3] R. Sethi. Control flow aspects of semantics-directed compiling. *ACM Transactions on Programming Languages and Systems*, 5(4):554–595, October 1983.

[4] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In E. Börger et al., editors, *Selected Papers from CSL'92 (Computer Science Logic)*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308. Springer Verlag, New York, NY, 1993.

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

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

[7] M. Norrish. An abstract dynamic semantics for C. Technical Report TR-421, University of Cambridge, Computer Laboratory, May 1997.

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

[9] R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, August 1976.

[10] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 11, pages 577–631. Elsevier Science Publishers B.V., 1990.

[11] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, Laboratory for Foundations of Computer Science, 1990.

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

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