An abstract dynamic semantics for C

Michael Norrish
Computer Laboratory, University of Cambridge

May 6, 1997

Abstract

This report is a presentation of a formal semantics for the C programming language. The semantics has been defined operationally in a structured semantics style and covers the bulk of the core of the language. The semantics has been developed in a theorem prover (HOL), where some expected consequences of the language definition have been proved.

Contents

1 Introduction

This work presents a formal description of the C language [1] in the tradition of the formal definition of SML [17]. It is significant in two important ways. Firstly, it is a demonstration that a language as complicated and as inherently unco-operative as C can nonetheless be treated formally. Secondly, the model has been mechanised in the HOL theorem prover [10]. This provides the theorem proving community with a demonstration that large semantic descriptions are both possible, and that they can be used for the basis of further reasoning. It also suggests that mechanical work on real world verification problems may not be the practical impossibility that common wisdom suggests.

The C programming language was first fully defined in 1989, in the ANSI standard [1], subsequently adopted as an ISO standard [15]. However it had been in existence for a long time prior to that date. The very effort of standardisation was made "to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of computing systems" [1, abstract], an effort necessary because of the danger of diverging language implementations blurring exactly what was and was not a C program. However, the standard also makes admirable efforts to accommodate practice as it existed at the time.

1.1 Motivation

Why C? Formal language semantics have been developed in the past, notably the specification of the language SML in [17]. Such formalisations can be very valuable exercises in their own right, and numerous languages have been studied in this light. Nonetheless, the question remains as to why C is a worthwhile object of study.

The answer to this question has two facets. Pragmatically, a formal specification of C has the potential to be very useful. C is a very widely used language: it was used to write operating systems such as Unix, OS/2, and Windows; applications such as Netscape and Microsoft Word; and it is used in fields from avionics to compiler construction. Though formal verification is still an embryonic field as far as industry is concerned, tools and techniques to perform verification (with the advantages that this brings) must necessarily build on a fully formal semantics.

On the theoretical front, C combines a number of interesting features, the study of which is a worthwhile endeavour in its own right. Firstly, C's expressions have side effects. Although one would like to be able to consider programming language expressions as if they were the mathematical idealisations that they resemble, a C expression such as `f(3) + g(5)` is able to do more than simply compute a value, but may alter the contents of memory as well. Many theoretical treatments of verification in the past have preferred to deal with languages where expressions are side-effect free, so there is still considerable interest in the development of methods for dealing with side-affecting expressions.

C is also subject to the problem of non-determinism. For certain forms within the language, the standard does not specify exactly how these are to be executed, but merely requires that execution must satisfy certain constraints. Thus an implementation is given latitude to choose from a number of possible choices. This "under-determinedness" of C is not a mere curiosity confined to the inner workings of a C program, but is something that can manifest itself in the program's external behaviour. Therefore, if one is to prove that a program's behaviour is to satisfy certain constraints, then one will need to verify that this behaviour happens for all possible variations in the execution of the program.

Why in a theorem prover? Theorem provers are also a very useful way of organising logical material that might otherwise be overwhelming in its complexity. A user might forget the niggling flaw in the definition made six months ago, but the machine will not. One can be confident that a theorem proved is one which will not be later revealed to have some distressing hole in its proof.

C is a complicated language and embedding its semantics in the HOL theorem prover has kept my theorems correct, at least with respect to my definitions. What's more, the same theorem proving support has ensured that many flaws in my definitions have been detectable. For example, if the standard leads one to expect a certain conclusion but the proof of this purported theorem does not succeed, it might be because it has not been properly expressed, or it may be because the definitions it relies upon are wrong themselves.

2 The C language

2.1 What has been omitted

C's specification divides into two parts: the language and the library. The library is as carefully specified as the language, and a logical specification of its functions' behaviour would be a necessary prerequisite to any useful work verifying C programs in practice. Nonetheless, it is possible to get a good understanding of the way in which C programs work by simply studying the language core, as the library functions specified in the standard are on the whole concerned with interacting with the operating system, or are utility functions that one could imagine defining oneself. Therefore, this semantics concerns itself just with the C language and not the library.

Perhaps more significantly, this semantics omits `switch` and `goto` statements. Both statement forms would require a fairly significant effort to incorporate into the semantics, as each would require a traversal of a statement's abstract syntax in the search for a particular label to jump to.[1] Though this could be implemented, it has remained at a fairly low priority so far.

Less significantly, I omit qualified types (types qualified with the keywords `const` and `volatile`), union types, string constants and bit-fields within `struct`s.

It is also worth pointing out that the semantics does not model the pre-processor, but assumes that it is presented with an abstract syntax tree after the initial phases of the pre-processor have occurred. It also makes the convenient (though strictly speaking, incorrect) assumption that `typedef`s[2] are little more than a form of pre-processing, and that they have been "compiled out", leaving only basic and constructed types. Further, there are straightforward equivalences defined in the standard for the `->`, `[]`, pre-increment, and pre-decrement operators. I choose not to model these explicitly, but assume that they have been replaced at the syntactic level.[3]

What follows is what I term an abstract dynamic semantics. It does not concern itself unduly with the type correctness of the program (a static issue), it tries to treat memory as abstractly as possible, and it does not specify the behaviour of the basic operators (`+`, `-`, &c). In treating memory abstractly, I follow the lead

[1] Furthermore, this traversal and search is not an entirely straightforward process, as it must be prepared to allocate or deallocate space for variables if the label found is within a different block.

[2] C's `typedef` declarations establish transparent type aliases.

[3] The equivalences are: `rp->fld` is the same as `(*rp).fld`, `a[i]` is the same as `*(a + i)`, `++i` is the same as `i += 1`, and `--i` is the same as `i -= 1`.

of the standard itself. By glossing over the behaviour of the operators, I am able to make a presentation of the core semantics with an elegant separation of concerns.

The semantics defined here is a subset of that defined in HOL. The material omitted is precisely those aspects of the semantics which are not abstract in the sense used above. Thus the HOL semantics does include the operator definitions that this article omits. The HOL semantics is called Cholera, and I will use that name to refer to the semantics presented here as well. Where the distinction needs to be made, I will clarify whether I mean the abstract or full semantics, but what is true of one is typically true of the other.

2.2 Basic definitions

2.2.1 Degrees of under-specification

The standard under-specifies the semantics of the C language in three different ways. Each of the three is described below, and the way in which each form of behaviour is dealt with in Cholera is sketched (*c.f.* the glossary in[1, §1.6]):

Implementation-defined: An implementation-defined construct is one which must have a definite meaning, but one for which the standard has passed responsibility to the implementation. The implementation is required to document its choice of meaning. An example is the byte-ordering within multi-byte numeric objects (big-endian *vs* little-endian).

Cholera handles implementation-defined behaviour by defining, but under-specifying some constant. In this way, it models the fact that there is *a* well-defined behaviour, but makes it impossible for a user to rely on it being a *particular* behaviour. For example `CHAR_BIT`, the number of bits in a byte, is defined to be at least eight, but this is all one can rely upon.

Unspecified: A construct or program form for which behaviour is unspecified is one where the standard imposes no requirement. For example, the order of evaluation of expressions is unspecified. Here an implementation need not document its behaviour, and thus may choose to do different things in quite similar, if not identical, situations.

Cholera handles unspecified behaviours by always allowing all possible behaviours. In the case of expression evaluation, all possible evaluation orders can arise. One can not then claim that a program's behaviour will have a particular result without confirming that all possible behaviours lead to the same result.

Undefined: Undefined behaviour results when a program attempts to do something which is semantically invalid. Cholera treats all such behaviour as equivalent to a transition into a special state where no further action takes place. In implementations, a program which attempts an undefined behaviour will in all likelihood do something, and this something may in fact be quite reasonable. Nonetheless, there is no way of relying on undefined behaviour to do anything in particular, so the Cholera approach of effectively aborting it as soon as it occurs is safest.

Undefined behaviour occurs when uninitialised memory is accessed, when a null pointer is dereferenced, when a side effect attempts to update a memory object which has already been accessed in the same phase of expression evaluation, and in a host of other situations.

2.2.2 Program states and side effects

Ultimately, a semantics gives a program meaning by describing the way in which it transforms a "program state". The state in Cholera models the state of an abstract computer on which the described program is being run. In its simplest form, a model of program state would necessarily include a description of the abstract computer's memory, but in Cholera it proves necessary to include rather more than just memory.

If a formal semantics is to retain the connection between the high level syntax of the language and what happens to memory, then it must also include the sort of details that one might at first associate with compiler symbol tables. The types of variables, the fields that go to make up `struct` types, and the mapping from variable names to locations in memory must all be recorded. In the literature, these details make up what is commonly known as the *environment*. In the semantics that follows, I shall loosely use the term "state" to refer to both the environment and the contents of memory. After all, both can be seen to change as a program executes.

In C, changes to memory come about solely through the action of side effects. These are created as the result of evaluating certain expression forms, principally assignment expressions. However, side effects are not applied immediately upon their creation, but can be kept pending for a while. Nor do multiple side effects have to be applied in order. The motivation for this is that implementations should have licence to make changes to memory in ways that are convenient for them. This licence may allow useful optimisations to be performed, for example.

Cholera models this state of affairs by keeping a bag of pending side effects as part of the state. As the side effects are generated, they are put into the bag. The bag is emptied in a non-deterministic order, and at non-deterministic times, subject only to the constraint that it must be empty when what is known as a *sequence point* is encountered. Sequence points occur in certain syntactically well-defined places, and will be flagged in the description that follows.

Finally, the standard imposes an important restriction on the way in which side effects can be applied. Sequence points divide an execution up into a series of consecutive phases. A piece of memory that is updated in a given phase may not

μ	conversion to memory value
φ	conversion to fn. reference value

Table 1: state independent functions

α	address of variables	τ	typing function
ν	retrieves values from memory	o	operator semantics
ρ	memory references made		
$\varphi\alpha$	argument details about a function	$\varphi\beta$	body of a function
ξo	structure field offsets	$\xi\tau$	structure field types

Table 2: state functions

be updated again or examined within the same phase, otherwise the behaviour is undefined. The only exception to this rule allows an examination of an object if its value is to be used in an update of the same object. These rules forbid expressions such as `i++ + i` (a reference and an update in the same phase), and `i = i++ + 4` (two updates of the same object), but allow `i = i + 1` (`i` is both updated and referred to, but the reference is part of the computation of `i`'s new value).

These restrictions have an important rôle in expression evaluation.

2.2.3 Notation—general observations

Each section of the semantics defines its own operators and miscellaneous notation, but there are a number of general observations about notation that can be made here. Each section of the semantics presents an "arrow" relation of the form $\langle v_0, \sigma_0\rangle \to \langle v, \sigma\rangle$, where σ_0 and σ are the initial and final states, and where v_0 is the form of C syntax being defined. The nature of v depends on the precise relation being defined.

The following tables summarise the functions used in the rule definitions.

Table 1 shows functions used to abstract away the details of value representation in the abstract computer's memory. Both functions are left to the implementation to define, subject to various constraints. For example, implementations have licence to use one's complement as a representation for signed numbers. This decision is abstracted away in the use of μ.

Table 2 "state functions", which calculate commonly needed values from the current state of the program. These functions appear as superscripts to state variables.

Finally, Table 3 shows those functions which are used to modify state values.

`add_se`	adds a side effect to the state
`apply_se`	applies a pending side effect
`clear_se`	clears side effect records
`decl_var`	declares a variable, allocating space and type information
`init_decl_var`	as with `decl_var` but also gives an initial value
`install_params`	installs parameters in a function call
`mark_ref`	records references to memory
`mem_trans`	copies memory component of a state onto another
`remove_refs`	removes references to memory
`struct_decl`	declares a new `struct` type

Table 3: state modification functions

2.3 Expression evaluation

Expression evaluation uses what is commonly called a reduction or "small-step" semantics. The relevant relations are $\rightarrow_e$, and its transitive closure $\rightarrow_e^*$. The action of $\rightarrow_e$ can be seen as a gradual transformation of a piece of syntax into a value. Of course, for this relation to be properly typed, the syntactic forms have to be extended to include C values, as these are not valid pieces of C syntax.

What is a C value? In the standard, a normal value is essentially a finite sequence of bytes. A value has a type, and if numeric can be treated as a number. All values of the same type take up the same amount of space. Values may be *aggregate*, (i.e., contain other values), and a change to a constituent value is considered a change to the whole. In this report, values are emphasised by underlining, and are accompanied by their type. C refers to values resident in memory as *objects*. A second form of value, the *lvalues*, also play an important part in the semantics. An lvalue is essentially a reference to an object, and so may often "decay" into that object's value.

2.3.1 Extra syntactic forms and undefined expressions

In the course of performing a small-step expression evaluation, it is necessary to introduce new intermediate forms that do not correspond to any genuine piece of C syntax. These intermediate forms are used to signal to the abstract machine that a certain stage has been reached.

For example, the short circuiting logical operators (`&&` and `||`) use an intermediate form corresponding to the stage of evaluation where the first argument has been evaluated, but the second has not yet finished. Cholera's treatment of lvalues also requires two new forms. These are Λ and R. All of these new forms will be explained further when the rules that use them are discussed.

Finally, we need to add an undefined value to the semantics. When an expression does something that the standard classes as undefined behaviour, then the expression in question reduces to a special form (written $\mathcal{U}$). Once created, this will subsequently bubble its way up to the root of the syntax tree.

2.3.2 Expression evaluation contexts

Following the example of other presentations of reduction semantics (e.g., [8], and ultimately [7]) Cholera's expression semantics makes use of evaluation contexts. Informally, an evaluation context is a piece of syntax "with a hole in it". Contexts provide a convenient way to generalise a whole family of evaluation rules where independent reductions occur in sub-expressions. The following rule, using the $\mathcal{E}$ context, captures most of the ways in which evaluation can proceed in the sub-terms of an overall expression:

$$\frac{\langle e_0, \sigma_0 \rangle \rightarrow_e \langle e, \sigma \rangle}{\langle \mathcal{E}[e_0], \sigma_0 \rangle \rightarrow_e \langle \mathcal{E}[e], \sigma \rangle}$$

This rule should be read as "If expression e_0 can reduce to e, altering state σ_0 to σ in the process, then the expression formed by inserting e_0 into the hole of the context $\mathcal{E}$ can start in σ_0 and reduce to the corresponding expression with e in place of e_0, finishing in state σ." The allowed forms for $\mathcal{E}$ are:

$$\begin{array}{rl} \mathcal{E}[-] ::= & - \odot e \mid e \odot - \mid \Box - \mid - \mathtt{\&\&}\ e \mid - \mathtt{||}\ e \mid - \mathtt{,}\ e \mid \\ & -\mathtt{++} \mid -\mathtt{--} \mid - \mathtt{.}\ f \mid - \odot\mathtt{=}\ e \mid - \mathtt{=}\ e \mid (t)- \mid R- \mid \\ & - \mathtt{?}\ e_1 \mathtt{:}\ e_2 \mid -(e_1, \ldots, e_n) \mid e(e_1, \ldots, -, \ldots e_n) \end{array}$$

Here $\odot$ stands for any of the standard binary operators (`+`, `*`, `|` etc.), and $\Box$ stands for any of the unary operators `-`, `!`, `~`, `&`, `*` and the intermediate form $\widehat{\mathtt{\&|}}$. Finally, the t is a type in the type-cast expression, and the f is the field name of some `struct` type.

The sequenced behaviour of the logical operators `&&` and `||`, as well as the comma operator and the `?:` operator is apparent here; only an evaluation that looks at the first argument is acceptable. Conversely, there are two contexts for the binary operators, giving rise to a significant non-determinism; the evaluation of the operands can be interleaved at all levels. In particular, evaluation of $(a + b) + (c + d)$ can see evaluation of the sub-expressions a, b, c and d occur in *any* order.

The $\mathcal{E}$ context is also used in the propagation of the $\mathcal{U}$ value.

$$\frac{}{\langle \mathcal{E}[\mathcal{U}], \sigma \rangle \rightarrow_e \langle \mathcal{U}, \sigma \rangle}$$

2.3.3 Base cases—values out of memory

In this section I present the expression evaluation relation's fundamental, "base case" rules. These are the rules which bring values out of memory and into syntax trees as they evaluate. Later sections define the ways in which values can be combined, and subsequently put back into memory.

Constants Cholera assumes that constants' types have already been statically determined. A value consists of two parts, a list of bytes and an accompanying type. The function μ takes a value and a type and returns an appropriate bit-pattern for that type.

$$\frac{}{\langle (n,t),\sigma\rangle \rightarrow_e \langle \underline{(\mu(n,t),t)},\sigma\rangle}$$

As before, the underlining indicates that the syntax on the right-hand side is actually a value.

Variables and lvalues A variable denotes a piece of memory and its contents. Cholera translates all variables into lvalues, using the following rule (the Λ lvalue-constructor takes the variable's address and the object's type as arguments):

$$\frac{}{\langle \mathsf{id},\sigma\rangle \rightarrow_e \langle \Lambda(\sigma^\alpha(\mathsf{id}),\sigma^\tau(\mathsf{id})),\sigma\rangle}$$

For any state σ, the function σ^τ returns the type of the variable.[4] Similarly, σ^α returns the address of a variable.

In many contexts, the semantics requires lvalues to become normal values. This is controlled by another relation, $\leadsto$. This relation can not be part of $\rightarrow_e$ because there are contexts in which we don't want to have the lvalue information disappearing (assignment, for one).

$$\frac{t \text{ not an array type}}{\langle \Lambda(n,t),\sigma_0\rangle \leadsto \langle \underline{(\sigma_0^\nu(n,t),t)}, \texttt{mark\_ref}(\sigma_0,n,t)\rangle}$$

For any state σ, $\sigma^\nu(n,t)$ returns the object at address n with type t. The specification of the type is necessary because objects in memory overlap, and one needs, for example, to be able to specify that one wants a field of a `struct` rather than the whole thing. The `mark_ref` function takes a state, an address and a type and returns a new state which differs from the first argument only in that it marks the piece of memory denoted by the combination of the address and the type as having been referred to.

[4] As all variables are declared before being used, we can use a function here, rather than a typing relation, as is done in SML where types are inferred.

Now, `mark_ref` is a partial function. It is undefined if the part of memory it attempts to mark as read has also been updated, or if it is uninitialised or unallocated. The corresponding rule is

$$\frac{\texttt{mark\_ref}(\sigma_0, n, t) = \mathcal{U} \quad t \text{ not an array type}}{\langle \Lambda(n,t), \sigma_0 \rangle \leadsto \langle \mathcal{U}, \sigma_0 \rangle}$$

The case when the lvalue is of an array type needs to be dealt with separately. C makes no provision for the manipulation of array values, and instead specifies that array lvalues are converted into pointers to their first element, which will necessarily share the same address. It is this rule which prompts the common (though inaccurate) comment to the effect that "arrays and pointers are the same in C".

$$\frac{}{\langle \Lambda(n, \texttt{Array}(t,m)), \sigma \rangle \leadsto \langle \underline{(\mu(n, \texttt{Ptr}\,t), \texttt{Ptr}\,t)}, \sigma \rangle}$$

The only issue to resolve is when the $\leadsto$ relation should be allowed to "fire". This is done with another evaluation context, called $\mathcal{L}$. $\mathcal{L}$ is the same as $\mathcal{E}$ except that it omits both arguments of assignment expressions[5], the arguments of `++` and `--`, the unary ("address-of") `&` operator, and the field selection `.` operator. The rule in $\to_e$ is

$$\frac{\langle e_0, \sigma_0 \rangle \leadsto \langle e, \sigma \rangle}{\langle \mathcal{L}[e_0], \sigma_0 \rangle \to_e \langle \mathcal{L}[e], \sigma \rangle}$$

Note that this rule means that lvalues at the very top level of an expression evaluation will not convert into normal values. This is handled specially at the statement level. The context $\mathcal{L}$ is also unnecessarily conservative when it comes to array lvalues, as the standard calls for array lvalues to decay into normal values in even more contexts than non-array lvalues. In Cholera, this failing only means that the assignment operators need to check that their left hand sides are not of array type.

Structure values The field selection operator `.` pulls values out of memory through `struct` values. This operator is unique in that it can produce either lvalues or normal values depending on the nature of its first argument. When the operand is an lvalue, the rule is

$$\frac{}{\langle \Lambda(n, \texttt{Struct}\,s)\ .\ f, \sigma \rangle \to_e \langle \Lambda(n + \sigma^{\xi o}(s,f), \sigma^{\xi \tau}(s,f)), \sigma \rangle}$$

[5]While it should be clear why the left-hand side of the assignment should be immune to decay, it is less clear why the right-hand side needs this protection; this issue is addressed in section 2.3.5

Here $\sigma^{\xi o}$ returns the offset of fields within a `struct` type, and $\sigma^{\xi\tau}$ returns the type of fields within a `struct` type.

If the first argument to field selection is a normal value, then the result is also a normal value, created by copying out the list of bytes within the value corresponding to the field in question.

$$\frac{}{\langle \underline{(m, \texttt{Struct}\, s)}\ \texttt{.}\ f, \sigma\rangle \rightarrow_e \langle \underline{(m', \sigma^{\xi\tau}(s, f))}, \sigma\rangle}$$

where

$$\begin{array}{lcl} n & = & \texttt{sizeof}(\sigma^{\xi\tau}(s,f)) \\ m' & = & m[o \ldots o+n-1] \\ o & = & \sigma^{\xi o}(s,f) \end{array}$$

Pointers As we have already seen, the notion of pointers is fundamental to the way in which lvalues are manipulated. There are two pointer specific operators, dereferencing (`*`) and taking addresses (`&`). The first of these takes a pointer value and returns an lvalue, subject only to the constraint that the pointer not be a `(void *)`, a void pointer. Such a pointer is used as the generic pointer, and can be used to store pointer values of any type, but it can never be dereferenced.

$$\frac{t \neq \texttt{Void}}{\langle \texttt{*}\underline{(\mu(n, \texttt{Ptr}\, t), \texttt{Ptr}\, t)}, \sigma\rangle \rightarrow_e \langle \Lambda(n,t), \sigma\rangle}$$

The `&` operator is the inverse of this, requiring an lvalue as an argument, and returning a pointer value.

$$\frac{}{\langle \texttt{\&}(\Lambda(n,t)), \sigma\rangle \rightarrow_e \langle \underline{(\mu(n, \texttt{Ptr}\, t), \texttt{Ptr}\, t)}, \sigma\rangle}$$

Function reference values The C Standard promulgates the view that there is such a thing as a pointer to a function. This is a superficially appealing view as it enables the exploitation of a syntax shared with normal pointers. However, this view is extremely misleading semantically. The standard itself recognises this with both its constant need to distinguish between pointers to objects and pointers to functions, and with the admission that dereferenced pointers to functions and pointers to functions play exactly the same rôle.

It is much more fruitful to think of a pointer to function type as a variable of a function reference type. The constants of this type will be the functions defined in the program, and a special null value. In a C function call `f(x,y)`,`f` is either a function in the program called `f` (i.e., `f` is a constant of the function reference type), or `f` is a pointer to a function (a variable of the reference type, which will either be a reference to an existing function in the program, or an undefined value).

The Cholera semantics keeps pointers to objects and function references entirely apart.[6] The rule for function reference constants is

$$\frac{}{\langle f, \sigma\rangle \rightarrow_e \langle \varphi(f, \sigma^\tau(f)), \sigma\rangle}$$

where φ is a function that takes a function identifier and returns the bit pattern that will map to the function in question. Function "pointers" are dealt with by the rule for variables as they are no more than variables of the function reference type.

2.3.4 Value producing operators

The term "value producing operator" is meant to contrast with the side effect operators detailed in Section 2.3.5. Value producing operators generate new values, but do not modify the memory part of the program state. The first rule is the most general:

$$\frac{\odot \in B}{\langle \underline{(m_1, t_1)} \odot \underline{(m_2, t_2)}, \sigma\rangle \rightarrow_e \langle \underline{\sigma^o(\odot, (m_1, t_1), (m_2, t_2))}, \sigma\rangle}$$

The operator function σ^o calculates the effect of the given operator ($\odot$ above), returning both the value and the type of the result. This is a state-based function because the addition and subtraction operators admit the possibility of being applied to pointers. In this case the calculation of the correct values will require the size of a type; which in the case of `struct`s requires reference to information about the types of fields. The types of the operator's arguments are also required to allow the operator to perform appropriate conversions.[7] Finally, the operator function will return $\mathcal{U}$ if the operation is undefined (as in division by zero, for example). The set B includes all of the standard binary operators (addition, subtraction &c), but excludes `&&`, `||` and the `,` operator.

There is a very similar rule for the unary operators `~`, `!` and `-`:

$$\frac{\Box \in \{\texttt{~}, \texttt{!}, \texttt{-}\}}{\langle \Box\underline{(m, t)}, \sigma\rangle \rightarrow_e \langle \sigma^o(\Box, (m, t)), \sigma\rangle}$$

There is one further unary operator, the type-cast. This takes values and converts them to values of the specified type. This is not possible for all pairs of types, but where it is possible, the semantics is very easy to specify in the abstract.

[6] Strictly the standard calls for a third type of pointer, one to an incomplete type, but these are an artifact of the static semantics and do not feature at the dynamic level.

[7] For example, adding an `int` and a `long` will cause the `int` to be converted to a `long` and the result will also be a `long`

$$\frac{t_1 \text{ and } t_2 \text{ are conversion compatible}}{\langle (t_1)\underline{(\mu(x,t_2),t_2)}, \sigma\rangle \to_e \langle \underline{(\mu(x,t_1),t_1)}, \sigma\rangle}$$

Note that it is also possible for μ to return $\mathcal{U}$ when a value is not representable in a given type. (This can happen when attempting to convert `float` values to an integral type).

The rule for the comma-operator is the first to involve a sequence point. Before evaluation can proceed to the second argument of the expression, it must be the case that all pending side effects have been applied. Furthermore, when the evaluation then proceeds, the state must be updated to forget the references and updates made up to this point. This is because the restrictions on references and updates to the same object only hold between two consecutive sequence points. The `clear_se` function updates the state in this way.

$$\frac{\text{no pending side effects in } \sigma_0}{\langle \underline{(m,t)} \texttt{ , } e, \sigma_0\rangle \to_e \langle R(e), \texttt{clear\_se}(\sigma_0)\rangle}$$

The R constructor forces an expression to become a normal value. It is used here to prevent the result of the comma-operator expression from being an lvalue. The only rule for R is

$$\frac{}{\langle R\underline{(m,t)}, \sigma\rangle \to_e \langle \underline{(m,t)}, \sigma\rangle}$$

The logical operators: `&&` and `||` Because of the presence of their sequence points, and like the comma-operator, evaluation of both operators proceeds with a definite ordering imposed. The $\mathcal{E}$ and $\mathcal{L}$ contexts already make it clear that their first argument will evaluate as far as a normal value without the second argument being looked at. The following rules specify what happens next. First the short circuit cases, where the sequence points don't come into play because the first expression is able to determine the value of the whole expression on its own.

$$\frac{t \text{ is scalar} \quad m = \mu(0,t)}{\langle \underline{(m,t)} \texttt{ \&\& } e, \sigma\rangle \to_e \langle \underline{(\mu(0,\texttt{int}),\texttt{int})}, \sigma\rangle}$$

And for `||`:

$$\frac{t \text{ is scalar} \quad m \neq \mu(0,t)}{\langle \underline{(m,t)} \texttt{ || } e, \sigma\rangle \to_e \langle \underline{(\mu(1,\texttt{int}),\texttt{int})}, \sigma\rangle}$$

A type is scalar if it is *arithmetic* (i.e., numeric), or if it is a pointer. In the case where t is a pointer type, $\mu(0,t)$ is to return the null pointer value, which may not necessarily be a bit pattern corresponding to the number 0.

When the logical operators do not short-circuit, a sequence point is reached, and the evaluation must proceed with the second argument. The fact that this point has been reached is recorded through the use of the intermediate constructor $\widehat{\texttt{\&|}}$. The rule for `&&` in this situation is

$$\frac{\text{no pending side effects in } \sigma_0 \quad t \text{ is scalar} \quad m \neq \mu(0,t)}{\langle \underline{(m,t)}\ \texttt{\&\&}\ e, \sigma_0\rangle \to_e \langle \widehat{\texttt{\&|}}(e), \texttt{clear\_se}(\sigma_0)\rangle}$$

The rule for `||` is similar:

$$\frac{\text{no pending side effects in } \sigma_0 \quad t \text{ is scalar} \quad m = \mu(0,t)}{\langle \underline{(m,t)}\ \texttt{||}\ e, \sigma_0\rangle \to_e \langle \widehat{\texttt{\&|}}(e), \texttt{clear\_se}(\sigma_0)\rangle}$$

The rules for $\widehat{\texttt{\&|}}$ are

$$\frac{m = \mu(0,t) \quad t \text{ is scalar}}{\langle \widehat{\texttt{\&|}}\underline{(m,t)}, \sigma\rangle \to_e \langle \underline{(\mu(0,\texttt{int}),\texttt{int})}, \sigma\rangle}$$

$$\frac{m \neq \mu(0,t) \quad t \text{ is scalar}}{\langle \widehat{\texttt{\&|}}\underline{(m,t)}, \sigma\rangle \to_e \langle \underline{(\mu(1,\texttt{int}),\texttt{int})}, \sigma\rangle}$$

The conditional operator The conditional operator `?:` is ternary. Evaluation begins with the evaluation of its first argument, and then depending on the resulting value, the value of the whole expression is either the second or third argument. This value can not be an lvalue. The type of the result is the same regardless of which argument is chosen, and is calculated according to some relatively complicated rules, which I omit. I assume that the type t_c of the result has been calculated already, given the types of the two sub-expressions. The result of the sub-expression's evaluation is cast to this type using the type-cast operator.[8]

$$\frac{\text{no pending side effects in } \sigma_0 \quad t \text{ is scalar} \quad m \neq \mu(0,t)}{\langle \underline{(m,t)}\ \texttt{?}\ e_1 : e_2, \sigma_0\rangle \to_e \langle (t_c)e_1, \texttt{clear\_se}(\sigma_0)\rangle}$$

$$\frac{\text{no pending side effects in } \sigma_0 \quad t \text{ is scalar} \quad m = \mu(0,t)}{\langle \underline{(m,t)}\ \texttt{?}\ e_1 : e_2, \sigma_0\rangle \to_e \langle (t_c)e_2, \texttt{clear\_se}(\sigma_0)\rangle}$$

[8] Incidentally, this is the only place in the semantics where it proves necessary to know an expression's type without actually having to evaluate it.

2.3.5 Side effect operators

An expression that causes side effects will do so through the action of one of a limited set of operators. Before describing these operators, I'll first describe the way in which side effects alter the contents of memory. Side effects are denoted $\clubsuit(n, m)$, meaning that the value m is to be written to the address n.[9] The variables η, η_1 etc. are also used to denote side effects.

Recall that side effects are not applied immediately upon generation. Instead, they are "queued" in a bag of pending side effects. At any time, the abstract machine can pull a side effect from this bag, and apply it, updating memory with the new value. The rule for this is

$$\frac{\eta \text{ is pending in } \sigma}{\langle e, \sigma\rangle \rightarrow_e \langle e, \texttt{apply\_se}(\sigma, \eta)\rangle}$$

The `apply_se` function applies the side effect chosen, and also records the fact that the affected part of memory has been updated. The `apply_se` function is undefined if the update it selects from the bag tries to update a piece of memory that has already been updated or referred to:

$$\frac{\eta \text{ is pending in } \sigma \quad \texttt{apply\_se}(\sigma, \eta) = \mathcal{U}}{\langle e, \sigma\rangle \rightarrow_e \langle \mathcal{U}, \sigma\rangle}$$

The assignment operator(s) C defines both a traditional assignment operator (written `=`) and a number of compound assignments, where the action of the assignment is combined with that of a normal binary operator. The meaning of $e_1 \odot\!\!= e_2$ is defined to be the same as $e_1 = e_1 \odot e_2$ ($\odot$ a binary operator), except that the expression e_1 is only evaluated once (clearly important in the presence of side effects).

The Cholera semantics generalises these operators by defining rules for a general compound assignment. The simple assignment case is assumed to be a compound assignment where the operator is a special "right hand side operator" $\diamondsuit$. This is defined so that $e_1 \diamondsuit e_2 = e_1$.

Assignment expressions need to include an extra component, a bag containing a record of all the references made to memory in the course of the evaluation of an assignment expression's right hand side. Assignment expressions will thus be written as $e_1 \overset{\beta}{\odot\!\!=} e_2$, where β is the bag of references to memory, and $\odot$ is the binary operator compounded with the assignment. Furthermore, it is assumed that the bag in an assignment expression is always empty when an evaluation begins.

[9] The use of the club suit is meant to suggest that this is something that is about to clobber memory.

Finally, it is assumed that the right-hand side of the expression will be wrapped in an R constructor. This ensures that the expression there will decay to a value, but also that it will do so in a way that the following rule for assignment will be able to "see".

A naïve version of the semantics might do away with the R and extend the $\mathcal{L}$ to include the right-hand sides of assignment expressions. Unfortunately, this would allow the following transition, which leaves the bag β unchanged, to take place:

$$\frac{\overline{\langle \Lambda(n,t), \sigma_0 \rangle \rightsquigarrow \langle m, \sigma \rangle}^{\;\rightsquigarrow}}{\langle e_1 \overset{\beta}{\odot\!=} \Lambda(n,t), \sigma_0 \rangle \to_e \langle e_1 \overset{\beta}{\odot\!=} m, \sigma \rangle}\ \mathcal{L}\text{ context}$$

The correct rule controls the evaluation of the right hand side and monitors the way in which references to memory are made (recall the `mark_ref` function). Allowing the naïve version of the rules would give two rules scope over the situation, and the monitoring of references could be bypassed, as in the above example.

So, the first rule for assignment controls the way in which evaluation of the assignment's right hand side proceeds:

$$\frac{\langle e_2, \sigma_0 \rangle \to_e \langle e_2', \sigma \rangle}{\langle e_1 \overset{\beta}{\odot\!=} e_2, \sigma_0 \rangle \to_e \langle e_1 \overset{\beta'}{\odot\!=} e_2', \sigma \rangle}$$

where

$$\beta' = \beta + (\sigma^\rho - \sigma_0^\rho) - (\sigma_0^\rho - \sigma^\rho)$$

I use $+$ and $-$ above to represent the bag operations of addition and subtraction. The σ^ρ "function" represents the bag of recorded references to memory in the state σ. The expression for determining β' mimics the changes to the ρ component of the state. Thus, if the evaluation of the right hand side e_2 makes additional references β_1 such that $\sigma^\rho = \sigma_0^\rho + \beta_1$, then β' will equal $\beta + \beta_1$.

However, it is also possible that the reference maps of program states will decrease. This will happen every time a sequence point is encountered, for example. The β component of the assignment syntax is therefore a bag of references to memory made in the course of the evaluation of the right hand side, and which are still current.

The motivation for keeping track of these references is revealed in the next rule. Previously (in section 2.2.2) it was noted that the abstract machine is allowed to make references to memory that may seem to clash with an update, if the references were made in the service of calculating the updated object's new value. This is precisely what happens in an expression such as `i = i + 1`. This "permission to refer" is implemented in the Cholera model of the semantics by removing the appropriate references when the assignment takes place. The rule is

$$\frac{\langle (t_1)\,\sigma_0^o(\odot, (m_1, t_1), (m_0, t_0)), \sigma_0\rangle \to_e \langle \underline{(m,t)}, \sigma_0\rangle \quad t_1 \text{ not an array type}}{\langle \Lambda(n, t_1) \overset{\beta}{\odot\!=} \underline{(m_0, t_0)}, \sigma_0\rangle \to_e \langle \underline{(m,t)}, \sigma\rangle}$$

where

$$\begin{array}{lcl} m_1 & = & \sigma_0^\nu(n, t_1), \\ \sigma & = & \texttt{remove\_refs}(\sigma_1, \beta, n, t_1) \;\; \text{and} \\ \sigma_1 & = & \texttt{add\_se}(\sigma_0, \clubsuit(m, n)) \end{array}$$

The value $\underline{(m,t)}$ is found by applying the operator $\odot$ to the appropriate memory values, and then casting this to the correct type for the lvalue into which the value is going. The state σ_0 is modified by removing references in β to memory corresponding to the object assigned from the reference map, and by adding the updating side effect to those pending in σ_0.

The term $\texttt{remove\_refs}(\sigma, \beta, n, t)$ returns a state identical to σ, except that β's recorded references to or within the object at location n with type t are removed from the recorded references of σ. The function application in $\texttt{add\_se}(\sigma, \eta)$ returns a state identical to σ except that it has the additional side effect η pending.

Now, the question that naturally arises when confronted with these ugly two assignment rules is whether or not they conform to the natural language of "references to the object assigned are allowed on the right hand side". There are two criteria to assess: the rules should not make things undefined that are defined in the standard, and *vice versa*, they should not give meaning to things that are undefined.

Consider the first case. The only way in which these rules might make something incorrectly undefined would be if an allowable reference in the bag β and thus also in σ^ρ were to clash with an attempt to update the same object elsewhere, thereby falling afoul of the rule forbidding reference and update of the same object. However, if such an update takes place, then the expression is undefined, because the allowable reference is only a prelude to another update of the same object. This has to cause undefined behaviour because two updates of the same object are also illegal. So, the rules will not make a difference in this way.

We must also consider the possibility that the two rules might give meaning to something which is actually undefined. This can not be, as our method of emulating what is required actually records references that shouldn't be recorded. This can only make things worse as far as defined-ness is concerned.

Post increment and decrement The rules for the post-increment (`++`) and post-decrement (`--`) operators differ only slightly. Although the rule does look at the value of the object (using σ^ν), it does not record this as a reference to the object with `mark_ref`. This is because, as with the assignment, this is clearly a reference made in order to calculate the fresh value. After pulling the fresh value out of memory, all that is required is to make the new value a pending side effect.

So, the rule for `++` is

$$\frac{t \text{ is a scalar type}}{\langle \Lambda(n,t)\texttt{++}, \sigma_0\rangle \rightarrow_e \langle \underline{(m,t)}, \sigma\rangle}$$

where

$$\begin{array}{rcl} m & = & \sigma_0^{\nu}(n,t) \\ \sigma & = & \texttt{add\_se}(\sigma_0, \clubsuit(n, \sigma_0^{o}(\texttt{+}, \underline{(m,t)}, \mu(1, \texttt{int})))) \end{array}$$

The rule for `--` is just the same except that `+` signs have been replaced by `-` signs:

$$\frac{t \text{ is a scalar type}}{\langle \Lambda(n,t)\texttt{--}, \sigma_0\rangle \rightarrow_e \langle \underline{(m,t)}, \sigma\rangle}$$

where

$$\begin{array}{rcl} m & = & \sigma_0^{\nu}(n,t) \\ \sigma & = & \texttt{add\_se}(\sigma_0, \clubsuit(n, \sigma_0^{o}(\texttt{-}, \underline{(m,t)}, \mu(1, \texttt{int})))) \end{array}$$

Recall that pre-increment (`++x`) and pre-decrement (`--x`) are both handled by assuming translation to the equivalent `x += 1` and `x -= 1`.

2.3.6 Function calls—the interface with statement evaluation

The final form of expression is the function call. There are two rules for this as there is a sequence point after the arguments and the function designator are evaluated. The $\mathcal{E}$ context ensures that these "inner" evaluations take place, so the following rule merely records the fact (using the new "hat" intermediate syntactic form) that the sequence point has been reached.

$$\frac{\text{no pending side effects in } \sigma_0 \quad \text{all of } f \text{ and the } e_i\text{s are values}}{\langle f\,\texttt{(}e_1, \ldots, e_n\texttt{)}, \sigma_0\rangle \rightarrow_e \langle \hat{f}\,\texttt{(}e_1, \ldots, e_n\texttt{)}, \texttt{clear\_se}(\sigma)\rangle}$$

The call to the function is evaluation in this final expression rule.

$$\frac{\langle \sigma_0^{\varphi\beta}(f), \texttt{install\_params}(\sigma_0, \sigma_0^{\varphi\alpha}(f), [e_1 \ldots e_n])\rangle \rightarrow_s \langle \texttt{RetVal}\underline{(m,t)}, \sigma_1\rangle}{\langle \hat{f}\,\texttt{(}e_1, \ldots, e_n\texttt{)}, \sigma_0\rangle \rightarrow_e \langle \underline{(m,t)}, \sigma\rangle}$$

where

$$\sigma = \texttt{mem\_trans}(\sigma_1, \sigma_0)$$

The $\varphi\alpha$ function returns information (names and types) about function arguments. This information is then used by the `install_params` function to install the argument values (e_1 to e_n) in memory, and to update the environment information stored in the state. The $\varphi\beta$ function returns the body of a function, a statement. The $\rightarrow_s$ relation is the statement relation, while the `RetVal` constructor packages a function's return values. These statement specific constructs are further explained in the next section. Finally, the `mem_trans` function copies the memory component of the first argument over the rest of the second argument, returning the composite state thus constructed.

2.4 Statements

C's statements are rather fewer in number than its expressions, and the rules are also rather simpler. In particular, it is possible to write the rules in what is often called "an evaluation-order", "natural" or "big-step" style. The statement relation $\rightarrow_s$ maps from statement-state pairs to *statement value*-state pairs. A statement value is one of the following limited set of possibilities, each describing how statement execution has come to finish at this point.

`BreakVal`	a break statement was encountered
`ContVal`	a continue statement was encountered
$\mathtt{RetVal}\underline{(m,t)}$	a return statement (with value) was encountered
`StmtVal`	an ordinary evaluation termination

All but `StmtVal` interfere with the statement sequencing rule. There is also an undefined value, represented as before by $\mathcal{U}$. I use v to vary over statement values.

2.4.1 Simple statements

Empty statements The first statement evaluation rule is that for the empty statement, written here just using the `;`.

$$\frac{}{\langle \mathtt{;}, \sigma\rangle \rightarrow_s \langle \mathtt{StmtVal}, \sigma\rangle}$$

Expression statements An expression statement consists only of an expression, which is evaluated for the side effects caused. Note the use of R which gives the expression a valid $\mathcal{L}$ context in which to evaluate values which may be lvalues.

$$\frac{\langle R(e), \sigma_0\rangle \rightarrow_e^* \langle \underline{(m,t)}, \sigma\rangle \quad \text{no pending side effects in } \sigma}{\langle e, \sigma_0\rangle \rightarrow_s \langle \mathtt{StmtVal}, \sigma\rangle}$$

Of course, the expression evaluation may go astray, in which case we promote the expression undefinedness to the level of statements:

$$\frac{\langle R(e), \sigma_0\rangle \rightarrow_e^* \langle \mathcal{U}, \sigma\rangle}{\langle e\,;, \sigma_0\rangle \rightarrow_s \langle \mathcal{U}, \sigma\rangle}$$

2.4.2 Interruptions

The **break**, **continue** and **return** statements all interrupt the normal flow of control. It is gratifyingly easy to express their semantics:

$$\overline{\langle \mathtt{break}, \sigma\rangle \rightarrow_s \langle \mathtt{BreakVal}, \sigma\rangle}$$

$$\overline{\langle \mathtt{continue}, \sigma\rangle \rightarrow_s \langle \mathtt{ContVal}, \sigma\rangle}$$

$$\frac{\langle R(e), \sigma_0\rangle \rightarrow_e^* \langle m, \sigma\rangle \quad \text{no pending side effects in } \sigma}{\langle \mathtt{return}\ e, \sigma_0\rangle \rightarrow_s \langle \mathtt{RetVal}(m), \sigma\rangle}$$

$$\overline{\langle \mathtt{return}, \sigma\rangle \rightarrow_s \langle \mathtt{RetVal}(\emptyset), \sigma\rangle}$$

There are two rules for `return` because it exists in two forms, both with a value to be returned and without. In the latter case, I use the $\emptyset$ symbol to stand for a null (empty) value. No well-behaved program will attempt to make use of such a value, as it will only be returned by functions returning `void`.

2.4.3 Compound statements

Blocks Statements can be grouped together in a block. A block consists of a list of variable declarations followed by a list of statements. When the statements of a block finish evaluating, they will do so in an environment different from the external one. This is rectified as the block exits by using the `mem_trans` function, which restores the environment components of the state.

$$\frac{\langle [d_1 \dots d_n], \sigma_0\rangle \rightarrow_v \langle \mathtt{VarDeclVal}, \sigma_1\rangle \quad \langle [s_1 \dots s_m], \sigma_1\rangle \rightarrow_s \langle v, \sigma_2\rangle}{\langle \{[d_1 \dots d_n][s_1 \dots s_m]\}, \sigma_0\rangle \rightarrow_s \langle v, \mathtt{mem\_trans}(\sigma_2, \sigma_0)\rangle}$$

Even variable declarations are not immune to failure, so that the following rule is also necessary:

$$\frac{\langle [d_1 \dots d_n], \sigma_0\rangle \rightarrow_v \langle \mathcal{U}, \sigma\rangle}{\langle \{[d_1 \dots d_n][s_1 \dots s_m]\}, \sigma_0\rangle \rightarrow_s \langle \mathcal{U}, \sigma\rangle}$$

(The `VarDeclVal` value and $\rightarrow_v$ relation are explained in section 2.5.)

Statement sequencing A list of statements, as occurs in a block, is executed in order, with the requirement that for execution to continue, the statement value of the last statement executed must have been `StmtVal`. The base case is the empty list:

$$\frac{}{\langle [\,], \sigma\rangle \rightarrow_s \langle \texttt{StmtVal}, \sigma\rangle}$$

A normal evaluation proceeds according to the following rule:

$$\frac{\langle s_1, \sigma_0\rangle \rightarrow_s \langle \texttt{StmtVal}, \sigma_1\rangle \quad \langle [s_2 \ldots s_n], \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle}{\langle [s_1, s_2 \ldots s_n], \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}$$

On the other hand, an interrupted evaluation (here an undefined result is also effectively an interruption) will look like

$$\frac{\langle s_1, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle \quad v \neq \texttt{StmtVal}}{\langle [s_1, \ldots], \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}$$

This means that interruption statements will cause all further statements in a sequential composition to be skipped.

2.4.4 Conditional statements

There are three rules for the `if` statement. The first copes with the failure of the guard to evaluate properly.[10]

$$\frac{\langle R(e), \sigma_0\rangle \rightarrow_e^* \langle \mathcal{U}, \sigma\rangle}{\langle \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2, \sigma_0\rangle \rightarrow_s \langle \mathcal{U}, \sigma\rangle}$$

The two rules where the expression does actually evaluate fully are entirely straightforward. If the guard expression evaluates to a non-zero value then the first branch is evaluated, and its statement value preserved.

$$\frac{\langle R(e), \sigma_0\rangle \rightarrow_e^* \langle \underline{(m, t)}, \sigma_1\rangle \quad \langle s_1, \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle}{\langle \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}$$

where

t is scalar, $m \neq \mu(0, t)$, and there are no pending side effects in σ_1.

Otherwise, the second branch is chosen:

[10] C supports `if` statements without `else` branches, but these are trivially modelled with `else` branches of the empty statement.

$$\frac{\langle R(e), \sigma_0\rangle \to_e^* \langle \underline{(m,t)}, \sigma_1\rangle \quad \langle s_2, \sigma_1\rangle \to_s \langle v, \sigma\rangle}{\langle \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2, \sigma_0\rangle \to_s \langle v, \sigma\rangle}$$

where

t is scalar, $m = \mu(0, t)$, and there are no pending side effects in σ_1.

2.4.5 Iteration

There are three looping constructions in C, the `while` loop, the `for` loop, and the `do-while` loop. Cholera models them all with two special intermediate syntactic forms, the O (loop) and T (trap) constructors. The first is the basis for a generic looping mechanism, and the second is a mechanism which allows interrupt values to be intercepted or "trapped". The O constructor takes the loop guard and the loop body as arguments, while the T constructor takes the interrupt value to be intercepted and the statement which will be executed. The translations for the three loop forms in C are

$$\begin{array}{ll}
\texttt{while (}g\texttt{)}\ s & \hat{=}\ T(\texttt{BreakVal}, O(g, T(\texttt{ContVal}, s))) \\
\texttt{for (}e_1\texttt{;}\ e_2\texttt{;}\ e_3\texttt{)}\ s & \hat{=}\ \{e_1\texttt{;}\ T(\texttt{BreakVal}, O(e_2, \{T(\texttt{ContVal}, s)\ \ e_3\texttt{;}\}))\} \\
\texttt{do}\ s\ \texttt{while (}g\texttt{);} & \hat{=}\ T(\texttt{BreakVal}, \{T(\texttt{ContVal}, s)\ \ O(g, T(\texttt{ContVal}, s))\})
\end{array}$$

Note that the juxtapositioning of elements inside a pair of braces above indicates sequencing; the semi-colon is used as a statement terminator, not as a statement separator. The rule for the `for` loop should suggest that while a continue statement in the loop body may interfere with the rest of the loop body, it will not prevent the third expression from being evaluated.

There are two very simple rules for the T construction. If the value returned by the statement wrapped up is the one trapped, then a `StmtVal` is returned instead of what was going to be returned.[11]

$$\frac{\langle s, \sigma_0\rangle \to_s \langle v, \sigma\rangle}{\langle T(v, s), \sigma_0\rangle \to_s \langle \texttt{StmtVal}, \sigma\rangle}$$

Otherwise, the return value is passed through unchanged.

$$\frac{\langle s, \sigma_0\rangle \to_s \langle v, \sigma\rangle}{\langle T(v', s), \sigma_0\rangle \to_s \langle v, \sigma\rangle}$$

where

$v' \neq v$

[11] There are very clear parallels here with the way in which exceptions in a language such as SML are caught; here the exceptions are very simple, atomic values.

There are four rules for O. The first two specify the behaviour when the guard expression doesn't evaluate to true, which can happen in two different ways. The guard might result in undefined behaviour, in which case the loop's behaviour is also undefined.

$$\frac{\langle R(g), \sigma_0\rangle \to_e^* \langle \mathcal{U}, \sigma\rangle}{\langle O(g,s), \sigma_0\rangle \to_s \langle \mathcal{U}, \sigma\rangle}$$

Alternatively, the loop guard may evaluate to a zero value:

$$\frac{\langle R(g), \sigma_0\rangle \to_e^* \langle \underline{(\mu(0,t),t)}, \sigma\rangle \quad t \text{ is a scalar type} \quad \text{no pending side effects in } \sigma}{\langle O(g,s), \sigma_0\rangle \to_s \langle \mathtt{StmtVal}, \sigma\rangle}$$

If, however, the guard does evaluate to a non-zero value, then the loop is entered. The first rule below covers those cases where the body doesn't evaluate to a `StmtVal`; this causes the loop to exit. Consider then how this rule interacts with the translations of the standard C forms; if a `continue` statement is encountered while evaluating a loop body, then this will have been trapped by the T that is always wrapped around the occurrences of the body in the translation. Thus, this rule will not apply. In the case of the `break` statement, this rule *will* apply, but the `BreakVal` will be trapped by the trap around the entirety of the loop.

$$\frac{\langle R(g), \sigma_0\rangle \to_e^* \langle \underline{(m,t)}, \sigma_1\rangle \quad \langle s, \sigma_1\rangle \to_s \langle v, \sigma\rangle \quad v \neq \mathtt{StmtVal}}{\langle O(g,s), \sigma_0\rangle \to_s \langle v, \sigma\rangle}$$

where

t is a scalar type, $m \neq \mu(0,t)$, and there are no pending side effects in σ_1.

Finally, if the loop body's execution does terminate normally, the loop is entered once more:

$$\frac{\langle R(g), \sigma_0\rangle \to_e^* \langle \underline{(m,t)}, \sigma_1\rangle \quad \langle s, \sigma_1\rangle \to_s \langle \mathtt{StmtVal}, \sigma_2\rangle \quad \langle O(g,s), \sigma_2\rangle \to_s \langle v, \sigma\rangle}{\langle O(g,s), \sigma_0\rangle \to_s \langle v, \sigma\rangle}$$

where

t is a scalar type, $m \neq \mu(0,t)$, and there are no pending side effects in σ_1.

2.5 Variable declarations

Variables can be declared in two possible contexts, either at the start of a block, in which case they are by default *automatic* variables (which will cease to exist in any meaningful sense after the block exits), or they can be declared at the top level, in which case they are *static*, and have lifetimes equal to the duration of the program. It is possible to declare static variables inside a block, but this possibility is one that this semantics side-steps. Instead I claim that static local variables are actually global variables, but with unique names that only occur in the block where the variable is declared.

Upon declaration, variables can also be initialised. This naturally involves use of the $\to_e$ relation. Finally, though not strictly a variable declaration, structure declarations can also occur wherever a variable declaration is permitted.

The $\to_v$ relation takes declaration-state pairs and returns value-state pairs. There are only two possible return values for variable declarations, `VarDeclVal`, representing a successful execution, and $\mathcal{U}$ for a descent into undefinedness.

The rule for basic (automatic) variable declaration is:

$$\frac{}{\langle t\ v, \sigma_0\rangle \to_v \langle \texttt{VarDeclVal}, \texttt{decl\_var}(\sigma_0, v, t)\rangle}$$

The `decl_var` function updates the state argument's environment, allocating the variable in question some space in memory (space appropriate for its type), and changing the name to address, and name to type maps in the state (σ^α and σ^τ respectively).

When a variable is initialised with an expression, the abstract machine must first evaluate the expression, and then place the resulting value into the space allocated. Again choosing to abstract away from memory management mechanisms, I merely assume the existence of a state-modifying function to perform this latter function. Note also the use of the cast, in order to ensure that the value generated is of the appropriate type for the variable.

$$\frac{\langle \texttt{(}t\texttt{)}e, \sigma_0\rangle \to_e^* \langle m, \sigma\rangle \quad \text{no pending side effects in } \sigma}{\langle t\ v\ \texttt{=}\ e, \sigma_0\rangle \to_v \langle \texttt{VarDeclVal}, \texttt{init\_decl\_var}(\sigma, v, t, m)\rangle}$$

Of course, expressions can go astray, and this is the only variable declaration rule where undefined behaviour can result:

$$\frac{\langle \texttt{(}t\texttt{)}e, \sigma_0\rangle \to_e^* \langle \mathcal{U}, \sigma\rangle}{\langle t\ v\ \texttt{=}\ e, \sigma_0\rangle \to_v \langle \mathcal{U}, \sigma\rangle}$$

Finally, the third type of declaration is for declaring new `struct` types. This requires both a name for the type, and the names and types for all of the constituent fields. This information is stored in the ξo and $\xi\tau$ components of the state record.

$$\frac{}{\langle \texttt{struct}\ \mathit{tag}\ \mathit{fields}\,\texttt{;}, \sigma\rangle \to_v \langle \texttt{VarDeclVal}, \texttt{struct\_decl}(\sigma, \mathit{tag}, \mathit{fields})\rangle}$$

3 Results

As mentioned before, the Cholera semantics above has been defined in the HOL theorem prover. This has subsequently supported the proof of a number of results relating to the semantics. Most of these have been proved with an eye to their later use in verification. They are presented here with that as the underlying motivation (see also [18]).

3.1 A derived "axiomatic" logic

Following Gordon [9], I have proved a number of theorems mimicking the axiomatic rules of Hoare [14]. Though Hoare's rules are axioms defining the semantics of his language, here the operational semantics is the ultimate authority. The theorems do not define the language, but are statements about the properties of an existing logical entity. It is the intention that their similarity to Hoare's axiomatic rules should allow them to be used in verification work, hiding the underlying complexity of the operational semantics.

C's various "ugly" features mean that it is impossible to derive rules that are exactly the same as Hoare's originals. Rather than attempt to encapsulate all of C's complexity in a series of complicated rules, I have chosen instead to try and characterise programs for which simpler rules hold. This results in rules accompanied by various side conditions. In addition, the emphasis has been on making these side conditions syntactic in nature. This means that simple static tests can determine the applicability or not of the rules.

The triple $\{P\}\ S\ \{Q\}$ is an assertion about the partial correctness of S; if S is executed in a state where P holds, then, if S terminates, it will result in a state where Q is true.

These principles are illustrated in the derived rule for sequencing:

$$\frac{\{P\}\ S_1\ \{P'\} \quad \{P'\}\ S_2\ \{Q\}}{\{P\}\ S_1\ S_2\ \{Q\}}$$

where

S_1 contains no interrupt statements.

It is easy enough to show that this rule is sound, given the rules for sequencing. It is too conservative a rule, in that it will not apply to examples where the first statement has (say) an unreachable `break` statement in it, but it should still be useful.

There is also a nice rule for `while` loops, which is a simple variant on the usual rule for loops that dodges the issue of expression evaluation in the guard g by referring to the semantics of the `if` statement.

```
while (P) {
  if (ex_cond)
     break;
  ...
}
```

Figure 1: loop with abnormal exit

$$\frac{\{I\}\ \texttt{if (}g\texttt{)}\ S\ \texttt{else ;}\ \{I\}}{\{I\}\ \texttt{while (}g\texttt{)}\ S\ \{I\}}$$

Another rule (which might well be combined with the above, but is presented separately here for the purposes of exposition) allows us to express a useful loop termination property:

$$\overline{\{P\}\ \texttt{while (}g\texttt{)}\ S\ \{\texttt{was(!}g\texttt{)}\}}$$

where

S contains no interrupt statements

The special `was` operator takes a C expression and specifies that the state to which it is applied is one which might result from the execution of the expression returning a non-zero value. In other words, it specifies that the expression evaluated to true and arrived at the current state in the process. This is reminiscent of a dynamic logic in which the modality operator is backwards looking.

Unfortunately, `was` has no obvious nice properties that might allow it be the basis for a C reasoning system. In particular, it doesn't seem likely that a system such as the one presented in Boehm's work [4] could grow out of it. Boehm's system relies on expressions being both terminating and deterministic, properties not true of C expressions in general. On the other hand, C does not fall into the category of languages for which "nice" descriptions are impossible, as described in [5], as it does not allow nested function definitions.

3.2 Further analysis of loops

The usual rule that one wants to be able to use when analysing loop executions states that the negation of the loop's guard is true when a loop terminates. In C this is not the case because the loop may have terminated because of a `return` or `break` statement. This means that a loop's post-condition must instead be a disjunction: it terminated normally and the negation of the guard is true, or the loop terminated abnormally and some other post-condition holds.

I have written a function within HOL that analyses loop bodies and automatically generates a post-condition corresponding to all of the abnormal ways in which a loop might exit. I have subsequently proven that the condition generated by this function does indeed hold when a loop terminates abnormally. The conditions generated are strong enough to be able to state that in the example of Figure 1, either `P` and `ex_cond` will be true, or `P` will be false. This assumes that neither expression introduces any side effects. If either does, the analysis that Cholera performs still generates correct statements of what must be true when the loop exits, but they are rather more complicated.

If one wishes to keep the analysis of the code at the "axiomatic" level of the previous section, alternatives to this approach involving multiple post-conditions are also possible. An example of this is the presentation of a *wp* semantics for a language with an **exit** statement [16]. Earlier work along the same lines is found in [2]. In C, one would want triples with four postconditions, corresponding to the three different forms of interruption and normal execution.

3.3 Purity analysis

Expressions cause would-be "axiomatic" rules about C so much difficulty because they might cause side effects. Yet there are plenty of C expression which will never cause side effects. I have developed a theory of these *pure* expressions.

A pure expression can be used much as are the expressions in traditional axiomatic semantics. Because their repeated evaluation does not alter the state of the machine, they can be the basis for rules such as

$$\frac{\{I \wedge G\}\ \ S\ \ \{I\}}{\{I\}\ \texttt{while}\ (G)\ S\ \{I \wedge \neg G\}}$$

where

G is pure and S is free of interrupt statements

Furthermore, it is the case that some expressions must be pure because of their syntactic form. This is a particularly useful subset of the pure expressions because this syntactic categorisation allows one to do a simple check to confirm that a rule applies.

Therefore, if one chose to program in a subset of C where impure expressions were uncommon, a verification tool based on the derived "axiomatic" semantics would frequently be able to apply rules such as that given above. With both side conditions syntactically checkable, the tool would be able to decide automatically that one was working within the scope of the easiest rules. Such a system might have to plunge the user into the depths of the operational semantics if particularly complicated forms were encountered, but might otherwise present quite a simple interface.

4 Related work

C has not been a popular language for formalisation work in the past. However, as both theorem proving technology and the theoretical semantics tools have improved, the prospect of managing what is inevitably a complicated and tedious project has become less intimidating. A number of other authors have attempted to formalise some aspect of C. However, none of them address the modelling of sequence points and side effects in expression evaluation.

The only work for which this is probably not a significant issue is that of Black [3], where the attempt is to describe C at a much higher level than this work. Here, an axiomatic semantics is developed *ex nihilo*, and expressions are characterised at a much higher level than in Cholera.

In other work, the details of expression evaluation are simply not correct. The evolving algebra semantics of Gurevich and Huggins [11, 12] relies on the mistaken assumption that side effects are applied as they are generated, and that expressions involving binary operators can be evaluated by evaluating all of one argument before switching to the other. Its rules for binary operators effectively assume a big-step style semantics:

$$\frac{\langle e_1, \sigma_0 \rangle \rightarrow_e^* \langle v_1, \sigma' \rangle \quad \langle e_2, \sigma' \rangle \rightarrow_e^* \langle v_2, \sigma \rangle}{\langle e_1 \odot e_2, \sigma_0 \rangle \rightarrow_e^* \langle v_1 \odot v_2, \sigma \rangle} \qquad \frac{\langle e_2, \sigma_0 \rangle \rightarrow_e^* \langle v_2, \sigma' \rangle \quad \langle e_1, \sigma' \rangle \rightarrow_e^* \langle v_1, \sigma \rangle}{\langle e_1 \odot e_2, \sigma_0 \rangle \rightarrow_e^* \langle v_1 \odot v_2, \sigma \rangle}$$

Subramanian and Cook present a semantics for a subset of C in [19]. This work has the distinction of having been done in the theorem-prover Nqthm. However, this work assumes a particular order of evaluation for binary operators, and also leaves out such features of the full language as interrupt statements (`break`, `continue` etc).

Finally, Cook *et al.* in the unpublished [6] present a denotational semantics that denotes C forms as expressions in a custom-built temporal logic. This semantics also makes the convenient assumption that C's expressions are evaluated left-to-right, but explicitly mentions ways in which this might be improved.[12] As with all the other work reviewed, there is no treatment of sequence points and the associated restrictions on expression evaluation.

5 Future work and conclusions

I intend to pursue a verification project using the Cholera semantics as the basis for a proof in HOL of the correctness of some C code. Because Cholera doesn't deal with system and library calls, this verification can't attempt anything like the `thttpd` code of [3], but will rather look at the C code written by John Harrison as part of his work on binary decision diagrams in [13]. This code is

[12] Although the improvements mentioned might result in the big-step non-determinism of [12].

over 300 lines long, uses a hash table, linked lists as buckets, and a promote-to-front strategy when searching the lists. It is thus quite a complicated example of data structure use, and seems a realistic small-scale verification exercise.

Cholera represents an important first step forward in the application of formal methods to the verification of C programs. On the way to this admittedly remote and difficult goal, a formalisation of the language has been achieved, demonstrating that the techniques of operational semantics can deal with the challenge of such an inconvenient language. Moreover, the development of a groundwork for program verification has required the proof of a number of "meta-level" results and the development of an "axiomatic" theory for abstracting away from the full tedium of the operational semantics. These results have an appeal of their own, independent of the verification context in which I expect to use them. This work suggests that C is not the monstrous language of received wisdom, but rather one about which it is possible to reason formally.

References

[1] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, December 1989.

[2] Michael A. Arbib and Suad Alagic. Proof rules for gotos. *Acta Informatica*, 11(2):139–148, 1979.

[3] Paul Black and Phil Windley. Inference rules for programming languages with side effects. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem proving in higher order logics. 9th international conference, TPHOLs 96*, volume 1125 of *Lecture notes in computer science*, pages 51–60. Springer, August 1996.

[4] Hans-Juergen Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4):637–655, October 1985.

[5] Edmund M. Clarke Jr. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the ACM*, 26(1):129–147, January 1979.

[6] J. V. Cook, E. L. Cohen, and T. S. Redmond. A formal denotational semantics for C. A draft document, available from Trusted Information Systems' web-site at `http://www.tis.com/docs/research/assurance/formal-c.html`, September 1994.

[7] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[8] Andrew D. Gordon. Bisimilarity as a theory of functional programming. Mini-course. BRICS Notes Series NS–95–3, BRICS, Aarhus University, 1995. Extended version of MFPS'95 and Glasgow FP'94 papers.

[9] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current trends in hardware verification and automated theorem proving*. Springer-Verlag, 1989.

[10] M. J. C. Gordon and T. Melham. *Introduction to HOL: a theorem proving environment*. Cambridge University Press, 1993.

[11] Yuri Gurevich. Evolving algebras: a tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[12] Yuri Gurevich and James K. Huggins. The semantics of the C programming language. In E. Borger, editor, *Selected papers from CSL '92*, volume 702 of *Lecture notes in computer science*, pages 274–308. Spring-Verlag, 1993. Corrected version available from University of Michigan web-site: `http://www.eecs.umich.edu/gasm`.

[13] John Harrison. Binary decision diagrams as a HOL derived rule. *Computer Journal*, 38(2), 1995.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[15] *Programming languages – C*, 1990. ISO/IEC 9899:1990.

[16] Steve King and Carroll Morgan. Exits in the refinement calculus. *Formal aspects of computing*, 7(1):54–76, 1995.

[17] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[18] Michael Norrish. Derivation of verificiation rules for C from operational definitions. In J. von Wright, J. Grundy, and J. Harrison, editors, *Supplementary proceedings of TPHOLs '96*, number 1 in TUCS General Publications, pages 69–75. Turku Centre for Computer Science, August 1996.

[19] Sakthi Subramanian and J. V. Cook. Mechanical verification of C programs. In *First workshop on Formal Methods in Software Practice (FMSP '96)*. Association for Computing Machinery, January 1996.