A case study in specifying the semantics of a programming language

Ravi Sethi

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

Abstract

On and off over the period of about a year I have worked on a semantic specification for the C programming language My objective was to construct a readable and precise specification of C, aimed at compiler writers, maintainers, and language pundits. This paper is a report on the project.

Introduction

C is a general purpose programming language in which most of the software running under the UNIX* operating system is written. Notes on the development of the language may be found in [rit78].

1.1. *What is C?* The de facto standard for C is D. M. Ritchie's compiler [rit79]. The portable C compiler by S. C. Johnson [joh78] (which runs on a number of dissimilar machines) is remarkably close to the Ritchie compiler. The compilers for C are permissive in that they compile and run more programs than the C reference manual in [ker78] would allow. The permissiveness of the compilers is balanced by a program called *lint* [joh79], which checks C programs and complains about dubious constructions ranging from type mismatches to possibly non-portable constructions. *lint* is restrictive in that it complains about programs that the C reference manual [ker78] allows.

The C programming language therefore lies somewhere between the compilers, *lint*, and the C reference manual.

1.2. *Choice of semantic method.* At one time or another three approaches towards defining the semantics of realistic programming languages have been suggested: the operational or interpretive approach; the denotational or mathematical approach; and the axiomatic approach that constructs a logic for the programming language. While logics of programs are interesting in their own right, they are not a semantics for the programming language. (See for example the discussion in [gre79, hoa78].)

Recent work on semantics has tended to use the denotational or mathematical approach [scs71]. In fact after doing an operational semantics for PL/I [luc70] the "Vienna group" too has turned to denotational semantics [bjo78]. The denotational method was chosen because it can yield abstract semantic specifications, and because it has been used to specify the semantics of a number of programming languages: Algol 60 [mos74, hen78]; Algol 68 [mil72]; Gedanken [ten76]; Pascal [ten77]; PL/CV [con79]; Snobol 4 [ten73]; Sal [mil76].

1.3. *The sequel.* This paper reports on the application of the denotational method to the C programming language.

The bulk of the paper deals with declarations. Initial drafts of the semantics of C treated all declarations together — but this led to a specification that was hard to read. In section 2, two small language fragments are used to gradually introduce the required concepts: the first allows simple data declarations of arrays and pointers, while the second allows structures of a simple sort. Using the first fragment type determination and storage management are discussed. Using the second fragment the extraction of recursively defined information is discussed. Together, these two fragments allow the introduction of all the concepts that are needed to specify the semantics of data declarations in C.

Since denotational semantics of statements and expressions have been discussed extensively in the

* UNIX is a trademark of Bell Laboratories.

literature (see for example [sto77]), only brief remarks about the semantics of statements and expressions appear in sections 3 and 4.

A few language issues relevant to declarations are mentioned in section 5, and the final section 6 contains some notes on the project.

1.4. *Literature.* Little of the folklore of the denotational method is available in print. Stoy [sto77] is a general introductory reference. The best reference I have found for describing languages is the unpublished set of lecture notes by Gordon [gor78]. As I progressed, I used [ten77] as a primary reference. The several sources where the semantics of an actual language is given tend to contain little discussion of why the particular presentation was chosen.

The discussion of declarations in section 2 draws heavily on the folklore of denotational semantics. If there is anything original in the section, it is in the way the material leads up to the semantics of data declarations in a real language.

2. Declarations

By "declarations" we mean the part of a programming language that allows meaning to be associated with identifiers. Identifiers are used not only to refer to basic values like characters, and data structures like arrays, but also to executable "functions", which take parameters and return values. A discussion of the meaning of declarations must therefore address issues suggested by the following phrases: basic and derived types; data declarations; type determination; block structure; storage allocation; function declarations.

2.1. *Discussion of declarations.* Rather than assume familiarity with C [ker78], we will introduce declarations in the language through a sequence of examples.

For the moment, the terms "type" and "location" will be used informally. Think of there being a set **Ty**, whose elements are called types. Included in **Ty** are basic types like integer, and derived types like array of 8 integers. One of the purposes of a data declaration is to associate a type with an identifier. The term location corresponds to a storage cell in a machine, except that a location can hold any basic value. A location will be associated with each identifier representing a basic value. This basic value will be determined from the identifier in two stages: first the location for the identifier will be determined, and then the value held in the location will be looked up.

Syntax. The following program fragment suggests the syntax of declarations in C.

```
int n = 3;
char select(x,c,d)
int x; char c; char d;
{
        char e;
        if (x > n) e = c; else e = d;
        return (e);
}
```

We take a program in C to consist of a sequence of data declarations like

```
int n = 3;
```

followed by a sequence of one or more function declarations. C-functions like select are similar to functions and subroutines in Fortran, or to procedures in Pascal, except that function declarations cannot be nested. As in Algol 60, the char in

```
char select(x,c,d)
```

specifies that select is a C-function that *returns* a character. Declarations of the formal parameters x, c, and d precede the body of the C-function.

The identifier n is *external* to all function declarations. External identifiers can be referenced inside any function without being explicitly redeclared. We assume that inside a C-function any declarations, like that of e, precede all statements within the function.

Declarators. The syntax of an identifier declaration in C mimics the syntax of expressions in which the identifier might appear. For example,

```
float xyz[3][5];
```

says that, in an expression, xyz[m][n] represents a value of type float. Then, xyz[m] must represent an array

of five elements of type float. xyz is therefore declared to be an array of three subarrays; each subarray being an array of five elements of type float.

The declaration

```
int *px;
```

says that the construction *px represents an integer. The * operator "dereferences" a pointer, so px is declared to be a pointer to an integer.

The constructions xyz[3][5] and *px in the above declarations are instances of *declarators*.

Structures. The essential difference between an array and a "structure" is that an array contains a fixed number of members of the same type, while a structure contains a fixed number of members of possibly different types. There is another, more subtle, difference: a structure may contain a pointer to its own type so we must deal with recursively defined types.

A structure contains a fixed number of *members*. Each member has a *name*, and may have any type. We illustrate structures by declaring a "complex number" z and a pointer to a complex number zp. The identifier complex in the following declarations is referred to as a *structure tag*.

```
struct complex {float re; float im;};
```

The declaration of a structure tag and its use to declare another identifier can be combined in C, so z and zp can also be declared by

```
struct complex {float re; float im;} z, *zp;
```

Declarations like the above not only reserve storage for z and zp, they have the side effect of associating a type with the structure tag complex.

2.2. *Simple data declarations.* A number of programming language concepts must be understood before we can give the semantics of even a simple declaration like

```
char ab[7];
```

In this section we will consider declarations that declare identifiers to be: integers, characters, or some other basic type; arrays of a fixed number of elements of some type; or, pointers to some type. The syntax is as follows

declaration:
    basic_specifier declarator ;

We will not specify the syntax of *basic_specifier* any further.

A *declarator* (e.g. *px and xyz[3][5]) contains the identifier being declared. Declarators have the syntax:

declarator:
    identifier
    (*declarator*)
    declarator [*constant*]
    * *declarator*

The * operator in a declarator has lower precedence than all other operators, so *apc[7] will be parsed as *(apc[7]).

Semantic domains. All the semantic domains we will refer to here are defined in Figure 2.1.

We will take a very simplistic view of the term "type". Informally speaking, a type will be just an abstract entity that permits us to distinguish between identifiers that are declared differently e.g. distinguish between an integer and a pointer to an integer.

The domain **Ty** of types will be the sum of the domain **Tb** of basic types and domains corresponding to each way of constructing derived types. For example, the element of **Ty** corresponding to array of seven characters will be a triple (**array**,7,*character*): **array**, the only element of the domain {**array**}, serves as a "keyword" and is included for clarity; 7 is an element of the domain **N** of integers; and, *character* is an element of **Ty**.

The association of types with identifiers will be performed by elements of the domain **Ent** of *type environments*. The term "environment" applies to any function that maps an identifier to something

Semantic Domains

	Tb	basic types
$t \in$	**Ty** = **Tb** + {**array**} × **N** × **Ty** + {**pointer**} × **Ty**	types
$ent \in$	**Ent** = **Ide** → **Ty**	type environments
$l \in$	**L**	locations
$l \in$	**Vl** = **L** + [**N** → **Vl**]	lvalues
$enl \in$	**Enl** = **Ide** → **Vl**	lvalue environments
	Vb	basic values
	Vs = **Vb** + **Vl**	storable values
$v \in$	**V** = **Vs** + [**N** → **V**]	values
$s \in$	**S** = **L** → [**Vs** + {**grb**} + {**unused**}]	states

Figure 2.1: Semantic domains for simple data declarations.

associated with the identifier. Since the same identifier may have a type and also some storage associated with it, there will be more than one kind of environment.

The value of an identifier y can be changed either by an explicit assignment to y, or by an indirect assignment through a pointer to y. The presence of pointers makes it convenient to have a two-stage mapping from identifiers to their values.

An identifier in a data declaration will be mapped by an "lvalue environment" *enl* to an "lvalue". For an identifier of basic or pointer type, this lvalue will be a "location". Locations are analogous to storage cells. Every location is included in the class of lvalues, but the lvalue for an array will not be a location. For example, after the declaration

```
char pair[2];
```

locations will be reserved for pair[0] and pair[1]. Let these locations be l_0 and l_1. The lvalue corresponding to the identifier pair, by itself, is a function mapping 0 to l_0 and 1 to l_1, and is not a location. In general, the members of an array may themselves be arrays, so there will be lvalues rather than locations for the members.

Making the lvalue of an array identifier a function from integers to lvalues makes it easy to determine the lvalues of the array members e.g. if the lvalue for ab is l, then the lvalue for ab[5] will be $l(5)$.

Following Strachey [str72], a useful distinction is often made between values that can be stored, denoted by identifiers, expressed by expressions, passed as parameters, can appear on right hand sides of assignments, can appear on left hand sides of assignments, and so on. For example, in most languages, even if an identifier like xyz in

```
float xyz[3][5];
```

has a value, the value cannot be stored. What is stored are the values of xyz[0][0], xyz[0][1], · · · . One situation in which the identifier xyz by itself will have a value is if array values are returned by procedures in the language.

Corresponding to each basic type is a domain of basic values of that type. We sum all these domains together into the domain **Vb** of *basic values*. Included in **Vb** is the domain **N** of *integers*.

In addition to basic values there are values associated with pointers and arrays. Since the assignment px=&x; assigns the lvalue of x to px, the value of an identifier may sometimes be an lvalue. We therefore include **Vl** in the domain **V** of values. An *array value* in **V** will be a function from integers to values. Array values are not be storable.

A "state" maps a location to a storable value. In addition there will be two special values: **grb** a garbage value corresponding to an uninitialized location, and **unused** corresponding to a location that is

Semantic Rules

$[\![declarator]\!]t \in$ **Ide**×**Ty**

| *identifier*

‡ (*identifier*,*t*)

| (*declarator*)

‡ $[\![declarator]\!](t)$

| *declarator* [*constant*]

‡ **let** $n = [\![constant]\!]$; $t' = arr(n,t)$; **in** $[\![declarator]\!](t')$

| * *declarator*

‡ **let** $t' = point(t)$; **in** $[\![declarator]\!](t')$

dt$[\![declaration]\!](ent) \in$ **Ent**

| *basic_specifier declarator* ;

‡ **let** $t = [\![basic\_specifier]\!]$; $(id,t') = [\![declarator]\!]t$; **in** $ent[t'/id]$

ds$[\![declaration]\!](enl,s) \in$ **Enl** × **S**

| *basic_specifier declarator* ;

‡ **let** $t = [\![basic\_specifier]\!]$;
$(id,t') = [\![declarator]\!]t$;
$(l,s') = new(t',s)$;
$enl' = enl[l/id]$;
in (enl',s')

Figure 2.2: Semantic rules for declarators and declarations in the language of simple data declarations. Lines beginning with "|" specify the syntactic rules, and are followed by lines beginning with "‡" which give the corresponding semantic rules. We write $enl[l/id]$ for the new environment enl' satisfying

$enl'(x) =$ **if** $x=id$ **then** l **else** $enl(x)$

unallocated.

Semantic rules. The meaning of declarators can be explained by considering the declarations

```
char *( apc[7] );  int ( *pai )[7];
```

From the discussion earlier in this section, a construction like *(apc[m]) can appear in any context where a character is expected, and a construction like (*pai)[n] can appear in any context where an integer is expected. Reading the declarators inside-out, apc is declared to be an array of seven pointers to — from the type specifier char — characters. Similarly, pai is a pointer to an array of seven integers.

After a declarator has been examined, in addition to uncovering the embedded identifier, the type of this identifier will also be known. The meaning of a declarator will therefore be a function from a type to an identifier and its type. In Figure 2.2, the meaning of the nonterminal *declarator* is represented by $[\![declarator]\!]$. Similarly, the meaning of *basic_specifier*, is represented by $[\![basic\_specifier]\!]$, and will be a type.

Since the semantic rules associated with syntactic rules, we must follow the syntax and read declarators "outside-in" rather than "inside-out" as in the discussion above. Proceeding outside-in, we start with the type t from the type specifier and build up the type associated with the embedded identifier.

New types are built from old using the functions *point* and *arr*: *arr* maps an integer n and a type t to a new type t' corresponding to array of n members of type t; *point* maps a type t to t' corresponding to pointer to type t.

Having considered declarators we now turn to declarations. There are two kinds of meanings for declarations: one associates a type and the other an lvalue with an identifier. We distinguish between these two types of meanings by writing **dt**[[*declaration*]] for the first and **ds**[[*declaration*]] for the second. **dt** and **ds** are actually functions which map the semantic object *declaration* to appropriate semantic objects. Such functions are called *valuations*.

According to the semantic rules in Figure 2.2, under valuation **dt**, each declaration modifies the type environment by associating a type with the declared identifier. Under valuation **ds** a declaration associates an lvalue with an identifier. Before explaining valuation **ds** we need to clarify the handling of storage allocation.

Auxiliary Function

$new(t,s) \in \mathbf{Vl} \times \mathbf{S}$

$new(t,s) =$

$scalar(t) \rightarrow$

let $s(l) = \mathbf{unused}$; $s' = s[\mathbf{grb}/l]$; **in** (l,s')

$t = arr(n,t') \rightarrow$

let $(l_0,s_0) = new(t',s)$; $\cdots (l_{n-1},s_{n-1}) = new(t',s_{n-2})$;

$f = \lambda i.\ \perp$;

$f_0 = f[l_0/0]$; $\cdots f_{n-1} = f_{n-2}[l_{n-1}/n-1]$;

in (f_{n-1},s_{n-1})

Figure 2.3: The auxiliary function *new* allocates an lvalue for a given type.

Allocation. Given a type t and a state s, the auxiliary function *new* in Figure 2.3 returns an lvalue, and changes the state to s'. In the changed state, all locations in the returned lvalue are initialized to the special garbage value **grb**.

A type t is said to be *scalar* if t is basic i.e. $t \in \mathbf{Tb}$, or if t is a pointer i.e. $t = point(t')$ for some t'. We use the auxiliary function $scalar(t)$ which yields **true** if t is scalar, and **false** otherwise.

If a type t is scalar, then $new(t,s)$ will return an lvalue that is a location. If the type t is an array type, then *new* is invoked recursively to determine lvalues for each member of the array. If these lvalues are $l_0, l_1, \cdots, l_{n-1}$, then the lvalue for the array will be a function that yields l_i when applied to i, for $0 \le i \le n-1$. This function is constructed by starting with a function f that maps every integer to an undefined value $\perp$. f is successively modified to yield $f_0, f_1, \cdots, f_{n-1}$, where $f_{n-1}(i)$ is l_i for $0 \le i \le n-1$ and is $\perp$ otherwise. f_{n-1} and an appropriate state are returned by the *new* function.

2.3. *Recursively defined types.* We will be concerned only with type determination in this section since the storage aspects of structure declarations are just like those of array declarations. The determination of types here is a good illustration of the extraction of recursively defined information from a program. (Other examples of recursively defined meanings are those of recursive procedures and statement labels.)

Syntax. The syntax and semantics of declarators are as in section 2.2. The only purpose of the nonterminal *all_decl* is to collect all the declarations together.

all_decl:
declaration

declaration:
struct *identifier* { *member_decl* };
type_specifier declarator ;
declaration declaration

member_decl:
type_specifier declarator ;
member_decl member_decl

type_specifier:
basic_specifier
struct *identifier*

Semantic domains. In addition to the basic, pointer, and array types of section 2.2 we now need a structure type. The structure tag, the names and types of the members, and the order in which the members appear, are all significant. A structure type will therefore include the structure tag and the sequence of member names and types.

$$\mathbf{Ty} = \mathbf{Tb} + \{\mathbf{array}\} \times \mathbf{N} \times \mathbf{Ty} + \{\mathbf{pointer}\} \times \mathbf{Ty} + \{\mathbf{struct}\} \times \mathbf{Ide} \times [\mathbf{Ide} \times \mathbf{Ty}]^{+}$$

Furthermore, the operator *str*, applied to a structure tag and a list of pairs of member identifiers and associated types, yields a type t' corresponding to a structure with the appropriate tag and members. For example, associated with the tag complex is the type

$$str(\text{complex},(\text{re},float)\cdot(\text{im},float))$$

After the declarations

```
struct complex {float re; float im;};
struct complex z;
```

both complex and z have the type structure with tag complex containing members named re and im of type float.

But there is a difference. complex is a tag, while z is a structure. We will need to distinguish between these two ways of associating types with identifiers by suitably defining the type environment

$$\mathbf{Ent} = \mathbf{Ide} \to [\mathbf{Ty} + \{\mathbf{tag}\} \times \mathbf{Ty}] \qquad \text{type environments}$$

Meaning of member declarations. A type specifier must clearly yield a type. In order to determine this type, we may need to refer to the environment for the types associated with previously declared structure tags. Thus the meaning of a type specifier is a function from an environment to a type.

$[\![type\_specifier]\!]ent \in \mathbf{Ty}$

| *basic_specifier*

‡ $[\![basic\_specifier]\!]$

| struct *identifier*

‡ t **where** $(\mathbf{tag},t) = ent(identifier)$

In the above rule, *identifier* is a structure tag, which is mapped by the environment to a pair consisting of a special marker **tag** and a type t. The marker **tag** is ignored here. Later in this section we discuss what happens if the structure tag has not already been declared.

From the member declarations in a structure we need to extract the member names and their associated types. The meaning of *member_decl* is a function from a type environment to a sequence, *pairlist*, containing the pairs of member names and types.

$[\![member\_decl]\!]ent \in [\mathbf{Ide} \times \mathbf{Ty}]^{+}$

| *type_specifier declarator* ;

‡ **let** t = [[*type_specifier*]]*ent*;

pair = [[*declarator*]](t);

in *pair*

| *member_decl*1 *member_decl*2

‡ **let** *pairlist* 1 = [[*member_decl*1]]*ent*;

pairlist 2 = [[*member_decl*2]]*ent* ;

in *pairlist* 1 · *pairlist* 2

The meaning of a basic specifier is assumed to be an appropriate element of the domain **Tb** of basic types, and will not be specified any further.

Recursively defined types. The structure tag z below, is used to declare one of the structure members so the type of z is recursively defined.

```
struct z { int count; struct z *p; };
```

Informally, the type t associated with tag z will be structure with tag z containing member count of type integer and member p of type pointer to t. We therefore have the following recursive definition of t:

t = *str*(z, (count,*integer*)(p,*point*(t)))

A slightly more complex case is that of the types of x and y in:

```
struct x { int count; struct y *py; };
struct y { int count; struct x *px; };
```

This time we get a pair of mutually recursive definitions:

tx = *str*(x, (count,*integer*)(py,*point*(ty)))

ty = *str*(y, (count,*integer*)(px,*point*(tx)))

The key problem in determining types is that of setting up the mutually recursive equations for these types. Actually, instead of setting up equations for types, we will recursively determine the environment that will map identifiers to types.

Updating environments. Valuation **de** enters the type of the declared identifier into the environment. The first rule that follows is for a data declaration similar to the ones in section 2.2, and this rule for valuation **de** is very similar to the corresponding rule for valuation **dt** in Figure 2.2.

de[[*declaration*]]*ent* ∈ **Ent**

| *type_specifier declarator* ;

‡ **let** t = [[*type_specifier*]]*ent*;

(id,t') = [[*declarator*]](t);

in *ent*[t'/id]

| *declaration*1 *declaration*2

‡ **de**[[*declaration*2]](**de**[[*declaration*1]]*ent*)

If the declared identifier is a structure tag, then an ordered pair (**tag**,t), where t is the type associated with the tag, is entered into the environment.

| struct *identifier* { *member_decl* } ;

‡ **let** *pairlist* = [[*member_decl*]]*ent*;

t = *str*(*identifier*,*pairlist*);

t' = (**tag**,t);

in *ent*[t'/*identifier*]

Let us consider what happens when the above rule is applied to the declaration:

struct z { int count; struct z *p; };

Suppose that the environment *ent* is such that *ent*(z)=*tz*. Using the environment *ent*, the valuation for member declarations will yield the *pairlist*:

(count,*integer*)(p,*point*(*tz*))

Then, the following type will be constructed:

t = *str*(z, (count,*integer*)(p,*point*(*tz*)))

If *tz* is an approximation to the type of z, then t above is a better approximation to the type of z. In fact, if $t_0 = \perp$, and

t_{i+1} = *str*(z, (count,*integer*)(p,*point*(t_i)))

then the least upper bound of the chain $t_0, \cdots, t_i, \cdots$ is the desired type of z.

Rather than setting up a recursive definition that allows t_{i+1} to be determined from t_i, valuation **de** is used to construct a sequence of environments $ent_0, ent_1, \cdots$ in which ent_i(z) = t_i. The starting environment ent_0 will map all declared identifiers to $\perp$, the least defined type.*

A valuation **dz** will be used to construct ent_0. There is another use for a valuation like **dz**: the identifier z may be used for some other purpose outside the current C-function and we need to ensure that on entering a function, the types of any identifiers declared in the current function are "reset".

Initial environments in a "block". For each identifier with which a type is associated in the current set of declarations valuation **dz** will enter $\perp$ as the type of the identifier.

dz[[*declaration*]]*ent* ∈ **Ent**

| *type_specifier declarator* ;

‡ **let** t = [[*type_specifier*]]*ent*;

(id,t') = [[*declarator*]](t);

in *ent*[$\perp$/*id*]

| struct *identifier* { *member_decl* } ;

‡ *ent*[$\perp$/*identifier*]

| *declaration*$_1$ *declaration*$_2$

‡ **dz**[[*declaration*$_2$]](**dz**[[*declaration*$_1$]]*ent*)

Sequence of environments. The ingredients for determining the types of structure tags have all been assembled. On collecting all the declarations under *all_decl* the valuation **dz** is applied to the starting environment *ents* to reset the types for all declared identifiers. The valuation **de** is then used repeatedly to construct the sequence of environments $ent_0, ent_1, \cdots$ mentioned above. The desired environment is the least upper bound of the sequence $ent_0, ent_1, \cdots$:

ent_0 = **dz**[[*all_decl*]]*ents*

ent_{i+1} = **de**[[*declaration*]](ent_i) $\quad i \geq 0$

$ent_f = \bigsqcup\{ent_i \mid i \geq 0\}$

Fix closure. The above least upper bound is similar enough to the least upper bounds while determining least fixed points that the reader might be tempted to equate ent_f with the least fixed point of **de**[[*declaration*]]. Note however that ent_0 is not $\perp$ (in a more general setting ent_0 will contain valuable information about external identifiers).

* The usual approach (see for example the treatment of statement labels in Milne and Strachey [mil76, pp. 52,54] or [gor78]) is to first determine the identifiers declared in a block. Once the identifiers have been determined, a k-tuple is constructed from these identifiers. Then, using a valuation like **de**, we determine the type associated with each identifier. But instead of entering the type in the environment — as done by valuation **de** — a k-tuple of types associated with the k-tuple of identifiers is constructed. We start with with a k-tuple giving $\perp$ as the type of each identifier: the valuation successively determines tuples of types that are better approximations to the final types.

We therefore introduce a new operator **clo** (from fix closure)* such that given a function $\tau \in \mathbf{D} \to \mathbf{D}$ and $x \in \mathbf{D}$,

$$\mathbf{clo}(x)(\tau) = \bigsqcup\{\ \tau^i(x)\ |\ i \geq 1\}$$

Using **clo** we get the semantic rule:

$[\![all\_decl]\!]ents \in$ **Ent**

| *declaration*

‡ **clo**(*ent* 0)(**de**[[*declaration*]]) **where** *ent* 0 = **dz**[[*declaration*]]*ents*

2.4. *Data declarations.* The approach of the last two sections suffices to treat all data declarations in C. Types can be determined as in section 2.3 even if the declaration of structure tags is nested, or if the declaration of a structure tag is combined with the use of the tag to declare another identifier. Some of the rules change because a type specifier may contain the declaration of a structure tag so a type specifier can have the side effect of changing the environment.

3. Statements

Since C contains goto's, the semantics of statements were given using continuations [abd75, mor70, stw74]. The section containing the semantics of statements was written by editing section 9 of the C reference manual [ker78]. English descriptions of the various constructs were retained, and subsections containing semantic rules were added.

The reader can easily reconstruct the semantics of statements after reading [gor78] or [sto77].

Alternatives to continuations for handling goto statements have been proposed in [jon78] and [ros77]. The basic idea of the exit approach [jon78, pp.285] is to associate with each statement a function from a state to a state-label pair. If evaluation proceeds normally, then the label is "NIL" and is ignored; otherwise the label gives the target of the jump. In [ros77, pp.41], entries (embedded labels within a statement that can be jumped to) and exits (targets of jumps from within the statement) are explicitly identified, and rather than the meaning of a statement, the meaning of an entry-statement-exit triple is considered. We have not explored the use of these alternatives to continuations.

4. Expressions

C has a rich set of operators. In addition to the expected operators are a class of assignment operators of the form *op*=. The behaviour of *exp*1 *op*= *exp*2 is roughly equivalent to *exp*1 = *exp*1 *op* *exp*2, except that *exp*1 is evaluated just once.

The semantics of expressions are relatively straightforward. Some points of interest are noted below.

4.1. *Side effects and order of evaluation.* The C reference manual leaves unspecified the order of evaluation, and the same expression may be evaluated differently on different machines. Since assignments may be embedded within expressions, the value of an expression depends on the evaluation order. In practice, this does not present a problem, since programmers tend to stay away from such expressions.

The principle we have used is: only those expressions that are not sensitive to the order of evaluation are legal.

Unfortunately there is no way of checking if a given expression conforms to this principle. So we have picked a particular order of evaluation in the semantic specification. For all legal expressions any order of evaluation gives the right value, so the semantic specification will clearly give the right value. There remains the problem of ruling out non-legal expressions. The best we can do is to devise a heuristic in the spirit of *lint* [joh79], which will complain if the value of an expression is likely to depend on the order of evaluation.

4.2. *Checks: error, type, ...* This point actually applies to the whole language, but this is a good place to discuss it.

---

* For the unsophisticated reader it is just as easy to explain the **clo** operator as it is the least fixed point operator. Use of the **clo** operator simplifies some of the rules. The alternative, using least fixed points, employed by Tennent in his Pascal specification [ten77], requires the initial environment to be supplied as an extra argument, so the rules contain two environments at the same time. However, care must be exercised in using the **clo** operator: before writing **clo**$(x)(\tau)$ it is important to show that $x \sqsubseteq \tau(x)$. By inserting $\perp$ for the type of each identifier declared in the current block, valuation **dz** ensures that *ent* 0 is weaker that **de**[[*declaration*]]*ent* 0. We later learnt that the theoretical properties of **clo** have been studied in [cou79].

Speaking operationally, it is convenient to take the view that an error causes execution to terminate with an appropriate message. (This remark applies only to the semantic specification: the compiler certainly does not have to work this way.) The alternative to terminating execution is to propagate errors i.e. any operator applied to an error value yields an error value, so when the program finally ends, the error value is produced. The disadvantage with propagating errors is that the various operators have to be extended to apply to error values, and this may complicate the specifications for the operators.

As in [gor78, ten77], it is more convenient to separate the semantics of checks from the semantics that assumes that nothing goes wrong. It is then possible to use different methods for these two kinds of semantics. In particular, treating errors as jumps to the end of the program, we use continuations in the semantics that performs checks; but it is not necessary to use continuations in the "normal" semantics.

A more concrete example is as follows. With one minor exception — the sizeof operator — it is not necessary to carry around the type of each identifier when nothing goes wrong. (Structure tags will continue to be mapped to types by the environment, but all other identifiers will be mapped to locations or lvalues.) When type checking is performed, we need to know the types of identifiers, but the storage aspects need not be considered. Thus modularizing checking and run-time semantics is quite natural in this case. The sizeof operator maps an identifier to an integer equal to the "size" of the storage for the identifier. (We can formalize size by defining it to be the number of locations in the lvalue for the identifier.) Since the size is determined by the type of an identifier, the semantics of the sizeof operator are given separately from the run-time semantics. Since continuations are not needed for the semantics of the sizeof operator, its semantics can be separated from the semantics of type checks as well.

Exception: since the types of the arguments of a function need not be declared fully, the checking for compatibility between the types of formal and actual parameters cannot be separated from the run-time semantics.

5. Language issues

Much of the time devoted to specifying C was spent in learning the language. Here we mention some of the considerations that arose in the process. (The compiler writers were most helpful in volunteering information about known troublespots.)

5.1. *Blocks versus unrestricted jumps.* The C reference manual [ker78] allows a goto to jump anywhere within the current function, even if the jump is into the middle of a compound statement containing declarations. The compilers for C [joh78, rit79] resolve the issue of jumps into blocks by preprocessing blocks away: declarations of local identifiers are moved to the head of the function in which the declarations appear, after suitably renaming identifiers and converting initializations into explicit assignments.

Several drafts of the semantics of statements attempted to stay close to the language as it is implemented by preprocessing blocks away. However, we later discovered that the compilers overlay storage for independent nested blocks. Thus preprocessing declarations by assigning distinct new "names" to distinct uses of an identifier was also not consistent with the implementations. We have therefore specified semantics assuming that the language has blocks in the usual sense. Even if goto's are restricted from jumping into blocks, this is not quite accurate, since the compilers presently disallow a label from being redeclared in an inner block.

5.2. *Types of structures.* Structure members were originally implemented as offsets from the address of a structure, and there is a description of the restrictions on member names in the reference manual [ker78]. The first attempt at specifying the semantics of structure references was an abstraction of the description in [ker78]. The attempt shared a problem with the compilers. Suppose that in an outer block we have a structure declaration:

```
struct tnode { int header; int count; } fat;
```

and in an inner block there is a declaration:

```
struct confuse { int count; char info[37]; };
```

Since count refers to a different offset in the inner block (from the offset in the outer block), all references to fat.count in the inner block will incorrectly refer to the first component of the structure fat.

The compilers now keep member names with the structure, and member names can be redeclared in inner blocks without running into the problem mentioned above.

5.3. *One pass nature.* Except in structure declarations, all identifiers must be declared before they are used. This is also true of identifiers that are declared to be synonyms for types by typedef declarations. (In

the typedef declaration that follows, PX is declared to be a synonym for the type corresponding to: pointer to structure with tag x.) Thus the following declarations are legal, but reordering the declarations, or trying to combine them into one declaration is not allowed.

```
typedef struct x *PX;
struct x { int count; PX point; };
```

The semantic specification does not attempt to be faithful to the compilers and would allow the order of the above declarations to be changed.

The one pass nature of the compilers also shows up in the fact that the scope of a declared identifier seems to be from the point of declaration to the end of the block. Thus if x is a structure tag in an outer block, it can still be used to declare structures in and inner block until it is redeclared to be something else. We chose to take the scope of an identifier to be the entire block in which the declaration of the identifier appears.

6. Notes

6.1. *Metalanguage.* A subsidiary goal was to study the metalanguage needed to specify a language like C. Recent work on the automatic construction of interpreters from semantic specifications has used denotational semantics as a starting point [mos78, bjo78, don78]. We therefore decided to construct a denotational specification of C, and wanted to see where in the specification the various "features" of the metalanguage were needed. The style of the semantic rules was motivated by the translator generator *yacc* [joh75]. Except for the avoidance of Greek letters, the metalanguage is essentially that of the Oxford school. Using the same nonterminal names, with essentially the same meaning, as in the reference manual [ker78] contributed materially to thez6readability of the semantic specification.

6.2. *Supporting documents.* Since no prior knowledge of denotational semantics was assumed, a detailed introduction to denotational semantics was circulated. The need for a formal notation was motivated by mentioning a familiar construct whose meaning is hard to specify in English: the reference manual [ker78] deliberately bows to readability when it gives the meaning of the for statement in terms of a code fragment; this code fragment misrepresents the meaning of the for when the body of the for has a continue statement in it.

In retrospect, it would have been better to have focused on declarations before launching into statements. The simple data declarations discussed in section 2.2 are an excellent vehicle for introducing the notations and conventions. With statements on the other hand, continuations were used because of goto, break, and continue statements, and fixed points were used for while statements.

6.3. *Sublanguages for declarations.* The order of presentation of the semantics of declarations (section 2) was selected after the early draft doing the semantics of all declarations together was taken on faith by all who read it.

6.4. static *or* own *identifiers.* The denotational semantics of own identifiers [gor78] is inelegant. Such identifiers were therefore preprocessed away by making them external to all C-functions, and suitably renaming identifiers to separate scopes. This is essentially what the denotational semantics does, but it is cleaner to algorithmically specify a preprocessor.

6.4. *Expressions.* The finiteness of machine arithmetic and the order of evaluation in the presence of side effects led to more discussions than we care to recall. Once we began to understand declarations and get a feel for the whole language these problems faded in prominence, although they did not go away.

Acknowledgements

It is a pleasure to acknowledge the support and assistance of my many colleagues at Bell Laboratories where much of the work described here was done. S. R. Bourne, F. T. Grampp, S. C. Johnson, B. W. Kernighan, A. R. Koenig, M. D. McIlroy, and D. M. Ritchie at Bell Labs were all very patient and helpful. R. D. Tennent of Queens University fielded a number of questions about his denotational semantics of Pascal [ten77]. But for Cary Coutant's ministrations to the typesetter at Arizona this paper would not have been typeset.

References

abd75 S. K. Abdali, A lambda calculus model of programming languages: I. simple constructs; II. jumps and procedures, *Computer Languages 1*, 4 (1975) 287-301,303-320.

bjo78 D. Bjorner and C. B. Jones, *The Vienna Development Method: The Meta-Language,* Lecture Notes in Computer Science 61, Springer Verlag, Berlin, (1978).

con79 R. L. Constable and J. E. Donahue, A hierarchial approach to formal semantics with application to the definition of PL/CS, *TOPLAS 1*, 1 (July 1979) 98-114.

cou79 P. Cousot and R. Cousot, Constructive versions of Tarski's fixed point theorems, *Pacific J. Math* to appear.

don78 V. Donzeau-Gouge, G. Kahn, and B. Lang, A complete machine-checked definition of a simple programming language using denotational semantics, Rapport de Recherche No 330, IRIA Laboria, Rocquencourt France, (October 1978).

gor78 M. Gordon, Notes on the descriptive techniques of denotational semantics, University of Edinburgh (February 1978).

gre79 I. Greif and A. Meyer, Specifying programming language semantics: a tutorial and critique of a paper by Hoare and Lauer, Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas (January 1979) 180-189.

hen78 W. Henhapl and C. B. Jones, A formal definition of Algol 60 as described in the 1975 modified report, in D. Bjorner and C. B. Jones (eds.), *The Vienna Development Method: The Meta-Language,* Lecture Notes in Computer Science 61, Springer Verlag, Berlin, (1978) 305-336.

hoa78 C. A. R. Hoare, Some properties of predicate transformers, *J. ACM 25*, 3 (July 1978) 461-480.

joh75 S. C. Johnson, YACC — Yet another compiler-compiler, Computing Science Technical Report No 32, Bell Laboratories, Murray Hill, New Jersey, (July 1975).

joh78 S. C. Johnson, A portable compiler: theory and practice, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona (January 1978) 97-104.

joh79 S. C. Johnson, Lint, a C program checker, in UNIX *Time Sharing System:* UNIX *Programmers Manual, Volume 2A,* Bell Laboratories, Murray Hill NJ (January 1979).

jon78 C. B. Jones, Denotational semantics of **goto**: an exit formulation and its relation to continuations, in D. Bjorner and C. B. Jones (eds.), *The Vienna Development Method: The Meta-Language,* Lecture Notes in Computer Science 61, Springer Verlag, Berlin, (1978) 278-304.

ker78 B. W. Kernighan and D. M. Ritchie, *The C Programming Language* Prentice-Hall, Englewood Cliffs, New Jersey, (1978).

luc70 P. Lucas and K. Walk, On the formal description of PL/I, *Annual Review in Automatic Programming 6*, 3 (1970) 105-182.

mil72 R. E. Milne, The mathematical semantics of Algol 68, unpublished manuscript, Programming Research Group, Oxford University, (1972).

mil76 R. E. Milne and C. Strachey, *A Theory of Programming Language Semantics, 2 Vols.* Chapman and Hall, London and John Wiley, New York, (1976).

mor70 F. L. Morris, The next 700 programming language descriptions, unpublished manuscript, (November 1970).

mos74 P. D. Mosses, The mathematical semantics of Algol 60, Technical Monograph PRG-12, Programming Research Group, Oxford University, (1974).

mos78 P. D. Mosses, SIS: A compiler-generator system using denotational semantics (Reference Manual), Draft, ref. nr. 78-4-3, Department of Computer Science, University of Aarhus, Denmark, (June 1978).

rit78 D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, The C programming language, *BSTJ 57*, 6 part 2 (July-August 1978) 1991-2020.

rit79 D. M. Ritchie, A tour through the UNIX C compiler, in UNIX *Time Sharing System:* UNIX *Programmers Manual, Volume 2B,* Bell Laboratories, Murray Hill NJ (January 1979).

ros77 B. K. Rosen, Applications of high level control flow, Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California (January 1977) 38-47.

scs71 D. S. Scott and C. Strachey, Towards a mathematical semantics for computer languages, Proceedings of the Symposium on Computers and Automata Polytechnic Press, Brooklyn, New York, (April 1971) 19-46.

sto77 J. E. Stoy, *Denotational Semantics — The Scott-Strachey Approach to Programming Language Theory,* MIT Press, Cambridge MA and London (1977).

str72 C. Strachey, Varieties of programming language, Proceedings, International Computing Symposium, Cini Foundation, Venice (April 1972) 222-233.

stw74 C. Strachey and C. Wadsworth, Continuations: a mathematical semantics which can deal with full jumps, Technical Monograph PRG-11, Programming Research Group, Oxford University, (1974).

ten73 R. D. Tennent, Mathematical semantics of Snobol 4, ACM Symposium of Principles of Programming Languages (October 1973) 95-107.

ten76 R. D. Tennent, The denotational semantics of programming languages, *Comm. ACM 19*, 8 (August 1976) 437-453.

ten77 R. D. Tennent, A denotational definition of the programming language Pascal, Technical Report 77-47, Department of Computing and Information Science, Queen's University, Kingston, Canada, (July 1977).