A Hierarchial Approach to Formal Semantics With Application to the Definition of PL/CS

ROBERT L. CONSTABLE and JAMES E. DONAHUE
Cornell University

We describe a means of presenting hierarchically organized formal definitions of programming languages using the denotational approach of D. Scott and C. Strachey. As an example of our approach, we give the semantics of PL/CS, an instructional variant of PL/I. We also discuss the implications of this approach to language design, pointing out some cases where the wrong choices may cause the hierarchy to collapse into chaotic rubble.

Key Words and Phrases: programming language semantics, denotational semantics, recursive functions, PL/I, PL/C, PL/CS
CR Categories: 4.20, 4.22, 5.24

1. INTRODUCTION

One of the important goals of research in the formal semantics of programming languages is to develop techniques for improving the language design process. In particular, formal semantics may serve as a proscriptive tool, warning the language designer of unexpected complexity introduced by unfortunate compositions of language constructs. This paper describes one approach to structuring language definitions so that unnecessary complexity can be exposed, and it shows the application of this technique to the definition of an instructional variant of PL/I, PL/CS [4].

The programming language PL/CS was designed as a simple PL/I dialect that would be suitable for teaching introductory programming. Thus many features of PL/I or PL/C have been omitted and others restricted, particularly in those cases where syntactic restrictions would improve the chance of error repair. (For example, all variables in PL/CS programs must be declared at the beginning of each procedure.) A full list of PL/CS statements is given in Figure 1.

We believe that a simple language like PL/CS should have a simple semantics. Using the denotational approach of Scott and Strachey [13], we tried to produce

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
This research was supported in part by the National Science Foundation under Grants DCR-74-14701 and MCS-76-14293.
Author's address: Department of Computer Science, Cornell University, Upson Hall, Ithaca, NY 14853.
© 1979 ACM 0164-0925/79/0700-0098 $00.75

```
numeric variable = arithmetic expression;
[string variable = string expression;]
[SUBSTR (s, f, n) = string expression;]
[array = array;]
[array = constant;]
[bit variable = (condition);]
[IF   (condition)
      THEN statement;]
 IF   (condition)
      THEN statement 1;
      ELSE statement 2;
 DO;
      body;
      END;
[select-name: SELECT;
       WHEN (condition 1) statement 1;
       WHEN (condition 2) statement 2;
       ...
       OTHERWISE statement n;
       END select-name;]
 loop-name: DO WHILE (condition);
       body;
       END loop-name;
[loop-name: DO UNTIL (condition);
       body;
       END loop-name;]
[loop-name: DO index var = expr 1 TO expr 2 BY expr 3;
       body;
       END loop-name;]
 LEAVE loop-name;
[GOTO target-label;
 ...
 target-label:;]
[ON ENDFILE GOTO DATAEND;
 ...
 DATAEND:;]
 CALL procedure-name;
 CALL procedure-name (arguments);
 RETURN;
 RETURN (expression);
[GET LIST (variables);
 GET DATA (variables);
 GET DATA;
 GET EDIT (variables) (format items);]
[PUT SKIP (expr) LIST (expressions);
 PUT SKIP (expr) DATA (variables);
 PUT SKIP (expr) EDIT (values) (format items);]
[ASSERT (condition);
 ASSERT (cond) FOR ALL index var = expr 1 TO expr 2 BY expr 3;
 ASSERT (cond) FOR SOME index var = expr 1 TO expr 2 BY expr 3;]
```

Fig. 1. PL/CS statements

a simple, but complete and rigorous, definition of PL/CS. (This semantics is given in our paper [3].) In this paper, we describe the structure of the PL/CS semantics; space limitations prevent us from presenting the entire language. Those statements enclosed in brackets in Figure 1 are not discussed in this paper. Many of

them are variations of statements for which we do supply semantics (e.g. IF (Cond) THEN Stmt; is a variant of IF (Cond) THEN Stmt; ELSE Stmt;) and others require more detailed descriptions than we can provide in this paper; we refer the reader to [3] for more complete details.

That we were able to define PL/CS in a formal manner is hardly surprising; others (Bekić et al. [1], Walk et al. [16], ECMA [8]) have produced definitions of either full PL/I or PL/I subsets much larger than PL/CS. However, these definitions were written primarily for an audience with knowledge of both PL/I and formal methods. The intended audience of our PL/CS definition was assumed to be far less knowledgeable in both areas; thus we have been concerned not only with "getting the definition right," but with presenting it in a pedagogically sound manner.

The organization of our definition was based on the following "hierarchy principle." In any language, the meaning of some features depends only inessentially on the meaning of others. For example, the meaning of procedures probably does not depend on the arithmetic operations provided in the language. In terms of a formal semantics, this allows us to give definitions which leave some details unspecified in much the same way that well-structured programs allow some parts to be written independently of others. If our language is "simple" enough (perhaps "well-structured" is more appropriate here), then these levels of detail can be arranged in levels, so that we can proceed by defining ever more detailed sublanguages, eventually producing the definition of the complete language. This is the approach we followed in the PL/CS definition and which we describe below.

Our definition requires two basic sets of values (or *domains*):

(1) The set of *states* manipulated by statements in PL/CS. The meaning of each PL/CS statement is defined as a "state transformation function."
(2) The set of *values* produced by expressions in PL/CS. The meaning of each expression will be defined by a "value-producing" function (relative to the current state, of course).

Our hierarchy of language constructs is based on the degree to which these domains need to be specified in order to define a particular statement. As we see below, the simplicity of the language is captured by the degree to which we have been able to separate levels of semantic detail and avoid a "spaghetti definition." Following the details of the definition presented in Section 2, we conclude by discussing some design alternatives that, had they been chosen, would have made this "hierarchical" approach to semantics impossible.

2. THE PL/CS SEMANTIC HIERARCHY

2.1 Organization

As noted above, PL/CS can be defined by isolating several important sublanguages and presenting, in turn, the semantics of each of these sublanguages. The order we use is the following.

We begin with the language of *while schemes*. Essentially, while schemes include the control constructs basic to structured programming: sequencing, binary selection, and iteration. This subset has a particularly elegant and easy to

understand semantics.

The next subset is the PL/CS *flowchart schemes*, formed by extending the while schemes with the simple (forward) transfers of control allowed in PL/CS. The introduction of control transfers causes the first refinement in the underlying state. As we describe below the simple jumps of PL/CS have a much more intuitive semantics and require a less complicated state than the more complex transfers of control of full PL/I (or PL/C).

We next introduce *procedure declaration*, *call*, and *return* to form the language of PL/CS *recursion schemes*. To define the semantics of recursion schemes, we are forced to give a still more detailed description of the underlying "state" manipulated by PL/CS programs and to define the "meanings" of identifiers in programs.

In the next PL/CS sublanguage, we introduce the notion of *program variables*, causing a further refinement of the "state" to include a "machine store" component. In this sublanguage we define the meanings of variable declarations and parameter passing to PL/CS procedures. Finally, we complete our description of PL/CS by giving a detailed definition of the meaning of the value-producing *expressions* in the language, as well as the semantics of assignment and function call and return.

2.2 While Schemes

The simplest PL/CS sublanguage is the language of while schemes. PL/CS while schemes are defined by the following syntax[1]:

```
Stmt = SimpleStmt
     | IF (Cond) THEN Stmt; ELSE Stmt;
     | DO WHILE (Cond); Stmt; END;
     | Stmt; Stmt;
     | DO; Stmt; END;
```

For the present, we leave the class of simple statements unspecified. However, we note in passing that simple statements compose the basic actions of programs in the language, including assignment, procedure call, and input/output commands.

The meaning of each of the statements in this language is defined in terms of a function *Ms* which maps each statement in the language into its associated state transformation, i.e. *Ms* has functionality

$$Ms\colon \text{Stmt} \rightarrow [S \rightarrow S].$$

At present, because of the simplicity of this sublanguage, none of the details of the eventual structure of S need be known to give the definition. As in [15], the definition of *Ms* is given by presenting a clause of the definition for each syntactic category of the class of statements. (The brackets ⟦and⟧ are used to bracket the syntactic arguments as an aid to the eye.) Additionally, because Boolean expressions appear as the conditions in while schemes, we will need some minimal

[1] We modify the presentation of productions in a grammar by using a name with an initial capital letter, such as Cond, to denote the class of all conditions (Boolean expressions). We then write "cond," a nonterminal, to denote an element of the class.

knowledge of

Me: Exp → S → V,

the meaning function for expressions. At present, we simply need that Boolean expressions can produce Boolean values, i.e. that there exists a function

Me_{Cond}: Cond → S → {true, false}.

The clauses of *Ms* follow:

Ms ⟦ simple_stmt;⟧ = some particular f: S → S associated with
each simple statement
Ms ⟦ IF (cond) THEN $stmt_1$; ELSE $stmt_2$;⟧ (s) =
if Me_{Cond} ⟦cond⟧ (s) then *Ms* ⟦$stmt_1$;⟧ (s)
else *Ms* ⟦$stmt_2$;⟧ (s)
Ms ⟦ DO WHILE (cond); stmt; END;⟧ (s) =
if Me_{Cond} ⟦cond⟧ (s)
then *Ms* ⟦DO WHILE (cond); stmt; END;⟧ (*Ms* ⟦stmt;⟧ (s))
else s
Ms ⟦ $stmt_1$; $stmt_2$;⟧ (s) =
Ms ⟦$stmt_2$;⟧ (*Ms* ⟦$stmt_1$;⟧ (s))
Ms ⟦ DO stmt; END;⟧ (s) = *Ms* ⟦stmt;⟧ (s)

This semantics relies only on simple notions of Boolean expression, function compositions, and recursive function definitions. To ensure that the recursive definition of the meaning of *while* defines a function, we assume that whatever internal structure S has it is a *domain* (as defined in [15]). Then we can appeal to the least fixed point construction of [15, sec. 4.3] to give a functional meaning to the *while* clause. (We could also view this recursion equation as having a computational meaning in terms of replacement rules [12] or we could give an inductive definition of its graph as in [2].)

2.3 Flowchart Schemes

The simple definition of *Ms* ⟦ ⟧ in subsection 2.2 fails as soon as we add explicit transfers of control, like the PL/CS LEAVE or GOTO. The difficulty then is that statement composition is not function composition. This is immediately obvious for

GOTO l; stmt;

but is more insidious in the case of

DO WHILE (cond); $stmt_1$; END; $stmt_2$

in which the $stmt_1$ contains a GOTO.

In PL/CS, all transfers of control (GOTOs and LEAVEs) are restricted to be forward only, may not effect a transfer into a composite statement (e.g. into the arms of a conditional), and may not involve crossing a declaration boundary (e.g. jumping out of a procedure). (Indeed, PL/CS has *no* block structure, but for our purposes the previously stated restrictions are sufficient.) Because of this simple form of jumping, a minor refinement of our earlier S is sufficient to allow a straightforward definition.

The basic idea is to add to the state a component specifying the context in which the program is evaluated; this context gives the meaning of all the

statements to which control could be transferred. In particular, it provides the meaning of the next statement in sequence to be executed. The interested reader will note that this method is a simple form of continuations described by Strachey and Wadsworth [14] and by Tennent [15].

The syntax of the new PL/CS statements allowing transfers of control is

```
Stmt  = Id: DO WHILE (Cond); Stmt; END;
      | LEAVE Id;
```

The first rule allows loops to be labeled, the second provides the syntax for specifying exits from labeled loops.

To define the language of PL/CS flowchart schemes, we extend the meaning function *Ms* defined above to take an additional argument, the *context* in which GOTO statements are to be evaluated. The new meaning function is of type[2]

Ms: Stmt → Context → S → S

where the second argument is of type

Context = [Id + {*rest*}] → [S → S].

Given some c ∈ Context,

(1) c ⟦id⟧ gives the meaning of the remainder of the program beginning *after* the loop labeled id, i.e. the meaning of the program to which a LEAVE id would transfer control;
(2) c ⟦*rest*⟧ gives the meaning of the remainder of the program after the current statement.

The clauses of the definition are given below. The meaning of conditional statements does not change, so it has been omitted. The major changes involve the meaning of statement composition and the meaning of *while* loop (because of the need to record the label).

Ms ⟦simple_stmt;⟧ (c) (s) = (c ⟦*rest*⟧ (f(s))

where f: S → S is the state transformation associated with simple_stmt. So the meaning of a simple statement, simple_stmt, in a context c on state s, is the meaning of the rest of the program applied (c ⟦*rest*⟧) to the state produced by executing the simple statement on s. The way in which the context is provided will be explained by subsequent equations.

In these equations we use the notation $c[id_1 \leftarrow f_1; id_2 \leftarrow f_2; ...; id_n \leftarrow f_n]$ for distinct id_i to mean the function defined by

$c[id_1 \leftarrow f_1; id_2 \leftarrow f_2; ...; id_n \leftarrow f_n]\ \ [\![id_i]\!] = f_i \quad i = 1, ..., n$
$c[id_2 \leftarrow f_1; id_2 \leftarrow f_2; ...; id_n \leftarrow f_n]\ \ [\![id']\!] = c\,[\![id']\!]$ otherwise

that is, $c[id \leftarrow f]$ denotes the effect of the "assignment id ← f" on the mapping c.

[2] We associate type equations to the right so that this means the following: Stmt → [Context → [S → S]].

Ms ⟦id: DO WHILE (cond); stmt; END;⟧ (c) (s) =
  if Me_{Cond} ⟦cond⟧ (s)
    then Ms ⟦stmt⟧ (c[id ← c ⟦*rest*⟧;
      rest ← λs.Ms ⟦id: DO WHILE (cond); stmt; *end*;⟧ (c) (s)]) (s)
    else c ⟦*rest*⟧ (s)

Notice that a new context is provided by assigning meaning to the label and by adding the while loop itself to the rest of the program (thereby causing a recursive definition). From the definition of composition we can see that adding the loop to the context is equivalent to using

Ms ⟦stmt; DO WHILE (cond); stmt; END⟧ in place of Ms ⟦stmt⟧

in the equations

Ms ⟦$stmt_1$; $stmt_2$⟧ (c) (s) =
  Ms ⟦$stmt_1$;⟧ (c[*rest* ← λs.Ms ⟦$stmt_2$;⟧ (c) (s)]) (s)
Ms ⟦LEAVE id;⟧ (c) (s) = c ⟦id⟧ (s)

LEAVE statements just continue with the program after the labeled loop.

2.3 PL/CS Recursion Schemes

The next control constructs to be added to the PL/CS language are procedure declaration, call, and return. These constitute the language of PL/CS recursion schemes. (For the moment, we ignore the possibilities of parameter passing; this aspect of procedures is discussed in (subsection 2.4).)

The syntax of PL/CS recursion schemes is the following. A PL/CS recursion scheme is a main procedure body followed by a finite list of procedure declarations:

```
Program = Id: MainProc; {Id: ProcDef;}
ProcDef = PROCEDURE; Stmt; END
MainProc = PROCEDURE OPTIONS(MAIN); Stmt; END
```

Procedure bodies are simply statements, as defined above for flowchart schemes, with the following additions to allow procedure call and return:

```
Stmt  = CALL Id;
      | RETURN;
```

To define this sublanguage, we obviously need some way to associate identifiers with the "meaning" of declared procedures. The important aspect of procedure declaration is that, because of the static scope rules for PL/CS, the meaning associated with procedure identifiers is fixed throughout the execution of a program. This contrasts with the dynamically changing values of the program variables. Thus we separate the "program state" used earlier into (1) a static *environment*, which for the present simply associates "procedure values" with procedure identifiers; and (2) a "store" component containing the values of the program variables. For now, we leave further details of the store unspecified.

Thus the new definition will use the following domains:

Env = Id → Proc	environments
Proc = S → S	procedure value
S	stores

and the following meaning functions:

Ms: Stmt → Context → Env → S → S
Mp: ProcDef → Env → Proc
Mprog: Program → Env → Proc

The new meaning function for statements involves no changes from the previous one, except for the addition of new clauses to define procedure call and return. In all other cases, one can simply replace the argument S used in the previous *Ms* by (e, s), the new environment, and store. The new clauses of *Ms* are given below:

$$Ms\ [\![\text{CALL id}]\!]\ (c)\ (e,s) = (c\ [\![rest]\!])\ (e\ [\![\text{id}]\!]\ (s))$$

A procedure call is a simple statement taking the procedure value associated with the identifier in the environment and applying it to the current store to produce a resulting store.

$$Ms\ [\![\text{RETURN}]\!]\ (c)\ (e,s) = s$$

Return statements produce the final result of a procedure.

The final question to be resolved by the semantics of recursive schemes is how procedure declarations bind identifiers to procedure values. Here, because PL/CS procedures may be mutually recursive, we must use a simultaneous recursive definition, i.e. a recursive definition which simultaneously gives the meaning of all of the declared procedures.

We evaluate procedures bodies in a context in which no labels are defined; the use of *rest* is simply to return from the procedure when it finishes. We let C_0 denote the context that binds $\perp$ to all identifiers, e.g. no value is assigned to any identifier,

$$Mp\ [\![\text{PROCEDURE; stmt; END}]\!]\ (e) = \lambda s.Ms\ [\![\text{stmt;}]\!]\ (C_0\ [rest \leftarrow \lambda s.s])\ (e,s)$$
$$Mprog\ [\![\text{id: mainproc; } id_1\text{: procdef}_1\text{; } id_2\text{: procdef}_2\text{; ...; } id_n\text{: procdef}_n]\!]\ (e) = Mp\ [\![\text{mainproc}]\!]\ (e')$$

where

$$\begin{aligned} e' = e[&id_1 \leftarrow Mp\ [\![\text{procdef}_1]\!]\ (e');\\ &id_2 \leftarrow Mp\ [\![\text{procdef}_2]\!]\ (e');\\ &\quad\vdots\\ &id_n \leftarrow Mp\ [\![\text{procdef}_n]\!]\ (e')] \end{aligned}$$

Note that this definition of e′ is no more complex than an ordinary recursive function definition of the form

$$f_1 = \lambda s.Ms\ [\![\text{stmt}_1\ (f_1, \ldots, f_n)]\!]\ (s)$$
$$\vdots$$
$$f_n = \lambda s.Ms\ [\![\text{stmt}_n\ (f_1, \ldots, f_n)]\!]\ (s)$$

Our decision to give a definition of recursive procedures in terms of a recursive definition of an environment is simply a matter of notational convenience. By rewriting the definition slightly, we can highlight the crux of the matter, which is the mutually recursive definition of a set of procedures, e.g.

$$e'\ [\![id_1]\!] = \lambda s.Ms\ [\![stmt_1]\!]\ (C_0[rest \leftarrow \lambda s.s])\ (e[id_1 \leftarrow e'\ [\![id_1]\!]; \ldots; id_n \leftarrow e'\ [id_n]\!], s)$$

$$\vdots$$

$$e'\ [\![id_n]\!] = \lambda s.Ms\ [\![stmt_n]\!]\ (C_0\ [rest \leftarrow \lambda s.s])\ (e[id_1 \leftarrow e'\ [\![id_1]\!]; \ldots; id_n \leftarrow e'\ [id_n]\!], s)$$

Again, if we assume that Env and S are domains, we can appeal to the usual least fixed point construction to give a functional meaning to this recursive system of equations.

2.4 PL/CS Variable Declaration and Parameter Passing

A "program state" merely indicates the status of the program's world. So far our abstract states have indicated only the *form* of this world without mention of its content. In general, the execution of a program will create objects and examine them to detect certain properties.

In PL/CS only simple objects can be directly created (objects like numbers, strings, or arrays of numbers and strings, but not objects like sets or functions or graphs). We call these objects *values*. The language provides basic operations for building values, such as addition of numbers, concatenation of strings, and certain tests of properties of objects, such as equality of numbers and strings. (These operations and tests form the *expressions* and *Boolean expressions* of PL/CS.) But regardless of exactly how we build objects in a language (i.e what semantics are given to expressions), we have the more basic problem of how to name them and pass them around to be modified or tested. So before discussing the way we build values, we examine how we name them and move them regardless of what they are.

We first introduce a domain Val of values which are produced by the expressions in the language and are stored in the state as values of the program variables. (At present, the structure of Val is unspecified.) Of major importance in designing a language is deciding how values are associated with the identifiers used to denote variables in a program. There are two essential ways to do this. One way is to allow identifiers to reference values directly, so identifiers are *reference objects*. Another way is to assume a mathematical object called a reference object (or *location*, or *address*) and let identifiers name such objects. These reference objects in turn name the values.

The first method, having identifiers directly reference values, is characteristic of ordinary mathematical notation. One says, "let x range over set S," or "let x be an integer"; and this creates a "variable" x whose values belong to S. For example, in ordinary mathematical notation we do not have an operation for explicitly changing the value of a variable. Even when we do change the value of a mathematical variable we do so by changing context, not by computation.

In programming languages, we have operations for explicitly changing the value associated with an identifier (such as assignment or new declarations on block entry). Because of this, the concept of a *reference object* or *location* is commonly introduced to allow straightforward definitions of the dynamic changes of identifier meaning (especially scope rules for identifiers and parameter-passing mechanisms). By carefully restricting a language, one can avoid the need for locations (c.f. Donahue [7]), but the notions of static and external variables in

PL/CS (see below) are not compatible with the necessary restrictions. So we adopt locations as a domain in this semantics.

In order to present the PL/CS language features associated with the manipulation and referencing of values and allocation of locations, we need to introduce the domain Val of *values* (mentioned above) and the domain Loc of *locations.* We refine the store so that it consists of mappings from locations to values, and we extend environments to include binding identifiers to locations. The new domain definitions include

Val is a set called the *values*
Loc is a set called the *locations*
S = Loc → Val
Env = Id → [Proc + Loc]
Proc = S → S

Given this interpretation of variables, we can ask two questions about the binding of a particular location to an identifier.

(1) What is the *scope* of such a binding, i.e. over what region of a program is it the case that this identifier is bound to this location? In PL/CS, there are two obvious choices for the scope of a variable. First, scope could be local to the procedure in which the variable is declared. This is the default scope in PL/CS. Second, scope could be global to the program, i.e. the binding potentially could be known to all procedures forming a program. In PL/CS, this is external scope. The bindings of external variables are known in all procedures in which the variable is declared to be *external.*

(2). What is the *extent* to such a binding, i.e. how long does it persist? Again, in PL/CS there are two possibilities. First, the bindings of local variables of a procedure could be established upon each procedure invocation (this is also the default in PL/CS). The second choice is to bind local identifiers to locations prior to evaluation of the program. In PL/CS, this is the meaning used if a variable is declared to be *static.*

One final wrinkle in our new notion of environment and store is introduced by the parameter-passing mechanisms of PL/CS. In the language (as in PL/I), all parameter passing is by reference, i.e. each argument to a procedure is a location, not a value. If the argument is a variable, then the location denoted by the identifier is passed; otherwise a temporary location containing the value of the expression argument is passed.

The new PL/CS sublanguage includes the following syntactic extensions to the preceding sublanguages:

Stmt = CALL Id (Arg*)
Arg = Id | Exp
ProcDef = PROCEDURE (Id*); Body; END
Body = Decl*; Stmt
Decl = DECLARE Id* [Access]
Access = EXTERNAL | STATIC
Program = Id: MainProc; {Id: ProcDef;}*
MainProc = PROCEDURE OPTIONS (MAIN); Body; END

The various semantic domains and functions used to define the language include

the following:

Val	values
Loc	locations
S = Loc → [Val × Tag]	stores
Tag = {true, false}	allocation tags

Allocation tags are used to record whether the location is currently accessible, i.e. is bound to a variable currently in use.

Env = Id → [Proc + Loc]	environments
Proc = Loc* → S → S	procedures

In this subset of the language, procedures are now parameterized.

Ms: Stmt → Context → Env → S → S
Mb: Body → Env → S → S
Mp: ProcDef → S → [[Env × Proc] → S]
Mprog: Program → Env → S → S

The new *Mp* must not only produce the procedure value specified by a procedure declaration, but must also allocate static variables declared within the procedure.

Finally, the new clauses of the definition are given below:

$$Ms\ [\![\text{CALL id(arg*)}]\!]\ (c)\ (e,s) = (c\ [\![rest]\!]\ (e\ [\![\text{id}]\!] \mid \text{Proc})\ (l')\ (s')$$

where $l', s' = Ma\ [\![\text{arg*}]\!]\ (e,s)$.

Notes. (1) We use $e\ [\![\text{id}]\!] \mid \text{Proc}$ to project values in Proc + Loc to Proc, i.e.

$$e[\![\text{id}]\!] \mid \text{Proc} = \begin{cases} e[\![\text{id}]\!] & \text{if } e[\![\text{id}]\!] \in \text{Proc} \\ \perp & \text{otherwise.} \end{cases}$$

(2) We use *Ma*: Arg → [Env × S] → [Loc × S] to give meaning to arguments to procedures. The details of *Ma* are omitted, as they are tedious and straightforward. Note that *Ma* produces a store as part of its result because the evaluation of arguments may require the allocation of "temporary" locations.

The definition of *Mb* (giving meaning to procedure bodies) is as follows:

$$Mb[\![\text{decl*; stmt}]\!]\ (e,s) = Ms\ [\![\text{stmt}]\!]\ (C_0\ [rest \leftarrow \lambda s.s])\ (e',s')$$

where $e', s' = \text{Alloc (Locals (decl*)}, e, s)$

Notes. (1) Locals: Decl* → Id* produces a list of all variable identifiers *not* declared as static or external variables, i.e. the declarations of local variables. The definition of Locals is not given to shorten the presentation.

(2) Alloc: [Id* × Env × S] → [Env × S] produces a new environment and store such that the identifiers referred to in the declaration are bound to previously unallocated storage locations, and these locations have been tagged as allocated in the resulting store. The formal definition of Alloc is straightforward and therefore omitted.

From earlier, the type of *Mp*, the meaning function for procedure declarations, was

Mp: ProcDef → S → [[Env × Proc] × S].

The definition of *Mp* is given by

Mp ⟦PROCEDURE (id*); decl*; stmt; END⟧ (s) = ⟨p, s′⟩

where p = λe.λl*.λs″.*Mb* ⟦decl*; stmt⟧ (e[id′ ← l′; id* ← l*, s″), with id′, l′, s′ = New(Statics (decl*), s).

Notes. (1) New: [Id* × S] → [Id* × Loc* × S] allocates new locations for each of the identifiers appearing in the sequence of declarations and returns (a) the identifiers for which storage was allocated, (b) the locations allocated, and (c) the altered store. Note that we can define the previously used Alloc function in terms of New by Alloc (id*, e, s) = ⟨e[id ← l], s′⟩ where id, l, s′ = New(id*, s).

(2) Statics: Decl* → Id* produces a list of all variable identifiers declared as *static* variables. (Statics is similar to Locals in its use and definition.)

Mprog ⟦ id: PROCEDURE OPTIONS (MAIN); body END;
Mprog ⟦id: PROCEDURE OPTIONS (MAIN); body END;
id_1: $procdef_1$; ...; id_n: $procdef_n$;⟧ (e, s) =
Mb ⟦body⟧ (e′, s′)

where

e_{ext}, s_{ext} = Alloc (Externals (Program), e, s);
p_1, s_1 = *Mp* ⟦$procdef_1$⟧ (s_{ext});
⋮
p_n, s_n = *Mp* ⟦$procdef_n$⟧ (s_{n-1});
s' = s_n;
e' = e_{ext} [id ← p_1 (e′); ...; id_n ← p_n (e′)]

where Externals: Program → Id* produces a list of all *external* variables declared in the program.

Some remarks on the meaning given to programs are clearly in order. The important point is that the definition includes both "sequential" and "simultaneous" components. The sequential component of the definition is the meaning given to procedures. Because procedures now may contain local static variables and references to external variables, it is necessary to sequentially allocate storage for the external variables and the local static variables of each procedure. Thus we form the sequence of stores s_{ext}, s_1, s_2, ..., s_n and the sequence of Env → Proc values p_1, ..., p_n.

The simultaneous component of the definition is the formation of the environment e′, similar to the simultaneous recursive definition of the environment used in the discussion of recursion schemes in subsection 2.3. It is the environment formed by this simultaneous definition which is used to evaluate the main procedure of the program.

2.5 Expressions and Assignment

We have now built up the semantic model of PL/CS to define most of the language. One aspect which remains undiscussed is the way in which *values* (elements of the semantic domain Val) may be produced. The evaluation of expressions forms the content of this final section. We introduce a new meaning function *Me* to define expressions and discuss the semantics of PL/CS function declaration, call, and return and the semantics of assignment.

In PL/CS, values can be constructed using certain basic functions, $f_i: Val^{n_i} \rightarrow Val$, and programmer-defined *function procedures*. These values are denoted by *expressions*. For example, various expressions for Boolean values (used as selectors in conditional and iteration statements) can be built up in terms of *atomic predicates*, $p_j: Val^{n_j} \rightarrow$ {true, false} and other programmer-defined predicates.

The basic syntax of PL/CS expressions is given below:

Exp = Id | Const | Exp Op Exp | Id(Arg$^+$) | Op Exp .

In PL/CS, all programmer-defined function procedures have the set theoretic type $0^n \rightarrow 0$, that is, of n-tuples of individuals to individuals. The declaration must specify the exact type of the function, e.g. number of arguments, their attributes, and attribute of the returned value,

FuncDef = Id: PROCEDURE(Id*)RETURNS(Type); Stmt; END

where Type is any allowable PL/CS type identifier, e.g. FIXED, FLOAT, etc. At this level of the semantic hierarchy we ignore the extra complications caused by type-checking—they constitute another level of semantics.

The meaning of a function declaration is similar to that of a procedure declaration, except that functions produce members of Val, rather than S, as final values. Thus there must be a method for specifying the value of the statement which makes up the function body. This is done by executing RETURN(Exp), which has the effect of storing the value of the expressions exp in a special component of the state called *fvalue*. Then after execution of the function procedure body, a special function, ValueOf: S → Val, is applied to return the value of the *fvalue* component. More precisely, we now have the following definitions of function declaration and return. First, we give the new semantic domains and meaning functions involved.

Env = Id → [Val + Proc + Loc + Fun]	environments
Fun = Loc* → S → Val	functions
S = Loc → [Val × Tag] × Val	stores

The second component of S will be used as the *fvalue* component to return the value produced by a function body.

Mf: FuncDef → Env → Fun

is the meaning function for function declaration.

Ms ⟦RETURN(exp)⟧(c)(e, s) = s[*fvalue* ← *Me* [⟦exp⟧(e, s)]
Mf: FuncDef → S → [[Snv → Fun] × S]

is defined by

Mf ⟦ PROCEDURE(id*)RETURNS(type); body END⟧(e) =
λl*.λs'.ValueOf(*Mb* ⟦body⟧(e[id* ← l*; id' ← l'], s'[*fvalue* ← ⊥])

Functions in PL/CS are not allowed to have side effects. Thus they may not contain STATIC variable declarations, nor may they change the values of any EXTERNAL variables.

We now define a PL/CS program to be a main procedure, followed by a sequence of procedure and function procedure declarations, i.e. we have the new syntactic definitions

Program = Id: MainProc {; Id: ProcDef}* {; Id: FuncDef}*

with the following new semantic definition of programs:

$Mprog$ ⟦id: PROCEDURE OPTIONS(MAIN); body END; id_1: $procdef_1$; ...; id_n: $procdef_n$; id_{n+1}: $funcdef_1$; ...; id_{n+m}: $funcdef_m$⟧(e, s) = Mb ⟦body⟧(e′, s′)

where

e_{ext}, s_{ext} = Alloc (Externals (program), e, s);

$p_1, s_1 = Mp$ ⟦$procdef_1$⟧(s_{ext});
.
.
.
$p_n, s_n = Mp$ ⟦$procdef_n$⟧(s_{n-1});
$f_1 = Mf$ ⟦$funcdef_1$⟧;
.
.
.
$f_{m'} = Mf$ ⟦$funcdef_m$⟧;
$s' = s_n$
$e' = e_{ext}[id_1 \leftarrow p_1(e'); \ldots; id_n \leftarrow p_n(e');$
$id_{n+1} \leftarrow f_1(e'_{Nop}); \ldots; id_{n+m} \leftarrow f_m(e'_{Nop})]$

where

$e_{Nop} = \lambda i$.if e⟦i⟧ is Proc then error else e⟦i⟧

i.e. e_{Nop} masks all procedure values in e. This is necessary in PL/CS to disallow the possibility of procedure calls inside the body of functions. (Remember that expressions are not allowed to have side effects.)

The meaning of expressions is given by a function

Me: Exp → Env → S → Val

defined as follows:

Me ⟦id⟧(e, s) = (e⟦id⟧ | Loc)(s)
Me ⟦const⟧(e, s) = const
Me ⟦op exp⟧(e, s) = Uop ⟦op⟧(Me ⟦exp⟧(e, s))

where Uop: Op → Val → Val defines each unary operation

Me ⟦exp_1 op exp_2⟧(e, s) = Bop ⟦op⟧ (Me ⟦exp_1⟧(e, s), Me ⟦exp_2⟧(e, s))

where Bop: Op → [Val × Val] → Val defines each binary operation in PL/CS

Me ⟦id(arg*)⟧(e, s) = (e⟦id⟧ | Fun)(l′)(s′)

where l′, s′ = Ma ⟦arg*⟧(e, s).

Now, given this definition of Me, we can define one of the basic statements of PL/CS, the simple assignment statement, as follows:

Ms ⟦id := exp⟧(c)(e, s) = (c ⟦$rest$⟧)(e, s[e⟦id⟧ ← Me ⟦exp⟧(e, s)]).

At this level, we have not included any definitions of particular unary or binary operators, because of our desire to leave the structures of Val unspecified. In [3] we define one further level of detail in which we refine Val to include the domain of integers. This further refinement allows us to define arrays and bounded iteration, two features of the language which depend on the meaning of integer-valued expressions. To shorten this presentation, this final refinement has been omitted.

3. CONCLUSION

The simplicity of PL/CS is evinced by its decomposability into distinct levels and the relative succinctness of its defining equations. To amplify this point, we note a few instances where a decision to extend the language to full PL/I would have severely complicated the definition. We also note some simplifications of PL/CS suggested by this semantics.

PL/CS severely restricts the forms of explicit transferred control allowed in PL/I. The only transfers in PL/CS are LEAVEs from loops and simple forward GOTOs that do not cross "phrase structure" boundaries, i.e. into the middle of a DO-group or conditional. Extending PL/CS jumps to include any modification to the complexity of jumps allowed would have affected the structure of the semantics.

For example, the most radical addition to PL/CS would be to allow jumps out of functions or procedures. But then the meaning of a function cannot be seen as a value-producing mechanism; it must include the semantics of never producing a value because of an abnormal exit. Such a change would affect the meaning of *every* expression, since function calls occur as part of expressions. Using "expression continuations" (see [15]), one can easily define the semantics of functions allowing these wild jumps; however, this would mean that when introducing GOTO statements (as part of the basic control structures) we would have to specify with far more care the meaning given to expressions. And, as simple as that, our hierarchy begins to collapse.

PL/I includes many other forms of transfer of control that would cause problems with the semantic hierarchy because of the global nature of their effect. Aside from the obvious need to disallow label variables and label parameters, we also must rule out the STOP statement and ON-units and the associated SIGNAL statement and condition prefixes. To maintain some compatibility with PL/I, PL/CS allows an ON ENDFILE(SYSIN) declaration that must be local to a procedure performing input. (PL/I provides no other way for testing end-of-file). Even this small control transfer causes problems for our semantic hierarchy. Without it, the introduction of input/output could be handled by a straightforward refinement of the store to include components representing the input and output files, as is done by Tennent [15] and Donahue [6]. But, because PL/CS input/output also requires the handling of an ON-unit, to give the semantics we are also required to redefine the context component of the state to include the meaning of the ON-unit as a distinguished component. This is one case where the semantics suggests that the language should be changed.

Another aspect of PL/CS, not discussed in this paper, that simplifies the

semantics is the lack of automatic conversions between types. In PL/I implicit conversions are provided whenever necessary, e.g. if X is a fixed (integer) variable, the assignment X = '123.45' is legal, because the character string '123.45' can be converted to an integer (123). Such coercions are disallowed in PL/CS; this simplifies the semantics by making the meaning of the basic value manipulation aspects of the language independent of any contextual information about the syntactic types of identifiers. Thus, the meaning of X = Y is always "update the store to give X the value of Y" and never requires any modification of the value Y produces. This means that we may legitimately add type-checking as a completely separate level in our semantic hierarchy without the need to undo previous semantic equations. (It is interesting to note that PL/I conversions are a common source of errors in student programs, especially where such a conversion has the unintended effect of altering the meaning of a procedure call by causing allocation of a temporary.)

One aspect of the semantics that suggests some possible changes in the language is the need to use locations in the definition. Were we able to remove them, the semantics would have a pleasing symmetry—we could use domains Context, Env, and State, each being functions with doman Id (i.e. of type Id → ...). Such domains would be particularly easy to describe ("names you may jump to, names you may call, names you may assign to"). However, given the use of STATIC and EXTERNAL variables in PL/CS it is difficult to prevent *simple* but reasonable syntactic restrictions so that the techniques of Donahue [7] could be applied to remove locations.

The use of locations also appears in parameter passing, which in PL/I and PL/CS is always "by reference." Here our semantics points up two difficulties:

(1) In describing the semantics of procedure calls, it is necessary to introduce a separate meaning function to handle expressions as arguments, e.g. the meaning of 1 as an argument is different from 1 in an assignment (in an argument list the meaning of 1 is "evaluate 1; get a new location; store 1 in it"). Particularly disconcerting is the difference in meaning between x and (x) as arguments; in the first case, the meaning is to produce the location associated with x, in the second a new location with the value of x. The fact that this is the only place in the language where parentheses are semantically meaningful is painfully obvious from the extra mechanism needed to get the equations right. Certainly the "surprise factor" of such a feature is very high.

(2) The semantics of procedure declarations is also encumbered by the inclusion of the READONLY attribute for parameters by which a parameter may be protected from change. In [3], READONLY parameters are handled by a "screening function" that transforms the list of locations passed as arguments to a list of locations and values (by replacing each location corresponding to a READONLY parameter by the current value at the location). Thus assignment to READONLY parameters becomes erroneous in the semantics because such parameters are not bound to locations. This semantics points out the dangers of READONLY; in particular, if the user says nothing, the default is to protect nothing. This means that the user of a procedure should always assume (to be safe) that any variable passed to a procedure will be modified. A clear syntactic separation between "variable" and "value" arguments, as suggested by Hoare [10], would clarify what

can be safely assumed to be protected and would avoid the need for "screening functions" in the semantics of procedure declarations.

Even with the complexity described above, the formal semantic definition of the actual PL/CS language is useful, compact (summarized in 6 pages of the report), and easily understood in complete detail by a bright undergraduate. Moreover, the most difficult parts of the definition correspond to the most subtle parts of the language as judged by other criteria, such as implementation strategies and ease of teaching.

REFERENCES

1. BEKIĆ, H., BJORNER, D., HENHAPL, W., JONES, C.B., AND LUCAS, P. A formal definition of a PL/I subset. IBM Tech. Rep. 25.139, IBM Lab., Vienna, 1974.
2. CONSTABLE, R., AND O'DONNELL, M. *A Programming Logic with an Introduction to the PL/CV Verifier.* Winthrop, Cambridge, Mass., 1978.
3. CONSTABLE, R.L., AND DONAHUE, J.E. An elementary formal semantics for the programming language PL/CS. Tech. Rep. TR76-271, Dept. Comptr. Sci., Cornell U., Ithaca, N.Y., 1976.
4. CONWAY, R.W. *A Primer on Disciplined Programming Using PL/I, PL/CS, and PL/CT.* Winthrop, Cambridge, Mass., 1978.
5. CONWAY, R.W., AND WILCOX, T.R. Design and implementation of a diagnostic compiler for PL/I. *Comm. ACM 16,* 3 (March 1973).
6. DONAHUE, J.E. *Complementary Definitions of Programming Language Semantics, Lecture Notes in Computer Science, Vol. 42.* Springer-Verlag, New York, 1976.
7. DONAHUE, J.E. Locations considered unnecessary. *Acta Informat. 8* (1977), 221–242.
8. European Computer Manufacturers' Association and American National Standards Institute. PL/I Basis/1-12. ANSI, New York, 1975.
9. HOARE, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12,* 10 (Oct. 1969), 576–581.
10. HOARE, C.A.R. Procedures and parameters, an axiomatic approach. *Symp. on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, No. 188,* E. Engeler, Ed., Springer-Verlag, New York, 1971, pp. 102–116.
11. HOARE, C.A.R., AND WIRTH, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica* 2(1973), 335–355.
12. O'DONNELL, M.J. Reduction strategies in subtree replacement systems. Ph.D. Th., Cornell U., Ithaca, N.Y., 1976. Also in *Lecture Notes in Computer Science,* Springer-Verlag, New York, 1978.
13. SCOTT, D., AND STRACHEY, C. Toward a mathematical semantics for computer languages. *Proc. Symp. on Computers and Automata,* Polytech Inst. of Brooklyn, N. Y., 1971, pp. 19–46.
14. STRACHEY, C., AND WADSWORTH, C. Continuations, a mathematical semantics for handling full jumps. Tech. Mono. PRG-11, Oxford U. Computing Lab., Oxford, England, 1974.
15. TENNENT, R.D. The denotational semantics of programming languages. *Comm. ACM 19,* 8 (Aug. 1976), 437–453.
16. WALK, K., ET AL. Abstract syntax and interpretation of PL/I. IBM Tech. Rep. 25.098, IBM Lab., Vienna, 1969.

Received December 1976; revised September 1977 and June 1978