LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-246

FROM DENOTATONAL TO OPERATIONAL AND AXIOMATIC SEMANTICS FOR ALGOL-LIKE LANGUAGES: AN OVERVIEW

B.A. Trakhtenbrot

J.Y. Halpern

A.R. Meyer

October 1983

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

From Denotational to Operational and Axiomatic Semantics for ALGOL-like Languages: An Overview*†

B. A. Trakhtenbrot, *Dept. of Computer Science, Tel Aviv Univ.*
Joseph Y. Halpern, *IBM Research, San Jose*
Albert R. Meyer, *Laboratory for Computer Science, MIT*

Abstract. The advantages of denotational over operational semantics are argued. A denotational semantics is provided for an ALGOL-like language with finite-mode procedures, blocks with local storage, and sharing (aliasing). Procedure declarations are completely explained in the usual framework of complete partial orders, but cpo's are inadequate for the semantics of blocks, and a new class of store models is developed. Partial correctness theory over store models is developed for commands which may contain calls to global procedures, but do not contain function procedures returning storable values.

CR Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory–*syntax, semantics*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs–*assertions, logics of programs, pre- and post-conditions*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages–*operational semantics, denotational semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs.

General Terms: Languages, Verification, Theory.

Additional Key Words and Phrases: lambda-calculus, partial correctness, relatively complete, copy-rule semantics, fixed-point semantics.

---

*This paper is to appear in *Logic of Programs, Proceedings*, Clarke and Kozen, eds., Lecture Notes in Computer Science, Springer, 1983. A shorter version was presented by the third author under the title "Understanding ALGOL: The View of a Recent Convert to Denotational Semantics," in *1983 IFIPS Proceedings*, R. E. Mason, ed., North Holland.

†The research reported here was supported in part by NSF Grant MCS80-10707, a grant to the MIT Lab. for Computer Science from the IBM Corporation, and a grant from the National Science and Engineering Research Council of Canada. A portion of this research was performed while the second author was a visiting scientist jointly at the MIT Lab. for Computer Science and the Aiken Computation Lab., Harvard Univ.

1. Introduction. Despite wide-spread though by no means unanimous acceptance of denotational semantics, the numerous papers developing logical systems for proving assertions about programs in the style of [Hoare, 1969] consistently deviate from a purely denotational approach (cf. [Apt, 1981]). In every case, this ideal approach has been deemed inconvenient and has been compromised in favor of operational formulations, notably in explaining inference rules for calls of recursive procedures and for declarations of local variables.. We now realize that at least one reason for past deviations from a denotational approach is that the denotational semantics of local storage for blocks has never, until now, adequately been worked out.

This paper gives an overview of our development of a denotational basis for proof systems concerning ALGOL-like programs. We believe our work demonstrates that the denotational ideal is achievable without excessive complication and with significant benefits, notably, a satisfactory treatment of free (global) variables in commands and formulas, and a semantically sound proof system in which the usual substitution rules naturally hold.

In particular, we have developed a denotational semantics and Hoare-like axiom system for *partial correctness assertions* about an ALGOL-like language we call PROG. PROG is a structured language exhibiting a number of nontrivial features including blocks with local variables, nested declarations of recursive procedures, procedure parameters, call-by-name, -value, and -reference parameters, and sharing (aliasing) among identifiers.

2. ALGOL-like Languages. Our focus in this paper is on the family of ALGOL-like languages, since these languages are rich in expressive power, yet are also sufficiently structured to yield a rich algebra and proof theory. Following [Reynolds, 1981], we formulate several of the principles which characterize this class of languages:

(1) There is a consistent distinction between *commands* (or *programs*) which alter the store but do not return values, and *expressions* which return values but have no side-effects on the store.

(2) The only explicit calling mechanism is *by-name*. (Other mechanisms such as *by-value* or *by-reference* are available by simulation (syntactic sugaring).)

(3) The language is fully typed. Higher-order procedures of all *finite* types (in ALGOL jargon, *modes*) are allowed. There is a clear distinction between *locations* and *storable values*.

(4) The stack discipline is an explicit aspect of the semantics. Note that this discipline should be understood as a language design principle encouraging modularity in program construction rather than as an implementation technique for efficient storage management. It is better called the *local storage discipline* to avoid misunderstanding, and we do so henceforth.

As it happens, ALGOL-60 is not ALGOL-like in our terminology, nor are numerous features of ALGOL-68; Pascal and ALGOL-W come closest to being ALGOL-like.

Some comprehensive structural restrictions on ALGOL-like languages are implied by these principles. In particular, a consequence of local storage discipline is that neither locations nor procedures can be freely storable, since otherwise locations allocated inside a block might be accessible after exit from the block via the stored objects.

Types in PROG. We have tried to arrange a syntax for PROG reflecting familiar programming languages, but one place where our denotational perspective persuades us to violate common practice is in maintaining an *explicit* type distinction between locations and storable values (also called "left" and "right" values of expressions), because we feel that using the same syntax both for expressions denoting locations and expressions denoting values is an unnecessary source of confusion. So we consistently distinguish locations from their contents, using the token **cont** for explicit dereferencing. These two *basic* types – storable values and locations – will be called **int** and **loc**, respectively. Thus, $\mathbf{cont}(x^{\mathbf{loc}})$ denotes the element of type **int** which is the contents of x, and assignment commands take the form $LocE := IntE$ where $LocE$ is a location-valued expression and $IntE$ is an **int**-valued expression. Equality tests in PROG can only be between elements of basic type.

The other primitive types are **prog**, **intexp**, and **locexp**. The domain **prog** is the domain of program meanings, namely, mappings from stores to sets of stores. (PROG has a nondeterministic choice construct. For the partial correctness theory we develop, as opposed to *termination* theory, nondeterminism creates no problems in the semantics or proof theory.) The last two "expression" types provide the semantical domains for expressions whose evaluation yields basic values, viz., the elements of **intexp** (**locexp**) are functions from stores to **int** (**loc**). (Elements of type **intexp** or **intexp** are called "thunks" in ALGOL jargon).

In order to avoid some complications in the partial correctness theory of PROG, we impose two extra un-ALGOL-like restrictions. (None of the semantical properties discussed in §3-5 below depend on these restrictions.)

(5) Declarations of functional procedures which return *basic* values are disallowed.

(6) Expressions of type **intexp** and **locexp** are forbidden as parameters in calls.

A consequence of (6) is that the value of an actual parameter is independent of the store. However, note that while the meaning of a procedure term is a mapping involving stores, *which* mapping the term denotes is independent of the store. Thus we *do* allow arbitrary terms of procedure type in calls, e.g., the procedure term $x := \mathbf{cont}(x) + 1$ appears as an actual parameter in a call of the procedure *iterate*2 of Example 1 below.

Hence, we define the allowable procedure types by the grammar

Procedure types:

$Proctype ::= \mathbf{prog} \mid Paramtype \rightarrow Proctype$

Parameter types:

Paramtype ::= *Proctype* | **loc** | **int**

Blocks and Binding in PROG. Procedure identifiers are bound in PROG via procedure declarations occurring at the head of a *procedure block*, e.g.,

$$\textbf{proc}\ p(x) \Leftarrow \mathit{DeclBody}\ \textbf{do}\ \mathit{BlockBody}\ \textbf{end}.$$

Identifiers of basic type are bound by either let-declarations or new-declarations at the head of *basic blocks* of the form

$$\textbf{let}\ x\ \textbf{be}\ \mathit{BasE}\ \textbf{in}\ \mathit{Cmd}\ \textbf{tel},$$
$$\textbf{new}\ y^{\textbf{loc}}\ \textbf{in}\ \mathit{Cmd}\ \textbf{wen}$$

where x is a variable of basic type and $BasE$ is an expression of the same type.

Call-by-value is available implicitly through let-declarations. A call-by-value of the form $p(BasE)$ can be simulated by the basic block

$$\textbf{let}\ n\ \textbf{be}\ \mathit{BasE}\ \textbf{in}\ p(n)\ \textbf{tel}.$$

The let-declaration causes the evaluation of the expression $BasE$ in the declaration-time store and causes identifier n to denote the result of the evaluation. Call-by-reference is likewise available by simulation since it is merely call-by-value using location-valued expressions.

There is a fundamental difference between basic and procedure blocks. Namely, *which* basic values are bound to identifiers by basic declarations depends on the store "at declaration time", whereas which procedures are bound to identifiers by procedure declarations is store-independent.

Moreover, basic declarations are not recursive, while procedure declarations are. A reflection of this fact is that bound and free occurrences of identifiers are defined differently for basic and procedure blocks. In particular, in procedure declarations, the declared procedure identifiers bind occurrences of these identifiers in the declaration bodies *and* in the block bodies, while in basic declarations, the occurrences of the declared basic identifier are bound only in the block body (which must be a command). (This is made explicit by their contrasting translations to λ-calculus in the next section: procedure blocks are translated using the letrec binding mechanism, whereas basic blocks are translated using λ-abstraction and appropriate constants.)

Example 1 illustrates recursion in procedure declarations, procedures of arbitrarily high finite type, simultaneous procedure declarations (nested declarations are also allowed), and complex expressions as actual parameters in calls, including a procedure abstraction of the form lam *p. body* mal. The example assumes that the underlying domain **int** is actually integer arithmetic and uses primitive recursion at higher types to define a program which sets the variable x to $A(\mathrm{m})$, where A is a function which grows more rapidly than any function on the integers definable by ordinary (first-order) primitive recursion.

It is convenient to apply higher-order procedure identifiers to varying numbers of actual parameters in calls. In Example 1, procedure *iterate*1 is declared with four formal parameters of respective types int, β, α, prog, but is understood to be of "Curried" type int $\to \beta \to \alpha \to$ prog $\to$ prog rather than (int $\times \beta \times \alpha \times$ prog) $\to$ prog. It is called first with three actual parameters corresponding to a call of type prog $\to$ prog, and is called a second time with one actual parameter in a call of type $\beta \to \alpha \to$ prog $\to$ prog.

Example 1. Let prog denote the objects of type *program*, and let $\alpha =$ prog $\to$ prog, $\beta = \alpha \to \alpha$, $\gamma = \beta \to \beta$ denote successively higher procedure types.

proc $repeat(q^{\alpha}, p^{\text{prog}}) \Leftarrow q(q(p))$,
  $iterate1(n^{\text{int}}, r^{\beta}, q^{\alpha}, p^{\text{prog}}) \Leftarrow$ if $n = 0$ then p else $r(iterate1(n-1, r, q), p)$ fi,
  $iterate2(n^{\text{int}}, s^{\gamma}, r^{\beta}, q^{\alpha}, p^{\text{prog}}) \Leftarrow$ if $n = 0$ then p else $s(iterate2(n-1, s, r), q, p)$ fi,
  $A(x^{\text{loc}}) \Leftarrow$ let n be cont(x) in
    $iterate2(n, iterate1(n), repeat,$ lam $p^{\text{prog}}.(p; p)$ mal$, (x :=$ cont$(x) + 1))$
  tel
do $x :=$ m; $A(x)$ end

Sharing in PROG. Sharing of locations between identifiers arises naturally from procedure calls. Explicit sharing can also be imposed by a basic block like

$$\text{let } x^{\text{loc}} \text{ be } y^{\text{loc}} \text{ in } Cmd \text{ tel}.$$

Example 2 is contrived to illustrate the sharing features of PROG. It includes a declaration of a procedure that swaps the contents of two locations unless they are equal. It also illustrates the new declaration for allocating storage local to a block. The simple idea which underlies local variables in ALGOL-like languages is the *local storage discipline*: execution of a block new z in *body* wen causes allocation of a "new" storage location denoted by the identifier z which is used in the body of the block and then de-allocated upon exit from the block.

Example 2.

proc $swap(x_1^{\text{loc}}, x_2^{\text{loc}}) \Leftarrow$ if $x_1 = x_2$ then $x_1 :=$ error
  else new z in
    $z :=$ cont(x_1);
    $x_1 :=$ cont(x_2);
    $x_2 :=$ cont(z) wen fi
do if f(a) = f(cont(y)) then y else z fi := null;
  $swap(y, z)$ end.

Note that we are allowing explicit equality testing between locations ("$x_1 = x_2$") in addition to the usual test of equality between storable values ("f(a) = f(cont(y))"). Expressions which

evaluate to locations are allowed, as in the "conditional variable" expression to the left of the final assignment command in the example.

In addition to declarations of purely functional procedures which return basic values, other significant language features *compatible* with ALGOL-like principles but *omitted* from PROG include exit control, arrays and user-defined data-types, own-variables, polymorphism, implicit coercion (overloading) and concurrency. These will have to be the subject of future studies.

2. Semantics via Syntax-Preserving Translation to λ-Calculus. The main issue in reasoning about imperative programming languages is that the computer memory or *store* altered by program execution is never mentioned explicitly in programs. A denotational explanation of programs requires that the role of the store be made explicit. This is usually done by, in effect, translating programs into ordinary mathematical expressions which mention stores. Our approach is to formalize the assignment of semantics to programs in two steps:

(1) a purely syntactic *translation* from PROG to a fully-typed λ-calculus enriched with a letrec-construct corresponding to procedure declarations, and

(2) assignment of semantics to the λ-calculus in the standard way. Programs simply inherit their semantics directly from the λ-terms into which they translate.

This two step process has also been utilized by [Damm and Fehr, 1980; Damm, 1982]. Our approach refines theirs in that the λ-calculus into which programs are translated is chosen so that its types are the *same* as those of the programming language and its constants correspond to program constructors. (It follows that stores do *not* appear as one of the types of the target language, although they do appear within the definitions of the domains of meaning of various program phrases; in particular there is no λ-abstraction over stores.) In this way, the abstract syntax, viz., parse tree, of the translation of a program is actually *identical* to that of the program; the translation serves mainly to make the variable binding conventions of PROG explicit.

In addition to the binding of free variables, the translation Tr mapping PROG to λ-calculus makes explicit an implicit coercion (the only one) which takes place in PROG. Namely, a store-independent term T of basic type can always be treated as an expression (Mkexp T) denoting the constant function of the store identically equal to the value of T. More generally, any function f taking basic values as arguments can also be coerced straightforwardly into a mapping $Mkexp(f)$ taking as arguments functions from stores to basic values. (There is a different constant Mkexp for every first-order type, but we usually omit mention of the types of Mkexp and similar constants such as Ifexp below.) The term (Mkexp f) corresponding to $Mkexp(f)$ is abbreviated as $\bar{\mathbf{f}}$.

Procedure blocks are translated using letrec, so for example,

$$Tr(\textbf{proc}\ p(x) \Leftarrow DeclBody\ \textbf{do}\ BlockBody\ \textbf{end}) =_{def} \textbf{letrec}\ p = \lambda x.Tr(DeclBody)\ \textbf{in}\ Tr(BlockBody).$$

For a basic block with a let-declaration of type **int**,

$$Tr(\textbf{let}\ x^{\textbf{int}}\ \textbf{be}\ IntE\ \textbf{in}\ Cmd\ \textbf{tel}) =_{def} \textbf{Dint}\,(\lambda x.Tr(Cmd))\,(Tr(IntE))$$

where Dint is a constant of type (int → prog) → intexp → prog. For any element d_1 of type (int → prog), d_2 of type intexp, and store s, the interpretation [[Dint]] satisfies

$$[\![\text{Dint}]\!]\, d_1\, d_2\, s = \begin{cases} (d_1\,(d_2(s)))(s) & \text{if } d_2(s) \neq \perp_{\text{int}}, \\ \emptyset & \text{otherwise.} \end{cases}$$

Note that the binding effect of the block on x^{int} is reflected in the binding effect of λx on $Tr(Cmd)$, namely, the declaration binds x in Cmd, but does not bind x in $IntE$.

The principal consequence of this syntax-preserving translation is that the basic properties of procedure declarations in ALGOL-like languages – such as renaming rules associated with "static scope" for declared identifiers, declaration denesting rules, and expansions of recursive declarations – can be recognized as direct consequences of corresponding properties of the purely functional λ-calculus.

For example, let E be a system of procedure declarations, and let Cmd_1, Cmd_2 be any commands. The following equivalence holds in *all* models for λ-calculus, viz., all Cartesian-closed models [Barendregt, 1980; Meyer, 1982].

Declaration Distributivity:

$$\textbf{proc}\, E\, \textbf{do}\, Cmd_1; Cmd_2\, \textbf{end} \quad \equiv \quad \textbf{proc}\, E\, \textbf{do}\, Cmd_1\, \textbf{end}; \textbf{proc}\, E\, \textbf{do}\, Cmd_2\, \textbf{end}\,.$$

In particular, this equivalence follows solely from the binding properties of procedure declarations independently of whether declarations are recursive, and also independently of the meaning of constants like the sequencing operation ; . (On the other hand, its validity depends crucially on the fact that procedure declarations, in contrast to let-declarations, have the same binding effect no matter what the declaration-time store.)

The translation for blocks with new declarations, is

$$Tr(\textbf{new}\ x\ \textbf{in}\ Cmd\ \textbf{wen}) =_{def} \textbf{New}\,(\lambda x.Tr(Cmd))$$

where New is a special constant of type (loc → prog) → prog. The effect of the translation will be that Cmd runs using a "new" location in place of x. The contents of this new location are initialized to some standard value denoted by the constant $\mathbf{a}_0$ at the beginning of the computation of Cmd and restored to its original value at the end.

In defining the semantics of new declarations, we imagine an ability to generate "new" locations via a *Select* operation mapping any a procedure of type loc → prog into a new location. We can then define the meaning of $[\![\text{New}]\!]d^{\text{loc}\to\text{prog}}$ as follows:

$$[\![Tr(\text{let } x^{\text{int}} \text{ be cont}(y) \text{ in } y := \mathbf{a}_0; p(y); y := x \text{ tel})]\!]e\,,$$

where e is an environment such that $e(y) = Select(d)$ and $e(p) = d$.

We illustrate the full translation process with

Example 3. Translation of Example 2.

$$
\begin{array}{l}
\text{letrec } swap = \\
\quad \lambda x_1^{\text{loc}} x_2^{\text{loc}}.\, \text{Ifexp } \bar{x}_1 \bar{x}_2 \\
\quad\quad (\text{Assign } \bar{x}_1\ \overline{\text{error}}) \\
\quad\quad (\text{New } \lambda z.\text{Seq} \\
\quad\quad\quad (\text{Assign } \bar{z}(\text{Cont } \bar{x}_1))(\ \text{Seq} \\
\quad\quad\quad (\text{Assign } \bar{x}_1(\text{Cont } \bar{x}_2)) \\
\quad\quad\quad (\text{Assign } \bar{x}_2(\text{Cont}\bar{z}))\ )) \\
\text{in } (\text{Seq } (\text{Assign} \\
\quad\quad\quad (\text{Ifexp } (\text{Mkexp } (\text{f a}))\, (\bar{\text{f}}(\text{Cont } \bar{y}))\, \bar{y}\, \bar{z}) \\
\quad\quad\quad \overline{\text{null}}) \\
\quad\quad (swap\ y\ z))
\end{array}
$$

Constants like **Assign**, **Ifexp**, **Seq**, ..., in λ-terms correspond to the tokens :=, **if...then... else...fi**, ;, ..., of PROG and must be appropriately interpreted.

Note that we have kept our promise that the translation preserves the true syntax of commands: Cmd and $Tr(Cmd)$ have the same abstract syntax.

3. Levels of Understanding. A denotational approach has led us to identify a half dozen levels of abstraction at which aspects of ALGOL-like languages can be understood. The highest level abstracts away all properties except for variable binding; these properties hold in all models of λ-calculus. For example, the fundamental rule:

Procedure-Context Replacement:

$$\frac{Cmd_1 \equiv Cmd_2}{\textbf{proc } E \textbf{ do } Cmd_1 \textbf{ end} \equiv \textbf{proc } E \textbf{ do } Cmd_2 \textbf{ end}}$$

is obvious from a denotational viewpoint, and like the distributing rule, actually holds in all λ-calculus models.

The next level reveals that procedure declarations are recursive; the corresponding proof theory is simply the equational theory of the fixed-point combinator and similar expansion rules for **letrec.**

Properties connecting different fixed-points require the further hypothesis that fixed-points in distinct domains be chosen harmoniously. This is usually captured by imposing an order structure on domains, keeping to order-respecting (monotone) functions on the domains, and choosing *least* fixed points as solutions to recursive equations. At this "monotone" level, we can justify:

Declaration Denesting:

$$\textbf{proc } (p(x) \Leftarrow \textbf{proc } E \textbf{ do } body \textbf{ end}), E' \textbf{ do } Cmd \textbf{ end} \quad \equiv \quad \textbf{proc } (p(x) \Leftarrow body), E, E' \textbf{ do } Cmd \textbf{ end}$$

providing none of the identifiers declared in E occurs in E' or Cmd, p is not declared in E or E', and x is not free in E.

At the fourth level of abstraction we entirely account for the procedure mechanism of ALGOL-like languages. Here, the familiar *continuous* models of λ-calculus based on complete partial orders (cpo's) [Scott, 1982; Milne and Strachey, 1976; Stoy, 1977] provide an adequate semantical basis. We refer to properties which are valid for all continuous models as *continuity* properties. The most fundamental continuity property is that every command can be understood as a limit of finite procedure-declaration-free commands.

The original ALGOL 60 report [Naur, et. al., 1963] gave a copy-rule semantics for the language. The equivalence of fixed-point and copy-rule semantics verifies that our choice of denotational "fixed-point" semantics is consistent with the prior operational understanding based on the copy-rule. We give a simplified proof of:

Theorem. In every continuous model, fixed-point and copy-rule semantics assign the same semantics to commands (which may contain global procedures) in PROG.

Still further refined levels are needed to explain the store-dependent aspects of programs, i.e., their side-effects. In particular, continuous models are not adequate to explain local storage allocation, and we must introduce a new class of *store models* discussed in §5.

4. Fixed-point versus Copy-rule Semantics. Copy-rule semantics for ALGOL-like languages have historical precedence over denotational semantics, and are widely regarded as more intuitive for computationally oriented students (cf. [Blikle, 1983]). It seems obvious to us, however, that the tricky and otherwise arbitrary-seeming renaming rules which are part of the definition of the copy-rule, and which are crucial in determining the properties of declarations, spring from a mathematical intuition with an even earlier historical claim. But arguments from intuition are always questionable; putting such arguments aside, we can identify the place where the denotational approach is clearer and more general than an operational approach to be the handling of "global" procedures, i.e., free procedure identifiers.

Reasoning about commands with calls to global procedures is essential in theory and in practice. The need for reasoning about commands containing globals arises, for example, when global procedures denote library procedures. Given assertions about the behavior of the library procedures, one should be able to reason about the behavior of commands incorporating these procedures, without necessarily being given the declarations of the procedures. After all, the code of these library procedures is typically unavailable or written in machine language, and in any case is not what one wants to see. Unfortunately, nearly all the operationally based proof systems apply only to programs without globals, viz., programs in which all procedures are declared.

One can give an operational-style explanation of the range of globals, namely, that global procedures range over textual objects such as "closures". This explanation is clearly unsatisfactory when library procedures are written in another language. Another difficulty with this explanation

is that any enrichment of the language enlarges the range of the global procedures, so that all the axioms and rules involving globals must be reexamined for soundness. In contrast, a denotational approach in which *environments* map free identifiers to meanings over a domain of functional objects, smoothly handles commands containing global procedures.

The desire to reason by induction on the structure of programs – which motivates the design of structured programming languages in the first place – also naturally requires reasoning about global procedures, since the procedure identifiers declared in a block inevitably have free occurrences in the body of the block. *Fixed-point induction* is an important instance of a structural inference rule in which free procedure identifiers are essential. The following special case illustrates the essence of the rule when applied to partial correctness assertions of the form $P\{\,Cmd\,\}Q$ where P and Q are first-order formulas. (The precise semantics of such assertions is given in §6.)

Fixed-Point Induction: Let p be an identifier and $ProcE$ an expression, both of the same procedure type, such that none of the free *first-order* variables in $ProcE$ are free in P or Q.

$$\frac{P\{\,[\text{diverge}/p]Cmd\,\}Q\,,\quad P\{\,Cmd\,\}Q \vdash P\{\,[ProcE/p]Cmd\,\}Q}{P\{\,\text{proc } p \Leftarrow ProcE \text{ do } Cmd \text{ end}\,\}Q}$$

where $[Expr/p]$ denotes syntactic substitution (with renaming to avoid capture of free variables) of $Expr$ for *free* occurrences of the procedure identifier p.

The soundness of fixed-point induction is an easily proved continuity property.

[Clarke, 1979], extending [Gorelick, 1975], introduced another proof rule for partial correctness of higher-order procedure calls closely resembling the fixed-point induction rule but justified using the copy-rule. Subsequent work [Langmaack and Olderog, 1980; Olderog, 1981, 1983a, 1983b; Apt, 1981] has followed this approach. Our axiom system in §6 includes a version of Clarke's rule called *copy-rule induction.*

[Langmaack and Olderog, 1980] have defended the use of copy-rule induction:

"In soundness and completeness proofs a semantics definition should be employed which yields shortest proofs. The question of equivalence of partly operational and purely denotational semantics should be considered separately."

We remain uncomfortable with this view. It is a useful technical insight that inductive proofs about calls in ALGOL-like languages can be based on copy-rule semantics. Yet this fact seems too coincidental to serve as a justification for postponing denotational arguments. For example, it seems fortuitous that the usual axioms for partial correctness happen not to require the procedure-context replacement rule above. The rule is obvious denotationally, but we know of no justification for it using copy-rule semantics which is any simpler than the proof that copy-rule and denotational semantics are equivalent. We expect that outside the special case of partial correctness proofs, and

perhaps even there, it will be disadvantageous to develop proof systems using copy-rule semantics alone.

Of course, whenever there is a nontrivial equivalence between two definitions, there are bound to be facts which are obvious starting from one definition and not from the other, and it should be expected that some important facts about program behavior, possibly such as Clarke's rule, would be seen more easily in terms of the copy-rule. If it were merely the case that the semantical soundness proof was more easily carried out using one of two equivalent definitions instead of the other, we would not be concerned. However, in contrast to the fixed-point induction rule, copy-rule induction is *not sound* in the usual logical sense, although only valid assertions are provable using it.

Namely, copy-rule induction, like fixed-point induction, is formulated in natural deduction style where the *provability* of one assertion from another serves as the antecedent for application of the rule. Because of the reference to the proof system in the antecedent, the meaning of such rules technically changes if we alter the proof system in any way, for example by adding further sound inference rules. This reference to the proof system will be harmless as long as soundness of the rule follows from soundness of the rest of the proof system – as opposed to facts about the detailed structure of proofs. This is what is meant by semantical soundness of a natural deduction style rule (cf. §6). Fixed-point induction is semantically sound in this sense, but copy-rule induction is not because it depends crucially on structural properties of proofs. In fact, we can show that a price to be paid for using copy-rule induction is that adding very simple, obviously sound rules makes the proof systems inconsistent! ([Olderog, 1981] claims to avoid this problem, but he does so by adopting a definition of validity which is not referentially transparent, so that substituting one equivalent command for another cannot be added as a rule in his system without yielding an inconsistency.)

We have not yet worked out as strong a completeness theorem using the fixed-point induction rule or other sound rule in place of Clarke's rule, although we have an idea how to do so. Meanwhile, as a temporary expediency we have included copy-rule induction in our own proof system. Insofar as the remarks of Olderog and Langmaack and others supporting operational semantics for proof systems are intended as a defense of copy-rule induction despite its unsoundness, we disagree with their view. We see no theoretical obstacle to discovering a denotationally sound alternative to copy-rule induction, and we regard developing such a rule as an interesting research problem.

5. Store Models. Although the local storage discipline seems intuitively simple, it raises a number of both practical and theoretical problems.

A well-known practical consequence of the mixture of blocks and recursive procedures is that *run-time* storage allocation is necessary: a block may appear within a recursive procedure which is itself called within the body of the block, so the number of different locations which must be *simultaneously* allocated for nested activations of the block during execution may grow unboundedly,

ruling out static ("compile time") allocation.

On the other hand, program schemes with neither free procedure identifiers nor recursive procedure declarations require only a bounded number of locations to be allocated – independent of the interpretation of the scheme – and static allocation is not difficult. The fact that static allocation is possible in this case can be formulated as the theorem that every such program scheme is equivalent to an effectively constructible scheme *without blocks* but with various constants denoting fixed locations. Since copy-rule semantics reduces the meaning of general programs to procedure-declaration-free programs, it follows that the semantics of block storage allocation *in the absence of free procedure identifiers* is easy to explain with the copy-rule. Free procedure identifiers raise problems which have not been dealt with using operational semantics.

From a theoretical viewpoint, the problem is to explain what is meant by a "new" location. Operationally, the "old" locations for a command correspond to the values of the free location variables in it. This is sometimes modeled denotationally by enriching the notion of stores to include with each location an indication of whether the location is "active". Execution of a new block on a store involves selecting the first inactive location as the one to be allocated. The problem with this approach is that the locations designated as inactive by the store may already be accessible to the body of the block, and so the first inactive location may *not* in fact be "new". For example, the block

$$\textbf{new } x \textbf{ in if } x = y \textbf{ then diverge else skip fi wen}$$

ought intuitively to be equivalent to skip since the "new" x should never equal the "old" y. But if this block is executed on a store in which (the location denoted by) y happens to be designated as the first inactive location, then the block will diverge. Validity of the expected properties of blocks thus hinges on hypotheses about how the locations designated as active by the store relate to the "old" locations which really are active, and we are in any case still left with the problem of explaining what a new location is.

The denotational meaning of a program is a mapping from stores to stores, and it is not hard to give a purely denotational characterization of what it means to say that such a mapping "reads" or "writes" a set of locations (cf. [Trakhtenbrot, 1979; de Bakker, 1980, Def. 5.9; Meyer and Mitchell, 1982]). The meaning of the body of a new block is in turn a mapping from locations to store mappings, and a denotational definition of what locations such a block body "knows about" can also be given with some care. In general, we define a notion of the set of locations which form the *support* of any procedure of finite type. The locations outside the support of a procedure are the"new" ones for it. The support of a command is thus the denotational concept corresponding to the syntactic notion of free location variables appearing in the command. This would appear to provide the desired denotational semantics of block storage allocation.

However, an amusing technical problem arises. Monotonicity is normally required of the functions defining the semantics of λ-terms in order to ensure that the fixed-points necessary to

explain recursive definitions exist in the domain of meanings. The operation of allocating and later de-allocating "new" storage turns out not to be monotonic, essentially because of the possibility of running out of new storage locations!

One apparent way out of this difficulty is to admit a new **overflowed** error-object which is maximal in the partial order on programs. This complicates the logic of programs since we now must explicitly reason about the possibility of running out of storage. For example, we might expect that

$$\textbf{new } x \textbf{ in } Cmd \textbf{ wen} \equiv Cmd$$

when x does not occur free in Cmd. But this equivalence is only valid providing Cmd denotes a program whose support does not contain *all* locations; if it does, then the left-hand side is equivalent to **overflowed**. Similarly, the equivalence

$$(Cmd; \textbf{diverge}) \equiv \textbf{diverge}$$

becomes questionable in the case that Cmd denotes **overflowed**. A more serious problem is that although an **overflowed** object circumvents nonmonotonicity, it does not restore continuity. Namely, a command whose support contains all locations is the limit of a sequence of approximating commands with smaller support; allocating new storage is therefore possible for the approximations but yields **overflowed** for the limit. (The discontinuity of new storage allocation is noted in [Milne and Strachey, 1976], with a reference to further discussion in Milne's thesis.)

In general, objects with "large" support force us to face the discontinuity of storage overflow. It would be reasonable to rule out such objects, especially in view of the fact that *definable* objects, viz., objects which are the denotations of phrases in PROG, can be proved to depend on only finitely many locations. Unfortunately, the domain of programs with finite support is not *complete* (closed under least upper bounds) because the lub of a sequence of programs each with finite support may have infinite support.

A simple way around this incompleteness would be to use, for each finite set, L, of locations, a separate domain consisting of those programs with support contained in L. This works as long as no procedure parameters appear, but the mixture of *higher order* recursive procedures and block structure turns out to be explosive. A procedure which takes a program parameter might be called upon recursively within a new block, and might be applied to programs constructed within the block, as in:

$$\begin{aligned} p(q^{\mathbf{prog}}, n^{\mathbf{int}}) \Leftarrow \textbf{if } n \neq 0 \textbf{ then} \\ \textbf{new } x \textbf{ in } p(r(q,x), n-1) \textbf{ wen} \\ \textbf{else } q \textbf{ fi}. \end{aligned}$$

The domain of such a procedure includes programs with unbounded finite support, and we are no longer able to confine ourselves to the cpo of programs with any particular finite support L.

Difficulties of this sort have led [Reynolds, 1981] and [Oles, 1983] to consider more sophisticated functor categories as domains of interpretation.

Store models overcome these difficulties: they are domains of mappings with *countably infinite* support. Such domains are not complete, but they are ω-complete – closed under *countable* lubs – and it is known that ω-completeness is sufficient to develop the semantics of recursive programs [Meseguer, 1978; Plotkin, 1982].

Allowing elements with countably infinite support is merely a mathematical contrivance to preserve closure under countable directed limits. The countable covering restriction works, despite some intuitively jarring consequences, the oddest of which is that we must hypothesize an *uncountable* number of locations! (But after all, we do not complain about an uncountable set of real numbers even when we compute only with rationals.)

We outline below the main properties of store models and the support notion.

Primitive Domains: Given a set Loc (of locations) and a set Int (of storable values) we define the domains

$$D_{\mathbf{loc}} =_{def} Loc \cup \{\perp_{\mathbf{loc}}\},\ D_{\mathbf{int}} =_{def} Int \cup \{\perp_{\mathbf{int}}\}$$

to be the flat cpo's.

For the other primitive domains, we select some subset, $Store$, of $Int^{Loc} =_{def}$ the set of *all total* functions from Loc to Int. $Store$ must be closed under finite updates and under permutations μ of Loc, i.e., if $s \in Store$, then $s \circ \mu \in Store$. Then

$$D_{\mathbf{intexp}} \subseteq (D_{\mathbf{int}})^{Store},\ D_{\mathbf{locexp}} \subseteq (D_{\mathbf{loc}})^{Store},\ D_{\mathbf{prog}} \subseteq (\mathcal{P}(Store))^{Store}.$$

Here $\mathcal{P}(Store)$ denotes the power-set of stores, so elements of $D_{\mathbf{prog}}$ correspond to nondeterministic mappings between stores.

Higher Domains: The domains $D_{\beta\to\gamma}$ consist of subsets of the ω-continuous functions from D_β to D_γ, and form a Cartesian-closed type-frame with least fixed-points.

Uniformity on Locations: For each type α and permutation μ of Loc, the domain D_α is closed under the permutation μ_α induced by μ, e.g., if $\alpha = \beta \to \gamma$, then

$$\mu_\alpha(f) =_{def} \mu_\gamma \circ f \circ \mu_\beta^{-1}.$$

We define a *covering* relation between subsets $L \subseteq Loc$ and elements $d \in D_\alpha$. Let L be a subset of Loc. Two stores s, t *agree on* L, written $s =_L t$, iff $\forall l \in L.\, s(l) = t(l)$. Similarly, two sets $S, T \in \mathcal{P}(Stores)$ agree on L if there is a bijection $f : S \to T$ such that $\forall s \in S.\, s =_L f(s)$. Covering has the properties that

(a) if $\pi \in D_{\mathbf{prog}}$, then L covers π iff $\forall s, t \in Store.(s =_L t \Rightarrow \pi(s) =_L \pi(t)) \wedge (t \in \pi(s) \Rightarrow s =_{Loc-L} t)$,

(b) L covers $l \in D_{\mathbf{loc}}$ iff $l \in L \cup \{\perp_{\mathbf{loc}}\}$,

(c) if L covers d^α, then $\forall \mu$ fixing $L.\, \mu_\alpha d = d$,

(d) if L_1 covers $d_1 \in D_{\beta\to\gamma}$ and L_2 covers $d_2 \in D_\beta$, then $L_1 \cup L_2$ covers $(d_1\, d_2)$.

Countable Covering Restriction: Every element $d \in D_\alpha$, has a *countable* set covering it.

Interpretability of the Constants: Let

$$Support(d) =_{def} \bigcap \{ L \subseteq Loc \mid L \text{ is countable and covers } d \}.$$

There must be elements in the model giving appropriate interpretations to each of the constants used in translating PROG, and these elements must have *empty* support.

We remark that another odd consequence of allowing countable rather than finite covers is that $Support(d)$ may not cover d if no finite set covers d.

Now it is easy to show that all the constants other than New are continuous and have empty support. The sole purpose of the covering restriction is to ensure that New is interpretable. In particular, a function $Select : \mathcal{P}(Loc) \to Loc$ will be called a *selection function* iff $Select(L) \notin L$ for all *countable* sets $L \subseteq Loc$. As long as Loc is uncountable, selection functions exist, and using *any* selection function as $Select$ in the definition of New indicated in §2 yields the same ω-continuous function with empty support. This is the desired denotation for **New**.

Some typical equivalences about new-declarations are given below. Support properties of store models are essential to guarantee their validity.

$$\textbf{new } x \textbf{ in } x := \mathrm{b} \textbf{ wen} \equiv \textbf{skip},$$

$$\textbf{new } x \textbf{ in } x := \mathrm{a}_0; Cmd \textbf{ wen} \equiv \textbf{new } x \textbf{ in } Cmd \textbf{ wen},$$

$$\textbf{new } x \textbf{ new } y \textbf{ in } Cmd \textbf{ wen wen} \equiv \textbf{new } y \textbf{ new } x \textbf{ in } Cmd \textbf{ wen wen},$$

$$\textbf{new } x \textbf{ in if } x = y \textbf{ then } Cmd_1 \textbf{ else } Cmd_2 \textbf{ fi wen} \equiv y := \mathbf{cont}(y); Cmd_2 .$$

More generally, we can define an operational semantics for interpreting basic blocks and prove:

Theorem. In every store model, fixed-point and operational semantics assign the same semantics to commands (which may contain global procedures) in PROG.

The natural classes of objects to allow in store models are those with finite support, because PROG-definable programs only read or write a finite number of locations. These objects form a Cartesian-closed type-frame with least fixed-points, but, as we noted, not a cpo. The construction outlined above embeds the desired domains of elements with finite support into (ω-)cpo's. Hence all the requisite properties of cpo's are inherited by the embedded domains, and therefore we are now justified in restricting ourselves to the objects with finite support. In our proof theory in the next section, we in fact restrict variables to range over elements with finite support.

(A closer analysis of the role of limits in program semantics reveals that not all directed sets, but only "algebraic" directed sets $\bot, f(\bot), f(f(\bot)), \ldots$, where $\bot$ is the least element in D_α and $f \in D_{\alpha \to \alpha}$, must have lub's [Guessarian, 1982]. These lub's *do* exist in the frame of elements with finite support. The unexpected peculiarities of countable supports would have been avoided had we developed our semantics using such algebraically closed partial orders instead of cpo's. However, we did not want to take the time to reformulate and prove for algebraically closed partial orders all the well documented properties which make cpo's suitable as semantical domains, so we have kept to the better known cpo framework.)

6. Partial Correctness Theory. Instead of the usual first-order language of storable values, we use a two-sorted first-order language with sorts **int** and **loc**. We also add special atomic formulas for reasoning about the support of global procedures, namely, for each identifier p of **loc** or procedure type and each variable x of type **loc**, there is an atomic formula $\mathbf{Support}(x, p)$ which means that x is in the support of p. This language has the same constructive properties as ordinary (one-sorted) first-order language, e.g., the formulas valid in all interpretations are nicely axiomatizable.

We require assertions about support because we are reasoning about commands with global procedure identifiers. Without global procedure identifiers, one can determine the support of a command by inspection – namely, the support is contained in the denotations of the free location variables. This is obviously not possible for command with global procedure identifiers unless we are told which locations are in the support of the globals.

Any $s \in \mathit{Store}$ provides an interpretation for the contents function denoted by the token **cont**. Thus, given a store model D, a store s, and an environment e to assign values to variables, first-order expressions can be interpreted as elements of $D_{\mathbf{int}}$ and $D_{\mathbf{loc}}$ in the usual way.

We let $P, Q, \ldots$ denote first-order formulas. The satisfaction relation between a first-order formula P and an interpretation D, e, s, written $D, e, s \models P$, is defined as usual.

In defining satisfaction for assertions it is sometimes convenient to ignore the values an environment assigns to basic variables.

Definition. Two environments e, e' are said to *match* iff $e(x) = e'(x)$ for all *procedure* variables x. An environment e *uses finite supports* iff $\mathit{Support}(e(x))$ is finite for each identifier x.

Definition. Let D be a store model. The satisfaction relation $\models$ is extended to partial correctness assertions as follows:

$$
\begin{aligned}
D, e, s \models P\{\mathit{Cmd}\}Q \quad &\text{iff} \quad (\text{if } D, e, s \models P \text{ then } D, e, t \models Q \text{ for all } t \in [\![\mathit{Cmd}]\!]_D es), \\
&\quad \text{where } [\![\mathit{Cmd}]\!]_D e \in D_{\mathbf{prog}} \text{ is the denotation of } \mathit{Cmd} \text{ in environment } e, \\
D, e \models P\{\mathit{Cmd}\}Q \quad &\text{iff} \quad D, e', s \models P\{\mathit{Cmd}\}Q \text{ for all } e' \text{ matching } e \text{ and all } s \in \mathit{Store}, \\
D \models P\{\mathit{Cmd}\}Q \quad &\text{iff} \quad D, e \models P\{\mathit{Cmd}\}Q \text{ for all environments } e \text{ using finite supports}, \\
\models P\{\mathit{Cmd}\}Q \quad &\text{iff} \quad D \models P\{\mathit{Cmd}\}Q \text{ for all } \textit{store} \text{ models } D.
\end{aligned}
$$

This notation is also used for formulas, e.g.,

$$D, e \models P \text{ iff } D, e', s \models P \text{ for all } e' \text{ matching } e \text{ and all } s \in \mathit{Store}.$$

It may be helpful to note that by this convention D, e satisfies a first-order formula iff D, e satisfies the first-order universal closure of the formula. That is,

$$D, e \models P \text{ iff } D, e \models \forall x^{\gamma}.P$$

where γ is **int** or **loc**.

Distinguishing locations and their contents becomes particularly beneficial when making assertions about programs. With two-sorted language, our axiom system is able to deal with sharing among program variables explicitly and without excessive complication.

An awkward feature of common programming logic systems which do not allow preconditions to distinguish locations from values is that renaming free variables does not preserve validity. For example, although

$$x_1 \neq x_2\{\, x_3 := x_2 \,\}x_1 \neq x_3$$

is valid in such systems, it fails to be valid if we substitute x_1 for x_3, which can be a source of confusion (cf. [Manna and Waldinger, 1981]). We therefore judge the soundness of arbitrary renamings, which follows routinely for our system, to be a valuable feature.

Definition. A *first-order substitution* is a type-respecting mapping, σ, from *basic* identifiers to first-order terms which *do not contain* occurrences of the token cont. Let $\sigma(P)$ and $\sigma(Cmd)$ denote the result of simultaneously substituting $\sigma(x)$ for all free occurrences of basic variables x (renaming bound variables as necessary) in the formula P or command Cmd.

Substitution Lemma: Let σ be a first-order substitution. For any first-order model D and environment e, if $D, e \models P\{\, Cmd \,\}Q$, then $D, e \models \sigma(P)\{\, \sigma(Cmd) \,\}\sigma(Q)$.

This justifies the following:

Rule of Substitution: Let σ be a first-order substitution.

$$\frac{P\{\, Cmd \,\}Q}{\sigma(P)\{\, \sigma(Cmd) \,\}\sigma(Q).}$$

Another standard source of confusion as a result of sharing occurs in axiomatizing simple assignments. Clearly, no matter what the initial conditions are, after setting $x^{\mathbf{loc}}$ to some constant $\mathbf{a}$, the contents of x will equal $\mathbf{a}$, namely,

$$\mathbf{true}\{\, x := \mathbf{a} \,\}\mathbf{cont}(x) = \mathbf{a}$$

is a valid partial correctness assertion. But the naive generalization of this example to the case in which x is replaced by some expression whose evaluation yields a location is not valid. For example, let $LocE_0$ be the location-valued conditional expression

$$\mathbf{if}\ \mathbf{cont}(x) = \mathbf{a}\ \mathbf{then}\ y\ \mathbf{else}\ x\ \mathbf{fi}.$$

Setting $LocE_0$, rather than x, to $\mathbf{a}$, makes the following assertion valid

$$\big(\mathbf{cont}(x) = \mathbf{a} \wedge \mathbf{cont}(y) \neq \mathbf{a}\big)\{\, LocE_0 := \mathbf{a} \,\}\mathbf{cont}(LocE_0) \neq \mathbf{a}.$$

The problem here is that the meaning of $LocE_0$ depends on the store; $LocE_0$ has different meanings before and after the assignment.

To deal with assignment, we construct for any formula P, a first-order formula $[LocE \leftarrow IntE]P$ such that for each D, e, s, if l, d are the interpretations of $LocE$, $IntE$ in D, e, s, then

$$D, e, s \models [LocE \leftarrow IntE]P \text{ iff } D, e, s' \models P$$

where $s' =_{Loc-\{l\}} s$ and $s'(l) = d$. Now the assertion that after executing the assignment some first-order formula Q holds, is equivalent to asserting that the formula

$$[LocE := IntE]Q =_{def} (LocE = \bot \vee IntE = \bot \vee [LocE \leftarrow IntE]Q)$$

holds in the initial store. (The $\bot$ clauses handle the case that the evaluation of either of the expressions diverges.) Difficulties with assignments are thus completely handled by the:

Assignment axiom:

$$([LocE := IntE]Q)\{\, LocE := IntE \,\}Q\,.$$

Our axiom system AX_D is defined as usual *relative* to an underlying model D. However, it is worth noting that we do not need support predicates in the axioms about D. That is, we only include in AX_D the set $Th(D)$ of all first-order formulas *without* support predicates which are valid in D.

Axioms and rules for the programming constructs diverge, sequencing, conditionals, choice, as well as "logical" rules for consequence, conjunction and quantification, along with substitution and assignment above, are as usual. Also familiar is the:

Rule of let-Declarations: If x is a variable of basic type, $BasE$ is an expression of the same type, and y a variable of the same type which is not free in P, Q, $BasE$ or Cmd, then

$$\frac{(P \wedge (y = BasE) \wedge (y \neq \bot))\{\, [y/x]Cmd \,\}Q}{P\{\, \textbf{let } x \textbf{ be } BasE \textbf{ in } Cmd \textbf{ tel} \,\}Q\,.}$$

A command Cmd is *distributed* iff the block body of every procedure block which occurs in Cmd consists solely of a procedure call. Using rules like declaration distributivity (§2), one can easily transform any command Cmd into another distributed command $DIST(Cmd)$ which is equivalent to Cmd in all λ-calculus models. To ensure applicability of the preceding rules to procedure blocks, we include the:

Distributing Rule: If $Cmd = DIST(Cmd')$, then

$$\frac{P\{\, Cmd \,\}Q}{P\{\, Cmd' \,\}Q\,.}$$

We now enumerate the rules which rely on properties of support.

The usual rules of predicate calculus are applicable to first-order formulas with support predicates, treating $\mathbf{Support}(x,p)$ as a monadic first-order predicate in x for each variable p. We also use the axiom $\mathbf{Support}(x,y) \equiv (x = y \wedge y \neq \bot)$ when y is of type **loc**.

Rule of new declaration: Let $y^{\mathbf{loc}}, z^{\mathbf{int}}$ be variables not free in P, Q, or Cmd.

$$\frac{(([y := z]P) \wedge (\bigwedge_{p \in \mathcal{V}(Cmd) - \{x\}} \neg \mathbf{Support}(y,p)) \wedge (\mathbf{cont}(y) = \mathbf{a}_0))\{[y/x]Cmd\}([y := z]Q)}{P\{\,\mathbf{new}\ x\ \mathbf{in}\ Cmd\ \mathbf{wen}\,\}Q}$$

where $\mathcal{V}(Cmd)$ is the set of free variables of procedure and location type in Cmd. (Recall that $\mathbf{a}_0$ is the constant used for initialization.)

The two-sorted assertion language also yields descriptions of invariance (noninterference) among programs and properties. For example, if P implies that the truth of Q is invariant under changes to the contents of any of the locations in the support of Cmd, then Q holds before execution of Cmd iff it holds afterward. This gives the:

Rule of invariance:

$$\frac{P \Rightarrow \forall y^{\mathbf{loc}}[\bigvee_{p \in \mathcal{V}(Cmd)} \mathbf{Support}(y,p) \Rightarrow \mathbf{Invariant}(Q,y)])}{(P \wedge Q)\{\,Cmd\,\}Q\,.}$$

Here the assertion $\mathbf{Invariant}(Q,y)$ is an abbreviation for the first-order formula

$$\forall z^{\mathbf{int}}(Q \equiv [y \leftarrow z]Q)$$

where z is not free in Q. This means that Q is invariant under changes to the contents of y. We remark that invariance is the only rule whose soundness actually depends on the fact that environments use *finite*, as opposed to countable, supports.

Given a set of (mutual) procedure declarations E, let

(i) H_E be the first-order formula which asserts that any location which is in the support of one of the procedure identifiers declared in E is also in the support of one of identifiers of procedure or location type which is free in E,

(ii) $Del_E(Cmd) =_{def}$ the result of replacing in Cmd all occurrences of identifiers declared in E by the constant **diverge** denoting the divergent procedure,

(iii) $CpyExp_E(Cmd) =_{def}$ the result of expanding in Cmd all *outermost* calls of identifiers declared in E by the bodies declared for them as usual according to the copy-rule. In addition, actual parameters of outermost calls to global procedures not declared in E are also expanded.

Copy-Rule Induction:

$$\frac{\begin{array}{c}(H_{E_j} \wedge P_j)\{\, Del_{E_j}(Cmd_j)\,\}Q_j \text{ for } j = 1, \ldots, n, \\ (H_{E_j} \wedge P_j)\{\, Cmd_j\,\}Q_j \mid_{j=1}^{n} \vdash (H_{E_j} \wedge P_j)\{\, CpyExp_{E_j}(Cmd_j)\,\}Q_j \mid_{j=1}^{n}\end{array}}{P_j\{\,\textbf{Proc}\, E_j\, \textbf{do}\, Cmd_j\, \textbf{end}\,\}Q_j \text{ for } j = 1, \ldots, n.}$$

Theorem. For store models D, only D-valid assertions can be proved with AX_D.

In fact, we prove that except for copy-rule induction, the system AX_D is semantically *sound* in the usual logical sense. Namely, an axiom A is sound iff it is valid, and an ordinary inference rule is sound iff, for any store model D and environment e using finite supports, the D, e-validity of its antecedents implies the D, e-validity of its consequent.

Natural deduction style rules of the form

$$\frac{A_1, \quad A_2 \vdash A_3}{A_4}$$

are semantically sound for D iff

> if $D, e \models A_1$ for an environment e using finite supports,
> and $\forall e'$ using finite supports ($D, e' \models A_2$ implies $D, e' \models A_3$),
> then $D, e \models A_4$.

We can show that copy-rule induction is not sound in this sense. (It is sound providing there are no procedure parameters.) We justify the rule instead, following [Olderog, 1981], by showing that if the antecedents are true facts about provability in the particular system AX_D, then the consequent is valid, using special "syntax-directed" properties of AX_D.

It is impossible to find a sound and relatively complete axiomatization for partial correctness assertions about a language as powerful as PROG [Clarke, 1979]. Hence, in our full paper we define PROG′ – a fragment of PROG for which our axiom system is complete in the sense of [Cook, 1978]. Intuitively, this fragment captures commands which generate only a finite number of distinct calls (up to a renaming of first-order variables) when we expand them by replacing a call by its associated body, as in [Clarke, 1979; Olderog, 1981]. We give a direct syntactic condition sufficient to ensure that a program is in PROG′, and demonstrate that our axiom system is complete for as large a class of commands as any other systems in the current literature.

Theorem. AX_D is relatively complete (in the sense of Cook) for PROG′.

7. History. The thesis that λ-calculus underlies programming languages was first extensively argued in [Landin, 1965]. The general denotational approach to semantics which we pursue was developed by Strachey and Scott (cf. [Milne and Strachey, 1976; Scott, 1982]). [Reynolds, 1981a] emphasized the role of typed λ-calculus in explaining the procedure mechanism of ALGOL-like languages. [Damm and Fehr, 1980] and [Damm, 1982] were the first to use explicit translation to typed λ-calculus to analyze ALGOL procedures.

Our proof of equivalence of copy-rule and fixed-point semantics made use of denesting transformations, and seems easier than the corresponding proof in [Apt, 1978], which was for a simpler language (without procedure parameters). [Langmaack, 1973] has used transformations similar to our explicit parameterization and declaration denesting rules to show that a class of ALGOL-like programs can be denested under copy-rule semantics. [Damm, 1982] also contains a proof of the equivalence of copy-rule and fixed-point semantics for denested programs without global procedures.

A comprehensive survey of research on partial correctness of programs is given in [Apt, 1981] to which we refer the reader for a description of the pioneering work of Floyd, Hoare, Cook and Dijkstra. A thorough development of partial correctness for recursive programs with *basic* parameters only is given in [De Bakker, 1980]. De Bakker comes close to achieving the purely denotational approach we propose, though even he compromises in defining the semantics of recursive procedures syntactically. He handles free procedure variables and proves completeness using a fixed-point induction rule. [Harel, Pnueli and Stavi, 1977] keep to a pure denotational approach. However, neither De Bakker nor Harel, et. al. treats sharing (De Bakker requires that his environments be *injections* from identifiers to locations) or procedure parameters, and Harel, et. al. do not even treat blocks; these of course are the source of most of the problems considered in this paper.

Programs with higher-order procedures have been studied in [Langmaack and Olderog, 1980; Olderog, 1981] and more recently in [Damm and Josko, 1982]. Langmaack and Olderog consider an untyped programming language with a copy-rule semantics and independently obtain essentially the same completeness results we described above, restricting themselves to programs without global procedures. (This paper was written well after, and has benefited from, those of Langmaack and Olderog, as well as Damm and Fehr; however, the same ideas were discovered independently by the first author in Novosibirsk.) [German, Clarke, and Halpern, 1983] extend the operational approach to handle globals via closures. Damm and Josko give an axiomatization for a similar typed language with procedure parameters but without basic parameters, local storage, or sharing. They prove a completeness result, but only relative to a higher-order assertion language of uncertain power. Halpern has recently obtained such completeness results for PROG [Halpern, 1983].

[Olderog, 1981] handles sharing using equivalence classes of identifiers. Our approach distinguishing locations and storable values seems technically smoother and more intuitive. [Janssen and van Emde Boas, 1977] also advocate distinguishing locations and their contents, and develop assignment axioms similar to ours. [Schwartz, 1979] also makes the distinction and suggests some axioms for reasoning about sharing; however, he does not justify soundness nor examine completeness of his proposed axioms. Another approach to dealing with sharing is presented in [Cartwright and Oppen, 1981], but their programming language does not allow procedure parameters and places other restrictions on variables appearing in procedures; their assertion language also contains

higher-order features such as quantification over sequences.

8. Conclusions. By explicitly translating ALGOL-like programs to expressions in typed λ-calculus with **letrec**, we have clarified the source of a rich mathematical structure of ALGOL-like programs which guides reasoning, both informally and with formal axiom systems, about side-effects.

The denotational approach identifies a half dozen different levels at which successive features of ALGOL-like languages can be explained. It will be interesting to see whether this mathematical classification serves as a useful guide for teaching such languages.

Although the main texts on denotational semantics gave the impression that the local storage discipline had been assigned a denotational semantics, we discovered none of them gave a "correct" semantics capturing the properties desired for the local storage discipline. Indeed, we argued that correct semantics for the local storage discipline could not be constructed using standard continuous models, but we indicated how to construct satisfactory ω-continuous store models for this discipline.

The one place where we have failed to keep to a purely denotational approach is in our use of copy-rule induction. We believe that this failing can be avoided.

Conjecture: Copy-rule induction can be replaced by fixed-point induction supplemented with a few additional semantically sound rules to yield a semantically sound proof system for partial correctness which is complete for at least as large a fragment of PROG as the current AX_D.

We see two natural next steps building on the work we have presented. The most apparent one is to axiomatize side-effect-free function procedures returning storable values, thereby eliminating the need for the two un-ALGOL-like restrictions imposed on PROG. (See [De Bakker, Klop, Meyer, 1982] for a careful development of functional procedures returning storable values in a language without blocks.) The other, more important step in our view, is to develop the partial correctness theory of commands with global procedure identifiers given partial correctness assertions about the globals.

If partial correctness assertions about arbitrary commands are allowed as hypotheses, then the distinction between partial and *total* correctness blurs. For example, if *Cmd* has no free variables, then it is *total* iff **true**{ *Cmd* }**false** implies **false**. No completeness theory (in the sense of Cook) of total correctness has been developed, and it appears that total completeness can only be proved relative to higher-order theories, e.g., weak second-order. However, it still seems possible to prove (relative) completeness if the assertions allowed as hypotheses are about "pure" procedure calls whose actual parameters contain neither blocks nor commands, and this would be quite interesting (cf. [Meyer and Mitchell, 1982]).

We mention one further open problem of particular interest to us. Note that the obstacles to axiomatizing partial correctness for all of PROG noted by [Clarke, 1979] do not apply if **new** declarations are forbidden.

Open problem: Can PROG – { **new** } be axiomatized (cf. [Damm and Josko, 1982])?

Acknowledgments. The first author benefited from discussions with his colleagues in Novosibirsk, particularly V. Yu. Sazonov, where the approach to using an imperative language and procedure mechanism based on a fully-typed call-by-name λ-calculus was crystallized. Conversations with J. Reynolds significantly influenced our understanding of types and local storage discipline in ALGOL-like languages. We thank W. Damm for many comments and references to related work; B. Josko, who pointed out the definitive counterexample to an earlier "static allocation" approach to new declarations; E. R. Olderog, for his meticulous comments and references for an earlier draft; G. Plotkin, for his elegant counterexamples to soundness of Clarke's procedure call axiom; and J. Mitchell, R. Schwartz, and R. Waldinger for comments. We are very grateful to Flavio Rose for expert help with TEX formatting.

References

K. R. Apt, Equivalence of operational semantics for a fragment of Pascal, in *Formal Descriptions of Programming Language Concepts*, E. J. Neuhold, ed., North Holland, 1978.

K. R. Apt, Ten years of Hoare's logic: a survey – part I, *ACM Trans. Programming Languages and Systems* **3**, 1981, 431–483.

K. R. Apt, Ten years of Hoare's logic, a survey, part II: nondeterminism, *Foundations of Computer Science IV, Mathematical Center Tracts*, **159**, Mathematisch Centrum, Amsterdam, 1983, 101–132.

H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, Studies in Logic **103**, North Holland, 1981.

R. Cartwright and D. Oppen, The logic of aliasing, *Acta Informatica* **15**, 1981, 365–384.

E. M. Clarke, Programming language constructs for which it is impossible to obtain good Hoare-like axioms, *J.ACM* **26**, 1979, 129–147.

S. A. Cook, Soundness and completeness of an axiom system for program verification, *SIAM J. Computing* **7**, 1978, 70–90.

W. Damm, The IO- and OI-hierarchies, *Theoretical Computer Science* **20**, 1982, 95–207.

W. Damm and E. Fehr, A schematological approach to the procedure concept of ALGOL-like languages, *Proc. 5ieme colloque sur les arbres en algebre et en programmation*, Lille, 1980, 130-134.

W. Damm and B. Josko, A sound and relatively* complete Hoare-logic for a language with higher type procedures, Lehrstuhl fur Informatik II, RWTH Aachen, Bericht No. 77, 1982, 94pp.

J. De Bakker, *Mathematical Theory of Program Correctness*, Prentice-Hall International, 1980, 505pp.

J. De Bakker, J. W. Klop, and J.-J.Ch. Meyer, Correctness of programs with function procedures, *Logics of Programs*, D. Kozen, ed., Lecture Notes in Computer Science **131**, Springer, 1982, 94–112.

S. German, E. Clarke, and J. Halpern, Reasoning about procedures as parameters, 1983, *this volume.*

I. Guessarian, Survey on some classes of interpretations and some their applications, Laboratoire Informatique Theorique et Programmation, 82-46, Univ. Paris 7, 1982.

G. A. Gorelick, A complete axiom system for proving assertions about recursive and non-recursive programs, University of Toronto, Computer Science Dept. TR–75, 1975.

J. Y. Halpern, A good Hoare axiom system for an ALGOL-like language, 1983, to appear.

D. Harel, A. Pnueli, and J. Stavi, A complete axiomatic system for proving deductions about recursive programs, *Proc.* 9^{th} *ACM Symp. Theory of Computing*, 1977, 249–260.

C. A. R. Hoare, An axiomatic basis for computer programming, *Comm. ACM*, **12**, 1969, 576–580.

T. M. V. Janssen and P. van Emde Boas, On the proper treatment of referencing, dereferencing and assignment, 4^{th} *Int'l. Coll. Automata, Languages, and Programming*, Lecture Notes in Computer Science **52**, Springer, 1977, 282-300.

P. J. Landin, A correspondence between ALGOL 60 and Church's lambda notation, *Comm. ACM* 8, 1965, 89-101 and 158-165.

H. Langmaack, On procedures as open subroutines, *Acta Informatica* 2, 1973, 311–333.

H. Langmaack and E. R. Olderog, Present-day Hoare-like systems, 7^{th} *Int'l. Coll. Automata, Languages, and Programming*, Lecture Notes in Computer Science **85**, Springer, 1980, 363–373.

Z. Manna and R. Waldinger, Problematic features of programming languages: a situational-calculus approach, *Acta Informatica* **16**, 1981, 371–426.

J. Meseguer, Completions, factorizations and colimits of ω-posets, *Coll. Math. Soc. Janos Bolyai* **26**. *Math. Logic in Computer Science*, Salgotarjan, Hungary, 1978, 509-545.

A. R. Meyer, What is a model of the λ-calculus? *Information and Control* **52**, 1982, 87–122.

A. R. Meyer and J. C. Mitchell, Axiomatic definability and completeness for recursive programs, 9^{th} *ACM Symposium on Principles of Programming Languages*, 1982, 337–346. Revised as: Termination assertions for recursive programs: completeness and axiomatic definability, MIT/LCS/TM-214, MIT, Cambridge, Massachusetts, March, 1982; to appear *Information and Control*, 1982.

R. E. Milne and C. Strachey, *A Theory of Programming Language Semantics*, 2 Vols., Chapman and Hall, 1976.

P. Naur et al., Revised report on the algorithmic language ALGOL 60, *Computer J.* **5**, 1963, 349–367.

E. R. Olderog, Sound and complete Hoare-like calculi based on copy rules, *Acta Informatica* **16**, 1981, 161–197.

E. R. Olderog, A characterization of Hoare's logic for programs with Pascal-like procedures, *Proc.* 15^{th} *ACM Symp. Theory of Computing*, 1983a, 320-329.

E. R. Olderog, Hoare's logic for program with procedures – what has been accomplished?, *Proc. Logics of Programs*, Carnegie-Mellon Univ., Pittsburgh, 1983, *to appear, Lecture Notes in Computer Science*, Springer, 1983b.

F. J. Oles, Type algebras, functor categories, and block structure, Computer Science Dept., Aarhus Univ. DAIMI PB-156, Denmark, Jan. 1983.

G. D. Plotkin, A Powerdomain for countable non-determinism, 9^{th} *Int'l. Coll. Automata, Languages, and Programming*, Lecture Notes in Computer Science 140, Springer, 1982, 412–428.

J. C. Reynolds, The essence of ALGOL, *International Symposium on Algorithmic Languages*, de Bakker and van Vliet, eds., North Holland, 1981a, 345–372.

J. C. Reynolds, The Craft of Programming, Prentice Hall International Series in Computer Science, 1981b, 434pp.

J. C. Reynolds, Idealized ALGOL and its specification logic, Syracuse University, Technical Report 1-81, 1981c.

R. L. Schwartz, An axiomatic treatment of ALGOL 68 Routines, 6^{th} *Int'l. Coll. Automata, Languages and Programming*, Lecture Notes in Computer Science 71, Springer, 1979, 530–545.

D. S. Scott, Domains for Denotational Semantics, 9^{th} *Int'l. Conf. Automata, Languages, and Programming*, Lecture Notes in Computer Science 140, Springer, 1982, 577–613; to appear, *Information and Control.*

J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Massachusetts, 1977.

B. A. Trakhtenbrot, On relaxation rules in algorithmic logic, *Mathematical Foundations of Computer Science 1979*, (J. Becvar, ed.), Lecture Notes in Computer Science 74, Springer, 1979, 453–462.

Cambridge, Massachusetts
October 3, 1983