MIT/LCS/TM-245

UNDERSTANDING ALGOL: A VIEW OF A RECENT CONVERT TO DENOTATIONAL SEMANTICS

Albert R. Meyer

October 1983

Understanding ALGOL: The View of a Recent Convert to Denotational Semantics* †

Albert R. Meyer
Laboratory for Computer Science,
Massachusetts Institute of Technology

Abstract. The advantages of denotational over copy-rule semantics are argued. A denotational semantics is indicated 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 described. The semantics justifies a proof system for partial correctness of commands containing global procedures.

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 based on the author's invited lecture at the IFIPS Symposium, Paris, September, 1983, and will appear in the IFIPS Proceedings, R. E. Mason, ed., North Holland. It is a shortened version of a paper by Trakhtenbrot, B. A., Halpern, J. Y., and Meyer, A. R., "From denotational to operational and axiomatic semantics for ALGOL-like languages: an overview," to appear in *Logic of Programs*, Proceedings, Clarke and Kozen, eds., Lecture Notes in Computer Science, Springer, 1983.

†The research reported here was supported in part by NSF Grant MCS80-10707 and a grant to the MIT Lab. for Computer Science from the IBM Corporation.

The announced topic of my invited address was on a subject other than the present paper, but the research on the logic of ALGOL-like programs which I have been pursuing for nearly two years jointly with Boris Trakhtenbrot and Joe Halpern has proved so absorbing, and has led to the unexpected discovery of such rich and elegant mathematical structure, that I could not focus my attention on my planned topic. So I have chosen to write and speak about the subject which has been preoccupying me these past several months.

This short paper gives an overview of our development of a denotational basis for proof systems concerning ALGOL-like programs. Originally we found it curious that despite wide-spread (though by no means unanimous) acceptance of denotational semantics, the numerous papers (cf. [Apt, 1981]) developing Hoare-style logics consistently deviated from a purely denotational approach. In every case, this ideal approach had been deemed inconvenient and had 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 adequately been worked out. We believe our work demonstrates that the denotational ideal is achievable without excessive complication and with important benefits.

1. 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.

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.

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). So we consistently distinguish locations from their contents, using the token **cont** for explicit dereferencing. The 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, partial (actually, nondeterministic) mappings from stores to stores. 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**).

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

$$\mathbf{proc}\ p(x) \Leftarrow DeclBody\ \mathbf{do}\ BlockBody\ \mathbf{end}\,.$$

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

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

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

$$\mathbf{let}\ n\ \mathbf{be}\ BasE\ \mathbf{in}\ p(n)\ \mathbf{tel}\,.$$

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.

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

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

The following program, which includes a procedure that swaps the contents of two locations unless they are equal, is contrived to illustrate the sharing features of PROG. 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. The procedure *swap*:

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

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 ("$\mathrm{f(a)} = \mathrm{f}(\mathrm{cont}(y))$"). Expressions which evaluate to locations are allowed, as in the "conditional variable" expression to the left of the final assignment command in the example.

2. Semantics via 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. 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. 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.

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**,

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

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 $[\![\textbf{Dint}]\!]$ is such that

$$[\![\textbf{Dint}]\!]d_1 d_2 s =_{def} (d_1\,(d_2(s)))(s)$$

providing $d_2(s) \neq \perp_{\text{int}}$, and diverges ($= \emptyset$) otherwise. 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 $[\![\textbf{New}]\!]d^{\textbf{loc}\to\textbf{prog}}$ 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 $e(p) = d$.

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.

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

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

The equivalence of fixed-point and copy-rule semantics is the most fundamental continuity property.

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 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.

There is 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. (The proof theory we develop focuses on the class of "before – after" assertions about commands known as partial correctness assertions. A *partial correctness assertion* is a triple consisting of two formulas and a command, written $P\{\,Cmd\,\}Q$. It asserts that, if the precondition P holds before execution of the nondeterministic command Cmd, then the postcondition Q is satisfied by every terminating state (if any) of Cmd.)

Let P and Q be first-order assertions, and 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. Then

Fixed-Point Induction:

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

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 induction is an easily proved continuity property.

[Clarke, 1979], extending [Gorelick, 1975], introduced a proof rule for partial correctness of higher-order procedure calls closely resembling the fixed-point induction rule but justified using the copy-rule, and subsequent work [Langmaack and Olderog, 1980; Olderog, 1981, 1983a, 1983b; Apt, 1981] has followed this approach. In our full paper we formulate 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. 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. Free procedure identifiers raise special 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. 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! In particular, the function *Select* hypothesized in defining the **New** constant in §2 is not monotonic.

In general, objects with "large" support force us to deal with 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 a program with infinite support can be the lub of an infinite sequence of programs each with finite support. As with monotonicity, completeness is normally required of the domains defining the semantics of λ-terms, in this case to ensure that fixed-point and symbolic-operational semantics agree.

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:

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

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. The discontinuity of new storage allocation is also noted in [Milne and Strachey, 1976].

Store models overcome these difficulties: they are domains of mappings with finite and 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 thus 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.)

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

$$\begin{aligned}
\textbf{new } x \textbf{ in } x := \mathrm{b} \textbf{ wen} &\equiv \textbf{skip}\,,\\
\textbf{new } x \textbf{ in } x := \mathrm{a}_0; Cmd \textbf{ wen} &\equiv \textbf{new } x \textbf{ in } Cmd \textbf{ wen}\,,\\
\textbf{new } x \textbf{ new } y \textbf{ in } Cmd \textbf{ wen wen} &\equiv \textbf{new } y \textbf{ new } x \textbf{ in } Cmd \textbf{ wen wen}\,,\\
\textbf{new } x \textbf{ in if } x = y \textbf{ then } Cmd_1 \textbf{ else } Cmd_2 \textbf{ fi wen} &\equiv y := \mathbf{cont}(y); Cmd_2\,.
\end{aligned}$$

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 (with global procedures) in PROG.

5. 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 **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 a command with global procedure identifiers unless we are told which locations are in the support of the globals.

Distinguishing locations and their contents becomes particularly beneficial when making assertions about programs. With this 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. We therefore judge the soundness of arbitrary renamings, which follows routinely for our system, to be a valuable feature.

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

a, the contents of x will equal 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 $LocE$ whose evaluation yields a location is not valid. The problem here is that the meaning of $LocE$ depends on the store; in general $LocE$ 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 of first-order formulas *without* support predicates which are valid in D.

Axioms for the programming constructs **diverge**, sequencing, conditionals, **let** declarations, as well as "logical" rules for consequence, conjunction and quantification, and substitution, along with assignment above, are as usual. We enumerate below 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 the range of globals is restricted to objects with *finite*, as opposed to countable, supports.

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 demonstrating that our axiom system is complete for as large a class of commands as any other systems in the current literature.

6. 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.

This paper is a sketch of ongoing research. A more extended survey of our results will appear in Trakhtenbrot, B. A., Halpern, J. Y., and Meyer, A. R., "From denotational to operational and axiomatic semantics for ALGOL-like languages: an overview," in *Logic of Programs*, Proceedings, Clarke and Kozen, eds., Lecture Notes in Computer Science, Springer, 1983.

References

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

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

A. Blikle, Naive denotational semantics, *Proc. IFIP Congress*, 1983.

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.

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

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.

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.

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, 1983b, *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.

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
September 26, 1983