THE SEMANTICS OF LOCAL STORAGE,
OR WHAT MAKES THE FREE-LIST FREE?

J.Y. Halpern
A.R. Meyer
B.A. Trakhtenbrot

April 1984
The Semantics of Local Storage, or What Makes the Free-List Free?  
(Preliminary Report)*†

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

Abstract. Denotational semantics for an ALGOL-like language with finite-mode procedures, 
blocks with local storage, and sharing (aliasing) is given by translating programs into an 
appropriately typed λ-calculus. Procedures are entirely explained at a purely functional level – 
indendent of the interpretation of program constructs – by continuous models for λ-calculus. 
However, the usual (epo) models are not adequate to model local storage allocation for blocks 
because storage overflow presents an apparent discontinuity. New domains of store models are 
offered to solve this problem.

CR Categories and Subject Descriptors: D.3.1 [Programming Languages]: Formal Definitions 
and Theory—syntax, semantics; F.3.2 [Logics and Meanings of Programs]: Semantics of Pro-
gramming Languages—operational semantics, denotational semantics; F.3.3 [Logics and Mean-
ings of Programs]: Studies of Program Constructs.

General Terms: Languages, Semantics, Theory.

Additional Key Words and Phrases: lambda-calculus, copy-rule, stack discipline, block structure.

* This research was supported in part by NSF Grants MCS80-10707, MCS-8304498, and a grant to the MIT 
Laboratory for Computer Science from the IBM Corporation.

†This report is a slightly revised draft of a paper in the Conference Record of the 11th ACM Symposium on Principles 
of Programming Languages, January, 1984, 245-257.
1. The Problem of Free Locations.

ALGOL-like languages obey a "stack discipline" in which local storage for blocks is allocated from the top of a memory stack at block entry time. For object-oriented languages like LISP orCLU requiring heap storage, new memory locations (also known as local variables) are usually allocated from a "heap" or linked list of free locations.

In both cases, there is a simple idea behind local variables in blocks: execution of a block
begin new \textit{z} in \textit{Body} end
causes allocation of a "new" storage location denoted by the identifier \textit{z} which is used in the body of the block. In ALGOL-like languages obeying stack discipline, the location is deallocated upon exit from the block. Understood in this way, stack discipline is a language design principle - encouraging modularity in program construction - rather than an implementation technique for efficient storage management. It is better called the local storage discipline to avoid misunderstanding, and we do so henceforth.

The simple idea behind local storage raises a theoretical puzzle: what is a "new" location? We were disappointed to discover that the mathematical models of storage allocation which appear in the denotational semantics literature [Milne and Strachey 76; Stoy 77; Gordon 79] do not adequately address this problem. Instead, these models merely reflect the bookkeeping mechanisms used in implementations. Specifically, new storage allocation is modeled by enriching the notion of stores to include with each location an indication of whether the location is "active". Execution, starting on some store, of a block with local storage involves selecting the first "free" (i.e., not marked "active") location of the store as the one to be allocated.

The problem with this approach is that the locations designated by the store as free may already be accessible from the body of the block, and so may not in fact be free. For example, let \textit{z} be an identifier of location (in this context, also called a \textit{reference}) type, and let \textit{p} be a parameterless pure procedure identifier. Then, the block

\begin{verbatim}
begin new \textit{z} in
\hspace{1em} \textit{z} := 1; \textit{p}; \text{if} \text{cont}(\textit{z}) = 0 \text{then} \text{skip} \text{else} \text{diverge} \text{fi}
end
\end{verbatim}

ought to diverge since the "new" location allocated for \textit{z} should not be affected by the call to \textit{p}. But if \textit{p} happens to denote the program which assigns the value zero to some location \textit{l}, and this block is executed on a store in which location \textit{l} happens to be designated as the first free location, then the block will not 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 locations which \textit{really are} active, and we are in any case still left with the problem of explaining what a free location "really" is.

The semantics using activity marks does behave properly on programs without calls to global (undeclared) procedure identifiers. For example, the block above will diverge in any program context in which the global identifier \textit{p} is declared (in a declaration which itself does not contain
global procedures). In this case, execution of the overall program will correctly update the free list so that the locations affected by \( p \) will be marked as active by block execution time. This can be proved by induction on the length of computation of programs without procedure globals. However, this observation leaves several matters unresolved:

1. Suppose we add some new command to the language – say one which initializes some special portion of the store? This enriches the possible ways \( p \) might be declared, requiring re-verification of the allocation mechanism for the richer class of \( p \)'s. (In fact, this enrichment invalidates the mechanism unless all locations in the special portion of the store are permanently marked active).

2. More generally, suppose \( p \) is a call to a program written in another language – say a system program in machine language? Allocation from the free list will not be safe.

3. The simple reasoning that goes with the idea that “new” storage is allocated at block entry must be replaced by reasoning about the details of particular allocation mechanisms.

We address these problems by explaining \textit{semantically} when a location is active or free with respect to a procedure. In general, we define how a set of locations \textit{covers} a procedure of finite type, by induction on types. The \textit{support} of a procedure is its minimal cover. The locations in the support are “active” with respect to a procedure, and the locations outside the support are “free”. The desired semantical explanation of new storage allocation is then simply that any location free for the block body is to be allocated – no other details of the allocation mechanism need be considered.

An amusing technical problem must be faced with this approach. Some kind of continuity condition is normally required of the functions defining the semantics of procedures in order to ensure that the fixed-points necessary to explain recursive definitions exist. Unfortunately, in the usual formulations the operation of allocating and later de-allocating “new” storage turns out not to be continuous, essentially because of the theoretical possibility of running out of storage – even if we assume there are an infinite number of locations in memory! For example, suppose \( \pi \) is a store to store mapping whose only cover is the set of \textit{all} locations – \( \pi \) might be the denotation of a procedure which “sweeps” memory searching for an untagged location. Now \( \pi \) can be expressed as the limit of a sequence of approximating mappings \( \pi_i \) which only sweep the first \( i \) locations. Since storage is infinite but a finite number of locations cover \( \pi_i \), there is always a location free to allocate for a block whose body behaves like \( \pi_i \). On the other hand, allocating new storage for \( \pi \) yields an \textit{overflowed error}, viz., allocating local storage and taking limits do not commute as required by the definition of continuity. (The discontinuity of new storage allocation was noted in [Milne and Strachey, 76], 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. We would like to rule out such objects, especially in view of the fact that \textit{definable} objects, viz., objects which are the denotations of phrases in ALGOL-like languages, can be proved to depend
on only finitely many locations. However, once we have mappings (like \( \pi \)) which depend on only finitely many locations, the usual requirement that semantical domains be \textit{complete partial orders} (cpo's) which are closed under taking least upper bounds of all increasing chains forces us to admit programs (like \( \pi \)) with infinite support [Sloy 77; Scott 81, 82]. Difficulties of this sort have led [Reynolds, 81] and [Oles, 83] to consider more sophisticated functor categories as domains of interpretation. For further discussion see [Meyer, 83; Trakhtenbrot, Halpern and Meyer, 83].

Our solution is to relax the requirement that domains be closed under all (increasing) limits. We require closure only under certain “algebraic” limits sufficient to ensure that domains will obey the fixed-point and other properties required for program semantics. This theory of \textit{algebraically closed partial orders} is less well known than the cpo theory, but has been developed extensively [Nivat, 75; Guessarian 81; Guessarian 82; Gallier, 1983; Courcelle, 1983]. In this framework, we give a general definition of the notion of covering, and define \textit{store models}: systems of algebraically closed partial orders containing only elements with finite support but including enough elements to interpret all the programming constructs of ALGOL-like languages.

Store models justify all the intended properties of \textit{new}-declarations. For example, in store models the block mentioned above with global call to \( p \) indeed diverges in all environments. Another illustrative equivalence is:

\[
\textbf{begin new } z \textbf{ in if } z = y \textbf{ then } Cmd_1 \textbf{ else } Cmd_2 \textbf{ end } \equiv (y := \text{cont}(y); Cmd_2).
\]

(The “useless” assignment to \( y \) appears in case \( y \) denotes the divergent \( (\bot) \) location.)

2. \textbf{ALGOL-like Languages.}

The focus of our proof-theoretic studies has been on the family of idealized ALGOL-like languages. We review several of the principles which characterize this class of languages [cf. Reynolds, 81; Meyer, 83; Trakhtenbrot, Halpern, and Meyer, 83; Halpern, 84]:

(1) \textit{Commands}, which alter the store but do not return values, are distinguished from \textit{expressions}, which return values but have no side-effects.

(2) Calling is \textit{by-name}. (Calls by-value, etc., are treated as syntactic sugar.)

(3) Higher-order procedures of all \textit{finite} types (in ALGOL 68 jargon, \textit{modes}) are allowed.

(4) The local storage discipline is an explicit aspect of the semantics.

In this section we sketch a few of the features of an illustrative ALGOL-like language we call \textbf{PROG}.

\textbf{Types in PROG}. The distinction between locations and storable values – in our semantics they behave as disjoint domains – is one of several structural restrictions on ALGOL-like languages implied by local storage discipline. For example, it is well-known that locations (and likewise procedures) cannot be storable without restriction, since otherwise locations allocated inside a block might be accessible after exit from the block via the stored objects.
For simplicity, we consider storable values of only one type. The two basic types storable values and locations are abbreviated int and loc, respectively. PROG syntax mandates an explicit type distinction between locations and storable values (also called “left” and “right” values of expressions), using the token cont for explicit dereferencing. Thus, \( \text{cont}(x^{\text{loc}}) \) denotes the element of type int which is the contents of \( x \), and assignment commands take the form \( \text{loc}E := \text{int}E \) where \( \text{loc}E \) is a location-valued expression and \( \text{int}E \) is an int-valued expression.

Equality tests in PROG can only be between elements of basic type. We do allow explicit equality testing between locations, \( x^{\text{loc}} = y^{\text{loc}} \), in addition to the usual test of equality between storable values, \( a = f(\text{cont}(y^{\text{loc}})) \). Expressions which evaluate to locations are allowed, as in the "conditional variable" expression on the lefthand side of the assignment command

\[
\text{if } a = f(\text{cont}(y)) \text{ then } y \text{ else } z \text{ fi } := a.
\]

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. Since we do not attempt to distinguish "failing" from diverging, nondeterminism is adequately modeled with mappings to sets as opposed to the more complex power-domains of [Plotkin, 76,82; Smyth, 78].) The other two "expression" types are the denotations of expressions whose evaluation yields basic values, viz., the elements of intexp (locexp) are functions from stores to int (loc), i.e., "thunks" in ALGOL jargon.

Blocks and Binding in PROG. Procedures of all higher finite types formed from the five primitive types may be declared, passed as parameters, and returned as values.

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

\[
\text{proc } p(x) = \text{DeclBody} \text{ do BlockBody end.}
\]

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

\[
\begin{align*}
\text{let } x^{\text{int}} & \text{ be } \text{int}E \text{ in } \text{Cmd} \text{ tel,} \\
\text{let } y^{\text{loc}} & \text{ be } \text{loc}E \text{ in } \text{Cmd} \text{ tel,} \\
\text{begin new } y^{\text{loc}} & \text{ in } \text{Cmd} \text{ end.}
\end{align*}
\]

The let-declaration causes the evaluation of the expression \( \text{int}E \) in the declaration-time store and causes identifier \( x \) to denote the result of the evaluation. (A call-by-value of the form \( p(BasE) \) can be simulated by the basic block \( \text{let } n \text{ be } BasE \text{ in } p(n) \text{ tel.} \) Basic and procedure declarations have quite different scopes and meaning, as will be revealed below.

3. Syntax-Preserving Translation to \( \lambda \)-Calculus.

We formalize the assignment of semantics to programs in two steps:

(1) a purely syntactic translation from PROG to a fully-typed \( \lambda \)-calculus enriched with a letrec-construct as in [Damm and Fehr, 1980; Damm, 1982; cf. Landin, 65], and
(2) assignment of semantics to the \( \lambda \)-calculus in a standard referentially transparent way [Barendregt, 81; Meyer, 82].

Our \( \lambda \)-calculus is chosen so that its constants correspond to program constructors, its binding operations, \texttt{letrec} and \( \lambda \), correspond to program declarations and procedure abstraction, and its types are the \textit{same} as those of the programming language. In fact, the abstract syntax, viz., parse tree, of the translation of a program is actually \textit{identical} to that of the program; the translation serves mainly to make the variable binding conventions of \texttt{PROG} explicit.

Procedure blocks are translated using \texttt{letrec}, so for example,

\[
Tr(\text{proc } p(x) = \text{DeclBody do BlockBody end}) \equiv \texttt{letrec } p = \lambda x. Tr(\text{DeclBody}) \text{ in } Tr(\text{BlockBody}).
\]

This recursive declaration of \( p \) binds occurrences of \( p \) in both the declaration and the block bodies. Procedure declarations in this way inherit the \textit{static scoping rules} of \( \lambda \)-calculus.

Basic blocks are handled with constants and \( \lambda \)'s, e.g.,

\[
Tr(\text{let } z^\text{int} \text{ be } \text{IntE in } \text{Cmd tel}) = \texttt{Dint} (\lambda x. Tr(\text{Cmd})) (Tr(\text{IntE}))
\]

where \texttt{Dint} is a constant of type \( \text{(int} \rightarrow \text{prog}) \rightarrow \text{intexp} \rightarrow \text{prog} \). Note that the binding effect of the block on \( z^\text{int} \) is reflected in the binding effect of \( \lambda x \) on \( Tr(\text{Cmd}) \), namely, the declaration binds \( z \) in \( \text{Cmd} \), but does not bind \( z \) in \( \text{IntE} \), in contrast to the case for procedure declarations. Basic blocks with declarations of location type are translated using a corresponding combinator \texttt{Dloc}. Similarly,

\[
Tr(\text{begin new } z \text{ in } \text{Cmd end}) \equiv \texttt{New} (\lambda x. Tr(\text{Cmd}))
\]

where \texttt{New} is a special constant of type \( \text{(loc} \rightarrow \text{prog}) \rightarrow \text{prog} \). The semantics of \texttt{New} will be defined so that \( \text{Cmd} \) runs in an environment in which \( z \) is bound to some location outside a cover of \( \text{Cmd} \). The contents of this new location are initialized to some standard value denoted by the constant \( a_0 \) at the beginning of the computation of \( \text{Cmd} \) and restored to their original value at the end.

Other commands and expressions are translated directly by introducing suitable constants (but no binding operators), e.g.,

\[
Tr(\text{cont}(\text{LocE})) = \texttt{Cont}(Tr(\text{LocE})),
\]

\[
Tr(\text{LocE := IntE}) = \texttt{Update} (Tr(\text{LocE})) (Tr(\text{IntE})),
\]

\[
Tr(\text{Cmd}_1; \text{Cmd}_2) = \texttt{Seq} (Tr(\text{Cmd}_1)) (Tr(\text{Cmd}_2)),
\]

etc.

The principal consequence of this syntax-preserving translation is that all the properties of \textit{procedure} declarations in \texttt{ALGOL}-like languages such as renaming rules associated with static scope, declaration denesting rules, and expansions of recursive declarations, can be recognized as direct consequences of the corresponding purely functional properties of the \texttt{letrec-\lambda}\textit{-calculus} – which have nothing at all to do with side-effects. Before elaborating this point, we review the properties of the \texttt{letrec-calculus}.
4. Typed Lambda Calculus.

Let $T$ be a set of primitive type symbols, $C$ be a set of typed constants, and $X$ be a set of typed variables.

Type expressions are defined inductively: the primitive type symbols are type expressions, and if $\alpha, \beta$ are type expressions, then so are $\alpha \to \beta$ and $\alpha \times \beta$. With each type expression $\alpha$ we associate a (possibly empty) set of constants $C_\alpha$, disjoint from $C_\beta$ for $\alpha \neq \beta$. With each $\alpha$ we also associate an infinite set of variables $X_\alpha$, disjoint from $X_\beta$ for $\alpha \neq \beta$. We use the notation $x^\alpha$ when we wish to emphasize $x \in X_\alpha$. By definition, $C = \cup_\alpha C_\alpha$ and $X = \cup_\alpha X_\alpha$.

We define $L^\alpha$, the terms of letrec-$\lambda$-calculus of type $\alpha$, by induction.

1. $C_\alpha \cup X_\alpha \subseteq L^\alpha$.

2. Application: If $u \in L^{\alpha\to\beta}$, $v \in L^\alpha$, then $\langle u \, v \rangle \in L^\beta$.

3. Abstraction: If $x \in X_\alpha, u \in L^\beta$, then $\lambda x. u \in L^{\alpha\to\beta}$.

4. Block with mutual procedure declarations: If $x_j \in X^{\alpha_j}, u_j \in L^{\alpha_j}, j = 1, \ldots, k$, $x_j$ all distinct, and $v \in L^\beta$ then $\langle \text{letrec } x_1 = u_1 \text{ and } \ldots \text{ and } x_k = u_k \text{ in } v \rangle \in L^\beta$. We say $x_j$ is declared in this block with declaration body $u_j$, and $v$ is the block body.

Free and bound occurrences of variables are defined as usual [Hindley, Lercher and Seldin, 1972; Stoy, 1977; Barendregt, 1981]. Note we are allowing recursion here: the variables $x_j$ may occur in $u_i$ as well as $v$. In particular, "letrec $x_j$" binds all free occurrences of $x_j$ in $u_1, \ldots, u_k, v$.

As usual, we omit parentheses in compound applications with association to the left being understood. In contrast, the operations $\to$ and $\times$ associate to the right in compound type expressions. Thus $uvw$ abbreviates $((uv)w)$ while $\alpha \to \beta \to \gamma$ abbreviates $(\alpha \to (\beta \to \gamma))$. We let $[v/z]u$ denote the result of substituting the term $v$ for free occurrences of $z$ in $u$ subject to the usual provisos about renaming bound variables in $u$ to avoid capture of free variables in $v$ [Stoy, 1977, Def. 5.7; Barendregt, 1981, Appendix C].

5. Cartesian Closed Models.

For any sets $D_1, \ldots, D_n$, let $D_1 \times \cdots \times D_n$ be the set of all ordered $n$-tuples $(d_1, \ldots, d_n)$ of elements $d_i \in D_i$. Let $\text{tuple}_{D_1, \ldots, D_n} : D_1 \to \cdots \to D_n \to (D_1 \times \cdots \times D_n)$ be defined by:

$$\text{tuple} \, d_1 \cdots d_n = (d_1, \ldots, d_n),$$

and let $\text{proj}_{D_1, \ldots, D_n}^i : (D_1 \times \cdots \times D_n) \to D_i$ be projection on the $i^{th}$ coordinate.

A Cartesian Closed type-frame consists of a family of sets $\{D_\alpha\}$ called domains or types, one for each type expression $\alpha$, such that

1. $D_\alpha \to \beta$ consists of some nonempty family of functions from $D_\alpha$ to $D_\beta$ and $D_\alpha \times \beta = D_\alpha \times D_\beta$, and
(2) there are elements \( S_{\alpha,\beta,\gamma} \in D_{(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma} \) and \( K_{\alpha,\beta} \in D_{\alpha \rightarrow \beta \rightarrow \alpha} \) for every \( \alpha, \beta, \gamma \) such that
\[
S_{\alpha,\beta,\gamma} d_0 d_1 d_2 = (d_0 d_2)(d_1 d_2),
\]
\[
K_{\alpha,\beta} d_3 d_4 = d_3.
\]

(3) \( \text{tuple}_{D_{a_1}, \ldots, D_{a_n}} \in D_{(a_1 \times \ldots \times a_n) \rightarrow a} \), and similarly \( \text{proj}^i_{D_{a_1}, \ldots, D_{a_n}} \in D_{(a_1 \times \ldots \times a_n) \rightarrow a_i} \).

An environment for a type-frame \( \{D_\alpha\} \) is a mapping \( \epsilon : X \rightarrow D \), where \( D = \bigcup_\alpha D_\alpha \), which respects types, i.e., \( \epsilon(x^\alpha) \in D_\alpha \). Given an environment \( \epsilon \), let \( \epsilon[d/z] \) denote the environment which differs from \( \epsilon \) only at \( z \), and \( (\epsilon[d/z])(x) = d \). Let \( \epsilon[d_1/p_1, \ldots, d_k/p_k] \) abbreviate \( \epsilon[d_1/p_1, \ldots, d_k/p_k][d_k+1/p_k+1] \). (We define the “patch”, \( f[b/a] \), of any function \( f : A \rightarrow B \), at \( a \in A \), by \( b \in B \) similarly.) Let \( \text{Env}_D \) be the set of all environments for \( D \).

A Cartesian closed model consists of a Cartesian closed type frame together with an interpretation of the constants, i.e., a mapping \( \llbracket \rrbracket_D : C \rightarrow D \) which respects types. The model is standard iff the constant symbols \( S_{\alpha,\beta,\gamma} \in C_{(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma} \) and \( K_{\alpha,\beta} \in C_{\alpha \rightarrow \beta \rightarrow \alpha} \) are interpreted as the corresponding \( S \) and \( K \) functions, and similarly for the constants \( \text{tuple} \) and \( \text{proj}^i \). Let \( L_1 \subseteq L \) be the usual typed \( \lambda \)-calculus (without \( \text{letrec} \)). The justification for this peculiar definition is that for any Cartesian closed model \( D \), there exists a unique mapping \( \llbracket \rrbracket_D : L_1 \rightarrow \text{Env}_D \rightarrow D \) which respects types such that

(a) \( \llbracket c \rrbracket_D \epsilon = \llbracket c \rrbracket_D \alpha \),
(b) \( \llbracket x \rrbracket_D \epsilon = \epsilon(x) \),
(c) \( \llbracket u v \rrbracket_D \epsilon = (\llbracket u \rrbracket_D \epsilon)(\llbracket v \rrbracket_D \epsilon) \),
(d) for all \( d \in D_\alpha, (\llbracket \lambda x. u \rrbracket_D \epsilon) d = \llbracket u \rrbracket_D (\epsilon[d/x]) \).

A fixed-point frame is a Cartesian closed frame such that there is an element \( Y_\alpha \in D_{(\alpha \rightarrow \alpha) \rightarrow \alpha} \) such that
\[
Yf = f(Yf)
\]
for all \( f \in D_{\alpha \rightarrow \alpha} \) and all type expressions \( \alpha \). A fixed-point model is a model whose type frame is a fixed-point frame; it is standard iff the constants above have the standard interpretation and the constant symbols \( Y_\alpha \in C_{(\alpha \rightarrow \alpha) \rightarrow \alpha} \) are interpreted as fixed point operators \( Y_\alpha \).

Let \( \lambda(x_1, \ldots, x_n).u \) abbreviate
\[
\lambda z . ((\text{proj}^1 z)/x_1) \ldots ((\text{proj}^n z)/x_n) u
\]
for \( z \) not free in \( u \).

Terms \( u \) and \( v \) are equivalent for some model \( D \), written \( u \equiv_D v \), iff \( \llbracket u \rrbracket_D = \llbracket v \rrbracket_D \). If \( M \) is a class of models, \( u \) and \( v \) are \( M \)-equivalent iff \( u \equiv_D v \) for all models \( D \in M \).

For any Cartesian closed fixed-point model \( D \), there exists a unique mapping \( \llbracket \rrbracket_D : L \rightarrow \text{Env}_D \rightarrow D \) which respects types, satisfies (a d) above, and such that

(c) \( \text{letrec} p_1 = u_1 \ and \ldots \ and \ p_n = u_n \ in \ v \equiv_D (\lambda(p_1, \ldots, p_n).v)(Y(\lambda(p_1, \ldots, p_n).\text{tuple} u_1 \ldots u_n)) \).
We abbreviate a mutual procedure declaration of the form \( \text{letrec } p_1 = u_1 \text{ and } \cdots \text{ and } p_n = u_n \text{ in } v \) by \( \text{letrec } \text{Dec in } v \), where \( \text{Dec} = \{ p_1 = u_1, \ldots, p_n = u_n \} \).

The following fundamental inference rule verifies the referential transparency of \( L \). It is sound in any Cartesian closed model when we merely regard \( \text{letrec } \text{Dec in } v \) as an abbreviation for 
\( (\lambda(p_1, \ldots, p_n).v)(\text{tuple } u_1 \ldots u_n) \)
without assuming any facts (such as fixed-point properties) about the constant \( c \).

**Replacement Rule.** If \( u \equiv v \) and \( w_2 \) is the result of literally replacing (without renaming bound variables) an occurrence of \( u \) by \( v \) in \( w_1 \), then \( w_1 \equiv w_2 \).

The following equivalences hold in any Cartesian closed model as well.

**Variable renaming,** viz., \( \alpha \)-conversion:

\[
\lambda x. u \equiv \lambda y. [y/ x] u,
\]

\[
(\text{letrec } \{ p = \text{body } \} \cup \text{Dec in } u) \equiv (\text{letrec } \{ q = [q/p] \text{body } \} \cup [q/p] \text{Dec in } [q/p] u),
\]

where \( y \) is not free in \( u \), and \( q \) is not free in \( u \), \( \text{body } \), or \( \text{Dec } \), and is not declared in \( \text{Dec} \).

**Evaluation by substitution,** viz., \( \beta \)-conversion:

\[
(\lambda x. u) v \equiv [v/ x] u.
\]

**Declaration distributivity:**

\[
(\text{letrec } \text{Dec in } uv) \equiv (\text{letrec } \text{Dec in } u)(\text{letrec } \text{Dec in } v).
\]

**Declaration elimination:**

\[
(\text{letrec } \text{Dec in } u) \equiv u
\]
providing no variable declared by \( \text{Dec } \) is free in \( u \).

**Variable binding commutativity:**

\[
\lambda x. (\text{letrec } \text{Dec in } u) \equiv (\text{letrec } \text{Dec in } \lambda x. u),
\]
providing \( z \) is neither free nor declared in \( \text{Dec} \).

**Extensionality,** viz., \( \eta \)-conversion:

\[
\lambda x. (u \ x) \equiv u
\]
providing \( u \in L^\alpha \rightarrow \beta \) for some types \( \alpha, \beta \).

A term \( u \) is in **normal form** iff for every application \( (u_1 u_2) \) which is a subterm of \( u \), the operator \( u_1 \) is neither an abstraction nor a block. The following result is well-known for typed \( \lambda \)-calculus (cf. [Barendregt, 1981, Appendix C]), and extends directly to include \text{letrec}.  

\[9\]
Normal Form Theorem: Every term \( u \) is effectively transformable using \( \alpha, \beta \)-conversion, declaration distributivity and the replacement rule to a normal form \( NF(u) \) which is unique up to \( \alpha \)-conversion.

As an immediate consequence, we have in any Cartesian closed model that:

**Normal Form:** \( u \equiv NF(u) \).

The preceding equivalences did not even require assumptions about fixed points. The fixed-point property indeed comes into play in justifying declaration-expanding transformations.

**Declaration expansion:** In any fixed point model,

\[
(\text{letrec } \{ p = \text{body} \} \cup \text{Dec} \text{ in } [p/q]\nu) \equiv (\text{letrec } \{ p = \text{body} \} \cup \text{Dec} \text{ in } [\text{body}/q]\nu).
\]

6. Algebraically Closed Models.

Cartesian closed fixed-point models are still too general to justify even routine transformations of declarations. To establish soundness of such transformations, it is necessary that the fixed point operators be chosen consistently with the structure of the type frame; for example, designated fixed-points should be preserved under isomorphisms induced by reassociating Cartesian products. Frames whose types have some order structure which ensures the existence of least fixed-points can provide a harmonious system of fixed-point operators. One well-known least fixed-point frame is the frame of complete partial orders (cpo's) with continuous functions. However, we need more general classes of least fixed-point frames we call algebraically closed frames.

If \( D \) and \( E \) are partially ordered, then a function \( f : D \rightarrow E \) is monotone iff \( d_1 \sqsubseteq d_2 \) implies \( f(d_1) \sqsubseteq f(d_2) \). If a subset \( Z \subseteq D \) has a least upper bound, \( \sqcup Z \), then \( f : D \rightarrow E \) is continuous along \( Z \) iff it is monotone and \( f(\sqcup Z) = \sqcup \{ f(z) \mid z \in Z \} \).

An algebraically closed (acl) type frame is a Cartesian closed type frame \( \{ D_\alpha \} \) such that

(1) each primitive domain \( D \) is partially ordered with least element \( \bot_D \),

(2) function and product domains of higher type are partially ordered by the inherited pointwise and coordinatewise partial orders,

(3) for all types \( \alpha \) and functions \( f \in D_\alpha \rightarrow \alpha \), the least upper bound \( \sqcup_k f^k(\bot) \) exists, where \( f^0(z) = z \) and \( f^{k+1}(z) = f(f^k(z)) \) (sequences of this form \( \bot, f(\bot), f(f(\bot)), \ldots \) are called algebraic),

(4) for all types \( \alpha \), every function in \( D_\alpha \rightarrow \beta \) is monotone, and is continuous along every algebraic sequence of elements in \( D_\alpha \),

(5) for all types \( \alpha \), the least fixed point operators \( Y_\alpha \) defined by \( Y_\alpha(f) = \sqcup_k f^k(\bot_D) \) are in \( D_{(\alpha \rightarrow \alpha) \rightarrow \alpha} \).

An acl model is a fixed point model with an acl type frame; it is standard iff the constants \( S, K, \text{ tuple, proj}^i \) have the standard interpretation, the constants \( Y_\alpha \) are interpreted as the
corresponding least fixed-point operators \( Y_\alpha \) and for all primitive \( \alpha \), the constants \( \text{diverge}^\alpha \in C_\alpha \) are interpreted as \( \bot_{D_\alpha} \). We let \( \text{diverge}^{\beta \times \alpha} \) abbreviate \( \lambda x.\beta.\text{diverge}^\alpha \) and handle \( \beta \times \alpha \) similarly so that in standard acl models, \( [\text{diverge}^\alpha] = \bot_{D_\alpha} \) for all \( \alpha \).

The following equivalences connect fixed-points between distinct domains and hence depend on choosing fixed-points harmoniously, viz., choosing least fixed-points. We refer to properties like these which are valid for all acl models as acl properties.

Declaration collection:

\[
(\text{letrec } Dec \text{ in } (\text{letrec } Dec' \text{ in } u)) \equiv (\text{letrec } Dec \cup Dec' \text{ in } u)
\]

providing none of the variables declared in \( Dec' \) occurs free or has a distinct declaration in \( Dec \).

Explicit parameterization:

\[
(\text{letrec } \{ p = body \} \cup Dec \text{ in } u) \equiv (\text{letrec } \{ q = \lambda x.[qx/p]body \} \cup [qx/p]Dec \text{ in } [qx/p]u)
\]

providing \( q \) does not appear in \( u, Dec, \) or \( body \), and \( p \) is not declared in \( Dec \).

Declaration denesting:

\[
(\text{letrec } \{ p = \text{letrec } Dec \text{ in } body \} \cup Dec' \text{ in } u) \equiv (\text{letrec } \{ p = body \} \cup Dec \cup Dec' \text{ in } u)
\]

providing none of the variables declared in \( Dec \) is free in \( u \) or \( Dec' \) or declared in \( Dec' \), and \( p \) is not declared in \( Dec \) or \( Dec' \).

A term \( u \in L \) is denested iff neither the body of any variable declaration nor the body of any block in \( u \) contains a declaration. Every term can be effectively transformed into an equivalent denested term using the equivalences above.

The following general induction principle is a basis for induction rules about programs. A predicate \( P \) on a domain \( D_\alpha \) in an acl frame is acl-inclusive iff \( (\forall i \geq 0. P(f^{(i)}(\bot))) \Rightarrow P(Y(f)) \) for all \( f \in D_{\alpha \rightarrow \alpha} \).

Fixed-point Induction: Let \( D_\alpha \) be a domain in an acl frame, \( P \) an inclusive predicate on \( D_\alpha \) and \( f \in D_{\alpha \rightarrow \alpha} \). If \( P(\bot_{D_\alpha}) \land \forall d \in D. (P(d) \Rightarrow P(f(d))) \), then \( P(Y(f)) \) holds.

The equivalences and rules for \( \lambda \)-terms immediately yield rules for PROG phrases; we indicate a few. Let \( E \) represent a finite system of mutual PROG procedure declarations; procedure blocks of the form \( \text{proc } E \text{ do } ProcT \text{ end} \) will be abbreviated as \( E \mid ProcT \) where \( ProcT \) is a procedure term.

Declaration distributivity in PROG:

\[
(E \mid (ProcT_1 \mid ProcT_2)) \equiv (E \mid ProcT_1)(E \mid ProcT_2),
\]

\[
(E \mid ProcT_1^{\text{prog}}; ProcT_2^{\text{prog}}) \equiv ((E \mid ProcT_1^{\text{prog}}); (E \mid ProcT_2^{\text{prog}})),
\]

etc.
Note that declaration distributivity depends crucially on the fact that $E$ denotes a set of procedure declarations, whose meaning is necessarily store-independent. So the declaration distributivity rule is valid despite the possible side-effects on the store between evaluations of different copies of $E$. In contrast, distributivity fails for basic (viz., let-) declarations because the value bound to an identifier by a basic declaration depends on the store "at declaration time". This contrast was reflected in the use of constants in translating basic blocks, compared to the \texttt{letrec} construct used to translate procedure blocks.

**Variable binding commutativity in \texttt{PROG}:**

\[
(E \mid \text{let } x \text{ be } \texttt{Base}E \text{ in } \texttt{Proc}T^{\text{prog}} \text{ tel } ) \equiv \text{let } x \text{ be } \texttt{Base}E \text{ in } (E \mid \texttt{Proc}T^{\text{prog}}) \text{ tel,}
\]

\[
(E \mid \text{begin new } y \text{ in } \texttt{Proc}T^{\text{prog}} \text{ end}) \equiv \text{begin new } y \text{ in } (E \mid \texttt{Proc}T^{\text{prog}}) \text{ end}
\]

providing $x, y$ do not occur free in $E$.

**Fixed-Point Induction for Approximation in \texttt{PROG}:** Let $p$ be an identifier and $\texttt{Proc}T$ a \texttt{PROG} term, both of the same type, such that $p$ is not free in $\texttt{Proc}T_2$. Then

\[
[\text{diverge/p}]\texttt{Proc}T_1 \subseteq \texttt{Proc}T_2,
\]

\[
(\texttt{Proc}T_1 \subseteq \texttt{Proc}T_2) \Rightarrow (([\texttt{Proc}T/p]\texttt{Proc}T_1 \subseteq \texttt{Proc}T_2)
\]

\[
\texttt{proc } p \Leftarrow \texttt{Proc}T \texttt{ do } \texttt{Proc}T_1 \texttt{ end } \subseteq \texttt{Proc}T_2
\]

7. **The Equivalence of Fixed-Point and Computational Semantics.**

The most fundamental acl property is that every term in \texttt{L} can be understood as a limit of finite \texttt{letrec}-free terms (in normal form if desired) which approximate the given term. These finite approximations are obtained by repeatedly "unwinding" the \texttt{letrec} declarations using the declaration expansion rule. This provides an effective computational rule for simulating the effects of \texttt{letrec}'s and the corresponding procedure declarations in \texttt{PROG}. It also shows that two procedures which expand to the same infinite declaration-free procedure are equivalent in all acl models for \texttt{PROG}, independent of the meaning of any \texttt{PROG} constructs.

The original ALGOL 60 report [Naur, ct.al., 1963] gave a "copy-rule" semantics for the language. The copy-rule can be understood as a particular computational strategy for generating the infinite expansion of a command. It follows that another acl property is that fixed-point and copy-rule semantics (appropriately extended to \texttt{letrec}-terms and \texttt{PROG} commands with free variables) assign the same meanings to terms [ct., Danm 82]. This confirms that our choice of denotational "fixed-point" semantics is consistent with the usual operational understanding based on the copy-rule. For the development here, however, we have no need of these facts, and so we omit further explanation.

Thus procedure declarations of ALGOL-like languages are entirely explained by acl semantics for \texttt{L}. On this basis we assert that the typed $\lambda$-calculus is the \textit{true mathematical syntaz} for these languages. For example, several of the language design principles of [Tennent, 81] can be recognized as proposing that syntactic restrictions of programs to subsets of \texttt{L} be removed.

12
8. Store Semantics of PROG.

Particular instances of ALGOL-like languages are determined by their types and the interpretations of their constants. Properties related to stores and side-effects appear only, at this level. We now specify the domains and constants which determine PROG.

Store Frames: Given an infinite set Loc (of locations) and a set Int (of storable values) we define the domains

\[ D_{loc} = \text{def} \ Loc \cup \{ \bot_{loc} \}, \quad D_{int} = \text{def} \ Int \cup \{ \bot_{int} \} \]

to be the flat cpo's.

For sets \( A, B \), let \( A^B = \text{def} \) the set of all total functions from \( B \) to \( A \). For the other primitive domains, we select some subset, \( \text{Store} \subseteq \text{Int}^{Loc} \). \( \text{Store} \) must be closed under finite patching. (Note that no store maps a location to \( \bot_{int} \). There is no need to introduce such "partial" stores in modeling the behavior of sequential languages like PROG.) Then

\[ D_{\text{intexp}} \subseteq (D_{\text{int}})^{\text{Store}}, D_{\text{locexp}} \subseteq (D_{\text{loc}})^{\text{Store}}, D_{\text{prog}} \subseteq (\mathcal{P}(\text{Store}))^{\text{Store}}. \]

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

A Store model is any standard acl model with the above five primitive types such that there are elements in the domains of the frame which interpret the constants required in the translation of PROG to \( L \) as specified below. These constants are: If, Mkexp, Cont, Update, diverge, Ifprog, Seq, Choice, Dint, Dloc, and New.

The constant \( \text{If}_{\alpha,\beta} \) for basic types \( \alpha, \beta \) has type \( \alpha \rightarrow \beta \rightarrow \beta \rightarrow \beta \). A store model interprets

\[
[\text{If}_{\alpha,\beta}]d_{1}^{\alpha}d_{2}^{\beta}d_{3}^{\beta} = \begin{cases} 
\bot_{\beta} & \text{if } d_{1} = \bot_{\alpha} \text{ or } d_{2} = \bot_{\beta}, \\
d_{3} & \text{if } d_{1} = \beta_{1} \\
d_{4} & \text{otherwise.}
\end{cases}
\]

Any first order function \( f \) of type \( \delta = \text{int}^k \rightarrow \text{int} \) can be coerced into a mapping \( \text{Mkexp}_{\delta}(f) \) taking as arguments functions from stores to \( \text{int} \). Namely,

\[
\text{Mkexp}_{\delta} f_{1}^{\text{intexp}} \ldots d_{k}^{\text{intexp}} = f(d_{1}(s), \ldots, d_{k}(s))
\]

for any store \( s \). The constant \( \text{Mkexp}_{\delta} \) of type \( \delta \rightarrow (\text{intexp} \rightarrow \ldots \rightarrow \text{intexp}) \) is interpreted as the coercer \( \text{Mkexp}_{\delta} \).

The constant \( \text{Cont} \) of type \( \text{locexp} \rightarrow \text{intexp} \) is defined in store models so that

\[
[\text{Cont}]d_{1}^{\text{locexp}}s = \begin{cases} 
\text{if } d(s) \neq \bot_{\text{loc}}, \\
\bot_{\text{int}} & \text{otherwise.}
\end{cases}
\]

For assignments, the constant \( \text{Update} \) of type \( \text{locexp} \rightarrow \text{intexp} \rightarrow \text{prog} \):

\[
[\text{Update}]d_{1}^{\text{locexp}}d_{2}^{\text{intexp}}s = \begin{cases} 
\{ s[d_{2}(s)/d_{1}(s)] \} & \text{if } d_{1}(s), d_{2}(s) \neq \bot, \\
\emptyset & \text{otherwise.}
\end{cases}
\]

13
For conditional commands, if \( \text{prog}_\alpha \) of type \( \alpha \cdot \text{exp} \to \alpha \cdot \text{exp} \to \text{prog} \to \text{prog} \to \text{prog} \):

\[
[\text{if} \text{prog}_\alpha] d_1^{\alpha \cdot \text{exp}} d_2^{\alpha \cdot \text{exp}} d_3^{\text{prog}} d_4^{\text{prog}} s = \begin{cases} 
\emptyset & \text{if } d_1(s) = \bot_\alpha \text{ or } d_2(s) = \bot_\alpha, \\
\{d_3(s)\} & \text{if } d_1(s) = d_2(s) \neq \bot, \\
\{d_4(s)\} & \text{otherwise.}
\end{cases}
\]

Command constructors Choice, Seq of type \( \text{prog} \to \text{prog} \to \text{prog} \):

\[
[\text{Seq}] d_1^{\text{prog}} d_2^{\text{prog}} s = \cup \{ d_2(s') \mid s' \in d_1(s) \},
\]

\[
[\text{Choice}] d_1^{\text{prog}} d_2^{\text{prog}} s = d_1(s) \cup d_2(s).
\]

For let blocks, Dint of type \( \text{int} \to \text{prog} \to \text{intexp} \to \text{prog} \):

\[
[D\text{int}] d_1^{\text{int} \to \text{prog}} d_2^{\text{intexp}} s = \begin{cases} 
(d_1(d_2(s)))(s) & \text{if } d_2(s) \neq \bot_{\text{int}}, \\
\emptyset & \text{otherwise.}
\end{cases}
\]

The constant Dloc is defined similarly.

The semantics of the constant New of type \( \text{loc} \to \text{prog} \to \text{prog} \) is handled in the next section.

9. Domains for the Local Storage Discipline.

To explain the semantics of New, we must define the notion of covering. For primitive types this is fairly straightforward.

Let \( L \) be a subset of \( \text{Loc} \). Two stores \( s, t \) agree on \( L \), written \( s =_L t \), iff \( \forall l \in L. s(l) = t(l) \).

Similarly, two sets \( S, T \subseteq P(\text{Stores}) \) agree on \( L \) if there is a bijection \( f : S \to T \) such that \( \forall s \in S. s =_L f(s) \).

Define the unary predicate \( \text{Access}_{L}^\alpha(d) \) on \( D_\alpha \) by induction on \( \alpha \) according to the rules below. If \( \text{Access}_{L}^\alpha(d) \) holds, we say that \( d \) accesses only the locations in \( L \).

1. \( \text{Access}_{\text{loc}}^L(l) \) iff \( l \in L \cup \{ \bot_{\text{loc}} \} \),
2. \( \text{Access}_{\text{int}}^L(d) \equiv \text{true} \),
3. \( \text{Access}_{\text{prog}}^L(\pi) \) iff \( \forall s, t \in \text{Store}. (s =_L t \Rightarrow \pi(s) =_L \pi(t)) \land (t \in \pi(s) \Rightarrow s =_{\text{Loc} \setminus L} t) \),
4. \( \text{Access}_{\text{intexp}}^L(\tau) \) iff \( \forall s, t \in \text{Store}. s =_L t \Rightarrow \tau(s) = \tau(t) \),
5. \( \text{Access}_{\text{locexp}}^L(\sigma) \) iff \( \forall s, t \in \text{Store}. s =_L t \Rightarrow \sigma(s) = \sigma(t) \in L \cup \{ \bot_{\text{loc}} \} \),
6. \( \text{Access}_{D_L}^L(f) \) iff \( \forall d \in D_L, L' \subseteq L. \text{Access}_{D_L}^L(d) \Rightarrow \text{Access}_{D_L}^L(f(d)) \),
7. \( \text{Access}_{D_L}^L(d_1, d_2) \) iff \( (\text{Access}_{D_L}^L(d_1) \land \text{Access}_{D_L}^L(d_2)) \).

For higher-type objects, we also need a notion of uniformity with respect to "new" locations.
Definition. Let $\mu : \text{Loc} \to \text{Loc}$ be a bijection; extend $\mu$ to $D_{\text{Loc}}$ so that $\mu(\bot) = \bot$. Let $\mu_{\text{Store}} : \text{Store} \to \text{Store}$ be the bijection defined by the rule

$$\mu_{\text{Store}}(s) = s \circ \mu^{-1}$$

where $\circ$ denotes functional composition, and let $\mu_{\mathcal{P}(\text{Store})} : \mathcal{P}(\text{Store}) \to \mathcal{P}(\text{Store})$ be the bijection defined by applying $\mu_{\text{Store}}$ elementwise. Define a bijection $\mu_{\alpha} : D_{\alpha} \to D_{\alpha}$ by the rules:

1. $\mu_{\text{Loc}} = \mu$,
2. $\mu_{\text{Int}}(d_{\text{Int}}) = d$,
3. $\mu_{\text{Prog}}(\pi) = \mu_{\mathcal{P}(\text{Store})} \circ \pi \circ \mu_{\text{Store}}^{-1}$,
4. $\mu_{\text{IntExp}}(\tau) = \mu_{\text{Int}} \circ \tau \circ \mu_{\text{Store}}^{-1}$,
5. $\mu_{\text{LocExp}}(\sigma) = \mu_{\text{Loc}} \circ \sigma \circ \mu_{\text{Store}}^{-1}$,
6. $\mu_{\beta \to \gamma}(f) = \mu_{\gamma} \circ f \circ \mu_{\beta}^{-1}$,
7. $\mu_{\beta \times \gamma}(d_1, d_2) = (\mu_{\beta}(d_1), \mu_{\gamma}(d_2))$.

Note that $(\mu^{-1})_{\alpha} = (\mu_{\alpha})^{-1}$, so the notation $\mu_{\alpha}^{-1}$ is unambiguous. A bijection $\mu : \text{Loc} \to \text{Loc}$ fixes $L$ if $\mu(l) = l$ for all $l \in L$. Define the unary predicate $\text{Unif}^L_{\alpha}$ on $D_{\alpha}$ by the rule:

$$\text{Unif}^L_{\alpha}(d) \text{ iff } \forall \mu \text{ fixing } L. \mu_{\alpha}(d) = d.$$ 

If $\text{Unif}^L_{\alpha}(d)$ holds, we say that $d$ is uniform on $L$.

We henceforth omit subscripts $\alpha$ when they are clear from context.

Definition. A set $L \subseteq \text{Loc}$ covers an element $d$ iff $\text{Access}^L(d) \wedge \text{Unif}^L(d)$.

Note that for primitive types, $\text{Access}^L(d)$ iff $L$ covers $d$. (We remark that covering is a logical relation in the sense of [Plotkin, 80; Statman, 82].)

Some key properties of covering are:

1. If $L$ covers $d$, then $L \cup L'$ covers $d$,
2. If $L$ covers $f^a \to \beta, d^\beta$, then $L$ covers $(f \cdot d)$,
3. If $L$ covers all $d \in Z \subseteq D_{\alpha}$ and $\bigcup Z$ exists, then $L$ covers $\bigcup Z$,
4. The functions $K, S, Y, \text{tuple}, \text{proj}^i$ have empty covers.

These facts immediately imply that for any environment $e$ and term $u \in L$, the element $[u]e$ is covered by a union of covers for $[e]$ and $e(x)$ for all the constants $e$ and free variables $x$ in $u$.

It not hard to show that all the constants other than $\text{New}$ are continuous and have empty covers. To ensure that $\text{New}$ is interpretable, we impose a further condition on store models:

Covering Restriction: Every element has a finite cover.
Under the covering restriction it follows that covers are closed under (infinite) intersection, so that every element has a minimum cover which is called its support.

**Definition.** A function \( \text{Select} : \mathcal{P}(\text{Loc}) \to \text{Loc} \) will be called a selection function iff \( \text{Select}(L) \notin L \) for all finite sets \( L \subseteq \text{Loc} \). (Selection functions exist because \( \text{Loc} \) is infinite.) For each selection function \( \text{Select} \), let \( \text{NewSelect} : D_{\text{loc-prog}} \to D_{\text{prog}} \) be defined by

\[
\text{NewSelect} \rho = \text{def} \ [\text{Tr}(\text{let } x \text{ be cont}(y) \text{ in } y := a_0; p(y); y := x \text{ tel})] \epsilon,
\]

where \( \epsilon(y) = \text{Select}(\text{Support}(\rho)) \), \( \epsilon(\rho) = \rho \).

**Lemma.** Let \( \text{Select}_1, \text{Select}_2 \) be selection functions. Then

(a) \( \text{NewSelect}_1 = \text{NewSelect}_2 \),

(b) \( \text{NewSelect}_1 \) is continuous along algebraic sequences and has empty support.

It follows that if we take any selection function \( \text{Select} \), then \( \text{NewSelect} \) unambiguously determines a meaning for \( \text{New} \) in store models, and we require this meaning to be in \( D_{(\text{loc-prog})-\text{prog}} \).

To demonstrate rigorously that the theory of \( \text{PROG} \) is consistent, it is sufficient to show that store models exist. We now indicate how to construct one.

For primitive types \( \alpha \), define partially ordered sets \( D_{\alpha}, E_{\alpha} \) as follows:

1. \( E_{\text{loc}} = D_{\text{loc}} \),
2. \( E_{\text{int}} = D_{\text{int}} \),
3. \( E_{\text{prog}} = (\mathcal{P}(\text{Store}))^{\text{Store}} \),
4. \( E_{\text{loc-exp}} = (D_{\text{loc}})^{\text{Store}} \),
5. \( E_{\text{int-exp}} = (D_{\text{int}})^{\text{Store}} \),
6. \( D_{\alpha} = \{ d \in E_{\alpha} \mid d \text{ has a finite cover} \} \).

For partially ordered sets \( D, E \), let \( D \to E \) be the set of functions from \( D \) to \( E \) which are continuous, i.e., preserve all least upper bounds which exist in \( D \). For higher types,

7. \( E_{\beta \to \gamma} = D_{\beta} \to D_{\gamma} \),
8. \( E_{\beta \times \gamma} = D_{\beta} \times D_{\gamma} \),
9. \( D_{\alpha} = \{ d \in E_{\alpha} \mid d \text{ has a finite cover} \} \), (i.e., same as (6) with \( \alpha \) any higher type).

It is not hard to verify that \( \{ D_{\alpha} \} \) is an acl frame which provides a store model.

We can further justify our store model semantics by demonstrating that it coincides with familiar operational semantics based either on stack implementations or on copy-rule semantics in which \( \textbf{new} \) declarations are explained through renaming of local identifiers (cf. [Langmaack and Olderog, 80; Olderog, 82]). These results will be developed in our full paper.
10. Reasoning about Support.

Because all the PROG constants have empty support, a cover for (the meaning of) any PROG phrase is easily characterized: take the union of covers for the free procedure and location identifiers. In particular, if the phrase has no global calls – so the only free identifiers are of location type – then a cover is available by inspection: the union of the (denotations of) the free location variables in the phrase. This follows because the support of any location \( l \in \text{Loc} \) is the singleton \( \{ l \} \). (In general, the support of a command may be strictly smaller than the supports of its free identifiers, e.g., \( x := \text{cont}(x) \) has empty support.)

These observations are the basis for a variety of axioms for program correctness suggested in [Meyer, 83; Trakhtenbrot, Halpern, and Meyer, 83; Halpern, 84].

11. Critique of PROG.

PROG fails as an example of satisfactory language design in many ways, even with respect to the limited set of features it is intended to model. For example,

1. there are no Boolean types,
2. there is no \textbf{while} command or other structured control statement,
3. only one identifier at a time can be declared in a basic declaration,
4. there are no \textbf{let} blocks of basic expression type.
5. Conditionals are not uniformly available at all types [cf. Reynolds, 1981a].

However, these pragmatic features are all inessential for our purposes since they can be simulated at the level of uninterpreted program schemes by commands already in PROG, i.e., each of the constants corresponding to these constructs is directly \( \lambda \)-definable in terms of the constants already introduced. Therefore they raise no semantical or proof-theoretical issues beyond those already treated.

An important feature in actual ALGOL-like languages but missing from PROG is that locations can be storable subject to restrictions (as in ALGOL 68) to ensure local storage discipline is preserved. Another extension improving uniformity involves introducing \( \alpha \text{-exp} \) types for \( \alpha \) other than \textbf{int} and \textbf{loc} (with a corresponding block \( \text{let } z^\alpha \text{ be } \text{ProcT}^{\alpha \cdot \text{exp}} \text{ in } \text{ProcT}^{\beta \cdot \text{exp} \cdot \text{tel}} \)). 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.

References.


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


Cambridge, Massachusetts
February 12, 1984