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 – independent of the interpretation of program constructs – by continuous models for* λ*-calculus. However, the usual (cpo) 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.*

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 on block entry. For object-oriented languages like LISP or CLU requiring heap storage, new locations (aka program variables) are usually allocated from a linked list of free locations.

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

In both cases, there is a simple idea behind local variables in blocks: execution of a block **begin new** *z* **in** *Body* **end** causes allocation of a "new" storage location denoted by the identifier *z* which is used in the body of the block. In ALGOL-like languages obeying stack discipline, the location is moreover 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.

There is a problem in explaining this apparently simple idea behind local storage, namely, what is a "new" location? 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 they directly reflect the bookkeeping mechanisms used in implementations. Thus, new storage allocation is typically modeled denotationally 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 x be an identifier of location (aka reference) type, and let p be an identifier of parameterless pure procedure type. Then, the block

```
begin new x in
  x := 0; p;
  if cont(x) = 0 then skip else diverge fi
end
```

ought to be equivalent to **skip** since the "new" location allocated for x should not be affected by the call to p. But if p happens to denote the program which assigns a nonzero value to some location l, and this block is executed on a store in which location l happens to be designated as the first free 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 locations which *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 behave like **skip** in any program context in which the global identifier 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 reverification 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 *semantically* when a location is active or free with respect to a procedure. In general, we define how a set of locations *covers* a procedure of finite type, by induction on types. The locations outside the cover of a procedure are the free ones for it. 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 π is a store to store mapping whose only cover is the set of *all* locations – π might be the denotation of a procedure which "sweeps" memory searching for an untagged location. Now π can be expressed as the limit of a sequence of approximating mappings π_i which only sweep the first i locations. Since storage is infinite but a finite number of locations cover π_i, there is always a location free to allocate for a block whose body behaves like π_i. On the other hand, allocating new storage for π yields an 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" covers force us to face the discontinuity of storage overflow. We would like to rule out such objects, especially in view of the fact that *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 π_i) which depend on only finitely many locations, the usual requirement that semantical domains be *complete partial orders* (cpo's) which are closed under taking least upper bounds of all increasing chains forces us to admit programs (like π) with infinite covers [Stoy 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 *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 *store models*: systems of algebraically closed partial orders containing only elements with proper covers but including enough elements to interpret all the programming constructs of ALGOL-like languages.

Store models justify all the intended properties of new-declarations. For example, in store models the block mentioned above with global call to p is indeed equivalent to **skip** in *all* environments. Another illustrative equivalence is:

begin new x **in**
  if $x = y$ **then** Cmd_1 **else** Cmd_2 **fi end** $\equiv$
    $y :=$ **cont**(y); Cmd_2.

(The "useless" assignment to y appears in case y denotes the divergent ($\perp$) location.)

2. 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, 83]:

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

(2) Calling is *by-name*. (Calls by-value, etc., are treated as syntactic sugar.)

(3) Higher-order procedures of all *finite* types (in ALGOL 68 jargon, *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 PROG.

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, **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. We do allow explicit equality testing between locations, "$x^{\mathbf{loc}} = y^{\mathbf{loc}}$", in addition to the usual test of equality between storable values, "$\mathbf{a} = \mathbf{f}(\mathbf{cont}(y^{\mathbf{loc}}))$". Expressions which evaluate to locations are allowed, as in the "conditional variable" expression on the lefthand side of the assignment command

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

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.,

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

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

$$\mathbf{let}\,x^{\mathbf{int}}\,\mathbf{be}\,\mathit{IntE}\,\mathbf{in}\,\mathit{Cmd}\,\mathbf{tel}\,,$$
$$\mathbf{let}\,y^{\mathbf{loc}}\,\mathbf{be}\,\mathit{LocE}\,\mathbf{in}\,\mathit{Cmd}\,\mathbf{tel}\,,$$
$$\mathbf{begin}\,\mathbf{new}\,y^{\mathbf{loc}}\,\mathbf{in}\,\mathit{Cmd}\,\mathbf{end}\,.$$

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

3. Syntax-Preserving Translation to λ-Calculus. We 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 as in [Damm and Fehr, 1980; Damm, 1982; cf. Landin, 65], and

(2) assignment of semantics to the λ-calculus in a standard referentially transparent way [Barendregt, 81; Meyer, 82].

Our λ-calculus is chosen so that its constants correspond to program constructors, its binding operations, letrec and λ, correspond to program declarations and procedure abstraction, and its types are the *same* as those of the programming language. In fact, 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,

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

This recursive declaration of p binds occurrences of p in both the declaration and the block bodies. Procedure declarations in this way inherit the *static scoping rules* of λ-calculus.

Basic blocks are handled with constants and λ's, e.g.,

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

where **Dint** is a constant of type (**int** → **prog**) → **intexp** → **prog**. Note that the binding effect of the block on $x^{\mathbf{int}}$ is reflected in the binding effect of λx on $Tr(\mathit{Cmd})$, namely, the declaration binds x in *Cmd*, but does not bind x in *IntE*, in contrast to the case for procedure declarations. Similarly,

$$Tr(\mathbf{begin}\,\mathbf{new}\,x\,\mathbf{in}\,\mathit{Cmd}\,\mathbf{end}) =_{def}$$
$$\mathbf{New}\,(\lambda x.Tr(\mathit{Cmd}))$$

where New is a special constant of type (loc $\rightarrow$ prog) $\rightarrow$ prog. The semantics of New will be defined so that *Cmd* runs in an environment in which x is bound to some location outside a cover of *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 *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.,

$$
\begin{aligned}
&Tr(\mathbf{if}\, Term_1^\alpha = Term_2^\alpha \\
&\quad \mathbf{then}\, Term_3^\beta\, \mathbf{else}\, Term_4^\beta\, \mathbf{fi}) =_{def} \\
&\quad (\mathbf{If}_{\alpha,\beta}\,(Tr(Term_1^\alpha))\,(Tr(Term_2^\alpha)) \\
&\qquad (Tr(Term_3^\beta))\,(Tr(Term_4^\beta))), \\
&Tr(\mathbf{cont}(LocE)) =_{def} (\mathbf{Cont}(Tr(LocE))), \\
&Tr(Cmd\, \mathbf{return}\, IntE) =_{def} \\
&\quad (\mathbf{Return}(Tr(Cmd))Tr(IntE))), \\
&Tr(LocE := IntE) =_{def} \\
&\quad (\mathbf{Update}\,(Tr(LocE))\,(Tr(IntE))), \\
&Tr(Cmd_1;\, Cmd_2) =_{def} \\
&\quad (\mathbf{Seq}\,(Tr(Cmd_1))\,(Tr(Cmd_2))), \\
&\qquad \text{etc.}
\end{aligned}
$$

The principal consequence of this syntax-preserving translation is that all the properties of *procedure* declarations in 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 letrec-λ-calculus – which have nothing at all to do with side-effects. Before elaborating this point, we review the properties of the letrec-calculus.

4. Typed Lambda 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 α, β are type expressions, then so are $\alpha \rightarrow \beta$ and $\alpha \times \beta$. With each type expression α we associate a (possibly empty) set of constants C_α, disjoint from C_β for $\alpha \neq \beta$. With each α we also associate an infinite set of variables X_α, disjoint from X_β for $\alpha \neq \beta$. We use the notation x^α 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^α, the terms of letrec-λ-calculus of type α, by induction.

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

(2) Application: If $u \in L^{\alpha\rightarrow\beta}$, $v \in L^\alpha$, then

$$(u\, v) \in L^\beta.$$

(3) Abstraction: If $x \in X_\alpha, u \in L^\beta$, then

$$\lambda x.u \in L^{\alpha\rightarrow\beta}.$$

(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

$$(\mathbf{letrec}\; x_1 = u_1\; \mathbf{and} \cdots \mathbf{and}\; x_k = u_k\; \mathbf{in}\; v) \in L^\beta.$$

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 operation $\rightarrow$ associates to the right in compound type expressions. Thus uvw abbreviates $((u\,v)\,w)$ while $\alpha \rightarrow \beta \rightarrow \gamma$ abbreviates $(\alpha \rightarrow (\beta \rightarrow \gamma))$. We let $[v/x]u$ denote the result of substituting the term v for free occurrences of x 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].

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 λ-calculus (cf.[Barendregt, 1981, Appendix C]), and extends directly to include letrec.

Normal Form: Every term u is effectively transformable using (cf. §5) α, β-conversion, declaration distributivity and the replacement rule to a normal form $NF(u)$ which is unique up to α-conversion.

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 $\langle d_1, \ldots, d_n \rangle$ of elements $d_i \in D_i$. Let $tuple_{D_1,\ldots,D_n} : D_1 \rightarrow \cdots \rightarrow D_n \rightarrow (D_1 \times \cdots \times D_n)$ be defined by:

$$tuple\, d_1 \cdots d_n = \langle d_1, \ldots, d_n \rangle,$$

and let $proj^i_{D_1,\ldots,D_n} : (D_1 \times \cdots \times D_n) \rightarrow D_i$ 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 α, such that

(1) $D_{\alpha \rightarrow \beta}$ consists of some nonempty family of functions from D_α to D_β 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 α, β, γ 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) $tuple_{D_{\alpha_1},\ldots,D_{\alpha_n}} \in D_{\alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow (\alpha_1 \times \cdots \times \alpha_n)}$, and similarly $proj^i_{D_{\alpha_1} \ldots D_{\alpha_n}} \in D_{(\alpha_1 \times \cdots \times \alpha_n) \rightarrow \alpha_i}$.

An *environment* for a type-frame D is a mapping $e : X \rightarrow D = \bigcup_\alpha D_\alpha$ which respects types, i. e., $e(x^\alpha) \in D_\alpha$. Given an environment e, let $e[d/x]$ denote the environment which differs from e only at x, and $(e[d/x])(x) = d$. Let $e[d_1/p_1, \ldots, d_{k+1}/p_{k+1}]$ abbreviate $e[d_1/p_1, \ldots, d_k/p_k][d_{k+1}/p_{k+1}]$. (We define the "patch", $f[b/a]$, of any function $f : A \rightarrow B$, at $a \in A$, by $b \in B$ similarly.) Let $\mathbf{Env}_D$ 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 $[\![\,]\!]_0 : C \rightarrow D$ which respects types. The model is *standard* iff the constant symbols $\mathbf{S}_{\alpha,\beta,\gamma} \in C_{(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))}$ and $\mathbf{K}_{\alpha,\beta} \in C_{\alpha \rightarrow (\beta \rightarrow \alpha)}$ are interpreted as the corresponding S and K functions, and similarly for the constants **tuple** and $\mathbf{proj}^i$. Let $L_1 \subseteq L$ be the usual typed λ-calculus (without **letrec**). The justification for this peculiar definition is that for any Cartesian closed model D, there exists a unique mapping $[\![\,]\!]_D : L_1 \rightarrow \mathbf{Env}_D \rightarrow D$ which respects types such that

(a) $[\![\mathbf{c}]\!]_D e = [\![\mathbf{c}]\!]_0$,
(b) $[\![x]\!]_D e = e(x)$,
(c) $[\![(uv)]\!]_D e = ([\![u]\!]_D e)([\![v]\!]_D e)$.
(d) for all $d \in D_\alpha$, $([\![\lambda x^\alpha.u]\!]_D e)d = [\![u]\!]_D(e[d/x])$.

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 α. 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 $\mathbf{Y}_\alpha \in C_{(\alpha \rightarrow \alpha) \rightarrow \alpha}$ are interpreted as fixed point operators Y_α.

Let $\lambda \langle x_1, \ldots, x_n \rangle . u$ abbreviate

$$\lambda z.([(\mathbf{proj}^1\, z)/x_1] \ldots [(\mathbf{proj}^n\, z)/x_n] u)$$

for z not free in u.

For any Cartesian closed fixed-point model D, there exists a unique mapping $[\![\,]\!]_D : L \rightarrow \mathbf{Env}_D \rightarrow D$ which respects types, satisfies (a–d) above, and such that

(e) $[\![\mathbf{letrec}\, p_1 = u_1\, \mathbf{and} \ldots \mathbf{and}\, p_n = u_n\, \mathbf{in}\, v]\!]_D e$

$$=_{def} [\![v]\!]_D(e[d_1/p_1, \ldots, d_n/p_n])$$

where $\langle d_1, \ldots, d_n \rangle =$

$$[\![(\mathbf{Y}\, \lambda \langle p_1, \ldots p_n \rangle . \mathbf{tuple}\, u_1 \cdots u_n)]\!]_D e.$$

Terms u and v are *equivalent* for some model D, written $u \equiv_D v$, iff $[\![u]\!]_D = [\![v]\!]_D$. If $\mathcal{M}$ is a class of models, u and v are $\mathcal{M}$-*equivalent* iff $u \equiv_D v$ for all models $D \in \mathcal{M}$.

We abbreviate a mutual procedure declaration of the form (**letrec** $p_1 = u_1$ **and** $\cdots$ **and** $p_n =$

u_n in v) by (**letrec** Dec **in** v), where $Dec = \{p_1 = u_1, \ldots, p_n = u_n\}$.

The following fundamental inference rule verifies the referential transparency of L. It is sound in any Cartesian closed model when we merely regard **letrec** Dec **in** v as an abbreviation for $(\lambda\langle p_1, \ldots, p_n\rangle.v)(\mathbf{c}\,(\lambda\langle p_1, \ldots p_n\rangle.\mathbf{tuple}\, u_1 \cdots u_n))$ 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.

Variable renaming, viz., α-conversion:

$$(i) \quad \lambda x.u \equiv \lambda y.[y/x]u,$$

where y is not free in u, and

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

where q is not free in u, $body$, or Dec, and is not declared in Dec.

Evaluation by substitution, viz., β-conversion:

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

Declaration distributivity:

(**letrec** Dec **in** uv) $\equiv$
(**letrec** Dec **in** u)(**letrec** Dec **in** v).

Declaration elimination:

(**letrec** Dec **in** u) $\equiv u$

providing no variable declared by Dec is free in u.

Variable binding commutativity:

$\lambda x.$(**letrec** Dec **in** u) $\equiv$ (**letrec** Dec **in** $\lambda x.u$),

providing x is neither free nor declared in Dec.

Extensionality, viz., η-conversion:

$$\lambda x.(u\, x) \equiv u$$

providing $u \in L^{\alpha \to \beta}$ for some types $\alpha; \beta$.

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

The fixed-point property justifies declaration-expanding transformations.

Declaration expansion:

(**letrec** $\{p = body\} \cup Dec$ **in** $[p/q]v$) $\equiv$
(**letrec** $\{p = body\} \cup Dec$ **in** $[body/q]v$).

6. Algebraically Closed Models. Cartesian closed fixed-point models are still too general even to justify 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 $f : D \to E$ 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, $\bigsqcup Z$, then $f : D \to E$ is ***continuous along*** Z iff it is monotone and $f(\bigsqcup Z) = \bigsqcup\{f(x) \mid x \in Z\}$.

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 α and functions $f \in D_{\alpha\to\alpha}$, the least upper bound $\bigsqcup_k f^k(\bot)$ exists, where $f^0(x) = x$ and $f^{k+1}(x) = f(f^k(x))$ (sequences of this form $\bot, f(\bot), f(f(\bot)), \ldots$ are called *algebraic*),

(4) for all types α, every function in $D_{\alpha\to\beta}$ is continuous along every algebraic sequence of elements in D_α,

(5) for all types α, the *least fixed point operators* Y_α defined by $Y_\alpha(f) = \bigsqcup_k f^k(\perp_{D_\alpha})$ are in $D_{(\alpha\to\alpha)\to\alpha}$.

An *(acl) model* is a fixed point model with an acl type frame; it is standard iff the constants **S**, **K**, **tuple**, **proj**^{i} have the standard interpretation, the constants $\mathbf{Y}_\alpha$ are interpreted as the corresponding *least* fixed-point operators Y_α, and for all *primitive* α, the constants **diverge**$^\alpha \in C_\alpha$ are interpreted as $\perp_{D_\alpha}$. We let **diverge**$^{\beta\to\alpha}$ abbreviate λx^β.**diverge**$^\alpha$ and handle $\beta \times \alpha$ similarly so that in standard acl models, $[\![\mathbf{diverge}^\alpha]\!] = \perp_{D_\alpha}$ for all α.

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:

$$(\mathbf{letrec}\ Dec\ \mathbf{in}\ (\mathbf{letrec}\ Dec'\ \mathbf{in}\ u)) \equiv (\mathbf{letrec}\ Dec \cup Dec'\ \mathbf{in}\ u)$$

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

Explicit parameterization:

$(\mathbf{letrec}\ \{p = body\} \cup Dec\ \mathbf{in}\ u) \equiv$

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

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

Declaration denesting:

$(\mathbf{letrec}\ \{p = \mathbf{letrec}\ Dec\ \mathbf{in}\ body\} \cup Dec'\ \mathbf{in}\ u) \equiv$

$(\mathbf{letrec}\ \{p = body\} \cup Dec \cup Dec'\ \mathbf{in}\ u)$

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_α in an acl frame is *acl-inclusive* iff $(\forall i \geq 0.\ P(f^{(i)}(\perp))) \Rightarrow P(Y(f))$ for all $f \in D_{\alpha\to\alpha}$.

Fixed-point Induction: Let D_α be a domain in an acl frame, P be an inclusive predicate on D_α and $f \in D_{\alpha\to\alpha}$. If

$$P(\perp_{D_\alpha}) \wedge \forall d \in D.\ (P(d) \Rightarrow P(f(d))),$$

then $P(Y(f))$ holds.

The equivalances and rules for λ-terms immediately yield rules for PROG phrases; we indicate a few. Let E (possibly primed or subscripted) represent a finite system of mutual PROG procedure declarations; procedure blocks of the form **proc** E **do** $ProcT$ **end** will be abbreviated as $E \mid ProcT$ where $ProcT$ is a procedure term.

Declaration distributivity in PROG:

$$(E \mid (ProcT_1\, ProcT_2)) \equiv (E \mid ProcT_1)(E \mid ProcT_2),$$
$$(E \mid ProcT_1^{\mathrm{prog}};ProcT_2^{\mathrm{prog}}) \equiv ((E \mid ProcT_1^{\mathrm{prog}});(E \mid ProcT_2^{\mathrm{prog}})),$$

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 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 **letrec** construct used to translate procedure blocks.

Variable binding commutativity in PROG:

$$(E \mid \mathbf{let}\ x \Leftarrow BasE\ \mathbf{in}\ ProcT^{\mathrm{prog}}\ \mathbf{tel}\,) \equiv \mathbf{let}\ x \Leftarrow BasE\ \mathbf{in}\ (E \mid ProcT^{\mathrm{prog}})\ \mathbf{tel},$$
$$(E \mid \mathbf{begin\ new}\ y\ \mathbf{in}\ ProcT^{\mathrm{prog}}\ \mathbf{end}) \equiv \mathbf{begin\ new}\ y\ \mathbf{in}\ (E \mid ProcT^{\mathrm{prog}})\ \mathbf{end}$$

providing x, y do not occur free in E.

Fixed-Point Induction for Approximation in PROG: Let p be an identifier and $ProcT$ a PROG term, both of the same type, such that p is not free in $ProcT_2$. Then

$$\frac{[\mathbf{diverge}/p]ProcT_1 \sqsubseteq ProcT_2,\quad ProcT_1 \sqsubseteq ProcT_2 \vdash [ProcT/p]ProcT_1 \sqsubseteq ProcT_2}{\mathbf{proc}\ p \Leftarrow ProcT\ \mathbf{do}\ ProcT_1\ \mathbf{end} \sqsubseteq ProcT_2}$$

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

The original ALGOL 60 report [Naur, et. al., 1963] gave a "copy-rule" semantics for the language. The copy-rule can be understood as particular computational strategy for generating the infinite expansion of a command. Another acl property is that fixed-point and copy-rule semantics (appropriately extended to letrec-terms and PROG commands with free variables) assign the same meanings to terms [cf., Damm 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 L. On this basis we assert that the typed λ-calculus is the *true mathematical syntax* 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 L be removed.

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

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

to be the flat cpo's.

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

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

Here $\mathcal{P}(Store)$ denotes the power-set of stores (ordered by containment), so elements of $D_{\mathbf{prog}}$ 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**, **Return**, **Dint**, **Dloc**, and **New**.

The constant $\mathbf{If}_{\alpha,\beta}$ for *basic* types α, β has type $\alpha \to \alpha \to \beta \to \beta \to \beta$. A store model interprets **If** so that

$$[\![\mathbf{If}_{\alpha,\beta}]\!] d_1^{\alpha} d_2^{\alpha} d_3^{\beta} d_4^{\beta} = \begin{cases} \perp_{\beta} & \text{if } d_1 = \perp_{\alpha} \text{ or } d_2 = \perp_{\alpha}, \\ d_3 & \text{if } d_1 = d_2 \neq \perp, \\ d_4 & \text{otherwise.} \end{cases}$$

Any first order function f of type $\delta = \mathbf{int}^k \to \mathbf{int}$ can be coerced into a mapping $Mkexp(f)$ taking as arguments functions from stores to **int**. Namely, the coercer $Mkexp_{\delta}$:

$$Mkexp_{\delta}\, f^{\delta}\, d_1^{\mathbf{intexp}}, \ldots, d_k^{\mathbf{intexp}} s = f(d_1(s), \ldots, d_k(s))$$

for any store s. The constant $\mathbf{Mkexp}_\delta$ of type $\delta \to (\mathbf{intexp} \to \ldots \to \mathbf{intexp})$ is interpreted as $Mkexp_\delta$.

The constant **Cont** of type **locexp** $\to$ **intexp** is defined in store models so that

$$[\![\mathbf{Cont}]\!]\, d^{\mathbf{locexp}} s = \begin{cases} s(d(s)) & \text{if } d(s) \not\equiv \bot_{\mathbf{loc}}, \\ \bot_{\mathbf{int}} & \text{otherwise.} \end{cases}$$

For assignments, the constant **Update** of type **locexp** $\to$ **intexp** $\to$ **prog**:

$$[\![\mathbf{Update}]\!] d_1^{\mathbf{locexp}} d_2^{\mathbf{intexp}} s = \begin{cases} \{\, s[d_2(s)/d_1(s)] \,\} & \text{if } d_1(s), d_2(s) \not\equiv \bot \\ \emptyset & \text{otherwise.} \end{cases}$$

For conditional commands, **Ifprog** of type α-**exp** $\to$ α-**exp** $\to$ **prog** $\to$ **prog** $\to$ **prog**:

$$[\![\mathbf{Ifprog}_\alpha]\!] d_1^{\alpha\text{-}\mathbf{exp}} d_2^{\alpha\text{-}\mathbf{exp}} d_3^{\mathbf{prog}} d_4^{\mathbf{prog}} s =$$

$$\begin{cases} \emptyset & \text{if } d_1(s) = \bot_\alpha \text{ or } d_2(s) = \bot_\alpha, \\ d_3(s) & \text{if } d_1(s) = d_2(s) \not\equiv \bot, \\ d_4(s) & \text{otherwise.} \end{cases}$$

Command constructors **Choice**, **Seq** of type **prog** $\to$ **prog** $\to$ **prog**:

$$[\![\mathbf{Seq}]\!] d_1^{\mathbf{prog}} d_2^{\mathbf{prog}} s = \bigcup \{\, d_2(s') \mid s' \in d_1(s) \,\},$$

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

For let blocks, **Dint** of type (**int** $\to$ **prog**) $\to$ **intexp** $\to$ **prog**:

$$[\![\mathbf{Dint}]\!] d_1^{\mathbf{int}\to\mathbf{prog}} d_2^{\mathbf{intexp}} s = \begin{cases} (d_1\,(d_2(s)))(s) & \text{if } d_2(s) \not\equiv \bot_{\mathbf{int}}, \\ \emptyset & \text{otherwise.} \end{cases}$$

We translate basic blocks with declarations of location type similarly, using a corresponding combinator **Dloc**. (The simpler definition which omits the "otherwise" clause seems to imply unavoidable implementation inefficiencies and (presumably for that reason) does not correspond to the behavior of actual languages.)

Return of type **prog** $\to$ **intexp** $\to$ **intexp**:

$$[\![\mathbf{Return}]\!] d_1\, d_2\, s = d_2(d_1\, s).$$

The semantics of the constant **New** of type (**loc** $\to$ **prog**) $\to$ **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 Loc. Two stores s, t *agree on* L, written $s =_L t$, iff $\forall l \in L.\, s(l) = t(l)$. Similarly, two sets $S, T \subseteq \mathcal{P}(Stores)$ agree on L if there is a bijection $f : S \to T$ such that $\forall s \in S.\, s =_L f(s)$.

For each primitive type α, define the unary predicate $Access_\alpha^L$ on D_α by the rules below. If $Access^L(d)$ holds, we say that *d accesses only the locations in L*. Note that $Access^L(d)$ will imply $Access^{L \cup L'}(d)$.

(1) $Access_{\mathbf{loc}}^L(l)$ iff $l \in L \cup \{\, \bot_{\mathbf{loc}} \,\}$,

(2) $Access_{\mathbf{int}}^L(d) \equiv true$,

(3) $Access_{\mathbf{prog}}^L(\pi)$ iff $\forall s, t \in Store.\, (s =_L t \Rightarrow \pi(s) =_L \pi(t)) \wedge (t \in \pi(s) \Rightarrow s =_{Loc-L} t)$,

(4) $Access_{\mathbf{intexp}}^L(\tau)$ iff $\forall s, t \in Store.\, s =_L t \Rightarrow \tau(s) = \tau(t)$,

(5) $Access_{\mathbf{locexp}}^L(\sigma)$ iff $\forall s, t \in Store.\, s =_L t \Rightarrow \sigma(s) = \sigma(t) \in L \cup \{\, \bot_{\mathbf{loc}} \,\}$.

For higher-type objects, we also need a notion of uniformity with respect to "new" locations.

Definition. Let $\mu : Loc \to Loc$ be a permutation; extend μ to $D_{\mathbf{loc}}$ so that $\mu(\bot) = \bot$. Let $\mu_{Store} : Store \to Store$ be the permutation defined by the rule

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

where $\circ$ denotes functional composition, and let $\mu_{\mathcal{P}(Store)} : \mathcal{P}(Store) \to \mathcal{P}(Store)$ be the permutation defined by applying μ_{Store} elementwise. For each primitive type α, define a permutation $\mu_\alpha : D_\alpha \to D_\alpha$ by the rules:

(1) $\mu_{\mathbf{loc}} = \mu$,

(2) $\mu_{\mathbf{int}}(d^{\mathbf{int}}) = d$,

(3) $\mu_{\mathbf{prog}}(\pi) = \mu_{\mathcal{P}(Store)} \circ \pi \circ \mu_{Store}^{-1}$,

(4) $\mu_{\mathbf{intexp}}(\tau) = \tau \circ \mu_{Store}^{-1}$,

(5) $\mu_{\mathbf{locexp}}(\sigma) = \mu_{\mathbf{loc}} \circ \sigma \circ \mu_{Store}^{-1}$.

Note that $(\mu^{-1})_\alpha = (\mu_\alpha)^{-1}$, so the notation μ_α^{-1} is unambiguous.

We now proceed to define the unary predicates $Access^L_\alpha$ on D_α and the permutations $\mu_\alpha : D_\alpha \to D_\alpha$ for higher-order α by induction on types.

Definition. At higher types define

$Access^L_{\beta\to\gamma}(f)$ iff
$\forall d \in D_\beta,\ L' \subseteq Loc.\, Access^{L'}_\beta(d) \Rightarrow Access^{L\cup L'}_\gamma(f(d))$,
$Access^L_{\beta\times\gamma}(d_1, d_2)$ iff $(Access^L_\beta(d_1) \wedge Access^L_\gamma(d_2))$,
$\mu_{\beta\to\gamma}(f) = \mu_\gamma \circ f \circ \mu_\beta^{-1}$,
$\mu_{\beta\times\gamma}(d_1, d_2) = \langle \mu_\beta(d_1), \mu_\gamma(d_2)\rangle$.

A permutation $\mu : Loc \to Loc$ *fixes* L iff $\mu(l) = l$ for all $l \in L$. Define the unary predicate $Unif^L_\alpha$ on D_α by the rule:

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

If $Unif^L_\alpha(d)$ holds, we say that d *is uniform off* L.

We henceforth omit subscripts α when they are clear from context.

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

Note that for primitive types, $Access^L(d)$ iff L covers d. Some key properties of covering are
(1) if L covers d, then $L \cup L'$ covers d,
(2) if L covers $f^{\alpha\to\beta}, d^\beta$, then L covers $(f\ d)$,
(3) if L covers all $d \in Z \subseteq D_\alpha$ and $\bigsqcup Z$ exists, then L covers $\bigsqcup Z$,
(4) The functions $K, S, Y, tuple, proj^i$ 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 $[\![c]\!]$ and $e(x)$ for all the constants c and free variables x in u.

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

Covering Restriction: Every element has a finite cover.

Definition. A function $Select : D_{\mathbf{loc}\to\mathbf{prog}} \to Loc$ will be called a *selection function* iff $\forall\rho \in D_{\mathbf{loc}\to\mathbf{prog}} \exists \text{cover } L \text{ of } \rho.\ Select(\rho) \notin L$. (Selection functions exist because of the covering restriction.) For each selection function $Select$, let $New_{Select} : D_{\mathbf{loc}\to\mathbf{prog}} \to D_{\mathbf{prog}}$ be defined by

$New_{Select}\,\rho =$
$[\![Tr(\mathbf{let}\ x^{\mathbf{int}} \Leftarrow \mathbf{cont}(y)\ \mathbf{in}\ y := \mathbf{a}_0; p(y); y := x\ \mathbf{tel})]\!]e$,
where $e(y) = Select(\rho)$, $e(p) = \rho$.

Lemma. Let $Select_1, Select_2$ be selection functions. Then
(a) $New_{Select_1} = New_{Select_2}$,
(b) New_{Select_1} is continuous along algebraic sequences and has an empty cover.

It follows that if we take any selection function $Select$, then New_{Select} unambiguously determines a meaning for New in store models – which we require to be in $D_{(\mathbf{loc}\to\mathbf{prog})\to\mathbf{prog}}$.

To demonstrate rigorously that the theory of PROG is consistent, we must show that store models exist. Let *Loc* be uncountable. An ω-cpo model [Meseguer 78; Plotkin 82] with the five store-model base types (and with higher function domains consisting of all ω-continuous functions) is an also an acl model which satisfies all the conditions for store models – including the existence of an ω-continuous function which behaves like $[\![\text{New}]\!]$ on elements with *countable* covers – except for the covering restriction. For each domain of the ω-cpo model, we take the subdomain of those elements which have a finite cover. Using the method of *logical relations* of [Plotkin, 80;Statman, 82] these subdomains can be taken together to form an acl frame which can be demonstrated to be 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 new declarations are explained through renaming of local identifiers (cf. [Langmaack and Olderog, 80; Olderog, 82]).

11. Reasoning about Covers.

Because all the PROG constants have empty covers, 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 a cover for any location $l \in Loc$ is the singleton $\{l\}$. (In general, a minimal cover of a command is strictly smaller than the covers of its free identifiers, e.g., $x :=$ cont(x) has an empty cover.)

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

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 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 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 λ-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 α-**exp** types for α other than **int** and **loc** (with a corresponding block **let** x^α **be** $ProcT^{\alpha\text{-}\mathbf{exp}}$ **in** $ProcT^{\beta\text{-}\mathbf{exp}}$ **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

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

B. Courcelle, Fundamental properties of infinite trees, *Theoretical Computer Science* **25**, 1983, 95–170.

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.

J. H. Gallier, n-Rational algebras, Parts I and II, Technical Report, Dept. of Computer and Information Sciences, Univ. of Pennsylvania, Philadelphia, 1983, 55pp. and 65pp.

M. J. C. Gordon, *The Denotational Description of Programming Languages*, Springer, 1979.

I. Guessarian, *Algebraic Semantics*, Lecture Notes in Computer Science **99**, Springer, 1981, 158pp.

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

J. Y. Halpern, A good Hoare axiom system for an ALGOL-like language, ACM Symp. on Principles of Programming Languages, 1983 (this volume).

R. Hindley, B. Lercher, and J. Seldin, *Introduction to Combinatory Logic*, London Math. Soc. Lecture Note Series 7, Cambridge University Press, 1972.

J. Lambek, From λ-calculus to Cartesian closed categories, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Hindley, eds., Academic Press, 1980, 375-402.

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

H. Langmaack and E. R. Olderog, Present-day Hoare-like systems, 7th *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.

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.

M. Nivat, On the interpretation of recursive polyadic program schemes, *Symposia Mathematica*, 15, Academic Press, 1975, 255-281.

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

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 construction, *SIAM J. Comp.* 5, 1976, 452–487, 1976.

G. D. Plotkin, Lambda-definability in the full type hierarchy, in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Hindley, eds., Academic Press, 1980, 363–373.

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

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

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

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

D. S. Scott, Lectures on a Mathematical Theory of Computation, Technical Monograph PRG-19, Oxford Univ. Computing Lab., 1981.

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

M. B. Smyth, Powerdomains, *J. Computer and System Sciences* 16, 1978, 23–36.

R. Statman, Logical relations and the typed lambda-calculus, *to appear*, 1982.

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

R. D. Tennent, *Principles of Programming Languages*, Prentice-Hall International Series, 1981, 271pp.

B. A. Trakhtenbrot, J. Y. Halpern, and A. R. Meyer, From denotational to operational and axiomatic semantics: an overview, *Proc. Logics of Programs*, Carnegie-Mellon Univ., Pittsburgh, 1983, *to appear, Lecture Notes in Computer Science*, D. Kozen and E. Clarke, eds., Springer, 1983.