Semantics of Local Variables

P. W. O'Hearn[*] *R. D. Tennent*[†]

Abstract

This expository article discusses recent progress on the problem of giving sufficiently abstract semantics to local-variable declarations in Algol-like languages, especially work using categorical methods.

1 Introduction

One of the first things a beginning programmer learns is how to "declare" a new variable, as in the following Algol-like block:

$$\mathbf{new}\, x\colon \mathbf{int}.\ \ x := 0\,;\cdots;\, x := x + 1\,;\cdots$$

It might be thought that a satisfactory semantic interpretation for such declarations would be well established by now. But when free identifiers, possibly of higher-order type, can appear within the bodies of such declaration blocks, there are serious difficulties. The aim of this expository paper is to survey recent progress on this problem, especially work using categorical methods.

The traditional denotational-semantic approach [9, 19, 18] is to assume a denumerable set L of "locations" (abstract storage addresses) and construct a set of states (also known as "stores") along the following lines:

$$S \;=\; L \longrightarrow (V + \{\mathit{unused}\})\,,$$

where V is the set of *storable values.* (For simplicity, we will assume there is a *single* type of storable value, say, integers.) The idea is that each state $s \in S$ records, for every location $\ell \in L$, *either* the $v \in V$ currently stored at that location, *or* the fact that location ℓ is not currently "in use." Then the

[*]School of Computer and Information Science, Syracuse University, Syracuse, New York, USA 13244. This author attended the Durham Symposium with the aid of a travel stipend from the S.E.R.C.

[†]Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6. This author was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada.

effect of a variable declaration is to bind the declared variable identifier to any currently unused location for the execution of the block body.

This kind of interpretation of local variables is adequate to show the correctness of the usual style of implementation of block structure [9]; however, many authors [3, 16, 13, 5, 14, 2, 8] have criticized it as being insufficiently abstract. For example, consider the simple equivalence

$$\begin{array}{l}\mathbf{new}\,\iota.\ \ \iota := 0;\\ \quad C\end{array} \quad \equiv \quad C\ ,$$

when identifier ι is not free in command C. The equivalence is a consequence of the inaccessibility of the "new" location to non-local entities in Algol-like languages. Unfortunately, the equivalence fails in general in the traditional semantics. To see the problem, assume that the initial state marks a location ℓ as being unused and ℓ is chosen as the meaning of ι in the block; but in general the meaning of C need not respect the convention that ℓ is not accessible to non-local entities and might, for example, branch on whether it is "unused."

Some researchers [9, 5] have addressed this kind of difficulty by attempting to define the set of "accessible" locations or *storage support* of semantic entities, in order to ensure that the "new" location allocated for a variable declaration is not already accessible in the block body. However, this has proved to be technically complex and delicate. Phrases of basic type, such as expressions and commands, can be treated fairly straightforwardly; but the storage support of a procedure is much more difficult to define, particularly if an extensional treatment of procedures is desired.

An alternative is to adopt a semantical approach in which storage supports are explicit and logically precede any assignment of meanings to phrases; this is the possible-world method of [16] and [13].

2 Possible Worlds

The basic idea underlying possible-world semantics is to parameterize interpretations by semantic entities termed "possible worlds" that determine certain "local" aspects of meanings. To treat storage support, we assume that each world is a *natural number*, interpreted as the number of locations currently in use; in implementation terms, this is the number of variables on the run-time stack. Then, for each world w, we can interpret basic phrase types **var** (variables), **exp** (expressions), and **comm** (commands) as the following computational domains:

$$[\![\mathbf{var}]\!]w \;=\; \{0, 1, 2, \ldots, w-1\},\ \text{discretely ordered;}$$

$$\begin{aligned}
[\![\mathbf{exp}]\!]w &= S(w) \rightarrow V_\perp; \\
[\![\mathbf{comm}]\!]w &= S(w) \rightarrow S(w)_\perp,
\end{aligned}$$

where $S(w) = [\![\mathbf{var}]\!]w \rightarrow V$ is the set of w-states, $A_\perp$ for any set A is the "lifted" domain obtained by adding a new least element $\perp$, and, for any sets or domains D and E, $D \rightarrow E$ is the domain of (continuous) functions from D to E, ordered pointwise.

Intuitively:

- a declared variable in world w denotes a location (natural number) less than w;
- an expression meaning in world w is a function from w-states to values (or to $\perp$, to allow for non-termination of evaluation), where a w-state is a vector of storable values with exactly w components; and
- a command meaning in world w is a function from w-states to w-states (or to $\perp$).

The worlds w determine the storage support of the basic semantic entities in a simple and natural way. Procedures will be discussed in Section 3.

To cope with variable declarations, we must define how to map phrase meanings from non-local to local worlds. This is most coherently done by defining a *category* $\mathbf{W}$ of possible worlds: the objects are natural numbers and the morphisms from w to x are the injective functions from $\{0, 1, \ldots, w-1\}$ to $\{0, 1, \ldots, x-1\}$, with functional composition as the composition in $\mathbf{W}$. We will explain later why *all* injections (and not merely the inclusions) are needed.

We can now extend the interpretations of basic phrase types θ to *functors* $[\![\theta]\!]$ from $\mathbf{W}$ to the category $\mathbf{D}$ of directed-complete posets and continuous functions by defining morphism parts $[\![\theta]\!]f$ for every $\mathbf{W}$-morphism $f: w \rightarrow x$. We define

$$[\![\mathbf{var}]\!]f : [\![\mathbf{var}]\!]w \rightarrow [\![\mathbf{var}]\!]x$$

to be the function f. To define

$$[\![\mathbf{exp}]\!]f : [\![\mathbf{exp}]\!]w \rightarrow [\![\mathbf{exp}]\!]x\,,$$

we note that, for any state $s \in S(x)$, $(f\,;s) \in S(w)$ is the corresponding "non-local part" of s, where ; denotes composition in diagrammatic order, and then

$$[\![\mathbf{exp}]\!]f\,e\,s = e(f\,;s)$$

for any $e \in [\![\mathbf{exp}]\!]w$ and $s \in S(x)$.

Similarly, $[\![\mathbf{comm}]\!] f\, c\, s$ is $\bot$ if $c(f;s) = \bot$, and otherwise is the state $s' \in S(x)$ such that, for any $\ell \in [\![\mathbf{var}]\!]x$, $s'(\ell) = c(f\,;s)(\ell')$ if $\ell = f(\ell')$ and $s'(\ell) = s(\ell)$ otherwise; that is, the effect of executing a non-local command meaning c is to change the non-local part of the state in the manner determined by c, but to preserve the local part of the state invariant.

To deal with free identifiers and environments, we note that finite products of functors from $\mathbf{W}$ to $\mathbf{D}$ are definable pointwise: for any $F, G\colon \longrightarrow \mathbf{D}$ and any world w, $(F \times G)(w) = F(w) \times G(w)$, and similarly for the morphism part. Then, for any assignment π of phrase types to a finite set of identifiers, we can define $[\![\pi]\!]$ to be the product of the functors for the types assigned by π to these identifiers. Then $[\![\pi]\!]w$ is the domain of π-compatible environments appropriate to world w, and $[\![\pi]\!]f$ for $\mathbf{W}$-morphisms $f\colon w \longrightarrow x$ maps such w-environments to the appropriate x-environments.

We use the notation $(\pi \mid \iota \mapsto \theta)$ to denote the type assignment that is like π except that identifier ι is assigned type θ, and similarly for the environment $(u \mid \iota \mapsto m)$, and the state $(s \mid \ell \mapsto v)$.

Finally, any phrase X that is well-formed and of type θ in the context of phrase-type assignment π (notation: $\pi \vdash X\colon \theta$) is interpreted (using a valuation function $[\![\cdot]\!]_{\pi\,\theta}$) as a *natural transformation* $[\![X]\!]_{\pi\theta}\colon [\![\pi]\!] \stackrel{\cdot}{\longrightarrow} [\![\theta]\!]$; i.e., the right-hand side of the following diagram commutes for all $\mathbf{W}$-morphisms $f\colon w \longrightarrow x$:

$$
\begin{tikzcd}
w \arrow[d, "f"'] & & [\![\pi]\!]w \arrow[r, "{[\![X]\!]_{\pi\theta}(w)}"] \arrow[d, "{[\![\pi]\!]f}"'] & [\![\theta]\!]w \arrow[d, "{[\![\theta]\!]f}"] \\
x & & [\![\pi]\!]x \arrow[r, "{[\![X]\!]_{\pi\theta}(x)}"] & [\![\theta]\!]x
\end{tikzcd}
$$

The commutativity requirement ensures that the parameterized interpretation of X is *uniform* with respect to all changes of world f. The phrase-type assignment (π) and phrase-type (θ) subscripts on phrase-valuation brackets will often be omitted when they are obvious from context.

Representative syntax rules and semantic valuations are presented in Tables 1 and 2, respectively. The semantic arguments in the latter are as follows: w is any $\mathbf{W}$-object (natural number), u is an environment in $[\![\pi]\!]w$, and $s \in S(w)$ is a state appropriate to world w.

$$\frac{\pi \vdash X\colon \mathbf{var} \quad \pi \vdash E\colon \mathbf{exp}}{\pi \vdash X := E\colon \mathbf{comm}}\ \textit{Assignment} \qquad \frac{(\pi \mid \iota \mapsto \mathbf{var}) \vdash C\colon \mathbf{comm}}{\pi \vdash \mathbf{new}\,\iota.\,C\colon \mathbf{comm}}\ \textit{New}$$

$$\frac{\pi \vdash C_1\colon \mathbf{comm} \quad \pi \vdash C_2\colon \mathbf{comm}}{\pi \vdash C_1 \,;\, C_2\colon \mathbf{comm}}\ \textit{Sequencing} \qquad \frac{\pi \vdash X\colon \mathbf{var}}{\pi \vdash X\colon \mathbf{exp}}\ \textit{Dereferencing}$$

Table 1. Selected Syntax Rules.

$$[\![X := E]\!]\,w\,u\,s \;=\; \begin{cases} (s \mid \ell \mapsto v), & \text{if } [\![E]\!]\,w\,u\,s = v \in V \\ & \quad\text{and } [\![X]\!]\,w\,u = \ell \\ \bot, & \text{if } [\![E]\!]\,x\,u\,s = \bot \end{cases}$$

$$[\![C_1 \,;\, C_2]\!]\,w\,u\,s \;=\; \begin{cases} [\![C_2]\!]\,w\,u\,s', & \text{if } [\![C_1]\!]\,w\,u\,s = s' \in S(w) \\ \bot, & \text{if } [\![C_1]\!]\,w\,u\,s = \bot \end{cases}$$

$$[\![X]\!]_{\pi\,\mathbf{exp}}\,w\,u\,s \;=\; s(\ell),\ \text{where } \ell = [\![X]\!]_{\pi\,\mathbf{var}}\,w\,u$$

Table 2. Sample Valuations.

We can now discuss the interpretation of the variable-declaration block. Let $v_0 \in V$ be a standard initial value for declared variables, and $f\colon w \longrightarrow w+1$ be the inclusion of $\{0, 1, \ldots, w-1\}$ in $\{0, 1, \ldots, w\}$; then $[\![\mathbf{new}\ \iota.\, C]\!]_{\pi\,\mathbf{comm}}\,w\,u\,s \;=\; f\,;s'$ when

$$[\![C]\!]_{(\pi \mid \iota \mapsto \mathbf{var})\mathbf{comm}}(w+1)(u')(s \mid w \mapsto v_0) \;=\; s',$$

where $u' = \big([\![\pi]\!]\,f\,u \mid \iota \mapsto w\big)$, and is $\bot$ otherwise.

Notice that the meaning of the block in world w is defined in terms of the meaning of the body of the block in world $w+1$. In the body of the block, the declared variable denotes the "new" location, and all other free identifiers denote meanings obtained by applying the morphism part of the relevant functor to the non-local meanings in the environment. After execution of the body, the new location is discarded. We leave it as an exercise for the reader to prove the validity of the problematical equivalence discussed in Section 1 using this semantics.

We conclude this section by discussing the naturality of $[\![\mathbf{new}\,\iota.\,C]\!]$; i.e., the commutativity of

$$\begin{tikzcd}
[\![\pi]\!]w \arrow[r, "{[\![\mathbf{new}\,\iota.\,C]\!]w}"] \arrow[d, "{[\![\pi]\!]f}"'] & {[\![\mathbf{comm}]\!]w} \arrow[d, "{[\![\mathbf{comm}]\!]f}"] \\
{[\![\pi]\!]x} \arrow[r, "{[\![\mathbf{new}\,\iota.\,C]\!]x}"] & {[\![\mathbf{comm}]\!]x}
\end{tikzcd}$$

for every $f\colon w \longrightarrow x$. Note that $[\![\mathbf{new}\,\iota.\,C]\!]w$ and $[\![\mathbf{new}\,\iota.\,C]\!]x$ bind ι to possibly *different* locations; the former uses $w \in [\![\mathbf{var}]\!](w+1)$, and the latter uses $x \in [\![\mathbf{var}]\!](x+1)$. The key to showing naturality is to relate w and x in a suitable fashion. This is done by extending f with the identity id_1 on 1; i.e., using the $\mathbf{W}$-morphism $(f+\mathrm{id}_1)\colon w+1 \longrightarrow x+1$ that renames $w \in [\![\mathbf{var}]\!](w+1)$ to $x \in [\![\mathbf{var}]\!](x+1)$, where, for any $f\colon w \rightarrow x$ and $g\colon y \rightarrow z$, the $\mathbf{W}$-morphism $(f+g)\colon w+y \longrightarrow x+z$ is given by

$$(f+g)(i) \;=\; \begin{cases} f(i), & \text{if } i < w \\ x+g(j), & \text{if } i = w+j \end{cases}$$

for $i \in [\![\mathbf{var}]\!](w+y)$. Naturality of $[\![C]\!]$ with respect to $(f+\mathrm{id}_1)$ gives us the commutativity of

$$\begin{tikzcd}
{[\![\pi]\!](w+1)} \arrow[r, "{[\![C]\!](w+1)}"] \arrow[d, "{[\![\pi]\!](f+\mathrm{id}_1)}"'] & {[\![\mathbf{comm}]\!](w+1)} \arrow[d, "{[\![\mathbf{comm}]\!](f+\mathrm{id}_1)}"] \\
{[\![\pi]\!](x+1)} \arrow[r, "{[\![C]\!](x+1)}"] & {[\![\mathbf{comm}]\!](x+1)}
\end{tikzcd}$$

which shows that the specific choice of fresh location does not affect the meaning of $\mathbf{new}\,\iota.\,C$, and then naturality of $[\![\mathbf{new}\,\iota.\,C]\!]$ follows easily.

The important role of "location renaming" in this discussion explains why we used all injective maps, and not just inclusions, as morphisms in the category of worlds: it is forced on us by naturality. This even shows up on the level of terms, via equivalences valid in the model; e.g.,

$$\mathbf{new}\,x.\,\mathbf{new}\,y.\,C \;\equiv\; \mathbf{new}\,y.\,\mathbf{new}\,x.\,C.$$

The principle that the specific "names" of declared locations are irrelevant turns out to be technically quite important; it plays a central role in [5, 2, 8] as well. In their treatment of support, Meyer and Sieber consider a semantic framework very similar to possible worlds, but using inclusion maps, rather than all injections, as the morphisms between worlds. They must then cut down the model by restricting to semantic entities invariant under certain location-renaming permutations. Here, the additional maps in our category of worlds provide us with stronger uniformity conditions.

It should be mentioned that, for languages *without* procedures, the kind of equivalences we have been discussing can be validated using "traditional" models. The main advantage of the possible-world approach to variable allocation is that it extends smoothly to procedural and other types. By building storage support into the "semantical spaces" (i.e., functors), one can find a notion of procedure meaning, based on exponentiation in functor categories, that itself incorporates support. This is the topic of the next section.

3 Procedures

In this section, we consider how *procedural* types of the form $\theta \longrightarrow \theta'$ are dealt with in the possible-world framework. A procedure defined in world w might be called in any world x accessible from w using a $\mathbf{W}$-morphism $f\colon w \longrightarrow x$; for example, there might be variable declarations occurring between the point of definition and the point of call. But it is the domain structure determined by f and x that should be in effect when the procedure body is executed. This suggests that we cannot in general define $[\![\theta \longrightarrow \theta']\!]w$ to be just $[\![\theta]\!]w \longrightarrow [\![\theta']\!]w$, where the arrow on the right is the exponentiation in $\mathbf{D}$. In general, the meaning of a procedure defined in world w must be a *family* of functions, indexed by changes of world $f\colon w \longrightarrow x$, and suitably constrained by a uniformity condition.

The key fact is that, for *any* small category $\mathbf{W}$, the functor category $\mathbf{D}^{\mathbf{W}}$ is Cartesian closed. This is proved for any complete Cartesian closed category $\mathbf{D}$ in [10]; the special case of $\mathbf{D}$ being the category of directed-complete posets and continuous functions is treated in [13].

Finite products in $\mathbf{D}^{\mathbf{W}}$ are definable pointwise. We now give a general definition of exponentiation and corresponding interpretations of abstraction and application satisfying the usual laws of the typed lambda calculus. For any $F, G\colon \longrightarrow \mathbf{D}$ and world w, $p \in (F \longrightarrow G)(w)$ is a *family* $p(\cdot)$ of functions, indexed by $\mathbf{W}$-morphisms with domain w, such that $p(f\colon w \longrightarrow x)$ is a continuous function from $F(x)$ to $G(x)$ and the family satisfies the following uniformity condition: for all $\mathbf{W}$-morphisms $f\colon w \longrightarrow x$ and $g\colon x \longrightarrow y$,

$$p(f)\,;G(g) \;=\; F(g)\,;p(f\,;g)$$

i.e., the following diagram commutes:

$$\begin{tikzcd} F(x) \arrow[r, "p(f)"] \arrow[d, "F(g)"'] & G(x) \arrow[d, "G(g)"] \\ F(y) \arrow[r, "p(f\,;g)"] & G(y) \end{tikzcd}$$

The domain $(F \longrightarrow G)(w)$ is ordered pointwise, and the morphism part of $F \longrightarrow G$ is defined as follows: for any $f\colon w \longrightarrow x$, $p \in (F \longrightarrow G)(w)$, and $g\colon x \longrightarrow y$,

$$(F \longrightarrow G)\, f\, p\, g \;\;=\;\; p(f\,;g).$$

We can now interpret procedural types by induction on the structure of types; for any phrase types θ and θ', $[\![\theta \longrightarrow \theta']\!] = [\![\theta]\!] \longrightarrow [\![\theta']\!]$, where the arrow on the right is the just-described exponentiation in the functor category $\mathbf{D}^{\mathbf{W}}$. The valuations for abstraction and application are then as follows; for worlds w and environments $u \in [\![\pi]\!]w$,

$$[\![\lambda\iota\colon\theta.\, P]\!]_{\pi(\theta\rightarrow\theta')}\, w\, u\, f\, a \;\;=\;\; [\![P]\!]_{(\pi|\iota\mapsto\theta)\theta'}\, x \left(\, [\![\pi]\!]\, f\, u \;\middle|\; \iota \mapsto a \right)$$

for all $f\colon w \longrightarrow x$ and $a \in [\![\theta]\!]x$, and

$$[\![P\; Q]\!]_{\pi\theta'}\, w\, u \;\;=\;\; [\![P]\!]_{\pi(\theta\rightarrow\theta')}\, w\, u\, (\mathrm{id}_w)\Big([\![Q]\!]_{\pi\theta}\, w\, u\Big),$$

where id_w is the identity $\mathbf{W}$-morphism on world w. Note that these definitions are applicable with *any* category $\mathbf{W}$ of possible worlds.

The above treatment of procedures is *not* in general extensional in the usual sense; however, for *any* category $\mathbf{W}$, the functor category $\mathbf{S}^{\mathbf{W}}$ (where $\mathbf{S}$ is the category of sets and functions) is a topos [4, 7, 1], and so there is an interpretation of logical connectives such as implication ($\Rightarrow$), universal quantification ($\forall$), and identity ($\equiv$) that validates the laws of intuitionistic logic, and also the following *formal* law of extensionality:

$$\Big(\forall\iota\colon\theta.\, P(\iota) \equiv Q(\iota)\Big) \;\;\Rightarrow\;\; P \equiv Q$$

where ι is not free in $P, Q\colon \theta \longrightarrow \theta'$. This allows us to reason about procedures in (intuitionistic) programming logics as if they were extensional. One such system will be discussed in Section 6.

4 States and Contravariance

At this point the reader may be wondering why functors for commands and expressions were defined pointwise in Section 2, whereas procedures were treated using exponentiation of functors in Section 3. Would it not be possible to obtain a more systematic treatment by using exponentiation in the definition of base types as well? That is, one might try to define a "states" functor S and proceed as follows:

$$\begin{aligned} [\![\mathbf{exp}]\!] &= S \rightarrow V_\perp \\ [\![\mathbf{comm}]\!] &= S \rightarrow S_\perp, \end{aligned}$$

where $(\cdot)_\perp$ is a pointwise "lifting" operation on functors from $\mathbf{W}$ to $\mathbf{D}$. However, this is not as straightforward as it might appear.

We can of course define a states functor as an exponentiation of the form $S = [\![\mathbf{var}]\!] \rightarrow V$, where V here is the obvious constant functor (of values). If w is a world and $s \in S(w)$, the uniformity condition on s requires commutativity of

$$\begin{tikzcd} {[\![\mathbf{var}]\!]w} \arrow[r, "s(\mathrm{id}_w)"] \arrow[d, "{[\![\mathbf{var}]\!]f}"'] & V \arrow[d, "\mathrm{id}_V"] \\ {[\![\mathbf{var}]\!]x} \arrow[r, "s(f)"] & V \end{tikzcd}$$

for any $\mathbf{W}$-morphism $f\colon w \rightarrow x$. But this does not constrain $s(f)$ outside the image of $[\![\mathbf{var}]\!]f$, and so there are far too many states; states $s, s' \in S(w)$ would be distinguished even if they agree on all locations in $[\![\mathbf{var}]\!]w$. There would even be many "global" states in $S(0)$, whereas one would expect $S(0)$ to be a singleton, containing only the "empty" state in which there are no locations.

An alternative is to use $S(w) = [\![\mathbf{var}]\!]w \rightarrow V$, as before, and, for any $f\colon w \rightarrow x$ in $\mathbf{W}$ and $s \in S(w)$, define

$$S(f)\, s\, i \;=\; \begin{cases} s(j), & \text{if } f(j) = i \\ v_0, & \text{otherwise,} \end{cases}$$

for some fixed storable value v_0. But it is difficult to justify the arbitrary choice of v_0; furthermore, this allows too many expression and command meanings. For example, if $e \in [\![\mathbf{exp}]\!]w$ and $f\colon w \rightarrow w+1$ is the inclusion, commutativity of

$$\begin{tikzcd} S(w) \arrow[r, "e(\mathrm{id}_w)"] \arrow[d, "S(f)"'] & V_\perp \arrow[d, "\mathrm{id}_{V_\perp}"] \\ S(w+1) \arrow[r, "e(f)"] & V_\perp \end{tikzcd}$$

in no way constrains $e(f)$ on states $s \in S(w+1)$ for which $s(w) \neq v_0$, and so the value yielded by a non-local expression meaning can be affected by changing the contents of a local variable.

Similarly, if $c \in [\![\mathbf{comm}]\!]w$, commutativity of

$$\begin{tikzcd} S(w) \arrow[r, "c(\mathrm{id}_w)"] \arrow[d, "S(f)"'] & S(w)_\perp \arrow[d, "S(f)_\perp"] \\ S(w+1) \arrow[r, "c(f)"] & S(w+1)_\perp \end{tikzcd}$$

does not preclude $c(f)$ from changing the contents of the local variable (whenever this differs from v_0). For example, c could be such that $c(f)(s)$ returns a state in which the contents of every variable is v_0, thus changing the contents of every variable whose contents in s is not v_0, including the local variable. This would invalidate expected equivalences such as the one mentioned in the Introduction.

In fact, it seems most natural to have a *contravariant* states functor: a state in a "big" world can be converted to one in a "small" world by simply dropping locations. We can make S from Section 2 into a functor from $\mathbf{W}^{\mathbf{op}}$ to $\mathbf{D}$ by defining $S(f)s = f\,;s$, when $f\colon w \longrightarrow x$ in $\mathbf{W}$ and $s \in S(x)$.

But then how can *covariant* functors $[\![\mathbf{exp}]\!]$ and $[\![\mathbf{comm}]\!]$ be constructed from *contravariant* S? A somewhat mysterious answer to these questions is given in [20] by defining a variant of exponentiation in functor categories, which we will term *contra-exponentiation*, that constructs a covariant functor from contravariant functors.

For example, a contra-exponential of the form $S \longrightarrow V_\perp$ is defined just like the usual exponential, but with a reversal of vertical arrows in uniformity diagrams to account for contravariance. That is, $e(\cdot) \in (S \longrightarrow V_\perp)w$ is a family of functions, indexed by morphisms out of w, such that

$$\begin{tikzcd}
S(x) \arrow[r, "e(f)"] & V_\perp \\
S(y) \arrow[u, "S(g)"] \arrow[r, "e(f\,;g)"] & V_\perp \arrow[u, "\mathrm{id}_{V_\perp}"']
\end{tikzcd}$$

commutes, where $f\colon w \longrightarrow x$ and $g\colon x \longrightarrow y$ in $\mathbf{W}$. The morphism part is defined as before to yield a *covariant* functor:

$$(S \longrightarrow V_\perp)\, f\, m\, g \;=\; m(f\,;g)$$

where $f\colon w \longrightarrow x$, $m \in (S \longrightarrow V_\perp)w$, and $g\colon x \longrightarrow y$. It can be seen that all of the $e(f)$ for $f\colon w \longrightarrow x$ are in fact determined by $e(\mathrm{id}_w)$, and so the "pointwise" definition of $[\![\mathbf{exp}]\!]$ given earlier is isomorphic to $S \longrightarrow V_\perp$.

We can also use contra-exponentiation to define a covariant functor $S \longrightarrow S_\perp$; the relevant diagram is

$$\begin{tikzcd}
S(x) \arrow[r, "c(f)"] & S(x)_\perp \\
S(y) \arrow[u, "S(g)"] \arrow[r, "c(f\,;g)"] & S(y)_\perp \arrow[u, "S(g)_\perp"']
\end{tikzcd}$$

However, it is evident that one of the problems with a covariant S arises here as well: this definition does not preclude non-local command meanings from changing the values of local variables. To get a definition of $[\![\mathbf{comm}]\!]$ comparable to that in Section 2, we need to impose the additional requirement on $c \in (S \longrightarrow S_\perp)w$ that, for every $f\colon w \longrightarrow x$, when $c(f)$ is defined it must preserve the values of locations not in the range of f.

Thus, contra-exponentiation allows us to build covariant functors from a contravariant states functor, but a further restriction is needed to get a satisfactory definition of $[\![\mathbf{comm}]\!]$. It is not known whether there is a more uniform way of dealing with these problems. Contra-exponentiation will be encountered again in Section 6, where it will be used to define $[\![\mathbf{comm}]\!]$ in a setting where a pointwise definition is *not* appropriate.

5 Generalized Variables

Reynolds [16] and Oles [13] argue that it is preferable to treat variables in Algol-like languages without assuming that states are structured by locations.

This can be achieved by defining a category $\mathbf{X}$ of possible worlds with *sets* (of allowed states) W, X, … as objects. (We assume that the small collection of $\mathbf{X}$-objects contains the set V of values, and is closed under cartesian products.) To interpret variable declarations we use morphisms that "expand" a world W to a world $W \times V$; the extra V-valued component holds the value of a new variable. Roughly, a state $x \in X$ is related to $\langle x, v\rangle \in X \times V$ similarly to the way $s \in S(w)$ is related to $(s \mid w \mapsto v) \in S(w+1)$ in the semantics of Section 2.

In general, the $\mathbf{X}$-morphisms from W to X will be pairs f, Q where f is a function from X to W and Q is an equivalence relation on X such that the following is a product diagram in the category of sets and functions:

$$W \xleftarrow{\;f\;} X \xrightarrow{\;q\;} X/Q$$

where q maps every element of X to its Q-equivalence class. Intuitively, f extracts the small stack embedded in a larger one, and Q relates large stacks with identical "expansion components." This is just the category-theoretic way of saying that larger stacks are formed from smaller ones by adding independent components for local variables. It can be verified that $X \cong W \times X/Q$, and that the restriction of f to each Q-equivalence class is bijective.

The identity morphism id_W on an object W has as its two components: the identity function on W, and the universally-true binary relation on W. The composition $(f, Q) \mathbin{;} (g, R) \colon W \longrightarrow Y$ of $\mathbf{X}$-morphisms $f, Q \colon W \longrightarrow X$ and $g, R \colon X \longrightarrow Y$ has as its two components: the functional composition of f and g, and the equivalence relation on Y that relates $s_0, s_1 \in Y$ just if they are R-related and Q relates $g(s_0)$ and $g(s_1)$. To see that these components have the required product property, let $h, S = (f, Q) \mathbin{;} (g, R)$ and note that

$$\begin{aligned}
Y &\cong X \times Y/R \\
&\cong (W \times X/Q) \times Y/R \\
&\cong W \times (X/Q \times Y/R) \\
&\cong W \times Y/S,
\end{aligned}$$

by the isomorphism between Y/S and $X/Q \times Y/R$ established by the function mapping $[y]_S$ to $\big([g(y)]_Q, [y]_R\big)$, whose inverse is the function mapping $\big([x]_Q, [y]_R\big)$ to $\big\{y' \in [y]_R \mid g(y') Q x\big\}$.

We can now define a states functor $S \colon \mathbf{X}^{\mathbf{op}} \longrightarrow \mathbf{D}$ as follows: $S(W) = W$, discretely-ordered, and $S(f, Q) = f$. Then, we can use contra-exponentiation to define $[\![\mathbf{exp}]\!]$ as $S \longrightarrow V_\perp$ and $[\![\mathbf{comm}]\!]$ as a restriction of $S \longrightarrow S_\perp$ (or use pointwise definitions).

All of this seems very similar to the location-based approach: components of products, instead of locations, are used to hold the values of variables. Furthermore, an n-fold product V^n can be viewed as a world that results from n variable declarations, and a state

$$\langle v_0, \ldots, v_{n-1} \rangle \in V^n$$

would correspond to a state $s \in S(n)$ with $s(i) = v_i$ for $i \in \{0, \ldots, n-1\}$, where $S(n)$ is as in Section 2. There are even morphisms that are similar to the "location renaming" maps needed for the naturality of variable declarations; e.g., isomorphisms on worlds of the form $W \times V \times V$ that exchange the two V-valued components.

However, this analogy between locations and components of products cannot be pushed too far. The reason is that the component structure is only used when defining local variables and there is no *a priori* structure on states that allows us to identify "location-like" components independently of declarations. Consider, for example, a world $V \times V$. If locations are components then this should correspond to a world with two locations. But in the state-set category of worlds, $V \times V$ is isomorphic to V if V is infinite. So how many locations does the world have, one or two?

The answer is neither. In fact, the state-set category of worlds seems to preclude identification of entities that behave like locations. But what then should $[\![\mathbf{var}]\!]$ be? The approach taken in [16, 13] is to treat variables in the following more general way:

$$[\![\mathbf{var}]\!] \;=\; (V \longrightarrow [\![\mathbf{comm}]\!]) \times [\![\mathbf{exp}]\!] \,.$$

Intuitively, a variable consists of two components, the first of which allows for changing the state when a storable value is supplied to it, and the second allows for accessing the state and yielding a stored value. Thus, the main difference between the natural-number category of worlds and the state-set category is that the former is oriented to a view of variables as locations, while the latter is oriented to states as abstract "objects" whose internal structure is deliberately left unspecified. These different views have important ramifications on how the type of variables is interpreted.

Even though the type of generalized variables does not consist of locations, the interpretation of declarations requires us to manufacture an appropriate new variable that has the location-like property that assigning a value to it results in a state in which the value in its expression component is the value just assigned. To do this, we make use of the product structure of morphisms.

A new variable $(a, e) \in [\![\mathbf{var}]\!](W \times V)$ is defined as follows, assuming pointwise definitions of $[\![\mathbf{comm}]\!]$ and $[\![\mathbf{exp}]\!]$:

$$a\,(f, Q)\,v\,s \;=\; s'$$

$$e\,\langle x, v\rangle \;=\; v$$

where $f\colon W \times V \to Y$, $v \in V$, $s \in S(Y)$, and s' is the unique state in $S(Y)$ such that $s\ Q\ s'$ and if $f(s) = \langle x, v'\rangle$ then $f(s) = \langle x, v\rangle$. (Uniqueness of s' follows from the bijectivity requirement on the f part of f, Q.) The idea behind the definition of a is that the V-component of s' is changed to v, while the X-component and the "expanded" part coming from f, Q remain unchanged. With these definitions, the semantics of **new** can be treated essentially as in Section 2, using an expansion $W \longrightarrow W \times V$ in place of the inclusion $w \longrightarrow w + 1$.

6 Non-Interference

The support-oriented approach to local variables represents a significant improvement on the models based on marked stores. However, there remain simple equivalences involving procedures that are not treated properly. For example, the following block should be equivalent to **diverge** (i.e., a never-terminating command):

$$\begin{array}{l}\mathbf{new}\, x.\;\; x := 0;\\ \quad P(x := x + 2);\\ \quad \mathbf{if\ even}\, x\ \mathbf{then\ diverge}\end{array}$$

when x is not free in $P\colon \mathbf{comm} \to \mathbf{comm}$. Intuitively, the block diverges because the *only* access to x that can result from the call to P must come from using the argument $x := x + 2$. This argument preserves the evenness of x and x is even before the call, so it should be even on termination. Therefore, the test in the conditional will succeed (if the call to P terminates) and the block will diverge; if the call does not terminate, neither will the block, and so in every case the block diverges.

This example (attributed in [8] to Allen Stoughton) fails in support-oriented interpretations because it is possible for P to denote a procedure that preserves non-local variables but sets to 1 any local variables whose values are changed by its argument. It also fails for models based on the state-set category of worlds from the preceding section. As with the location-based category, this can be shown by exhibiting a $p \in [\![\mathbf{comm} \to \mathbf{comm}]\!]X$ such that $p(f)(c)(\mathrm{id})(s, v) = (s', 1)$ if $c(\mathrm{id})(s, v) = (s', v')$ and $v' \neq v$, and $c(\mathrm{id})(s, v)$ otherwise, where $f\colon X \longrightarrow X \times V$ is the expansion morphism; with the state-set category it requires some work to show that such a p actually exists.

While this block is not itself a realistic program, it illustrates how the semantic problems with local variables involve principles that are important in programming practice. The procedure P is supplied with an argument that provides a *limited capability* for access to the local variable, and one reasons that the *only* way that the procedure call can access the variable is via this limited capability. This kind of informal reasoning is familiar from programming with "modules" or "objects." A module typically has a number of internal variables, which are accessed and modified by clients through the use of explicitly "exported" procedures only. Often, the exports provide only a limited capability for accessing the variables, in a way that presents to the client a view of the module that does not reveal details of its internal structure.

In this section we will consider a model that has arisen from work on finding a satisfactory semantic interpretation of the "specification logic" of [15, 17], which is a formal system for proving equivalences and partial-correctness properties of programs in Algol-like languages with higher-order procedures. For our purposes, the most important rule of the logic is one for reasoning about local-variable declarations; for clarity, we present it in natural-deduction format:

Local-Variable Declarations:

$$\frac{\begin{array}{c}\left[\begin{array}{c}\iota \mathrel{\#} E_1, \ldots, \iota \mathrel{\#} E_m \\ C_1 \mathrel{\#} \iota, \ldots, C_n \mathrel{\#} \iota\end{array}\right] \\ \vdots \\ \{P\}\, C\, \{Q\}\end{array}}{\{P\}\, \mathbf{new}\ \iota.\, C\, \{Q\}}$$

where the bound variable identifier ι is not free in assertions P or Q, in phrases E_i or C_j, or in uncancelled assumptions. Here, $\{P\}\, C\, \{Q\}$ is the familiar *Hoare triple* [6], and $C \mathrel{\#} E$ ("C doesn't interfere with E") essentially asserts that every way of using C preserves any value obtained by using E. The rule says that, when reasoning about a locally-declared variable, it is valid to assume that a non-local entity and the declared variable don't interfere with one another.

The divergence of the troublesome block above can in fact be proved in specification logic by deriving $\{\mathbf{true}\}\, B\, \{\mathbf{false}\}$, where B is the block. This is done using the rule for variable declarations, and instantiating R to $\mathbf{even}\ x$ and C to $x := x + 2$ in the following axiom [11, 12]:

Non-Interference Abstraction:

$$P \mathrel{\#} R \ \&\ \{R\}\, C\, \{R\} \ \Rightarrow\ \{R\}\, P(C)\, \{R\}$$

where P: **comm** $\rightarrow$ **comm**.

Meyer and Sieber [8] have shown how to modify support-oriented interpretations in a way that handles the "evenness" example as well. They use logical relations to cut down the model, restricting to procedures that preserve invariant properties of locations outside their support. For example, in the above equivalence the evenness of x is a property of a variable (x) outside the support of P, and the requirement is that any call to P must preserve the evenness of x if its argument does. The Non-Interference Abstraction axiom relates non-interference to these ideas of Meyer and Sieber about invariants.

The model for specification logic is based on a modification of the state-set category of worlds: the morphisms from X to Y are again pairs f, Q, where f is a function from Y to X, and Q is an equivalence relation on Y, but the restriction of f to a Q-equivalence class is only required to be injective, instead of bijective. We call this modified category $\mathbf{X}'$. The definition of composition given in Section 5 works here as well.

We still have the expansion maps from W to $W \times V$. But now we also have morphisms that "restrict" the possible world; for any $X' \subset X$, the *restriction morphism* $f, Q: X \rightarrow X'$ has as its components: the inclusion function f from X' to X and the everywhere-true binary relation Q on Y. We shall see how these additional morphisms give us stronger uniformity conditions.

A contravariant states functor is defined as in Section 5. Expression meanings can then be treated using contra-exponentiation: $[\![\mathbf{exp}]\!] = S \rightarrow V_\perp$. But now, because of the possibility of restricting to a subset of states, it is possible to prove that expressions cannot have side effects, not even "temporary" ones. For any $w \in W$, there is a "restriction" morphism $f: W \rightarrow \{w\}$ such that $S(f)$ is the inclusion of $\{w\}$ in W, and commutativity of the following diagram shows that the value of $e \in [\![\mathbf{exp}]\!]W$ at $w \in W$ is determined in the singleton world $\{w\}$ in which w is the *only* state:

$$\begin{tikzcd}
W \arrow[r, "e(\mathrm{id}_W)"] & V_\perp \\
\{w\} \arrow[u, "S(f)"] \arrow[r, "e(f)"] & V_\perp \arrow[u, "\mathrm{id}_{V_\perp}"']
\end{tikzcd}$$

This property plays an important role in reasoning about assignments. It is also reflected in the following equivalence:

$$\begin{array}{l} \textbf{if } x = 0 \\ \quad \textbf{then } P(x) \\ \quad \textbf{else } 1 \end{array} \quad \equiv \quad \begin{array}{l} \textbf{if } x = 0 \\ \quad \textbf{then } P(0) \\ \quad \textbf{else } 1 \end{array}$$

where $P\colon \mathbf{exp} \rightarrow \mathbf{exp}$. The idea is that execution of $P(x)$ never changes the value of x, so it should be equivalent to executing $P(0)$ in a state where the value of x is 0. This equivalence is not valid in the models discussed in previous sections because the semantic domains for expressions in these models allow for the possibility of "temporary" side effects, where the values of global variables are altered during expression evaluation and then restored on termination of the evaluation.

Command meanings must be treated somewhat differently. Because of restriction morphisms in $\mathbf{X}'$, $S \rightarrow S_\perp$ is too small, whereas previously it was too large; the commutativity of

$$\begin{tikzcd} W \arrow[r, "c(\mathrm{id}_W)"] & W_\perp \\ \{w\} \arrow[u, "S(f)"] \arrow[r, "c(f)"] & \{w\}_\perp \arrow[u, "S(f)_\perp"'] \end{tikzcd}$$

for any $w \in W$ would preclude commands from changing states at all!

Instead, a *partial* contra-exponentiation $\rightharpoonup$ can be defined in $\mathbf{D}^{\mathbf{X}'}$, and then commands are interpreted using $S \rightharpoonup S$. The uniformity requirement on $c \in (S \rightharpoonup S)(W)$ is that, for all $\mathbf{X}'$-morphisms $f\colon W \rightarrow X$ and $g\colon X \rightarrow Y$,

$$c(f\,;g)\,; S(g) \;\subseteq\; S(g)\,; c(f)$$

where the $c(\cdot)$ are *partial* functions on states, and the ordering is graph inclusion. This is *semi*-commutativity of

$$\begin{tikzcd} S(X) \arrow[r, "c(f)"] & S(X) \\ S(Y) \arrow[u, "S(g)"] \arrow[r, "c(f\,;\,g)"'] \arrow[ur, phantom, "\supseteq"] & S(Y) \arrow[u, "S(g)"'] \end{tikzcd}$$

which allows command meanings to be less-defined in more-restricted worlds. Notice that, in general, $F \to G_\perp$ is *not* isomorphic to $F \rightharpoonup G$ in functor categories. Furthermore, as before, an additional condition is needed to preclude non-local commands from changing local variables, or, more generally, from violating local constraints on state changes [20, 12].

An example will be helpful to illustrate the uniformity condition on $\rightharpoonup$. Consider a command meaning $c(\cdot) \in [\![\mathbf{comm}]\!](X \times V)$ corresponding to an assignment statement $x := x + 1$, where x denotes the local component in $W \times V$. The partial function for $c(\mathrm{id}_{W\times V})$ maps $\langle w, n\rangle \in W \times V$ to $\langle w, n+1\rangle$. But we also need to define $c(f)$ for any $\mathbf{X}'$-morphism with domain $W \times V$, including any restriction morphism. Suppose that we restrict to the world

$$Y = \{\langle w, v\rangle \in W \times V \mid v \text{ is even}\}.$$

If $f\colon W \times V \to Y$ is the restriction morphism and $c(f)\langle w, n\rangle$ is defined, then it must be equal to $c(\mathrm{id}_{W\times V})\langle w, n\rangle$ by the uniformity requirement on $S \rightharpoonup S$. But this is impossible, because $\langle w, n + 1\rangle$ is not in Y (assuming that n is even). Instead, $c(f)\langle w, n\rangle$ is undefined. More generally, if morphisms g and h are such that $s = S(h)s'$ and $c(g)s$ is defined but not in the range of $S(h)$, then $c(g\, ;\, h)s'$ must be undefined.

Now consider the composite $x := x + 1\, ;\, x := x + 1$, and its semantical counterpart $c\, ;\, c$, with c as above. Sequential composition is interpreted componentwise, so for command meanings c_1 and c_2, $(c_1\, ;\, c_2)(f)$ is just the compositition $c_1(f)\, ;\, c_2(f)$ of the partial functions for the components. Thus, we get that $(c\, ;\, c)(\mathrm{id}_{W\times V})\langle w, n\rangle$ is $\langle w, n + 2\rangle$. However, $(c\, ;\, c)f\langle w, n\rangle$ is undefined, where f is the restriction to Y, because $c(f)\langle w, n\rangle$ is undefined; the attempt to "stray" from Y, even at an intermediate state, leads to divergence.

This example shows how state-set restrictions and $\rightharpoonup$ interact: a command meaning at a restricted world terminates only if all the states encountered during "execution" are within the world. This is the key idea behind the semantics of non-interference in [20]. Given $c \in [\![\mathbf{comm}]\!]W$ and $e \in [\![\mathbf{exp}]\!]W$, we define

$$c \,\#\, e \iff \text{for all } f\colon W \to X \text{ and } s \in X,\ c(f)s = c(f\, ;\, g)s$$

where g is the restriction to

$$\{s' \in X \mid e(f)s' = e(f)s\}.$$

Intuitively, $c \,\#\, e$ holds iff any (terminating) execution of c can be restricted to a world for which the value of e is invariant. For discussion of axioms that make use of this invariance at intermediate states, see [17, 20].

In this model $[\![\mathbf{var}]\!]$ is again treated using the generalized form of variable discussed in the previous section. The rule for variable declarations can then be validated by showing that non-local entities don't interfere with the expression component of a local variable, and that no assignment to a local variable interferes with a non-local entity. For the generalization of # to higher types, and validation of Non-Interference Abstraction, see [12].

Our final example ties together these ideas about non-interference, restriction morphisms, and local variables:

$$\begin{array}{l}\mathbf{new}\,x.\;\; x := 0;\\ \quad P(x)\end{array}$$

is equivalent to $P(0)$, where x is not free in $P\colon \mathbf{exp} \rightarrow \mathbf{comm}$. This equivalence fails in other models. A sketch of its validity here is as follows. First, since P is non-local, none of its calls interfere with the local variable, unless the argument does. But the argument x is an expression, which doesn't cause any side effects, so $P(x)$ doesn't interfere with (the expression component of) x. Then, by the definition of non-interference given above, the result of evaluating $P(x)$ in a state $s \in W \times V$ is equivalent to evaluating the call in the restricted world

$$\{(w, v) \in W{\times}V \mid v = 0\},$$

using an appropriate restriction morphism. But in this restricted world identifier x and 0 have the same denotation: both are (families of) functions that return 0 in *any* state. This means that x can be replaced by 0 in the call, and the equivalence follows.

7 Discussion

We have concentrated here on models based explicitly on the possible-world approach of [16, 13]; but, it is interesting to note that, in a parallel line of development, Meyer and Sieber [8] have used methods that can be fairly said to have possible worlds at their core (combined with other principles for restricting models). If we count [8] as within the approach, then in fact the most satisfactory local-variable models are all based on possible worlds.

At present, there is no single "best" model for local variables. We have seen that the non-interference model validates a number of test equivalences that fail in other models. However, Arthur Lent has pointed out to us that there are equivalences that fail in this model because it is sensitive to intermediate states encountered during program execution, such as

$$\begin{array}{lcl}\mathbf{new}\,x.\;\; x := 0; & \equiv & \mathbf{new}\,x.\;\; x := 0;\\ \quad P(x := x + 2) & & \quad P(x := x + 1\,;\, x := x + 1)\end{array}$$

where x is not free in P: **comm** $\rightarrow$ **comm**. This fails in the non-interference model, essentially because the different arguments to P are not semantically equal, yet it *is* valid in the location-based model of Sections 2 and 3, the Reynolds/Oles state-set model of Section 5, and the models of [8].

Another significant point is state-sets *versus* locations. The major difference between the models of Sections 5 and 2 is that the former is predisposed against having a primitive type of locations in the language, while the latter is not. (Of course, whether one considers this a strength or weakness of the model depends on one's philosophy of what variables should be.)

An equivalence that fails in *all* known models is the following variation on Example 7 from [8]: execution of

$$\begin{array}{l}\mathbf{new}\,x.\ \ x := 0;\\ \quad P(x := x + 1)\end{array}$$

where x is not free in P: **comm** $\rightarrow$ **comm** might set x to any value; however, because P cannot *read* from x and x is discarded after execution of the body, the block should be equivalent to $P(\mathbf{skip})$. In each of the models discussed in this paper, a counterexample to this equivalence may be obtained by considering a procedure meaning p that, given a command argument c and a state s, terminates with result s if $c(s) = s$, and diverges otherwise. Intuitively, such a p reads from local variables, such as x in this example, because the operation of testing states for equality depends on their values.

The possible-world approach has resulted in models that better explain the sense in which a declared variable is "new." Nevertheless, the best current models are still far from perfect. It seems that the semantic ideas on support, non-interference, and preservation of invariants developed in [16, 13, 8, 20, 12] might prove to be just special cases of some more general and fundamental principle that is still lacking from our understanding of the interactions between non-local procedures and local variables in Algol-like languages.

Acknowledgements

We have benefitted from discussions with Eugenio Moggi, Arthur Lent, and Kurt Sieber, and comments on a draft by David Andrews.

References

[1] J. L. Bell. *Toposes and Local Set Theories.* Oxford University Press, 1988.

[2] S. D. Brookes. A fully abstract semantics and a proof system for an ALGOL-like language with sharing. In A. Melton, editor, *Mathematical Foundations of Programming Semantics*, volume 239 of *Lecture Notes in Computer Science*, pages 59–100, Manhattan, Kansas, 1985. Springer-Verlag, Berlin.

[3] J. Donahue. Locations considered unnecessary. *Acta Informatica*, 8:221–242, 1977.

[4] R. Goldblatt. *Topoi, The Categorial Analysis of Logic.* North-Holland, Amsterdam, 1979.

[5] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Conf. Record 11th ACM Symp. on Principles of Programming Languages*, pages 245–257, Austin, Texas, 1984. ACM, New York.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580 and 583, 1969.

[7] J. Lambek and P. J. Scott. *Introduction to Higher-Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

[8] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Conf. Record 15th ACM Symp. on Principles of Programming Languages*, pages 191–203, San Diego, California, 1988. ACM, New York.

[9] R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, and Wiley, New York, 1976.

[10] E. Nelson. On exponentiating exponentiation. *J. of Pure and Applied Algebra*, 20:79–91, 1981.

[11] P. W. O'Hearn. *The Semantics of Non-Interference: a Natural Approach.* Ph.D. thesis, Queen's University, Kingston, Canada, 1990.

[12] P. W. O'Hearn and R. D. Tennent. Semantical analysis of specification logic, part 2. Technical Report 91-304, Department of Computing and Information Science, Queen's University, Kingston, Canada, 1991. To appear in revised form in *Information and Computation.*

[13] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph.D. thesis, Syracuse University, Syracuse, 1982.

[14] F. J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 543–573. Cambridge University Press, Cambridge, 1985.

[15] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall International, London, 1981.

[16] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland, Amsterdam.

[17] J. C. Reynolds. Idealized Algol and its specification logic. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, Cambridge, 1982.

[18] D. S. Scott. Mathematical concepts in programming language semantics. In *Proc. 1972 Spring Joint Computer Conference*, pages 225–34. AFIPS Press, Montvale, N.J., 1972.

[19] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, and London, England, 1977.

[20] R. D. Tennent. Semantical analysis of specification logic. *Information and Computation*, 85(2):135–162, 1990.