Functor-Category Semantics of Programming Languages and Logics

by

R.D.Tennent

LFCS Report Series ECS-LFCS-86-3
May 1986

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Functor-Category Semantics of Programming Languages and Logics

R.D. Tennent[†]
Laboratory for the Foundations of Computer Science
Department of Computer Science
University of Edinburgh
Edinburgh, Scotland.

Abstract A category-theoretic technique for denotational-semantic description of programming languages has recently been developed by J.C. Reynolds and F.J. Oles. The first application was an "abstract" description of stack-oriented storage management in Algol 60-like programming languages. A more recent application has been to obtain a model of Reynolds's "specification logic" that is non-operational and validates certain intuitively-true axioms; this application required ideas from topos theory. This paper is an introduction to the Reynolds-Oles technique and its applications. A novel feature of the presentation is the systematic use in functor categories of analogues to conventional domain constructions.

In designing a programming language, the central problem is to organize a variety of concepts in a way which exhibits uniformity and generality. Substantial leverage can be gained in attacking this problem if these concepts can be defined concisely in a framework which has already proven its ability to impose uniformity and generality upon a wide variety of mathematics.

J.C. Reynolds

[†]On leave from the Department of Computing and Information Science, Queen's University, Kingston, Canada.

1. Motivations

1.1. Stack Implementability

Every programming-language implementor understands how to use a *stack* to manage storage for suitable programming languages. However, semanticists have found it difficult to capture an abstract notion of stack-implementability. Informally, a language is stack-implementable when it follows a last-in/first-out storage-allocation discipline and does not have "dangling references", that is to say, de-allocated variables are inaccessible. It is straightforward to describe last-in/first-out storage allocation using conventional denotational-semantic techniques [e.g., Tennent 1981]; however, verifying that there are no dangling references is very difficult [Milne and Strachey 1976, Tennent 1983, Halpern *et al.* 1983].

One difficulty is that state arguments of semantic valuations and procedure values range over a single "global" domain of states of unbounded size, rather than "local" domains of states of exactly the appropriate size; another problem is that environments bind identifiers to values that apparently can access any part of a (global) state. What is needed is a suitable way of *indexing* semantic domains by stack-depths or, for statically-typed languages, by state "shapes" specifying the "size" of each component of a stack.

1.2. Non-Interference in Specification Logic

Specification logic [Reynolds 1981a, 1982] is a programming logic for Algol 60-like languages with procedures. It is essentially a multi-sorted first-order theory with Hoare triples $\{P_0\}\ C\ \{P_1\}$ as atomic formulas and conventional logical operators such as conjunction, implication, and universal quantification. However, additional atomic formulas are needed to express certain kinds of assumptions about free identifiers, notably the "non-interference" formula $C \# E$, which is defined to be true just if every (terminating) execution of command C preserves the value of expression E invariant.

If this were to mean merely that the value of E *after* execution of C is the same as its initial value, there would be no semantic difficulty. However, $C \# E$ is intended to be interpreted in the stronger sense that *throughout* any terminating execution of C, the value of E remains the same as when the execution started. This cannot be specified in a conventional denotational-semantic description in which command meanings are functions or relations from initial-states to final-states, and so Reynolds [1981a] was forced to adopt an "operational" style of description for commands, using sequences of (intermediate) states. This is undesirable because it distinguishes between command meanings that follow different execution-paths, even if their "externally-observable" behaviour is identical.

A non-operational specification would be possible if one could specify which states are "allowable" for a command execution; for then the non-interference formula $C \# E$ could be interpreted as being true just if every terminating execution of C (within the allowed set of states) can *also* take place when the allowed set of states is further restricted to that subset for which E has the value it had initially. Again, it seems desirable to be able to *parameterize* semantic descriptions with respect to "local" constraints on what states are representable or allowed.

2. Functor-Category Semantics

In this section, we present the basic idea of a technique due to J.C. Reynolds [1981b] and F.J. Oles [1982,1985] which provides solutions to the problems of semantic description outlined above. Consider a phrase Z that conventionally would be interpreted so that $[\![Z]\!]$ is a function from a suitable domain of environments Env to a suitable domain of meanings M; that is,

$$\mathrm{Env} \xrightarrow{[\![Z]\!]} M$$

Suppose that x is an object that specifies some "local" aspect of storage structure. For example, for describing stack-oriented storage management, x would specify the stack-depth or state-shape; for describing non-interference, x would specify the set of allowed states. In logic, objects such as x are known as "possible worlds". Then, *both* the valuation function *and* the semantic domains should be "parameterized" with respect to possible worlds x:

$$\mathrm{Env}(x) \xrightarrow{[\![Z]\!]x} M(x)$$

This allows the environments and meanings to be tailored to whatever constraints on the local states might be appropriate. For example, the command meanings in a stack-implementable language might, as usual, be state-to-state functions, but the states should have just the correct depth or shape (as specified by x), and the environments should be constrained not to allow dangling references.

But of course the valuation functions and semantic domains for different possible worlds cannot be arbitrarily different from one another. To arrive at the appropriate uniformity condition, suppose that y is another possible world and that $f: x \to y$ specifies how to "change" from x to y. For example, if possible worlds are stack-depths, there would be a (unique) f for every y such that $x \leqslant y$. Or, if x and y are sets of allowed states and y is to be a *subset* of x (for example, the subset for which some expression has some value), f would be the function inserting y into x.

It is reasonable to require that, for any possible world x, there is a "null" change-of-possible-world $\mathrm{id}_x: x \to x$ and that, if $f: x \to y$ and $g: y \to z$ are changes-of-possible-worlds, there is a composite change $f;g: x \to z$ such that composition is an associative operation. In short, possible worlds and changes-of-possible-world must form a *category* X.

Now, every X-morphism $f: x \to y$ induces a change-of-meaning $M(f): M(x) \to M(y)$. For example, if f describes an increase in stack-depth, then the $M(f)$ for commands maps every command meaning c for the "small" stacks to a command meaning for the "larger" stacks: this would leave the new part of the stack unchanged while changing the old part of the stack like c. Similarly, $\mathrm{Env}(f)$ should do the same kind of thing component-wise for environments.

It is reasonable to require that these induced mappings on semantic domains should preserve identities and composites. In short, Env and M must be *functors* from X, a category of possible worlds, to a suitable category of semantic domains:

$$\begin{tikzcd}
x \arrow[d, "f"'] & \mathrm{Env}(x) \arrow[d, "\mathrm{Env}(f)"'] & M(x) \arrow[d, "M(f)"] \\
y & \mathrm{Env}(y) & M(y)
\end{tikzcd}$$

Finally, the condition that ensures uniformity of the valuations is that $[\![Z]\!]$ must be a *natural transformation* from Env to M; that is, the following diagram should commute for every phrase Z and X-morphism $f: x \rightarrow y$:

$$\begin{tikzcd}
x \arrow[d, "f"'] & \mathrm{Env}(x) \arrow[d, "\mathrm{Env}(f)"'] \arrow[r, "{[\![Z]\!]x}"] & M(x) \arrow[d, "M(f)"] \\
y & \mathrm{Env}(y) \arrow[r, "{[\![Z]\!]y}"'] & M(y)
\end{tikzcd}$$

Note that this picture reduces to the conventional one when X is the trivial (one-object and one-morphism) category.

To summarize, the basic idea of the Reynolds-Oles technique is to move from categories of domains and functions to *functor categories* $X \Rightarrow D$ whose objects are the functors from a (small) category X of possible worlds to a category D of semantic domains, and whose morphisms are the natural transformations of these functors.

3. Some Categories of Possible Worlds

In this section, we describe the categories of possible worlds that have been used to date in applications of the functor-category approach.

A category of possible worlds suitable for interpreting specification logic for a programming language without variable declarations [Tennent 1985] is as follows. The objects are sets $X,Y,...,$ interpreted as the sets of states *allowed* by the reasoning context; the morphisms from X to Y are the injective functions from Y to X. Intuitively, such a function maps every element of Y to the element of X that it represents; elements of X not in the range of the function are "unreachable" when executing in the more-restricted possible world. Composition of morphisms is just functional composition. An important kind of morphism in this category is "restriction to a subset": for X' a subset of X, the morphism $\lfloor X' : X \rightarrow X'$ is simply the insertion function from X' to X.

The category of possible worlds used by Oles [1982,1985] to model stack-implementable languages can be described as follows. The objects are again sets, now interpreted as the sets of states representable by the run-time stack; the morphisms from X to Y are pairs f,Q where f is a function from Y to X and Q is an equivalence relation on Y such that

$$\begin{tikzcd} X & Y \arrow[l, "f"'] \arrow[r, "q"] & Y/Q \end{tikzcd}$$

is a product diagram (in the category of sets and functions), where q maps every element of Y to its Q-equivalence class. Intuitively, f extracts the small stack embedded in a larger one, and Q relates large stacks with identical "extensions". This is just the category-theoretic way of saying that larger stacks are formed from smaller ones by adding independent components for local variables. The composition (in diagrammatic order) $f,Q;g,R : X \rightarrow Z$ of morphisms $f,Q : X \rightarrow Y$ and $g,R : Y \rightarrow Z$ has as its two components: the functional composition of f and g, and the equivalence relation on Z that relates $z_0,z_1 \in Z$ just if they are R-related and Q relates $g(z_0)$ and $g(z_1)$. An important kind of morphism in this category is "expansion by a set": if V is a set, then the morphism $\times V : X \rightarrow X \times V$ has as its components: the projection function from $X \times V$ to X, and the equivalence relation that relates $<x_0,v_0>$ and $<x_1,v_1>$ just if $v_0 = v_1$.

To allow for *both* expansions of state sets (e.g., to interpret variable declarations) *and* restrictions (e.g., to interpret non-interference formulas), the category of possible worlds must combine the properties of the two categories already described. The objects are pairs X,X' such that X is a set (interpreted as the set of representable states), and X' is a subset of X (interpreted as the subset allowed in the present context). A morphism from X,X' to Y,Y' is a pair of the form f,Q having the properties described in the preceding paragraph and the additional property that $f(Y')$ be included in X'; intuitively, the small state embedded in an *allowed* large state must itself be allowed. Composition of such morphisms is as described in the preceding paragraph. For any object X,X' in this "combined" category, there are both restriction morphisms $\rfloor X'' : X,X' \rightarrow X,X''$ for any subset X'' of X', and expansion morphisms $\times V : X,X' \rightarrow X\times V, X'\times V$ for any set V; their definitions should be evident.

4. Meaning Functors

In this section, we discuss how to define meaning functors $[\![\theta]\!]$ from categories $\mathbf{X}$ of possible worlds to categories $\mathbf{D}$ of domains for each of the phrase types θ for an Algol-like language and logic; these functors are generalizations of the "semantic domains" in a conventional denotational-semantic definition. Two categories of semantic domains are of particular interest: **Set** is the usual category of sets and (arbitrary) functions, and **Pdom** [Reynolds 1977] is the category of "pre-domains": directed-complete posets and continuous functions. The first of these is appropriate for "logical" phrase types, such as assertions and specifications, and the second for "computational" phrase types, such as expressions and commands. Both are Cartesian-closed and there are obvious embedding and forgetful functors between them.

It is sometimes possible to define meaning functors by using, for each component of the object part, the corresponding conventional-semantic domain with the usual "global" set of states replaced by a "local" set determined by the possible world. But this point-wise approach is inadequate for procedures and specification formulas, and it turns out to be much simpler consistently to use functorial analogues to the "domain constructions" of the conventional approach.

For example, there is a product operation $F_1 \times F_2$ on functors F_1 and F_2 which is derived point-wise from the product in **D**: for any X-object x and X-morphism $f: x \rightarrow y$,

$$(F_1 \times F_2)(x) = F_1(x) \times F_2(x)$$

and

$$(F_1 \times F_2)(f) = F_1(f) \times F_2(f) \quad .$$

Similarly, if, for any domain D and function f on domains, $D_\perp$ denotes D augmented by a new element $\perp$ (a new *least* element if the domain is a poset) and $f_\perp$ denotes the extension of f to the augmented domains that maps $\perp$ to $\perp$, then, for any functor F, there is a "lifted" functor $F_\perp$ such that $F_\perp(x) = F(x)_\perp$, and $F_\perp(f) = F(f)_\perp$. These operations will also be used on *contravariant* functors, yielding contravariant functors.

The functorial analogue to the function-domain construction is more sophisticated. To motivate this, consider that a procedure defined in possible world x might be called in any world y accessible from x using an X-morphism $f: x \rightarrow y$, and it is the storage structure determined by y which should be in effect when the procedure is executed. This suggests that the meaning of a procedure defined in possible world x must be a suitably-uniform *family* of functions, indexed by X-morphisms $f: x \rightarrow y$. So, for functors F_1 and F_2, $F_1 \rightarrow F_2$ is defined to be the functor whose object part is determined by

$$(F_1 \to F_2)(x)$$

$$= \{\ m \in \Pi\ f{:}\,x \to y\ .\ F_1(y) \to F_2(y)\ \mid$$

for all X-morphisms $f{:}\,x \to y$ and $g{:}\,y \to z$,

$$F_1(g)\ ;\ m(f;g) \ =\ m(f)\ ;\ F_2(g)\ \}\ ,$$

where the arrow in $F_1(y) \to F_2(y)$ denotes exponentiation in **D**.

Intuitively, a procedure meaning in possible world x is a family of functions (indexed by the changes of possible world from x) *from* the F_1-domain appropriate to the "changed" possible world, *to* the F_2-domain appropriate to that world. Furthermore, such a family must satisfy a natural-transformation-like uniformity condition: commutativity in **D** of all diagrams of the form

$$\begin{tikzcd}
F_1(y) \arrow[r, "m(f:x \to y)"] \arrow[d, "F_1(g:y \to z)"'] & F_2(y) \arrow[d, "F_2(g:y \to z)"] \\
F_1(z) \arrow[r, "m(f;g:x \to z)"'] & F_2(z)
\end{tikzcd}$$

This construction will also be used on contravariant functors; the appropriate uniformity condition is obtained by reversing the vertical arrows. In either case, the effect of $F_1 \to F_2$ on morphisms is defined as follows:

$$(F_1 \to F_2)(f{:}\,x \to y)(m \in (F_1 \to F_2)(x))(g{:}\,y \to z) = m(f;g)\ ,$$

so that $F_1 \to F_2$ is always covariant.

If **D** is **Set**, the functor category $\mathbf{X} \Rightarrow \mathbf{D}$ is Cartesian-closed for *any* **X** [Goldblatt 1979], and the construction just described is the exponentiation operation in that category.

Oles [1982] has shown that this is also the case if **D** is **Pdom** when the $(F_1 \rightarrow F_2)(x)$ are ordered point-wise.

We need another exponentiation-like operation, analogous to the construction of domains $D_1 \dashrightarrow D_2$ of *partial* functions in conventional semantics [Plotkin 1985]. For (covariant) functors F_1 and F_2, the object part is defined by

$$
\begin{aligned}
&(F_1 \dashrightarrow F_2)(x) \\
&= \{\; m \in \Pi\, f : x \rightarrow y \,.\, F_1(y) \dashrightarrow F_2(y) \;\mid \\
&\qquad\qquad \text{for all } \mathbf{X}\text{-morphisms } f : x \rightarrow y \text{ and } g : y \rightarrow z, \\
&\qquad\qquad\qquad F_1(g) \,;\, m(f;g) \;=\; m(f) \,;\, F_2(g) \\
&\qquad\qquad\qquad \text{whenever } m(f) \text{ has a defined result} \quad \} \quad .
\end{aligned}
$$

The uniformity condition for this construction only requires commutativity of

$$\begin{tikzcd}
F_1(y) \arrow[r, dashed, "m(f : x \rightarrow y)"] \arrow[d, "F_1(g : y \rightarrow z)"'] & F_2(y) \arrow[d, "F_2(g : y \rightarrow z)"] \\
F_1(z) \arrow[r, dashed, "m(f;g : x \rightarrow z)"'] & F_2(z)
\end{tikzcd}$$

when the result of the mapping along the top of the diagram is defined. For contravariant functors, the vertical arrows are reversed and commutativity is required only when the result of the mapping along the bottom of the diagram is defined. In either case, the morphism part of $F_1 \dashrightarrow F_2$ is defined in exactly the same way as that for $F_1 \rightarrow F_2$.

If **D** is **Set**, this construction yields a "representation of partial morphisms" for a "domain structure" that makes $\mathbf{X} \Rightarrow \mathbf{D}$ a *partial* Cartesian-closed category [Moggi 1986] for any category X, and also if **D** is **Pdom** when the $(F_1 \dashrightarrow F_2)(x)$ are ordered pointwise.

To construct the meaning functors for an Algol-like language using the above operations on functors, we start with functors that are analogous to "primitive" domains. We need (contravariant) constant functors V_τ such that, for all X-objects x, $V_\tau(x)$ is the set of values of data-type τ; for example, $V_{\text{Boolean}}(x)$ would be {true,false}. Also needed is a "states" functor S such that $S(x)$ is the set of states allowed in possible world x. For the categories of possible worlds described in Section 3, this functor is contravariant; for example, if X is the "combined" category described there, $S(f,Q: X,X' \to Y,Y')$ is evidently the restriction of the "projection" function $f \in Y \to X$ to domain $S(Y,Y') = Y'$ and co-domain $S(X,X') = X'$. Then, the meaning functors $[\![\theta]\!]$ for phrase types θ are defined in Table 1, where U denotes the (covariant) constant functor that maps X-objects to any one-element set. (We have simplified the presentation here by ignoring the fact that the appropriate category of domains is sometimes **Set** and sometimes **Pdom.**)

When X is the trivial category, these definitions yield the domains used in conventional denotational semantics. But, in general, meanings are X-morphism-indexed *families* of functions. For assertions, expressions, commands, acceptors and variables, such meanings are more complex than those in [Reynolds 1981b], [Oles 1982] and [Tennent 1985], but are more convenient because it is not necessary to use the morphism parts of meaning functors to define the families of functions "implicitly".

Note that assertions must always be properly truth-valued, but the meaning functors for expressions (including Boolean expressions) allow for non-definedness of evaluation. See [Tennent 1986] for a discussion of Hoare-style programming logics when the values of terms may be "undefined". The uniformity condition for $\to$ ensures that, during the evaluation of an expression or assertion at some state, its sub-expressions are evaluated at that same state (or an expansion of it); that is, there are no side effects to non-local variables, even "temporary" ones. The uniformity

θ	$[\![\theta]\!]$
assertions	$S \rightarrow V_{\text{Boolean}}$
expressions (of data-type τ)	$S \rightarrow V_{\tau\bot}$
commands	$S \dashrightarrow S$
acceptors (for data-type τ)	$V_\tau \times S \dashrightarrow S$
variables (of data-type τ)	$(V_\tau \times S \dashrightarrow S) \times (S \rightarrow V_{\tau\bot})$
procedures (of phrase-type $\theta_1 \rightarrow \theta_2$)	$[\![\theta_1]\!] \rightarrow [\![\theta_2]\!]$
specifications (i.e., formulas)	$U \dashrightarrow U$

Table 1. Meaning Functors

condition for $\dashrightarrow$ allows command execution in any possible world to be less-defined than in less-restricted possible worlds, but ensures that, when such execution is defined, it is consistent with execution in the less-restricted worlds.

Similarly, specification meanings are essentially X-morphism-indexed families of "partial elements", which are either undefined (interpreted as "false") or defined (interpreted as "true"); furthermore, if t is such a family and $t(f: x \rightarrow y)$ is true (i.e., defined) then the uniformity condition for $\dashrightarrow$ requires $t(f;g)$ to be true for all $g: y \rightarrow z$. Thus, t is (the characteristic function of) a "sieve on x", and the meaning functor for specifications is the truth-value object in the topos $\mathrm{X} \Rightarrow \mathbf{Set}$ [Goldblatt 1979].

Many variations are possible. If the language has non-deterministic commands, a "relational"-style semantics could be given by adopting $S \times S \dashrightarrow U$ as the meaning functor for commands, where here U would be contravariant. Programming languages with jumps require "continuation" semantics; for example, the meaning functor for commands would be $C \rightarrow C$, where, for any (contravariant) functor A for "answers", C, the functor for command continuations, would be $S \dashrightarrow A$ in the deterministic case, and $S \times A \dashrightarrow U$ in the non-deterministic case.

5. Semantic Valuations

In this section, we present "semantic equations" for typical logical and programming-language constructs. We begin by considering the purely-logical forms of specification. As one would expect, the valuation for specification conjunction (for *any* category X of possible worlds) is

$$[\![S_1 \ \& \ S_2]\!]xuf = [\![S_1]\!]xuf \text{ and } [\![S_2]\!]xuf \quad ,$$

where here, and throughout this section, x is an X-object, u is an environment appropriate to x and to the free identifiers of the phrase, and $f: x \rightarrow y$ is an X-morphism with domain x. The analogous treatment of specification implication does *not* work in general. It is known from topos theory [Goldblatt 1979] that the following non-classical semantics is needed:

$$[\![S_1 \Rightarrow S_2]\!]xuf$$

$$= \text{for all } g: y \to z, \text{ if } [\![S_1]\!]xu(f;g) \text{ then } [\![S_2]\!]xu(f;g) \quad .$$

The "implicit quantification" over changes of possible world is needed in general to satisfy the uniformity condition on specification meanings.

The valuation above provides a solution to a difficulty that Reynolds [1982] had with the interpretation of specification logic: showing the soundness of certain intuitively-true axioms which use non-interference formulas and implications as assumptions. One of these is the Strong Constancy axiom. Consider, first, the following weaker axiom:

$$C \# P \Rightarrow \{P\}\ C\ \{P\}$$

It asserts that if an assertion P is true before executing a command C, and is not interfered with by the command, then it will be true after execution of the command. This does not take advantage of the strong interpretation of non-interference: the assertion P is true *throughout* the execution, not merely *after* it.

To take full advantage of the non-interference assumption, first use the axiom of Specification Conjunction to obtain

$$C \# P \ \&\ \{P_0\}\ C\ \{P_1\} \Rightarrow \{P \textbf{ and } P_0\}\ C\ \{P \textbf{ and } P_1\}$$

Then strengthen this by weakening the second assumption, as follows:

$$C \# P \ \&\ (\{P\} \Rightarrow \{P_0\}\ C\ \{P_1\}) \Rightarrow \{P \textbf{ and } P_0\}\ C\ \{P \textbf{ and } P_1\}$$

where $\{P\}$ is an abbreviation for $\{\textbf{true}\}\ \textbf{skip}\ \{P\}$; i.e, P holds for all (allowable) states. Intuitively, this axiom (called "Strong Constancy") asserts that, if P holds before executing C and C preserves the truth of P throughout the execution, then it is possible to assume that P is a "local" mathematical fact in reasoning about C, because all states that might be encountered during any such execution of C must satisfy P. However, using the *classical* interpretation of the logical connectives, the axiom is invalid because P might not satisfy *all* allowable states.

With the non-classical interpretation of implication given above, one merely considers the change of possible world which is the restriction to the subset of states that satisfy assertion P. The second assumption of the axiom then ensures that P_1 will hold after execution of C (where, by hypothesis, P and P_0 hold before). The axiom of Leftside Non-Interference Composition [Reynolds 1982] also uses non-interference and implication formulas as assumptions, and can be validated in the same way.

The valuations for equivalence and universal quantification are similarly non-classical:

$$[\![Z_1 \equiv_\theta Z_2]\!]xuf$$

$$= \text{for all } g : y \to z,\ [\![\theta]\!](f;g)([\![Z_1]\!]xu) = [\![\theta]\!](f;g)([\![Z_2]\!]xu)$$

$$[\![\forall I : \theta . S]\!]xuf$$

$$= \text{for all } g : y \to z \text{ and } m \in [\![\theta]\!]z,\ [\![S]\!]z[\mathrm{Env}(f;g)u \mid I : m](\mathrm{id}_z) \quad ,$$

where Env is the appropriate environment functor, constructed as the product of the meaning functors for the free variables, and $[u \mid I : m]$ denotes the environment that is like u except that the I component is m. In general, these interpretations validate *intuitionistic*, rather than classical, logical rules.

The valuations for procedural abstraction and application are as follows:

$$[\![\lambda I:\theta.Z]\!]xuf(m \in [\![\theta]\!]y) = [\![Z]\!]y[\mathrm{Env}(f)(u)\,|\,I:m]$$

$$[\![Q(Z)]\!]xu = [\![Q]\!]xu(\mathrm{id}_x)([\![Z]\!]xu)$$

These are also applicable with *any* category of possible worlds.

Direct-semantic valuations for some of the forms of command and the atomic formulas of specification logic are given in Table 2. In the equation for sequential composition, the semi-colon on the right denotes composition of partial functions (in diagrammatic order). The valuation for the non-interference formula is only applicable when the category of possible worlds has restriction morphisms. It asserts that execution of command C in possible worlds for which the value of expression E is invariant is not less defined than unconstrained execution of C, because the uniformity condition on command meanings already ensures that the equation holds when execution in the constrained world terminates. Non-interference for assertions is similar.

Valuations for variable-declaration blocks may be given along the lines laid down in [Reynolds 1981b] and [Oles 1982]. Let X be the "combined" category of possible worlds of Section 3. We first define "expansion" functors $\exp_\tau : \mathrm{X} \to \mathrm{X}$ for each data-type τ as follows:

$$\exp_\tau(X,X') = X\times V, X'\times V \text{ where } V = V_\tau(X,X') \quad ,$$

and

$$\exp_\tau(f,Q: X,X' \to Y,Y') = f_\tau, Q_\tau$$

$$\text{where } f_\tau\langle y_0,v_0\rangle = \langle f(y_0),v_0\rangle$$

$$\text{and } \langle y_0,v_0\rangle Q_\tau \langle y_1,v_1\rangle \text{ iff } y_0 Q y_1 \quad .$$

For any X-morphism $f,Q: X,X' \to Y,Y'$, the following diagram commutes:

$$[\![\mathbf{skip}]\!]xuf = \mathrm{id}_{S(y)}$$

$$[\![C_1;C_2]\!]xuf = [\![C_1]\!]xuf \ ; \ [\![C_2]\!]xuf$$

$$[\![A :=_\tau E]\!]xufy_0 = \begin{cases} [\![A]\!]xuf\langle v,y_0\rangle, \text{ if } [\![E]\!]xufy_0 = v \in V_\tau(y), \\ \text{undefined, otherwise} \end{cases}$$

$[\![\{P_0\} \ C \ \{P_1\}]\!]xuf$

$= \text{for all } y_0 \in S(y), \text{ if } [\![P_0]\!]xufy_0 \text{ and } [\![C]\!]xufy_0 = y_1 \in S(y)$

$\quad \text{then } [\![P_1]\!]xufy_1$

$[\![C \ \#_\tau \ E]\!]xuf$

$= \text{for all } g : y \to z \text{ and } v \in V_{\tau\perp}(z),$

$$S(|Z_v) \ ; \ [\![C]\!]xu(f;g) = [\![C]\!]xu(f;g;|Z_v) \ ; \ S(|Z_v)$$

$$\text{where } Z_v = \{z_v \in S(z) \mid [\![E]\!]xu(f;g)z_v = v\}$$

Table 2. Commands

$$\begin{tikzcd}
X,X' \arrow[r, "\times V_\tau(X,X')"] \arrow[d, "f{,}Q"'] & \exp_\tau(X,X') \arrow[d, "\exp_\tau(f{,}Q)"] \\
Y,Y' \arrow[r, "\times V_\tau(Y,Y')"'] & \exp_\tau(Y,Y')
\end{tikzcd}$$

Then, the expression and acceptor components of a "new" variable of data-type τ in an expanded possible world $\exp_\tau(X,X')$ should be $e_\tau(X,X')$ and $a_\tau(X,X')$ such that, for every X-morphism $g,R: \exp_\tau(X,X') \rightarrow Z,Z'$,

$$e_\tau(X,X')(g,R)(z_0 \in Z') = v \text{ where } <x_0,v> = g(z_0) \quad ,$$

and

$$a_\tau(X,X')(g,R)<v_1, z_0 \in Z'>$$

$$= \begin{cases} z_1, \text{ if } z_1 \in Z' \\ \qquad \text{for the } z_1 \in Z \text{ such that } g(z_1) = <x_0,v_1> \text{ and } z_0Rz_1 \\ \qquad\qquad \text{where } <x_0,v_0> = g(z_0), \\ \text{undefined, otherwise.} \end{cases}$$

In the definition of a_τ, the state z_1 satisfying the two conditions must exist and be unique by the "product" property of X-morphism g,R. Intuitively, the effect of assigning a value v_1 to the acceptor is to replace the old value v_0 in the appropriate component of the stack by v_1, without changing more-local components (ensured by using R) or more-global components (i.e., x_0); however, if the resulting state happens to be disallowed in the current possible world, the assignment fails to terminate.

Then, a block command declaring a local τ-variable may be interpreted as follows:

$$[\![\mathbf{new}\ I\!:\tau\ \mathbf{in}\ C]\!]xufy_0$$

$$= \begin{cases} y_1, & \text{if } [\![C]\!](\exp_\tau(x)) \\ & \qquad [\mathrm{Env}(\times V_\tau(x))(u)\,|\,I\!: <a_\tau(x),e_\tau(x)>] \\ & \qquad (\exp_\tau(f)) \\ & \qquad <y_0,v_0> \\ & \quad = <y_1,v_1> \in S(\exp_\tau(y)) \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

where v_0 is a "standard" initial value for variables of type τ.

A similar valuation may be used for a form of block *expression* in which the value of the local variable after execution of command C is used as the value of the whole construct:

$$[\![\mathbf{result}\ I\!:\tau\ \mathbf{of}\ C]\!]xufy_0$$

$$= \begin{cases} v_1, & \text{if } [\![C]\!](\exp_\tau(x)) \\ & \qquad [\mathrm{Env}(\times V_\tau(x))(u)\,|\,I\!: <a_\tau(x),e_\tau(x)>] \\ & \qquad (\exp_\tau(f;|\{y_0\})) \\ & \qquad <y_0,v_0> \\ & \quad = <y_0,v_1> \in \{y_0\} \times V_\tau(x), \\ \perp, & \text{otherwise.} \end{cases}$$

Side effects to non-local variables are prevented by the restriction to a possible world z in which y_0, the initial state, is the *only* allowed state:

$$\begin{tikzcd}
x \arrow[r, "\times V_{\tau}(x)"] \arrow[d, "f"'] & \exp_{\tau}(x) \arrow[d, "\exp_{\tau}(f)"] \\
y \arrow[r, "\times V_{\tau}(y)"] \arrow[d, "\lceil\{y_0\}"'] & \exp_{\tau}(y) \arrow[d, "\exp_{\tau}(\lceil\{y_0\})"] \\
z \arrow[r, "\times V_{\tau}(z)"'] & \exp_{\tau}(z)
\end{tikzcd}$$

It would be reasonable for a compiler to *warn* the programmer if identifiers other than *I* had free command-like occurrences in *C*. But to deem this a syntactic *error* would create the same kind of difficulty that Reynolds [1978] had: syntactic well-formedness would not be invariant with respect to beta equivalence.

6. Discussion

There have been just two applications of the functor-category technique to date, but it seems very likely that others will be found. "Invariancy" properties can be difficult to prove with conventional denotational-semantic descriptions [Milne and Strachey 1976], and possible-world semantics provides a new and flexible tool for such problems.

Acknowledgements

Gordon Plotkin suggested that meaning functors should be constructed systematically, and Eugenio Moggi helped with the realization of this suggestion. Financial support was provided by an operating grant from the Natural Sciences and Engineering Research Council of Canada, and a fellowship from the Alvey Directorate and the British Science and Engineering Research Council. I am grateful to Robin Milner for arranging the fellowship, and for his hospitality.

References

[Goldblatt 1979]

R. Goldblatt, *Topoi, The Categorial Analysis of Logic,* North-Holland (1979, 2nd edition 1984).

[Halpern *et al.* 1983]

J.Y. Halpern, A.R. Meyer and B.A. Trakhtenbrot, "The semantics of local storage, or what makes the free-list free?", *Conf. Record 11th ACM Symp. on Principles of Programming Languages,* pp. 245-257, ACM, New York (1983).

[Milne and Strachey 1976]

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

[Moggi 1986]

E. Moggi, "Categories of partial morphisms and the λ_p-calculus", this volume.

[Oles 1982]

F.J. Oles, *A Category-Theoretic Approach to the Semantics of Programming Languages,* Ph.D. dissertation, Syracuse University (1982).

[Oles 1985]

F.J. Oles, "Type algebras, functor categories and block structure", in *Algebraic Methods in Semantics* (M. Nivat and J.C. Reynolds, eds.), pp. 543-573, Cambridge University Press (1985).

[Plotkin 1985]

G.D. Plotkin, "Types and partial functions", lecture notes, Computer Science Department, University of Edinburgh.

[Reynolds 1977]

J.C. Reynolds, "Semantics of the domain of flow diagrams", *J.ACM* 24 (3), pp. 484-503 (1977).

[Reynolds 1978]

J.C. Reynolds, "Syntactic control of interference", *Conf. Record 5th ACM Symp. on Principles of Programming Languages*, pp. 39-46, ACM, New York (1978).

[Reynolds 1981a]

J.C. Reynolds, *The Craft of Programming*, Prentice-Hall International, London (1981).

[Reynolds 1981b]

J.C. Reynolds, "The essence of Algol", in *Algorithmic Languages* (J.W. de Bakker and J.C. van Vliet, eds.), pp. 345-372, North-Holland (1981).

[Reynolds 1982]

J.C. Reynolds, "Idealized Algol and its specification logic", in *Tools and Notions for Program Construction* (D. Nèel, ed.), pp. 121-161, Cambridge University Press (1982); also Report 1-81, School of Computer and Information Science, Syracuse University (1981).

[Tennent 1981]

R.D. Tennent, *Principles of Programming Languages*, Prentice-Hall International, London (1981).

[Tennent 1983]

R.D. Tennent, "Semantics of interference control", *Theoretical Computer Science* 27, pp. 297-310 (1983).

[Tennent 1985]

R.D. Tennent, "Semantical analysis of specification logic (preliminary report)", in *Logics of Programs 1985* (R. Parikh, ed.), Lecture Notes in Computer Science, Vol. 193, pp. 373-386, Springer (1985).

[Tennent 1986]

R.D. Tennent, "A note on undefined expression values in programming logics", submitted for publication.