LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Semantic Analysis of Specification Logic

by

R. D. Tennent

LFCS Report Series ECS-LFCS-86-5
June 1986

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

Semantical Analysis of Specification Logic

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

Abstract The "specification logic" of J. C. Reynolds is a programming logic for Algol 60-like languages with procedures. It is interpreted here as an intuitionistic theory. The model is based on a form of possible-world semantics first used for programming-language interpretation by Reynolds and F. J. Oles to treat stack-oriented storage management.

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

1. Introduction

In the beginning, C. A. R. Hoare (1969) created a programming logic for specification triples of the form $\{P_0\}\ C\ \{P_1\}$. And Hoare's logic is good (for simple imperative programming languages without procedures or jumps). It is both *sound* (Hoare and Lauer 1974) and, independently of how the vexed question of its completeness might be answered, *usable* (Reynolds 1981a), providing a framework for rigorous specification, development, and verification of practical programs.

Many attempts have been made to extend Hoare's logic to languages with procedures or jumps. The "specification logic" of J. C. Reynolds (1981a, 1982) is perhaps the most important of these, because its generality, usability, and coherence have been clearly demonstrated. The aim of the research described here is to do the same for its *soundness*.

Specification logic is essentially a many-sorted first-order theory, with Hoare triples as atomic formulas and conventional logical connectives, such as conjunction, implication, and quantification. There are some additional atomic formulas to permit expression of certain kinds of assumptions about free identifiers, such as *non-interference*. A fairly conventional semantics for specification logic is outlined in Reynolds (1981a); however, there are two problems with this model.

The first difficulty is that commands are interpreted in an undesirably operational way: command meanings are functions from an initial state to the (possibly infinite) sequence of *all* states encountered during execution of the command. This kind of interpretation is undesirable because it distinguishes between commands whose "externally-observable" behaviour is identical. But it seemed to be necessary in order to interpret non-interference formulas: $C \# E$ asserts that the value of expression E is invariant *throughout* any (terminating) execution of command C.

The second difficulty is that Section 11 of Reynolds (1982) proposes two axioms, Strong Constancy and Leftside Non-interference Composition, which are intuitively *true* and seem to be very desirable or essential for verifying certain kinds of programs, yet which are *invalid* relative to the interpretation in Reynolds (1981a).

This paper describes a new approach to capturing the intended interpretation of specification logic. The rest of this introduction gives an informal (and simplified) presentation of the model.

The first idea is adapted from the treatment of block structure described by Reynolds (1981b) and Oles (1982, 1985): the semantics of a phrase is a suitably-related *family* of environment-to-meaning functions for different sets of allowable states. For example, the

semantics of a command for any state-set X is a continuous function from environments appropriate for X to partial functions (or binary relations) on X. For a control structure such as $C_1;C_2$, the meaning for any state-set is expressed as a function of the meanings of the immediate constituents C_1 and C_2 for the *same* state-set. This ensures that "intermediate" states between the executions of C_1 and C_2 will also belong to that state-set, without requiring that these intermediate states be explicit in command meanings. A specification now becomes a predicate about state-sets as well as environments. In logic, this is known as "possible-world" semantics; here, each state-set is a possible world.

A satisfactory interpretation of $C \# E$ for any possible world X is then definable as follows. For any value v that E might have, let X_v be the subset of X for which the value of E is v; then $C \# E$ if and only if, for all v, any terminating execution of C in X whose initial state is in X_v, is also a terminating execution of C in X_v.

Consider now the proposed axiom of Strong Constancy (Reynolds 1981a):

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

where C is a command, P, P_0 and P_1 are assertions, and the static-assertion specification $\{P\}$ is equivalent to $\{\textbf{true}\}$ **skip** $\{P\}$. Intuitively, if P holds before executing C and C does not interfere with P, then, while reasoning about C, it should be possible to treat P as a "local" mathematical fact. However, $\{P\}$ holds just if assertion P is true at *all* allowable states, not merely those that might be encountered while executing C. So Strong Constancy is invalid according to a conventional interpretation. There is a similar problem with the proposed axiom scheme of Leftside Non-interference Composition.

The essence of the solution proposed here is to adopt a non-classical interpretation of specification implication, inspired by Kripke's (1965) semantics for *intuitionistic* logic (Dummett 1977, van Dalen 1983). The reason for interpreting specification logic as an intuitionistic theory is to take advantage of what McCarty (1984) has termed *axiomatic freedom*,

> the recognition that intuitionistic logic allows axioms which are classically false but mathematically efficient to be consistent with powerful theories.

In fact, the equivalences, axioms, and rules presented by Reynolds are intuitionistically acceptable. (*Reductio ad absurdum* is not the correct name for rule R10 in Reynolds 1981a.) Furthermore, Reynolds has shown that if the classical but non-intuitionistic law $\neg\neg S \Rightarrow S$ were added to specification logic, then a formula asserting non-termination

of a simple assignment command would be derivable; this shows that the only models possible for a *classical* version of specification logic would be trivial ones in which commands do not terminate unless they leave the state unchanged. Reynolds's example is reproduced in an Appendix to this paper.

The new interpretation of the implication connective is essentially as follows: define $S_1 \Rightarrow S_2$ to be true for X just if, for all subsets X' of X, S_2 holds for X' whenever S_1 holds for X'. The implicit quantification over restricted possible worlds is needed in general to preserve the *monotonicity* of specification interpretation in the following sense: if any specification S holds for possible world X, then S holds for any more-restricted possible world X'.

To see how this interpretation of the implication connective helps with Strong Constancy, let X' be the subset of state-set X for which P holds, and suppose that execution of C in X can map initial state x_0 to final state x_1, where x_0 satisfies both P and P_0. Then assumption $C \# P$ ensures that the execution from x_0 to x_1 can also take place in X', so that x_1 satisfies P. Furthermore, with the new interpretation of implication, assumption $\{P\} \Rightarrow \{P_0\}\ C\ \{P_1\}$ ensures that x_1 also satisfies P_1, because, by definition, P holds for all states in X'. This validates Strong Constancy.

Similarly, consider the following weak form of the axiom scheme of Leftside Non-interference Composition (Reynolds 1982):

$$C \# E \ \&\ (I \# E \Rightarrow \{P_0\}\ C\ \{P_1\}) \Rightarrow \{P_0\}\ C\ \{P_1\} \quad ,$$

where I is a command identifier with a command-like free occurrence in command C, P_0 and P_1 are assertions, and E is an expression. Suppose again that execution of C in possible world X can map x_0 to x_1, where x_0 satisfies P_0. Let X' be the subset of X for which E has the same value as it does at x_0. Then assumption $C \# E$ ensures that the execution from x_0 to x_1 can also take place in X'. Because of examples like $(\lambda c\colon \mathbf{comm.\ skip})(c)$, it does *not* in general follow from $C \# E$ that $I \# E$; however, in possible world X' execution of I cannot interfere with E, so that, using the new interpretation of implication, the second assumption ensures that x_1 satisfies P_1. The same approach may be used to validate the stronger form of the axiom given in Reynolds (1982) involving all identifiers having command-like free occurrences in C.

In order to treat procedures and variable declarations, the model described formally in the rest of this paper is based on the category-theoretic formulation of possible-world semantics (Goldblatt 1979). In

this framework, possible worlds and changes of possible world form a *category*, **X**, and "semantic domains" are replaced by *functors* from **X** to either **S**, the usual category of sets and functions, for "logical" phrase types, or a category **D** of directed-complete posets and continuous functions for the "computational" phrase types. Then each phrase, Z, of the language is interpreted as a *natural transformation*, $[\![Z]\!]$, from an appropriate "environment" functor, Env, to an appropriate "meaning" functor, M, so that, if x and y are possible worlds and $f: x \rightarrow y$ is a change of possible worlds, the following diagram commutes in **S** or **D**:

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

2. Syntax

The type structure for the language is given in Table 1. For simplicity, we have avoided coercions and conventional variables (but retained *acceptors*, which are the "updating" components of variables). Informally, a *data type*, τ, denotes a set of values appropriate for some acceptor or expression, whereas a *phrase type*, θ, α, β or γ, denotes a set or poset of meanings appropriate for some kind of phrase. Assertional phrase types and specifications are distinguished from ordinary phrase types because meanings of "logical" phrases need not be computable. All phrase types except specifications are *bindable*; i.e., their meanings are denotable by identifiers.

The abstract syntax of the basic language of specification logic is given in Table 2. A *type assignment*, π, is a function from a finite set of identifiers, $\mathrm{dom}(\pi)$, to (bindable) phrase types. The notation $(\pi; I: \beta)$ denotes the type assignment π' such that

$$\mathrm{dom}(\pi') = \mathrm{dom}(\pi) \cup \{I\} \text{ , and}$$

$$\pi'(I') = \begin{cases} \beta, \text{ if } I' = I, \\ \pi(I'), \text{ otherwise,} \end{cases}$$

and similarly for $(\pi; I_1: \beta_1; \ldots; I_n: \beta_n)$. The same kind of notation will be used for extending environments.

Metavariables

τ	data types
θ	ordinary phrase types
α	assertional phrase types
β	bindable phrase types
γ	general phrase types

Productions

θ ::= τ-**exp**	expressions
| τ-**acc**	acceptors
| **comm**	commands
| $\theta \rightarrow \theta$	ordinary procedures
α ::= **assert**	assertions
| $\beta \rightarrow \alpha$	assertion procedures
β ::= θ | α	
γ ::= β	
| **spec**	specifications

Table 1. Phrase Types

$<(\pi; I:\beta) \vdash \beta>$ ::= I	identifier
$<\pi \vdash \theta \rightarrow \theta'>$ ::= $\lambda I:\theta.<(\pi; I:\theta) \vdash \theta'>$	ordinary abstraction
$<\pi \vdash \theta>$ ::= $<\pi \vdash \theta' \rightarrow \theta>(<\pi \vdash \theta'>)$	ordinary application
$<\pi \vdash \beta \rightarrow \alpha>$ ::= $\lambda I:\beta.<(\pi; I:\beta) \vdash \alpha>$	assertional abstraction
$<\pi \vdash \alpha>$ ::= $<\pi \vdash \beta \rightarrow \alpha>(<\pi \vdash \beta>)$	assertional application
$<\pi \vdash \mathbf{assert}>$ ::= **true** | **false**	constants
| **not** $<\pi \vdash \mathbf{assert}>$	negation
| $<\pi \vdash \mathbf{assert}>$ **and** $<\pi \vdash \mathbf{assert}>$	conjunction
| $\forall I:\tau.<(\pi; I:\tau\text{-exp}) \vdash \mathbf{assert}>$	universal quantification
| $\mathrm{D}_\tau(<\pi \vdash \tau\text{-exp}>)$	definedness
| $<\pi \vdash \tau\text{-exp}> =_\tau <\pi \vdash \tau\text{-exp}>$	equality
| ...	
$<\pi \vdash \mathbf{comm}>$ ::= **skip**	null
| $<\pi \vdash \mathbf{comm}>$; $<\pi \vdash \mathbf{comm}>$	sequencing
| $<\pi \vdash \tau\text{-acc}> :=_\tau <\pi \vdash \tau\text{-exp}>$	assignment
| **new** τ **var** I,I' **in** $<(\pi; I:\tau\text{-acc}; I':\tau\text{-exp}) \vdash \mathbf{comm}>$	variable declaration
| ...	
$<\pi \vdash \mathbf{spec}>$ ::= **absurd**	absurdity
| $<\pi \vdash \mathbf{spec}>$ & $<\pi \vdash \mathbf{spec}>$	conjunction
| $<\pi \vdash \mathbf{spec}> \Rightarrow <\pi \vdash \mathbf{spec}>$	implication
| $\forall I:\beta.<(\pi; I:\beta) \vdash \mathbf{spec}>$	quantification
| $<\pi \vdash \gamma> \equiv_\gamma <\pi \vdash \gamma>$	equivalence
| $\{<\pi \vdash \mathbf{assert}>\}$ $<\pi \vdash \mathbf{comm}>$ $\{<\pi \vdash \mathbf{assert}>\}$	Hoare-triple
| $<\pi \vdash \mathbf{comm}> \#_\tau <\pi \vdash \tau\text{-exp}>$	expression non-interference
| $<\pi \vdash \mathbf{comm}> \# <\pi \vdash \mathbf{assert}>$	assertion non-interference

Table 2. Abstract syntax

The phrase-class meta-variable $<\pi \vdash \gamma>$ ranges over well-formed phrases having type γ when the types of their free identifiers are

given by π. Productions for expressions and acceptors (other than the "generic" ones for identifiers and applications) and more complex control structures, such as conditionals, recursion and loops, are omitted. Jumps will be discussed briefly in Section 6. A "variable" declaration binds a pair of (distinct) identifiers to suitably-related acceptor and expression meanings, respectively.

Assertions are conventional first-order formulas over the data types, but non-terminating and partially-defined expressions (*not* assertions) are allowed, so there are also "definedness" predicates $\mathbf{D}_\tau(\cdot)$ for each data type τ.

The productions for specifications show that the specification language of Reynolds has been augmented by

(i) a constant **absurd**, which *never* holds (even for the null state-set, unlike the static-assertion specification {**false**} used for this purpose in Reynolds 1981a, 1982); and

(ii) an atomic formula for equivalence of phrases.

There should be no difficulty adding logical operators for disjunction and existential quantification, but there seems to be no reason to do so. The negation of S is definable as $S \Rightarrow$ **absurd**, and $\{P\}$, the static-assertion form of specification, is definable as {**true**} **skip** $\{P\}$. The "good-variable" specification and non-interference specifications for acceptors and procedures are also definable; see Reynolds (1982).

3. A Category of Possible Worlds

In the introduction, possible worlds were simply sets of allowable states, and changes of possible world were restrictions to subsets. But if the programming language has variable declarations, it must also be possible to *expand* the set of states, as in Reynolds (1981b) and Oles (1982, 1985). The appropriate generalization seems to be the following category. 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 this possible world). A morphism from X,X' to Y,Y' is a pair f,Q having the following properties:

(i) f is a function from Y to X;

(ii) Q is an equivalence relation on Y;

(iii) the following is a product diagram in the category of sets and functions:

$$X \xleftarrow{\ f\ } Y \xrightarrow{\ q\ } Y/Q$$

where q maps elements of Y to their Q-equivalence class; and

(iv) $f(Y')$ is a subset of X'.

The composition $f,Q\ ;\ g,R: X,X' \rightarrow Z,Z'$ of morphisms $f,Q: X,X' \rightarrow Y,Y'$ and $g,R: Y,Y' \rightarrow Z,Z'$ 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 $g(z_1)$.

For any object X,X' in this category, there is a restriction morphism $|X'': X,X' \rightarrow X,X''$ for any subset X'' of X', and an expansion morphism $\times V: X,X' \rightarrow X\times V,X'\times V$ for any set V. The restriction morphism $|X''$ has as its components: the identity function on X, and the universal binary relation on X. The expansion morphism $\times V$ has as its components: the projection function from $X\times V$ to X, and the equivalence relation that relates $<x_0,v_0>$ and $<x_1,v_1>$ in $X\times V$ just if $v_0 = v_1$.

4. Semantic Domains and Functors

In this section we define the "semantic domains" for our model. We begin by defining a number of constructions in categories of sets, (Scott) domains, and functors. Let **S** be the usual category of sets and functions. Let **D** be the category of directed-complete partial orders and continuous functions (Reynolds 1977); the objects of this category will be termed *domains*, whether or not they have least elements. Throughout, "id_x" will denote the identity morphism on object x and ";" will denote composition of morphisms or functors in diagrammatic order.

For any domains D and D', we define new domains as follows.

(i) product
$D \times D'$ is the Cartesian product of D and D', ordered component-wise.

(ii) lifting
$D_\perp$ is (an isomorphic copy of) D augmented by a new least element, denoted $\perp$.

(iii) exponentiation
$D \rightarrow D'$ is the poset of all continuous functions from D to D', ordered point-wise.

(iv) partial exponentiation
$D \rightsquigarrow D'$ is the poset of all continuous partial functions from D to D', ordered point-wise.[1]

---

[1]The following definitions are from Plotkin (1985): a partial function from D to D' is continuous just if the inverse image of any Scott-open subset of D' is Scott-open in D, where a subset U of a domain D is Scott-open just if, for every $u \in U$,

We also use exponentiations in category **S**: if S and S' are sets, $S \to S'$ is the set of all functions from S to S', and $S \rightsquigarrow S'$ is the set of all partial functions from S to S'.

We now define analogues of set and domain constructions in categories of *functors*. For *any* (small) category **X**, let $\mathbf{X} \Rightarrow \mathbf{D}$ be the category of all functors from **X** to **D**, with all natural transformations as the morphisms, and similarly for $\mathbf{X} \Rightarrow \mathbf{S}$. We use the following notation for products of families of sets. Suppose $S'(i)$ is a set (domain) for each $i \in S$; then $\prod i \in S.S'(i)$ denotes the set of functions f from S to the union of the (underlying sets of the) $S'(i)$ such that, for all $i \in S$, $f(i) \in S'(i)$. Let F and F' be functors from **X** to **D**, x and y be **X**-objects, and $f: x \to y$ and $g: y \to z$ be **X**-morphisms. Then the four constructions needed are as follows.

(i) product

$$(F \times F')(x) = F(x) \times F'(x) \text{ , and}$$

$$(F \times F')(f)(<d,d'> \in F(x) \times F'(x)) = <F(f)(d),F'(f)(d')>$$

(ii) lifting

$$F_\perp(x) = [F(x)]_\perp \text{ , and}$$

$$F_\perp(f)(d \in F_\perp(x)) = \begin{cases} \perp, \text{ if } d = \perp, \\ F(f)(d) \text{ in } F_\perp(y), \text{ otherwise.} \end{cases}$$

(iii) exponentiation

$$(F \to F')(x)$$

$$= \{ \; m \in \prod f: x \to y.F(y) \to F'(y) \; |$$

$$\text{for all } f: f \to y \text{ and } g: y \to z, \quad F(g) \; ; \; m(f;g) \; = \; m(f) \; ; \; F'(g) \; \}$$

ordered pointwise, and

$$(F \to F')(f)(m \in (F \to F')(x))(g) = m(f;g)$$

$d \in U$ whenever $u \leqslant_D d$, and, for every directed subset S of D, if $\mathbf{lub}(S) \in U$ then $d \in U$ for some $d \in S$. The "point-wise ordering" on partial functions is as follows: $f \leqslant_{D \rightsquigarrow D'} g$ iff, for all $x \in D$, if $f(x)$ is defined, then $g(x)$ is defined and $f(x) \leqslant_{D'} g(x)$.

(iv) partial exponentiation

$(F \rightsquigarrow F')(x)$

$= \{ \; m \in \Pi f: x \rightarrow y. F(y) \rightsquigarrow F'(y) \;\; |$

for all $f: x \rightarrow y$ and $g: y \rightarrow z$, $\;F(g) \; ; \; m(f;g) \;=\; m(f) \; ; \; F'(g)$

whenever $m(f)$ gives a defined result $\}$

ordered pointwise, and

$(F \rightsquigarrow F')(f)(m \in (F \rightsquigarrow F')(x))(g) = m(f;g)$

For functors to **S**, the $\rightarrow$ and $\rightsquigarrow$ constructions are defined in the same way, but are based on the exponentiations for **S**, rather than **D**, and the sets are not partially-ordered. Note that if **X** is the trivial (one-object and one-morphism) category, each of the constructions on functors reduces to the corresponding construction on domains or sets.

We also use some of these constructions on *contravariant* functors from **X** to **S** or **D**. The product and lifting operations construct contravariant functors in the obvious way, but we define the morphism parts of the exponentiation operations always to yield *covariant* functors; for example, if F and F' are contravariant functors from **X** to **S**,

$(F \rightsquigarrow F')(x)$

$= \{ \; m \in \Pi f: x \rightarrow y. F(y) \rightsquigarrow F'(y) \;\; |$

for all $f: x \rightarrow y$ and $g: y \rightarrow z$, $\;F(g) \; ; \; m(f) \;=\; m(f;g) \; ; \; F'(g)$

whenever $m(f;g)$ gives a defined result $\}$, and

$(F \rightsquigarrow F')(f)(m \in (F \rightsquigarrow F')(x))(g) = m(f;g)$

These constructions will be used to construct meaning functors $[\![\gamma]\!]$ for the phrase types γ of our language from the following "primitive" functors, where **X** is the category of possible-worlds defined in Section 3:

(i) S is the contravariant functor from **X** to **D** defined as follows. For any **X**-object X,X', $S(X,X') = X'$, and for any **X**-morphism $f,Q: X,X' \rightarrow Y,Y'$, $S(f,Q)$ is the restriction of the function $f \in Y \rightarrow X$ to Y' and X'. This is the only functor that depends directly on details of category **X**.

(ii) For every data type τ, $[\![\tau]\!]$ is the contravariant constant functor from **X** to **D** such that, for every **X**-object x, $[\![\tau]\!]x$ is the discretely-ordered domain of values of type τ.

(iii) T is the contravariant constant functor from **X** to **S** such that, for every **X**-object x, $T(x)$ is the set {true, false}.

(iv) U is the (covariant) constant functor from **X** to **S** such that, for every **X**-object x, $U(x)$ is any singleton set.

(v) F is the (covariant) functor from **D** to **S** that forgets partial-orderings and treats continuous functions as arbitrary functions.

(vi) E is the (covariant) functor from **S** to **D** that embeds sets into the category of domains by discretely ordering them and treating arbitrary functions as continuous functions. Note that $E;F$ is the identity functor on **S**.

We can now define functors $[\![\theta]\!]: \mathbf{X} \to \mathbf{D}$, and $[\![\alpha]\!],[\![\mathbf{spec}]\!]: \mathbf{X} \to \mathbf{S}$ as in Table 3. If **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 and acceptors, such meanings may appear to be more complex than those in Reynolds (1981b), Oles (1982) and Tennent (1985), but they are actually more convenient because it is no longer necessary to depend on the morphism parts of meaning functors to define families of functions "implicitly".

Finally, for any type-assignment π with $\pi(I)=\pi_I$ for every identifier $I \in \mathrm{dom}(\pi)$, we define a (covariant) functor $[\![\pi]\!]: \mathbf{X} \to \mathbf{D}$ as the product of the $[\![\pi_I]\!]$, where it is understood that, for any of the π_I that is an *assertional* (rather than ordinary) phrase type, $[\![\pi_I]\!]: \mathbf{X} \to \mathbf{S}$ is first composed with the embedding functor E to obtain a functor from **X** to **D**.

5. Semantic Valuations

We may now interpret the phrases of the language as natural transformations from environment functors to meaning functors:

$$[\![<\pi \vdash \theta>]\!]: [\![\pi]\!] \dot{\to} [\![\theta]\!]$$

$$[\![<\pi \vdash \alpha>]\!]: ([\![\pi]\!];F) \dot{\to} [\![\alpha]\!]$$

$$[\![<\pi \vdash \mathbf{spec}>]\!]: ([\![\pi]\!];F) \dot{\to} [\![\mathbf{spec}]\!]$$

where $[\![\pi]\!],[\![\theta]\!]: \mathbf{X} \to \mathbf{D}$, $[\![\alpha]\!],[\![\mathbf{spec}]\!]: \mathbf{X} \to \mathbf{S}$, and $F: \mathbf{D} \to \mathbf{S}$ are the functors defined in the preceding section.

We begin with the purely-logical fragment of the specification language. The key facts are that, for *any* category **X**, the functor

γ	$[\![\gamma]\!]$	functionality
τ-exp	$S \to [\![\tau]\!]_\perp$	$\mathbf{X} \to \mathbf{D}$
comm	$S \rightsquigarrow S$	$\mathbf{X} \to \mathbf{D}$
τ-acc	$[\![\tau]\!] \times S \rightsquigarrow S$	$\mathbf{X} \to \mathbf{D}$
$\theta \to \theta'$	$[\![\theta]\!] \to [\![\theta']\!]$	$\mathbf{X} \to \mathbf{D}$
assert	$(S;F) \to T$	$\mathbf{X} \to \mathbf{S}$
$\alpha \to \alpha'$	$[\![\alpha]\!] \to [\![\alpha']\!]$	$\mathbf{X} \to \mathbf{S}$
$\theta \to \alpha$	$([\![\theta]\!];F) \to [\![\alpha]\!]$	$\mathbf{X} \to \mathbf{S}$
spec	$U \rightsquigarrow U$	$\mathbf{X} \to \mathbf{S}$

Table 3. Meaning Functors

category $\mathbf{X} \Rightarrow \mathbf{S}$ is a topos (with truth-value object $U \rightsquigarrow U$), and intuitionistic logic may be interpreted in any topos (Goldblatt 1979). The resulting valuations are given in Table 4, where, for convenience, the identity function and the empty function on a singleton set are denoted by true and false, respectively. Here, and throughout this section, x is an $\mathbf{X}$-object, $f: x \to y$ is an $\mathbf{X}$-morphism with domain x, and u is a suitable environment; for example, here $u \in ([\![\pi]\!];F)(x)$ where π is the type-assignment component of the phrase-class of the phrase being interpreted. Note the "implicit" quantifications over changes of possible world in the valuations for implication, quantification, and equivalence.

In contrast, the logical operators in assertions may be treated "classically", as in Table 5, where $y_0 \in S(y)$. The equality operator is the "strict" one (yielding false if either operand is undefined) for reasons discussed in Tennent (1986), but the "strong" equality would

$[\![\mathbf{absurd}]\!]xuf$ = false

$[\![S_1 \ \& \ S_2]\!]xuf = [\![S_1]\!]xuf$ and $[\![S_2]\!]xuf$

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

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

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

= for all $g: y \to z$ and $m \in [\![\beta]\!]z$, $[\![S]\!]z(([\![\pi]\!];F)(f;g)(u); I: m)(\mathrm{id}_z)$

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

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

Table 4. Specifications

also be possible; each can be defined from the other using the definedness predicates (Scott 1967, 1979).

For procedures, the key facts are that, for *any* category **X**, the functor categories $\mathbf{X} \Rightarrow \mathbf{S}$ and $\mathbf{X} \Rightarrow \mathbf{D}$ are Cartesian-closed with product $\times$ and exponentiation $\to$ (Goldblatt 1979, Oles 1982), and the typed lambda calulus may be interpreted in any such category (Lambek 1980, Scott 1980). The resulting valuations (for ordinary procedures) are as follows:

$$[\![\lambda I: \theta.Z]\!]xuf(m \in [\![\theta]\!]y) = [\![Z]\!]y([\![\pi]\!]fu; I: m)$$

$$[\![P(A)]\!]xu = [\![P]\!]xu(\mathrm{id}_x)([\![A]\!]xu)$$

Assertional procedures may be treated in a similar way. For identifiers, $[\![I]\!]xu = u(I)$.

$[\![\mathbf{true}]\!]xufy_0 = \text{true} \qquad [\![\mathbf{false}]\!]xufy_0 = \text{false}$

$[\![\mathbf{not}\ P]\!]xufy_0 = \neg([\![P]\!]xufy_0)$

$[\![P_1\ \mathbf{and}\ P_2]\!]xufy_0 = ([\![P_1]\!]xufy_0 \text{ and } [\![P_2]\!]xufy_0)$

$[\![\forall I : \tau . P]\!]xufy_0$

$= \text{for all } m \in [\![\tau\text{-}\mathbf{exp}]\!]y,\ [\![P]\!]y(([\![\pi]\!];F)(f)(u); I : m)(\mathrm{id}_y)(y_0)$

$[\![\mathrm{D}_\tau(E)]\!]xufy_0 = ([\![E]\!]xufy_0 \neq \perp)$

$[\![E_1 =_\tau E_2]\!]xufy_0$

$= ([\![E_1]\!]xuy_0 = v_1 \in [\![\tau]\!]y) \text{ and } ([\![E_2]\!]xuy_0 = v_2 \in [\![\tau]\!]y)$
$\text{and } (v_1 = v_2)$

Table 5. Assertions

Valuations for the commands (other than the variable-declaration block) and for the Hoare-triple and expression-non-interference forms of specification are given in Table 6. In the equation for sequential composition, the semi-colon on the right denotes composition of partial functions (in diagrammatic order). Assertion non-interference is similar to expression non-interference, except that v ranges over the set {true, false}.

For new-variable declarations, we must first define "expansion" functors $\mathrm{exp}_\tau : \mathbf{X} \rightarrow \mathbf{X}$ for each data type τ as follows:

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

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

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

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

$=$ for all $y_0 \in S(y)$,

if $[\![P_0]\!]xufy_0$ and $[\![C]\!]xufy_0 = y_1 \in S(y)$ then $[\![P_1]\!]xufy_1$

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

$=$ for all $g: y \rightarrow z$ and $v \in ([\![\tau]\!]z)_\perp$,

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

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

Table 6. Commands

$\exp_\tau(X,X') = X\times V, X'\times V$

where $V = [\![\tau]\!](X,X')$,

and

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

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

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

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

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

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 $e_\tau(X,X') \in [\![\tau\text{-}\mathbf{exp}]\!](\exp_\tau(X,X'))$ and $a_\tau(X,X') \in [\![\tau\text{-}\mathbf{acc}]\!](\exp_\tau(X,X'))$ such that, for every **X**-morphism $g,R: \exp_\tau(X,X') \to Z,Z'$,

$$e_\tau(X,X')(g,R)(z_0 \in Z') = v$$

$$\text{where } \langle x_0,v\rangle = g(z_0) \ ,$$

and

$$a_\tau(X,X')(g,R)\langle v_1,\ z_0 \in Z'\rangle$$

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

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 is disallowed in the current possible world, the assignment fails to terminate.

Finally, the block command may be interpreted as follows:

$$[\![\textbf{new } \tau \textbf{ var } I,I' \textbf{ in } C]\!]xufy_0$$

$$= \begin{cases} y_1, & \text{if } [\![C]\!](\exp_\tau(x)) \\ & \quad ([\![\pi]\!](\times[\![\tau]\!]x)(u); I: a_\tau(x); I': e_\tau(x)) \\ & \quad (\exp_\tau(f)) \\ & \quad \langle y_0,v_0\rangle \\ & \quad = \langle y_1,v_1\rangle \in S(\exp_\tau(y)) \\ \text{undefined}, & \text{otherwise} \end{cases},$$

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

6. Formal System and Soundness

The purely-logical part of the formal system presented by Reynolds (1982) is essentially (the negative fragment of) many-sorted intuitionistic predicate logic with equality; however, occurrences of the static-assertion specification {**false**} should be replaced by the new constant **absurd** to allow for possible worlds in which the set of allowed states is empty. The non-logical axioms and derived rules are discussed in considerable detail by Reynolds and will not be repeated here; however, we have distinguished between assertions and (Boolean) expressions to allow the latter to be partial, so that axioms which use Boolean expressions as assertions must be slightly modified. For example, the **while** axiom would become

$$\{p \textbf{ and } e{=}\textbf{true}\}\ c\ \{p\} \Rightarrow \{p\}\ \textbf{while } e \textbf{ do } c\ \{p \textbf{ and } e{=}\textbf{false}\}\ ,$$

where **true** and **false** are Boolean constants. Also, axioms or formulas involving variables must be modified to suit our restricted language. For example, the equivalence defining the "good-variable" formula becomes

$$\mathbf{gv}_\tau(A,E)$$
$$\equiv_{\text{spec}} \forall I: \tau\text{-exp}.\ \forall I': \tau\text{-exp} \rightarrow \textbf{assert}.\ A \# I' \Rightarrow \{I'(I)\}\ A :=_\tau I\ \{I'(E)\}$$

where $A \in \langle \pi \vdash \tau\text{-acc}\rangle$, $E \in \langle \pi \vdash \tau\text{-exp}\rangle$, and I and I' are distinct

identifiers not in dom(π).

Our main result is the following

Theorem (soundness): for $S \in <\pi \vdash \mathbf{spec}>$, $[\![S]\!]xuf = \text{true}$ for all **X**-objects x, environments $u \in [\![\pi]\!]x$ and **X**-morphisms $f: x \to y$ whenever $S::\pi$ is derivable.

The proof is tedious but for the most part straightforward. The axioms of Strong Constancy and Leftside Non-interference Composition can be treated essentially as discussed in the introductory section. We present a detailed proof of only the "good-variable" property of newly-declared variables, which is needed to validate the axiom for variable-declaration blocks.

Consider any **X**-object x and let x' be $\exp_\tau(x)$. Consider any $e_0 \in [\![\tau\text{-}\mathbf{exp}]\!]x'$ and $p \in [\![\tau\text{-}\mathbf{exp} \to \mathbf{assert}]\!]x'$, and let a be $a_\tau(x)$ and e_1 be $e_\tau(x)$; i.e., $<a,e_1>$ is the new variable in the expanded possible world. Consider any **X**-morphism $f: x' \to y$ and any initial state $y_0 \in S(y)$. Suppose that $e_0(f)(y_0) = v \in [\![\tau]\!]y$ and $a(f)<v,y_0> = y_1 \in S(y)$. Then, from the definitions of a_τ and e_τ, we obtain that $e_1(f)(y_1) = v$.

Now, define the "constant" expression-meaning $e_v \in [\![\tau\text{-}\mathbf{exp}]\!]x'$ such that, for *all* $f: x' \to y$ and $y_i \in S(y)$, $e_v(f)(y_i) = v$. Then, using

Lemma 1: for all $p \in [\![\tau\text{-}\mathbf{exp} \to \mathbf{assert}]\!]$, and $e \in [\![\tau\text{-}\mathbf{exp}]\!]x'$, if $e(f)(y_i) = v$ then $p(\mathrm{id}_{x'})(e)(f)(y_i) = p(\mathrm{id}_{x'})(e_v)(f)(y_i)$,

we obtain

$$\begin{aligned} & p(\mathrm{id}_{x'})(e_1)(f)(y_1) \\ = \; & p(\mathrm{id}_{x'})(e_v)(f)(y_1) \quad \text{by lemma 1} \\ = \; & p(\mathrm{id}_{x'})(e_v)(f)(y_0) \end{aligned}$$

by the assumption that $a \# p$ and the obvious fact that $a \# e_v$,

$$= p(\mathrm{id}_{x'})(e_0)(f)(y_0) \quad \text{by lemma 1,}$$

which is the desired result.

To prove lemma 1, let $g: y \to z$ be the restriction morphism $[\{y_i\}$, so that $S(z)$ is the singleton set $\{y_i\}$. Then, using

Lemma 2: for all $e \in [\![\tau\text{-}\mathbf{exp}]\!]x'$, if $e(f)(y_i) = v$ then $[\![\tau\text{-}\mathbf{exp}]\!](f;g)(e) = [\![\tau\text{-}\mathbf{exp}]\!](f;g)(e_v)$,

we obtain

$$\begin{aligned} & p(\mathrm{id}_{x'})(e)(f)(y_i) \\ = \; & p(\mathrm{id}_{x'})(e)(f)(S(g)y_i) \quad \text{by the definition of } g \\ = \; & p(\mathrm{id}_{x'})(e)(f;g)(y_i) \quad \text{by the definition of } [\![\mathbf{assert}]\!] \\ = \; & [\![\mathbf{assert}]\!](f;g)(p(\mathrm{id}_{x'})(e))(\mathrm{id}_z)(y_i) \end{aligned}$$

$= p(f; g)([\![\tau\text{-exp}]\!](f; g)(e))(\mathrm{id}_z)(y_i)$ by the definition of $\rightarrow$
$= p(f; g)([\![\tau\text{-exp}]\!](f; g)(e_v))(\mathrm{id}_z)(y_i)$ by lemma 2
$= p(\mathrm{id}_{x'})(e_v)(f)(y_i)$ as with e.

To prove lemma 2, consider any $h: z \rightarrow w$ and $w_0 \in S(w)$; then,
$[\![\tau\text{-exp}]\!](f; g)(e)(h)(w_0)$
$= e(f; g; h)(w_0)$
$= e(f)(S(g; h)w_0)$ by the definition of $\rightarrow$
$= e(f)(y_i)$ because $g = [\{y_i\}$
$= v$
$= e_v(f)(y_i)$ by the definition of e_v
$= [\![\tau\text{-exp}]\!](f; g)(e_v)(h)(w_0)$ as with e.

This completes the proof of the good-variable property.

It should be straightforward to extend the interpretation to control structures such as conditionals, loops and recursion. Jumps would require continuation semantics for commands: for any contravariant functor A for "answers", the functor for command continuations would be $C = S \rightsquigarrow A$, and $[\![\mathbf{comm}]\!]$ would be re-defined as $C \rightarrow C$. The Hoare-triple form of specification would then be re-interpreted along the lines described by Reynolds (1981a, Section 4.2.3), and similarly for non-interference specifications.

7. Concluding Remarks

Specification logic is an *Algol-like* (Reynolds 1981b) programming logic. It is statically-typed. It avoids making references explicit. It treats substitution, binding and scope correctly. It includes the laws of the (typed) lambda calculus as a sub-system. It requires expressions to be side-effect-free. It distinguishes between the logic for data types and the logic for phrase types. It provides "generic" facilities such as quantification and non-interference uniformly for all relevant phrase types. And, finally, we have shown here that it "obeys a stack discipline" (though this operational terminology is inappropriate for logical phrase types) in that "local" changes to storage structure, such as state-set expansions and restrictions, can be allowed for by using possible-world semantics *without* making the worlds explicit in the logical language.

The only unsatisfactory aspect of specification logic is the inconvenience of having to deal explicitly with non-interference and good-variable assumptions. The most promising approach to avoiding these inconveniences (and also to optimizing the implementation of call-by-name) is that of Reynolds (1978).

Acknowledgements

John Reynolds suggested that Kripke semantics and the Reynolds-Oles approach to modelling block structure might be relevant to the interpretation of specification logic. The author is also grateful to Gordon Plotkin, Eugenio Moggi, and David McCarty for technical discussions. A preliminary report on this work appeared as Tennent (1985). 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

van Dalen, D. (1983), "Logic and Structure", 2nd edition, Springer.

Dummett, M. (1977), "Elements of Intuitionism", Oxford University Press.

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

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

Hoare, C. A. R., and Lauer, P. E. (1974), Consistent and complementary formal theories of the semantics of programming languages, *Acta Informatica* **3** (2), 135-153.

Kripke, S. A. (1965), Semantical analysis of intuitionistic logic I, *in* "Formal Systems and Recursive Functions", (J. N. Crossley and M. A. E. Dummett, Eds.), pp. 92-130, North-Holland.

Lambek, J. (1980), From λ-calculus to Cartesian-closed categories, *in* "To H. B. Curry, Essays in Combinatory Logic, Lambda Calculus and Formalism" (J. P. Seldin and J. R. Hindley, Eds.), pp. 375-402, Academic Press.

McCarty, D. C. (1984), Information systems, continuity and realizability, *in* "Logics of Programs, Proceedings 1983" (E. Clarke and D. Kozen, Eds.), Lecture Notes in Computer Science, vol. 164, pp. 341-359, Springer; *also* Chapter 7 of "Realizability and Recursive Mathematics", D.Phil. thesis, Oxford University, and technical report CMU-CS-84-131, Dept. of Computer Science, Carnegie-Mellon University.

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

Oles, F. J. (1985), 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.

Plotkin, G. D. (1985), "Types and partial functions", lecture notes, Computer Science Department, University of Edinburgh.

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

Reynolds, J. C. (1978), Syntactic control of interference, *in* "Conf. Record 5th ACM Symp. on Principles of Programming Languages",

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

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

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

Scott, D. S. (1967), Existence and description in formal logic, *in* "Bertrand Russell, Philosopher of the Century" (R. Schoenmann, Ed.), pp. 181-200, Allen and Unwin, London.

Scott, D. S. (1979), Identity and existence in intuitionistic logic, Lecture Notes in Mathematics, vol. 735, pp. 660-696, Springer.

Scott, D. S. (1980), Relating theories of the lambda calculus, *in* "To H. B. Curry, Essays in Combinatory Logic, Lambda Calculus and Formalism" (J. P. Seldin and J. R. Hindley, Eds.), pp. 403-450, Academic Press.

Tennent, R. D. (1985), 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.

Tennent, R. D. (1986), A note on undefined expression values in programming logics, *Inf. Proc. Letters* (to appear).

Appendix

The following example, from a personal communication to the author by J. C. Reynolds, demonstrates the undesirable consequences of adopting full classical (rather than intuitionistic) logic as the logical basis for specification logic.

From the rules for assignment, **skip**, and statement compounding, one derives

$$\mathbf{gv}(x) \Rightarrow \{x{=}3\}\ \mathbf{skip}; x{:=}4\ \{x{=}4\}\ .$$

By mathematical-fact introduction, static implication, and consequent weakening:

$$\mathbf{gv}(x)\ \&\ \{x{=}3\} \Rightarrow \{x{=}3\}\ \mathbf{skip}; x{:=}4\ \{\mathbf{false}\}\ .$$

Let S be the formula $\{x{=}3\}\ \mathbf{skip}; x{:=}4\ \{\mathbf{false}\}$. Then,

$$\mathbf{gv}(x)\ \&\ \neg S\ \&\ \{x{=}3\} \Rightarrow \mathbf{absurd}\ ,$$

and, since **absurd** implies anything,

$$\mathbf{gv}(x)\ \&\ \neg S\ \&\ \{x{=}3\} \Rightarrow \{\mathbf{true}\}\ \mathbf{skip}\ \{\mathbf{false}\}\ .$$

Now, Strong Constancy gives us

$$\mathbf{skip}\ \#\ (x{=}3)\ \&\ (\{x{=}3\} \Rightarrow \{\mathbf{true}\}\ \mathbf{skip}\ \{\mathbf{false}\}) \Rightarrow \{x{=}3\}\ \mathbf{skip}\ \{\mathbf{false}\}$$

and so, since $\mathbf{skip}\ \#\ (x{=}3)$, we get

$$\mathbf{gv}(x)\ \&\ \neg S \Rightarrow \{x{=}3\}\ \mathbf{skip}\ \{\mathbf{false}\}\ .$$

However, by assignment, $\mathbf{gv}(x) \Rightarrow \{\mathbf{false}\}\ x{:=}4\ \{\mathbf{false}\}$, so that statement compounding gives us

$$\mathbf{gv}(x)\ \&\ \neg S \Rightarrow \{x{=}3\}\ \mathbf{skip}; x{:=}4\ \{\mathbf{false}\}\ ,$$

i.e., $\mathbf{gv}(x)\ \&\ \neg S \Rightarrow S$, so that $\mathbf{gv}(x) \Rightarrow \neg\neg S$. Finally, with *classical* logic, we get $\mathbf{gv}(x) \Rightarrow S$, which means that an assignment like $x{:=}4$ does not terminate if x initially has a value other than 4.