MIT/LCS/TM-163

AXIOMATIC DEFINITIONS OF PROGRAMMING LANGUAGES: A THEORETICAL ASSESSMENT

Albert R. Meyer
Jospeh Y. Halpern

April 1980

Axiomatic Definitions of Programming Languages:

A Theoretical Assessment*

Albert R. Meyer
Laboratory for Computer Science, Massachusetts Institute of Technology,
Cambridge, Massachusetts 02139.

Joseph Y. Halpern
Mathematics Department, Harvard University,
Cambridge, Massachusetts 02138.

April 7, 1980

Abstract:

A precise definition is given of how partial correctness or termination assertions serve to define the semantics of classes of program schemes. Assertions involving only formulas of first order predicate calculus are proved capable of defining program scheme semantics, and effective axiom systems for deriving such assertions are described. Such axiomatic definitions are possible despite the limited expressive power of predicate calculus.

KEY WORDS: semantics of programming languages, partial correctness assertions, termination assertions, axiomatic definitions of programming languages.

*This is an expanded version of a paper with the same title which was given at the 7th Annual Symposium on the Principles of Programming Languages in Las Vegas, Nevada, January, 1980. This work was supported in part by the National Science Foundation, Grant No. MCS 7719754 and in part by a grant from the National Science and Engineering Research Council of Canada.

1. Introduction

The thesis that programming languages can be defined by axiomatic systems for proving properties about programs originated in [Floyd, 1967] (*cf.* [London, 1979]); it has been applied in practice allegedly to define a fragment of PASCAL in [Hoare and Wirth, 1973], advocated as desirable in [Hoare and Lauer, 1974], and further espoused and applied in [Dijkstra, 1975, 1976; Schwartz, 1978, 1979].

Nevertheless, it is not clear in what precise sense an axiom system for proving assertions serves to define a programming language. Moreover, the problem of explaining how axioms serve as definitions has hardly been considered in the literature. In what follows we consider, following [Greif and Meyer, 1979], two representative kinds of "before-after" assertions about programs, namely *partial correctness assertions* and *termination assertions.* We then go on to offer a precise, natural formulation of the property that an axiom system for deriving such assertions defines the input-output semantics of a programming language.[1]

For the sake of definiteness, we insist that the predicates within our assertions be expressible in a natural mathematical formalism, namely first order predicate calculus with equality. (This is in contrast to the treatment in [Greif and Meyer, 1979; Blikle, 1979] where predicates were purely set-theoretic, with no attention paid to the existence or derivation of formulas defining them.)

Our main result (see Sections 4.1, 4.5) is that

first order partial correctness assertions are capable of defining the input-output semantics of program schemes taken from quite general programming languages. Moreover, there is an effective procedure for generating enough true partial correctness assertions to define this semantics.

It remains open whether there is an embodiment of this effective procedure in terms of a reasonably elegant axiom system. But taking the abstract view that a sound axiom system is in general nothing other than an effective procedure for generating true or valid assertions, we conclude that defining programming language semantics by means of effective axioms for first order partial correctness assertions is indeed possible.

We found this result surprising since, in general, first order predicate calculus is inadequate to determine uniquely any infinite mathematical structure. For example, there are nonstandard models of the ring of integers which behave identically to the standard model with respect to all properties definable using first order language (i.e.

they are elementarily equivalent), but which are nevertheless not isomorphic to the standard structure. This might lead one to expect that assertions involving only first order formulas would also be consistent with nonstandard, perhaps pathological, interpretations of program semantics, but this is in fact not the case.

Despite this theoretically positive conclusion, we regard our study as revealing just how delicate and questionable is the thesis that languages can or should be given axiomatic definitions by means of partial correctness assertions. The subtlety of the proof that first order partial correctness assertions define semantics suggests, for example, that there is no simple, intuitive way to translate from axioms for partial correctness to the corresponding input-output semantics, even though this semantics is uniquely determined mathematically.

In fact, under apparently mild restrictions on formulas and a slight enrichment of the simplest class of finite flowchart schemes, the partial correctness assertions true of a program scheme are not adequate to define the semantics of the scheme. (See especially Section 4.7.)

We also consider termination assertions, which appear to be more tractable than partial correctness assertions from a theoretical viewpoint. For example, we exhibit a complete axiom system for first order termination assertions about **while**-program schemes and show that such assertions uniquely determine program semantics.

(Essentially the same results as our Theorems 4.1, 5.1, and 7.2, and the technical Lemmas 4.2 and 7.3 were obtained independently in [Bergstra, Tiuryn, Tucker, 1979] for the case of deterministic effective schemes, in response to a question posed by the first author.)

2. Partial Correctness, Termination, and Equivalence

2.1 *Definitions from first order logic.* A *type* (or *signature*) is a set of objects called *symbols.* Each symbol is either a *function* symbol or a *predicate* symbol and has an associated nonnegative integer *arity.* (Variables and constants are simply treated as zeroary function symbols.) A type is said to be *finite* if it has only finitely many function and predicate symbols. A *state* s, consists of a type $\tau(s)$, a domain D, and an assignment to each function (resp., predicate) symbol in $\tau(s)$ of a function (resp., predicate) on D of the associated arity. The type of formula F, denoted $\tau(F)$, is the set of function and predicate symbols appearing free in F. (Note that for first order formulas, all symbols of positive arity necessarily appear free.) A formula F of the first order predicate calculus with equality is defined to be true or false in a state s (such

that $\tau(s) \supseteq \tau(F)$) in the usual way; we write $s \models F$ iff F is true in state s. We write $\models F$ to indicate that F is valid, i.e., true in all states.

We extend the class of first order formulas as follows:

Let R denote some binary relation on states, and let F be any formula. Then [R]F is a formula and $s \models [R]F$ iff $t \models F$ for all t such that $(s, t) \in R$. Also, <R>F is a formula equivalent by definition to $\neg[R]\neg F$. (This is just Pratt's notation for dynamic logic (cf. [Harel, 1979]).) A relation R is *tidy* providing [R]F is equivalent to a first order formula of predicate calculus with equality whenever F is such a formula. (cf. [Pratt, 1976].)

2.2 Our results depend on the class of program schemes we consider. The richer the class, the larger the semantics to be specified, and therefore our positive results take their strongest form when we allow even more powerful program schemes than are reasonably realistic.

We consider four classes of schemes which are listed in decreasing order of generality.

a) *Arbitrary schemes* are nondeterministic, possibly infinite, flowcharts (finite, deterministic schemes are regarded as a special case) whose tests may be arbitrary first order formulas and whose basic instructions consist of *simple assignments* of the form "x := term", *array assignments* of the more general form "f(x) := term" and *random assignments* of the form "x := ?". (The random assignment "x := ?" means set x to any value in the domain (cf. [Harel, 1979]).)

(b) An arbitrary scheme is *recursively enumerable* if there is an effective procedure to generate the labels, boxes, and edges of the flowchart. Various weakenings of recursively enumerable schemes are possible by disallowing array assignments, random assignments, etc.

(c) *Effective schemes* are recursively enumerable schemes with open tests (i.e. tests consisting of quantifier-free formulas of predicate calculus with equality) and simple assignments only (i.e. no random or array assignments). (cf. [Friedman, 1971; Bergstra, Tiuryn, Tucker, 1979].)

(d) A special case of effective schemes are the familiar **while**-program schemes, with the following BNF-description:

<program>::=

<simple assignment> | <program>;<program> |

if <open test> **then** <program> **else** <program> **fi** |

while <open test> **do** <program> **od**.

The type $\tau(a)$ of an arbitrary scheme a is the set of function and predicate symbols appearing in the instructions and tests of the scheme.

2.3 *Definition*: If a is an arbitrary scheme, then R_a is the initial state - final state relation defined by a; i.e. $R_a = \{(s, t) |$ t is a state which is a possible result of performing a starting in state s$\}$. Let $R_a(s) = \{t | (s, t) \in R_a\}$. When there can be no confusion, we will use a instead of R_a; e.g. we will say $[a]F$ instead of $[R_a]F$.

2.4 *Definition*: Arbitrary schemes a and b are *equivalent* iff $R_a = R_b$; that is, they always take an initial state to equal sets of final states.

2.5 *Partial Correctness and Termination*: A *partial correctness* (resp., *termination*) *assertion* consists of a pair of formulas F,G and a program a, and is written $F\{a\}G$ (resp., $F(a)G$). The assertion is *true* iff $F \Rightarrow [R_a]G$ (resp., $F \Rightarrow \langle R_a \rangle G$) is a valid formula. Thus $F\{a\}G$ is true iff, whenever state s satisfies F and state t is a possible result of performing a starting in state s, then t satisfies G. Similarly, $F(a)G$ is true iff, whenever $s \models F$ there exists a t such that $(s, t) \in R_a$ and $t \models G$.

The *first order partial correctness* (resp., *termination*) *theory* of a, denoted $PC(a)$ (resp., $T(a)$) is $\{(F, G) \mid$ F,G are first order formulas of predicate calculus with equality and $F\{a\}G$ (resp., $F(a)G$) is true$\}$

For any type τ, we let $PC_\tau(a)$ denote $\{(F, G) \in PC(a) \mid \tau(F) \subseteq \tau$ and $\tau(G) \subseteq \tau\}$.

We observe trivially that

2.6 *Lemma*: If a and b are equivalent schemes, then $PC(a) = PC(b)$ and $T(a) = T(b)$.

2.7 *Lemma*: [Pratt, 1976] If a is a finite loop-free scheme, then a is tidy. Moreover, if a has only simple assignments and F is quantifier-free, then $[a]F$ is equivalent to a quantifier-free formula.

2.8 *Definition*: If a_i, i = 1, 2, 3, ... is a set of schemes, then $\cup_i a_i$ denotes the scheme which is the nondeterministic union of the a_i; i.e., $R_{\cup_i a_i}$ is equal to $\cup_i R_{a_i}$.

It is now easy to see that

2.9 *Lemma*: Any scheme a is equivalent to a union of finite, loop-free (and hence, by Lemma 2.7, tidy) schemes, such that each finite scheme uses only the instructions and tests which appear in a. Moreover, if a is recursively enumerable, we can effectively generate the finite schemes from (an index for) a.

3. Axiomatic Definitions of Language

3.1 The axiomatic definitions of programming languages proposed in the literature consist of systems for deriving assertions about classes of program schemes involving subroutine calls with parameter passing to other schemes. In nearly all cases (languages with a *pointer* data type are an exception; cf. [Janssen and van Emde Boas, 1977]), the programs are equivalent to arbitrary schemes of various types. We interpret the claim that such an axiom system *defines* the programming language semantics as meaning that enough assertions are provable to distinguish between inequivalent programs. (By Lemma 2.6, we cannot expect to make distinctions among equivalent programs using partial correctness or termination assertions.) More exactly, we offer the following

Definition: Let $\mathscr{P}$ be a set of true partial correctness assertions, and $\mathscr{A}$ be a set of arbitrary schemes. Then $\mathscr{P}$ *defines* $\mathscr{A}$ providing that for all inequivalent schemes a, $b \in \mathscr{A}$, there is an assertion in $\mathscr{P}$ about one of a or b which is not true of the other. That is, let $PC_{\mathscr{P}}(a) = \{(F, G) \mid F\{a\}G \in \mathscr{P}\}$. Then $\mathscr{P}$ defines $\mathscr{A}$ if

$$(PC_{\mathscr{P}}(a) - PC_{\mathscr{P}}(b)) \cup (PC_{\mathscr{P}}(b) - PC_{\mathscr{P}}(a)) \neq \varnothing.^{2}$$

3.2 *Example*:

Let *NOP* be the program which halts without changing anything, and let a_0 be the following program scheme:

if $(y \neq x \vee z \neq f(x))$ **then** *NOP*

else while $y \neq z$ **do** $z := f(f(z)); y := f(y)$ **od**;

$y := x; z := f(x)$ **fi**

It is easy to see that a_0, when it halts, behaves like *NOP*, namely, it has no effect. However, under some interpretations a_0 does not halt, for example if $x = y = 0$, $z = 1$, and f is the successor function on the integers, so *NOP* and a_0 have different meanings.

Take *Open* to be the set of all true quantifier-free partial correctness assertions. We will show in Appendix A that

3.3 *Lemma:* $PC_{Open}(a_0) = PC_{Open}(NOP)$,

Thus it follows that quantifier-free partial correctness assertions do *not* define **while**-program schemes. On the other hand, if we allow quantifiers we can distinguish a_0 and *NOP*. Indeed, the pair of formulas

$$\forall z(f(z) \neq x) \wedge \forall t \forall y(t \neq y \Rightarrow f(t) \neq f(y)),$$

$$w \neq w$$

is in $PC(a_0)$ but not in $PC(NOP)$, since if a state s satisfies the former formula, a_0 will diverge on s. In fact, Theorem 4.1 shows that first order partial correctness assertions define the semantics of **while**-program schemes.

4. Defining Semantics By Partial Correctness Assertions

In this Section we prove our main results about partial correctness assertions. We show in Theorem 4.1 that despite the general inadequacies of first order predicate calculus, the set of true partial correctness assertions defines the semantics of recursively enumerable schemes. In Theorem 4.5 we show that for effective schemes we can find a recursively enumerable set of first order partial correctness assertions which will define the semantics.

4.1 *Theorem*: For recursively enumerable schemes *b*, *c* of finite type, if *b* and *c* have the same first order partial correctness theories, then *b* and *c* are equivalent.

The proof of Theorem 4.1 rests on two lemmas. The first is purely model theoretic, and reveals that in a certain sense first-order formulas are closed under infinite recursively enumerable conjunctions. (This lemma, discovered independently by the authors, turns out to be a refinement of a classical result [Kleene, 1952][3].)

4.2 *Lemma*: Let F_0, F_1, ..., be any recursively enumerable sequence of first order formulas of finite type τ. Then one can effectively construct a first order formula G such that

a) $\models G \Rightarrow \wedge_i F_i$, and

b) if s is a state with infinite domain and $s \models \wedge_i F_i$, then there is an expansion[4] s' of $s|_\tau$ (the restriction of s to type τ) such that $s' \models G$.

The idea of the proof is to introduce new symbols +, · and include in G axioms involving these new symbols which define a (possibly nonstandard) copy of the ring of integers. Then we implicitly define a predicate T which acts like a truth predicate, and make sure (the Gödel number of) each F_i satisfies T. The proof is deferred to Appendix B.

4.3 *Lemma*: If b and c are arbitrary schemes of finite type and such that $R_b - R_c \neq \varnothing$, then there is a first order formula H such that $[c]H \wedge \neg[b]H$ is satisfiable.

Proof: Suppose $(s, t) \in R_b - R_c$. Let $x_1, \ldots x_n, f_1, \ldots, f_m$ be the free variables and functions mentioned in b and c. Choose fresh variables and functions $x_1', \ldots, x_n', f_1', \ldots, f_m'$ and define a state s' in which $x_1, \ldots, x_n, f_1, \ldots, f_m$ have the same respective values as they do in s, while $x_1', \ldots, x_n', f_1', \ldots, f_m'$ have the same respective values they do in t.

Let H be the formula

$$\neg(x_1 = x_1' \wedge \ldots \wedge x_n = x_n' \wedge \forall z(f_1(z) = f_1'(z) \wedge \ldots \wedge f_m(z) = f_m'(z))).$$

Then $s' \models [c]H \wedge \neg[b]H$, as the reader may verify. □

Proof of 4.1: Suppose b and c are inequivalent. Without loss of generality we may assume $R_b - R_c \neq \varnothing$.

By Lemma 4.3, we have a formula H for which there is a state s such that

$$s \models [c]H \wedge \neg[b]H.$$

Notice that this is equivalent to the assertion that $[c]H\{b\}H$ is false. Of course $[c]H\{c\}H$ is trivially true for all c and H. Thus the pair of predicates ([c]H, H) will serve to distinguish the partial correctness theories of b and c. The only difficulty comes from the fact that [c]H may not be equivalent to a first order formula.

However, by Lemmas 2.7 and 2.9, c is equivalent to $\cup_i c_i$, where each c_i is tidy. Hence [c]H is equivalent to $\wedge_i[c_i]H$. Taking F_i to be $[c_i]H$ (which is first order since c_i is tidy), by Lemma 4.2(a) we can effectively construct a first order G such that

$$\models G \Rightarrow [c]H$$

Moreover, if $s \models [c]H$ and s has an infinite domain, Lemma 4.2(b) says we can find an expansion of s to s′ such that $s' \models G$. Then $(G,H) \in PC(c) - PC(b)$.

If s has finite domain our task is even easier. For if $\tau = \tau(b) \cup \tau(c)$ then we can find a first order formula ISO_s such that $s' \models ISO_s$ iff $s'|_\tau$ is isomorphic to $s|_\tau$. (So $s' \models ISO_s$ iff s and s′ look the same as far as b and c are concerned.) Then clearly we have $(ISO_s, \neg ISO_t) \in PC(c) - PC(b)$, since $(s, t) \in R_b - R_c$. □

4.4 *Remark*: The hypotheses that b, c be recursively enumerable and of finite type are both necessary, as we show in Appendix C. □

Theorem 4.1 involves the set of all true partial correctness assertions, which is not recursively enumerable even in the the case of **while**-program schemes (in fact, it is Π^0_2 complete [Harel, Meyer, Pratt; 1977]). This leaves open the question of whether some, necessarily recursively enumerable, set of partial correctness assertions derivable from an effective axiom system can define semantics. But the proof of Theorem 4.1 is effective except for the portion concerning the behavior of schemes on states with finite domain. With a slight restriction on the class of recursively enumerable schemes, for example restricting to effective schemes will suffice, we can ignore the states with finite domain and give an affirmative answer to the question of effectiveness.

4.5 *Theorem*: There is a recursively enumerable set of first order partial correctness assertions $\mathscr{P}$ which defines the semantics of effective schemes and, *a fortiori*, of **while**-program schemes.

Proof: All the essential ideas of this proof are already in the proof of Theorem 4.1. For any program c and sentence H, let $G_{H,c}$ be the formula from the proof of Theorem 4.1 such that

$$\models G_{H,c} \Rightarrow \wedge_i [c_i]H.$$

(where again c is equivalent to $\cup_i c_i$, and each c_i is tidy.)

Then $G_{H,c}\{c\}H$ is always true.

Take $\mathscr{P} = \{G_{H,c}\{c\}H \mid H \text{ is a first order formula, } c \text{ is an effective scheme}\}$. Then $\mathscr{P}$ is recursively enumerable, since the proof of Lemma 4.2 actually shows how to construct $G_{H,c}$ effectively from H and c. Moreover, $\mathscr{P}$ defines the semantics of effective schemes. To see this suppose b and c are inequivalent, and $(s, t) \in R_b - R_c$. Without loss of generality we can assume s has an infinite domain. (Otherwise take s′

to be an extension of s with an infinite domain. Then there is a corresponding $(s', t') \in R_b - R_c$. The crucial point here is that since b and c are effective they only have open tests and simple assignments, so adding new elements to a domain will not affect the truth value of the tests). Take the H from Lemma 4.3 such that

$$s \models [c]H \wedge \neg[b]H.$$

Then the proof of Theorem 4.1 shows

$$(G_{H,c}, H) \in PC_{\mathscr{J}}(c) - PC_{\mathscr{J}}(b). \quad \square$$

4.6 *Remark*: We can extend Theorem 4.5 to larger subclasses of recursively enumerable schemes, in particular to **while**-program schemes which are enriched to include random and array assignments and first order tests. A technical complication arises with the introduction of first order tests and random assignments. Namely, we can no longer assume, as we did in Theorem 4.5 that we can find an s with in *infinite* domain such that $s \models \neg[b]H \wedge [c]H$. The details are left to Appendix D. $\square$

In Theorem 4.1 we showed that if b and c are inequivalent then $PC(b) \neq PC(c)$. In the proof we used the formulas G and H from Lemmas 4.2 and 4.3, which made use of new symbols such as $+$, $\cdot$, T, f_1', x_1', etc. which did not appear in either b or c. Can we still find a distinguishing partial correctness assertion if we restrict ourselves to $\tau(b) \cup \tau(c)$? In general the answer is no, as the following theorem shows.

4.7 *Theorem*: Let τ be a type with only finitely many function symbols of positive arity, finitely many predicate symbols, infinitely many variable symbols, and at least two function symbols of arity ≥ 1. Then there are inequivalent recursively enumerable schemes b, c of finite type such that $\tau(b) \cup \tau(c) \subseteq \tau$ and $PC_\tau(b) = PC_\tau(c)$.

Construction: Let f, g, f_1 ..., f_m, P_1, ..., P_n be the function and predicate symbols of τ with m, n ≥ 0 , where f_i is r_i-ary, P_j is s_j-ary. For convenience, let us assume f and g are unary functions.

Let Trivial $(x_1, ..., x_N)$ be the following predicate (where $N = \max\{r_1, ..., r_m, s_1, ..., s_n\}$):

$$P_1(x_1, ..., x_{s_1}) \wedge ... \wedge P_m(x_1, ..., x_{s_m}) \wedge$$

$$f_1(x_1, ..., x_{r_1}) = x_1 \wedge ... \wedge f_m(x_1, ..., x_{r_m}) = x_1 \wedge$$

$$f(g(x_1)) = x_1 \wedge g(f(x_1)) = x_1.$$

Then let $b = NOP$ and let c be defined as follows

if $x_1 \neq y \vee \ldots \vee x_N \neq y$ **then** *NOP*

else (x_1 := ?; ...; x_N := ?;

if $\neg$Trivial(x_1, ..., x_N) **then** $x_1 := y$; ...; $x_N := y$

else ($x_2 := f(x_1)$;

while $x_2 \neq x_1$ **do** $x_2 := f(x_2)$ **od**;

$x_1 := y$; ...; $x_N := y$) **fi**) **fi**

The program scheme c nondeterministically checks to make sure that $x_1 = x_2 = \ldots = x_N = y$, P_1, ..., P_m are all trivial predicates, f_1, ..., f_m are all projection on the first coordinate, f and g are inverse functions, and f has no "loops" (i.e. $f^k(x) \neq x$ for any x). If any of these conditions are not met, c acts like *NOP*; if they are all met c diverges. Hence $R_c \subseteq R_{NOP}$. But $R_c \neq R_{NOP}$, since if we take s to be the integers, with f successor and g predecessor, and $x_1 = \ldots = x_N = y = 0$, then $(s, s) \in R_{NOP} - R_c$. Finally, it is not hard to show (see Appendix E) that $PC_\tau(NOP) = PC_\tau(c)$.

In Section 7 we will take a more detailed look at the whole question of extra symbols. □

5. Defining Semantics By Termination Assertions

Termination assertions can also be used to define a class of program schemes, and they are, in general, a more powerful tool for doing so than partial correctness assertions, as the following theorems show.

5.1 *Theorem*: The set of all true termination assertions defines the semantics of arbitrary schemes of finite type.

Proof: Suppose b and c are inequivalent schemes of finite type. Without loss of generality, suppose $(s, t) \in R_b - R_c$. By Lemma 2.9, b is equivalent to $\cup_i b_i$, where each b_i is tidy. Hence there must be some b_{i_0} such that $(s, t) \in R_{b_{i_0}} - R_c$. By Lemma 4.3, there is a state s' and a formula H such that $s' \models \neg[b_{i_0}]H \wedge [c]H$, or equivalently

$$s' \models \langle b_{i_0} \rangle \neg H \wedge \neg \langle c \rangle \neg H.$$

Thus $(<b_{i_0}>\neg H, \neg H) \notin T(c)$, but we trivially have $(<b_{i_0}>\neg H, \neg H) \in T(b)$. □

5.2 *Remark*: The hypothesis of finite type is necessary. The programs b and c of Appendix C.1 demonstrate this.[6]

As the example in 3.2 showed, the set of true quantifier-free partial correctness assertions does not define the semantics of the class of **while**-programs. But termination assertions will do the job.

5.3 *Theorem*: The set of all true quantifier-free termination assertions defines the semantics of the class of arbitrary schemes of finite type with quantifier-free tests and simple assignments only (i.e. no array assignments or random assignments), and, *a fortiori*, defines the semantics of effective schemes and **while**-program schemes.

Proof: Let $x_1, ..., x_m$ be the variables of $\tau(b) \cup \tau(c)$. As in Theorem 5.1, we can assume without loss of generality that we have $(s, t) \in R_{b_{i_0}} - R_c$. Since b uses only simple assignments, only $x_1, ..., x_m$ could have been changed by b in going from s to t (no functions could have been changed since there are no array assignments).

Thus, H from Lemma 4.3 can be simplified to H′, where H′ is

$$\neg(x_1 = x_1' \wedge ... \wedge x_m = x_m'),$$

and we still have, as in Theorem 5.1,

$$(<b_{i_0}>\neg H', \neg H') \in T(b) - T(c).$$

But by Lemma 2.7, $<b_{i_0}>\neg H'$ is quantifier-free, so we are done. □

5.4 *Remark:* The hypothesis in Theorem 5.3 that there be only simple assignments is necessary, as we show in Appendix F.

6. A Complete Axiomatization For First-Order Termination Assertions

6.1 Consider the following deductive system for termination assertions about **while**-program schemes. (In what follows F, G, and H denote first order formulas.)

Axiom Schemes:

TA1(a). $F(NOP)F$.

TA1(b). $F^x_t(x := t)F$, where t is a term and F^x_t is the formula obtained from F by uniformly renaming all bound variables in F which occur in t, and then replacing all free occurrences of x in F by t.

TA2. $\neg F \wedge G(\textbf{while } F \textbf{ do } a \textbf{ od})G$.

Rules:

TA3. $F(a)G, G(b)H \vdash F(a;b)H$.

TA4(a). $G(a)H \vdash F \wedge G(\textbf{if } F \textbf{ then } a \textbf{ else } b \textbf{ fi})H$.

TA4(b). $G(a)H \vdash \neg F \wedge G(\textbf{if } F \textbf{ then } b \textbf{ else } a \textbf{ fi})H$.

TA5. $F(a)G, G(\textbf{while } E \textbf{ do } a \textbf{ od})H \vdash F \wedge E(\textbf{while } E \textbf{ do } a \textbf{ od})H$.

TA6. $F(a)G \vdash H(a)G$ whenever $\models F \equiv H$ (substitutivity of equivalent formulas).

TA7. $F(a)G \vdash F \wedge H(a)G$ (restricted rule of consequence).

TA8. $F(a)H, G(a)H \vdash F \vee G(a)H$.

6.2 *Theorem*: The quantifier-free termination assertions derivable from quantifier-free instances of the axiom schemes TA1-8 are precisely the quantifier-free termination assertions true for **while**-program schemes. That is, the axioms TA1-8 are complete for quantifier-free termination assertions.

Proof: The soundness of TA1-8 should be clear. We now prove completeness by induction on the structure of **while**-program schemes.

a) Suppose $F(NOP)G$ is true. Then $\models F \Rightarrow G$, so $\models (F \wedge G) \equiv F$. Hence we have:

1.	$\vdash G(NOP)G$	TA1(a)
2.	$\vdash (F \wedge G)(NOP)G$	TA7
3.	$\vdash F(NOP)G$ (since $\models (F \wedge G) \equiv F$)	TA6

(b) Suppose $F(x := t)G$ is true. Then $\models F \Rightarrow G^x_t$, so

$$\models (F \wedge G^x_t) \equiv F.$$

Using TA1(b), TA7, and TA6, we get $\models F(x := t)G$.

(c) Suppose $F(a;b)G$ is true. Since a and b are effective, they are respectively equivalent to $\cup_i a_i$, $\cup_j b_j$ and each a_i, b_j is tidy. Since $F(a;b)G$ is true we have

$$\models \vee_{i,j}(F \Rightarrow \langle a_i;b_j\rangle G).$$

By compactness it follows that there exist finite sets I, J such that

$$\models \vee_{i\in I,\ j\in J}(F \Rightarrow \langle a_i;b_j\rangle G).$$

Thus $F(a)(\vee_{j\in J}(\langle b_j\rangle G))$ is true. And clearly $(\vee_{j\in J}(\langle b_j\rangle G))(b)G$ is also true. So, by the induction assumption

$$\vdash F(a)(\vee_{j\in J}(\langle b_j\rangle G))$$

and

$$\vdash (\vee_{j\in J}(\langle b_j\rangle G))(b)G.$$

Then using TA3 we get $\vdash F(a;b)G$.

(d) Suppose G(**if** F **then** a **else** b **fi**)H is true. Then $F\wedge G(a)H$ and $\neg F\wedge G(b)H$ must both be true, so, by the induction assumption, derivable from TA1-8.

Using TA4(a) and TA4(b), we get

$$\vdash F\wedge G(\textbf{if } F \textbf{ then } a \textbf{ else } b \textbf{ fi})H$$

and

$$\vdash \neg F\wedge G(\textbf{if } F \textbf{ then } a \textbf{ else } b \textbf{ fi})H.$$

Using TA8, we get

$$\vdash((F\wedge G) \vee (\neg F\wedge G))\ (\textbf{if } F \textbf{ then } a \textbf{ else } b \textbf{ fi})\ H$$

and hence (since $((F\wedge G) \vee (\neg F\wedge G)) \equiv G$), from TA6 we have

$\vdash$ G(**if** F **then** *a* **else** *b* **fi**)H.

(e) Suppose G(**while** F **do** *a* **od**)H is true. For any formula of first order predicate calculus F′, let F′? denote the program scheme

if F′ **then** *NOP* **else** (**while** x = x **do** *NOP* **od**) **fi**.

Note $R_{F'?} = \{(t, t) | \ t \vDash F'\}$ and F′? is tidy; in fact <F′?>G is equivalent to F′∧G. Again, we can assume a is equivalent to $\cup_i a_i$, where a_i is tidy. Let $\mathcal{S}$ be the set of all finite sequences of natural numbers, and for $s = \langle s_1, \ldots, s_n \rangle \in \mathcal{S}$, define $a^{(s)} = F?; a_{s_1}; \ldots; F?; a_{s_n}; \neg F?$. The program scheme $a^{(s)}$ is tidy since F?, ¬F? are, and each a_i is. By hypothesis

$$\vDash G \Rightarrow \vee_{s \in \mathcal{S}} \langle a^{(s)} \rangle H.$$

By compactness there is a finite set $S \subset \mathcal{S}$ such that

$$\vDash G \Rightarrow \vee_{s \in S} \langle a^{(s)} \rangle H. \qquad (*)$$

Let |s| denote the length of the sequence s. We prove by induction on |s| that

$$\vdash \langle a^{(s)} \rangle H \ (\textbf{while F do } a \textbf{ od}) \ H.$$

If |s| = 0, then $\vDash \langle a^{(s)} \rangle H \equiv \langle \neg F? \rangle H \equiv \neg F \wedge H$, and by TA2

$$\vdash (\neg F \wedge H) \ (\textbf{while F do } a \textbf{ od}) \ H.$$

If $s = \langle s_1, \ldots, s_n \rangle$, $n \geq 1$ let $s' = \langle s_2, \ldots, s_n \rangle$.

Now $\langle a^{(s)} \rangle H \ (a) \ \langle a^{(s')} \rangle H$ is true, so by the main induction assumption,

$$\vdash \langle a^{(s)} \rangle H \ (a) \ \langle a^{(s')} \rangle H.$$

And by our induction on |s|,

$$\vdash \langle a^{(s')} \rangle H \ (\textbf{while F do } a \textbf{ od}) \ H.$$

Hence by TA5, TA6, and the fact that $\vDash (F \wedge \langle a^{(s)} \rangle H) \equiv \langle a^{(s)} \rangle H$, we have

$$\vdash \langle a^{(s)} \rangle H \ (\textbf{while F do } a \textbf{ od}) \ H.$$

Now using TA8, we get

$$\vdash (\vee_{s\in S}\langle a^{(s)}\rangle H)\ (\textbf{while F do } a \textbf{ od})\ H.$$

So by (*), TA6, TA7,

$$\vdash G(\textbf{while F do } a \textbf{ od})H. \qquad \square$$

It is now immediate from Theorem 5.3 that

6.3 *Corollary*: The quantifier-free theorems of the axiom system TA1-8 define the semantics of **while**-program schemes.

Note that the proof of Theorem 6.2 also goes through in the case of the full first order language. So we get

6.4 *Theorem*: The first order termination assertions derivable from first order instances of the axiom schemes TA1-8 are precisely the first order termination assertions true for **while**-program schemes. That is, TA1-8 are complete for first order termination assertions.

6.5 *Remark*: Theorem 6.4 continues to hold if we enrich the class of primitive instructions to include both random and array assignments, and indeed, any tidy primitive instruction, with the appropriate strengthening of TA1(b), (namely, to now read $\langle R_A\rangle F(A)F$ for A primitive). We may also allow first order tests in the **while**-statement.

Note that by allowing random assignments we are dealing with a highly nondeterministic class of program schemes, which illustrates the point that completeness of the above axiom system does not depend on the property of determinism. □

7. The Role of Extra Symbols

Although Theorem 4.1 showed that it is always possible to distinguish inequivalent recursively enumerable schemes by partial correctness assertions, Theorem 4.7 showed that in general one must use symbols that appear in neither scheme in order to do so. In this section we investigate the role of extra symbols more carefully, both in partial correctness and termination assertions.

Basically we observe that no extra symbols, not even extra variable symbols, are needed as long as first order tests and random assignments do not appear in programs (Theorem 7.2).

Two further results of purely technical interest are given. Allowing random assignments *or* first order tests makes extra symbols of *positive* arity necessary; extra variables do not suffice (Theorem 7.4). Finally, if array assignments are disallowed, then termination assertions do not require extra symbols aside from variables (Theorem 7.5), although in contrast partial correctness assertions do.

7.1 *Definitions*: We will say τ is a *sufficiently rich* type if it contains at least one function or predicate symbol of arity ≥ 2 or at least two unary function symbols. The type τ' is a *weak extension* of τ if $\tau' = \tau \cup \{\text{a finite number of variable symbols}\}$. Finally, τ has *enough variables* if it has at least one more variable symbol than the maximum arity of the function symbols and predicate symbols appearing in it.

Our major result is a positive one:

7.2 *Theorem*: For inequivalent effective schemes a, b, even allowing array assignments as primitives, if $\tau(a) \cup \tau(b) = \tau$ is sufficiently rich, then $PC_\tau(a) \neq PC_\tau(b)$ and $T_\tau(a) \neq T_\tau(b)$. That is we do *not* need extra symbols in order for either partial correctness assertions or termination assertions to distinguish inequivalent programs.[7]

Outline of Proof: We show that under our hypotheses we can replace the G and H of Lemmas 4.2 and 4.3 by new formulas, say G′ and H′, that perform the same function without using extra symbols. Using this G′ and H′ instead of G and H in the proofs of 4.1 and 5.1 we get our desired result.

Replacing the extra symbols in G and H is carried out as follows:

It is easy to show that we can replace all the extra symbols occurring in G and H by one predicate symbol of sufficiently large arity. Since τ is sufficiently rich, we can assume without loss of generality that it contains a binary predicate symbol, say R. Then using R, it is possible to write a first order formula which implies the existence of "new" elements which are not the values of terms of type τ. The formula asserts that the relation given by R between these new elements and the old ones serves to encode the large arity predicate on the old values.

This idea then leads to the following variation of Lemma 4.2:

7.3 *Lemma*: Let $F_0, F_1, \ldots$, be any recursively enumerable sequence of open formulas of finite type τ. If τ is sufficiently rich, then one can effectively construct a first order formula G such that

(a) $\tau(G) \subseteq \tau$;

(b) $\models G \Rightarrow \wedge_i F_i$, and

(c) if s is a state such that $s \models \wedge_i F_i$, then there is an extension[5] s′ of s such that $s' \models G$.

The details of the proof are omitted. □

As we already noted, Theorem 4.7 shows that if we allow either random assignment or first order tests, Theorem 7.2 fails in the case of partial correctness assertions. In fact, for schemes using random assignment or first order tests it is the case that extra symbols of *positive* arity are required for termination assertions as well as partial correctness assertions, as the following strengthening of Theorem 4.7 shows.

7.4 *Theorem*: For any finite type τ with enough variables and at least one function symbol of positive arity, there exist inequivalent recursively enumerable schemes a, b such that $\tau(a) \cup \tau(b) = \tau$ and for any weak extension τ' of τ,

$$PC_{\tau'}(a) = PC_{\tau'}(b);$$

and

$$T_{\tau'}(a) = T_{\tau'}(b).$$

Outline of Construction: The construction is an amalgam of the constructions used in Theorem 4.7 and Appendix F. We can assume without loss of generality that we have a unary function symbol f in τ. We define a flowchart *switch* which generates $f^n(x)$ and $f^m(x)$ for some nondeterministically chosen integers $m \neq n$ and then, using array assignments, switches the values of these two terms. Using random assignments as in the proof of Theorem 4.7, we also define a finite flowchart *trivial* which diverges on states in which every predicate and every function besides f is trivial, and acts like *NOP* otherwise. Then let a be *switch* ∪ *NOP* and b be *switch* ∪ *trivial*.

Again, we omit the remainder of the proof. □

Curiously, if *array* assignments are disallowed, then termination assertions

without any extra symbols aside from variables can distinguish inequivalent schemes -- even arbitrary schemes.

7.5 *Theorem*: Suppose a, b are inequivalent arbitrary schemes without array assignments. Then we can find a weak extension τ' of $\tau(a) \cup \tau(b)$ such that $T_{\tau'}(a) \neq T_{\tau'}(b)$.

Proof: The proof is just that of Theorem 5.3, except that since b_{i_0} might now have random assignments, $<b_{i_0}>\neg H$ is not in general quantifier-free (note that $<x := ?>F$ is equivalent to $\exists x F$). □

Note that the proof of Theorem 4.7 shows that partial correctness assertions do not share the property of termination assertions given above in Theorem 7.5. This provides another instance illustrating the somewhat nicer theoretical properties of termination as opposed to partial correctness assertions.

8. Conclusion

We have given a rigorous formulation of the thesis that programming languages can be specified axiomatically using before-after assertions. We examined this thesis in detail in the context of first order assertions about program schemes, and concluded that in theory there is an axiomatic system for effectively deriving enough assertions to define the behavior of any effective scheme.

We interpret the technical complication of the proofs of these positive results for the case of partial correctness assertions as suggesting that language specification by such assertions is likely to be awkward in practice. Termination assertions fare somewhat better in two respects: the proofs are straightforward, and these assertions have a simple complete axiomatization for **while**-programs. Another significant advantage of termination in contrast to partial correctness assertions is that the former can be viewed as a kind of operational specification -- with the proof of a termination assertion about a program corresponding to steps in a computation of the program. We shall not elaborate on this observation here, but we expect this property would make a specification using termination assertions much more useful to programming language implementors. (It would be interesting to examine extensions of the termination axioms to more realistic kinds of program schemes than **while**-schemes.)

The property of completeness plays a smaller role in this paper than in many others on proving assertions about programs (eg. [Cook, 1978; Apt, Bergstra, and Meertens, 1979]). The reason is simple -- the property of completeness of an axiom

system and the property of its yielding enough assertions to define semantics are independent. As we noted just before stating Theorem 4.5, no complete effective axiom system for partial correctness assertions even about **while**-programs is possible, yet a recursively enumerable set of such assertions does define semantics. Conversely there are also cases, e.g., using quantifier-free formulas, where the complete set of true assertions about some class of programs does not define semantics.

Some comments regarding our use of first order predicate calculus must be made here. A reasonable rejoinder to our concern about the technical subtleties arising from predicate calculus is that one could choose instead some richer logical language such as weak second order predicate calculus or first order arithmetic (augmented with the uninterpreted function and predicate symbols appearing in programs). For these richer logical languages the subtleties disappear; it is easy to show that partial correctness and/or termination assertions involving formulas from these languages define input-output semantics. Indeed these richer logical languages are the ones implicitly used in most of the literature on axiomatic definitions of programming languages.

But on adopting richer logical languages, one pays a high price in terms of effective proof procedures. The tautologies of weak second order predicate calculus and first order arithmetic augmented with uninterpreted function symbols are of quite high degree of undecidability (Π_1^1). Since one of the central aims of axiomatic programming language definitions is to ease the generation of proofs about programs, it is obviously important to retain constructive proof properties such as those of predicate calculus.

Further, we observe that with minor exceptions the use of arithmetic formulas in the literature on axiomatic programming languages arises only in the specification of arithmetic primitive operations in programs. Above the level of primitive operations (for example in the specifications of sequencing constructs or subroutine calls) the arithmetic properties of the primitives are not used. Thus the specifications fit within the framework of predicate calculus and uninterpreted flowchart schemes considered above, and we therefore believe our first order framework is the most appropriate one for assessing the thesis that programming languages can be specified axiomatically.

APPENDIX A

Proof of Lemma 3.3: We want to show that $F\{NOP\}G$ iff $F\{a_0\}G$ for F, G quantifier-free.

Note $R_{NOP} - R_{a_0} = \{(s, s) | s \vDash x = y \wedge z = f(x) \wedge (x, f(x), f^2(x), \ldots \text{ are all distinct})\}$. Since $R_{a_0} \subseteq R_{NOP}$, we have that $PC(NOP) \subseteq PC(a_0)$. Now suppose, in order to obtain a contradiction, $(F, G) \in PC(a_0) - PC(NOP)$, where F and G are quantifier-free. Then there must be a state s_0 such that $(s_0, s_0) \in R_{NOP} - R_{a_0}$ and $s_0 \vDash F \wedge \neg G$.

Now the truth of a quantifier-free formula in a state s only depends on the values in s of a finite number of terms in the Herbrand universe. More precisely, we mean the following:

Define a *partial state* to be a domain D and an arity-respecting assignment of *partial* functions and predicates on D to function and predicate symbols; (cf. [Meyer and Winklmann, 1979]). For a partial state s and a formula H, we say s *satisfies* H and write $s \vDash H$ iff $t \vDash H$ for every total state t which extends s[5]. Then it can easily be shown by induction on formulas that for any state s and quantifier-free formula H, if $s \vDash H$ then there is a partial state s′ with finite domain such that s extends s′ and $s' \vDash H$. Going back to our proof, let s_1 be a partial state with finite domain such that s_0 extends s_1 and $s_1 \vDash F \wedge \neg G$. Since $(s_0, s_0) \in R_{NOP} - R_{a_0}$, it must be the case that $s_0 \vDash (x, f(x), f^2(x), \ldots \text{ are all distinct})$. Let $n_0 = \max\{n | f^n(x) \in Dom(s_1)\}$. Finally, let t be a state which extends s_1 such that $t \vDash f^{n_0+1}(x) = f^{n_0}(x)$. Then $t \vDash F \wedge \neg G$, but $(t, t) \in R_{a_0}$, contradicting $F\{a_0\}G$. □

APPENDIX B

Proof of Lemma 4.2: For ease of notation, we will assume that the only symbols that occur in the type τ are one binary predicate symbol R, a unary function symbol f, and variables x_1 and x_2. We augment τ by adding new binary function symbols $+$, $\cdot$, and π (which will be a coding function), a binary predicate symbol T (which will be the truth predicate), a unary predicate symbol N (which will be interpreted as the natural numbers), constants 0 and 1, and a countable supply of variables x_3, x_4, ... We call this augmented type τ'.

Without loss of generality, the F_i's are built up from $\wedge$, $\neg$, $=$, and $\exists$, and x_1, x_2, ... (with the free variables of each F_i contained in $\{x_1, x_2\}$.) Further we may assume that the arguments of R and f are always variable symbols, and there is always a variable symbol on the left hand side of $=$. For example $R(f(f(x_1)), x_2)$ could be rewritten as:

$$\exists x_3 \exists x_4 (R(x_4, x_2) \wedge x_4 = f(x_3) \wedge x_3 = f(x_1))$$

We can attach a Gödel number to each formula of type τ in some standard way. In what follows, we will use the notation $\ulcorner F \urcorner$ to denote the Gödel number of formula F and ϕ_n to denote the formula F such that $\ulcorner F \urcorner = n$. The hypothesis that F_1, F_2, F_3, ... is a recursively enumerable set of formulas means that $\{\ulcorner F_1 \urcorner, \ulcorner F_2 \urcorner, \ulcorner F_3 \urcorner, ...\}$ is a recursively enumerable set of integers.

Let AX be a finite set of axioms true of arithmetic and sufficiently strong to allow representability of partial recursive functions (cf. [Machtey and Young, 1978; pp. 123 - 128]); i.e. for any partial recursive function $g : \mathbb{N}^k \mapsto \mathbb{N}$, we can effectively find a formula $F_g(x_1, ..., x_k, z)$ of type τ' such that

$$\models AX \Rightarrow [(F_g(x_1, ..., x_k, y) \wedge F_g(x_1, ..., x_k, z)) \Rightarrow y = z]$$

and for all natural numbers $n_1, ..., n_k, m$, if $g(n_1, ..., n_k) = m$ then

$$\models AX \Rightarrow F_g(n_1, ..., n_k, m)$$

(where we let k denote $(1 + (1 + ... (1 + 1) ...)$ (k times))

By definition of Gödel numbering one can effectively decide, given $m, i, j \in \mathbb{N}$, whether $m = \ulcorner R(x_i, x_j) \urcorner$. Thus, by recursive representability of partial recursive functions, we can effectively find a formula PRED, and similarly formulas EQ, FUN,

CON, NEG, EX, and ISF, all of type τ', such that if i, j, m, n, and p are integers and $s \vDash AX$ then:

1. $s \vDash PRED(\mathbf{m}, \mathbf{i}, \mathbf{j})$ iff $m = \ulcorner R(x_i, x_j) \urcorner$,

2. $s \vDash EQ(\mathbf{m}, \mathbf{i}, \mathbf{j})$ iff $m = \ulcorner x_i = x_j \urcorner$,

3. $s \vDash FUN(\mathbf{m}, \mathbf{i}, \mathbf{j})$ iff $m = \ulcorner x_i = f(x_j) \urcorner$,

4. $s \vDash CON(\mathbf{m}, \mathbf{n}, \mathbf{p})$ iff n and p are the Gödel numbers of formulas and $m = \ulcorner \phi_n \wedge \phi_p \urcorner$,

5. $s \vDash NEG(\mathbf{m}, \mathbf{n})$ iff n is the Gödel number of a formula and $m = \ulcorner \neg \phi_n \urcorner$,

6. $s \vDash EX(\mathbf{m}, \mathbf{i}, \mathbf{n})$ iff n is the Gödel number of a formula and $m = \ulcorner \exists x_i \phi_n \urcorner$,

7. $s \vDash ISF(\mathbf{m})$ iff $m = \ulcorner F_i \urcorner$ for some i. (Note that ISF(m) says that m is the Gödel number of one of the formulas in our recursively enumerable set.)

Let the formula G be the conjunction of the following ten clauses. (For readability we use q, r, x, y, etc. as variables instead of x_1, x_2, ...)

1. AX.

This ensures that if $s \vDash G$, s is a (possibly nonstandard) copy of the integers under addition and multiplication.

2. $N(0) \wedge \forall q(N(q) \Rightarrow N(q + 1))$.

The subset of Dom(s) which satisfies N will also be a (possibly nonstandard) copy of the integers.

3. $\forall t \forall q(N(q) \Rightarrow \forall u \exists t'(\forall r((N(r) \wedge r \neq q) \Rightarrow \pi(t, r) = \pi(t', r)) \wedge \pi(t', q) = u))$.

This says that the projection function π has the property that for any t, u, and integer q we can find an t' whose projections agree with t for every integer $r \neq q$. Moreover, $\pi(t', q) = u$.

Clauses 4 - 9 ensure that the truth predicate T acts properly on formulas:

4. $\forall x \forall t \forall q \forall r[(PRED(x, q, r) \wedge N(x) \wedge N(q) \wedge N(r)) \Rightarrow (T(x,t) \equiv R(\pi(t, q), \pi(t, r)))]$.

5. $\forall x \forall t \forall q \forall r[(FUN(x, q, r) \wedge N(x) \wedge N(q) \wedge N(r)) \Rightarrow (T(x, t) \equiv \pi(t, q) = f(\pi(t, r)))]$.

6. $\forall x \forall t \forall q \forall r[(EQ(x, q, r) \wedge N(x) \wedge N(q) \wedge N(r)) \Rightarrow (T(x, t) \equiv \pi(t, q) = \pi(t, r))]$.

7. $\forall x \forall t \forall y \forall z[(CON(x, y, z) \wedge N(x) \wedge N(y) \wedge N(z)) \Rightarrow (T(x, t) \equiv (T(y, t) \wedge T(z, t)))]$.

8. $\forall x \forall t \forall y[(NEG(x, y) \wedge N(x) \wedge N(y)) \Rightarrow (T(x, t) \equiv \neg T(y, t))]$.

9. $\forall x \forall t \forall q \forall y[(EX(x, q, y) \wedge N(x) \wedge N(q) \wedge N(y)) \Rightarrow (T(x, t) \equiv \exists t'(\forall r((N(r) \wedge r \neq q) \Rightarrow \pi(t', r) = \pi(t, r)) \wedge T(y, t')))]$.

Finally, clause 10 ensures us that if $s \vDash G$ then all the formulas in the recursively enumerable set $F_1, F_2, \ldots$ will be true in s:

10. $\forall x \forall t((N(x) \wedge ISF(x) \wedge \pi(t, 1) = x_1 \wedge \pi(t, 2) = x_2) \Rightarrow T(x, t))$.

Claim 1: Suppose $s \vDash G$ and $F(x_{i_1}, \ldots, x_{i_n})$ is a formula of τ whose free variables are contained in $\{x_{i_1}, \ldots, x_{i_n}\}$. Let $m = \ulcorner F \urcorner$, and suppose $a \in Dom(s)$. Then $s \vDash F(\pi(a, i_1), \ldots, F(\pi(a, i_n))$ iff $s \vDash T(m, a)$.

(There has been a small abuse of notation here since the domain element a is not a symbol of τ' and hence should not appear in the body of a formula. Technically, instead of $s \vDash T(m, a)$ we should have written $s^y_a \vDash T(m, y)$, where s^y_a denotes state s modified so that the value of variable y is a. Similar comments apply to all other formulas in which elements of Dom(s) appear).

Proof: Suppose $s \vDash G$. The cases where F is of the form $R(x_i, x_j)$, $x_i = x_j$, $x_i = f(x_i, x_j)$, $F_1 \wedge F_2$. or $\neg F$ are easily taken care of by clauses 4 - 8 in the definition of G. The only difficulty occurs when F is existentially quantified. For notational convenience, let us assume that F has only one free variable x_i and is of the form $\exists x_j H(x_i, x_j)$.

Let $m = \ulcorner F \urcorner$, $m' = \ulcorner H \urcorner$. Note we thus have

1. $s \vDash EX(m, j, m')$.

Suppose $a \in Dom(s)$ and $s \vDash F(\pi(a, i))$, or equivalently $s \vDash \exists x_j H(\pi(a, i), x_j)$. Then for some $b \in Dom(s)$,

2. $s \vDash H(\pi(a, i), b)$.

By clause 3 in the definition of G, there is a $c \in Dom(s)$ such that

3. $s \models \forall r((N(r) \wedge r \neq \mathbf{j}) \Rightarrow \pi(c, r) = \pi(a, r)) \wedge \pi(c, \mathbf{j}) = b$.

In particular we have

4. $s \models \pi(c, \mathbf{i}) = \pi(a, \mathbf{i}) \wedge \pi(c, \mathbf{j}) = b$.

Thus, from 2 and 4 it follows that

5. $s \models H(\pi(c, \mathbf{i}), \pi(c, \mathbf{j}))$.

By the induction hypothesis,

6. $s \models T(\mathbf{m}', c)$.

Finally by 1, 3, 6, and clause 9 in the definition of G, we have

$$s \models T(\mathbf{m}, a).$$

For the converse suppose $a \in Dom(s)$ and $s \models T(\mathbf{m}, a)$. By clause 9 again, there is some $a' \in Dom(s)$ such that $s \models \pi(a, \mathbf{i}) = \pi(a', \mathbf{i}) \wedge T(\mathbf{m}', a')$. By the induction hypothesis we have,

$$s \models H(\pi(a', \mathbf{i}), \pi(a', \mathbf{j})).$$

Thus it follows that

$$s \models \exists x_j H(\pi(a', \mathbf{i}), x_j)$$

or equivalently,

$$s \models F(\pi(a, \mathbf{i})). \quad \square$$

Claim 2: For any formula $F(x_1, x_2)$ with free variables contained in $\{x_1, x_2\}$, if $m = \ulcorner F \urcorner$ and $s \models G$, then

$$s \models F \text{ iff } s \models \forall t((\pi(t, \mathbf{1}) = x_1 \wedge \pi(t, \mathbf{2}) = x_2) \Rightarrow T(\mathbf{m}, t)).$$

Proof: Immediate from Claim 1. (Note that clause 3 in the definition of G ensures that we can always find a t such that $s \models \pi(t, \mathbf{1}) = x_1 \wedge \pi(t, \mathbf{2}) = x_2$.) $\square$

Claim 3: $\models G \Rightarrow \wedge_i F_i$.

Proof: If $s \models G$ and $m = \ulcorner F_i \urcorner$, we must have $s \models ISF(\mathbf{m})$. Then by clause 10 of the definition of G, it follows that

$$s \models \forall t((\pi(t, \mathbf{1}) = x_1 \wedge \pi(t, 2) = x_2) \Rightarrow T(\mathbf{m}, t)).$$

Since the only free variables of F_i are contained in $\{x_1, x_2\}$, from Claim 2 we have $s \models F_i$. And thus it follows that $s \models \wedge_i F_i$. □

Claim 4: If $s \models \wedge_i F_i$ and Dom(s) is infinite, then there is an expansion s' of $s|_\tau$ to type τ' such that $s' \models G$.

Proof: By the upward Lowenheim-Skolem Theorem, AX has a model of the same cardinality as s, so there is no problem expanding $s|_\tau$ to an s' such that $s' \models AX$. Let N be true only on the integers in Dom(s); then s' satisfies clause 2 in the definition of G.

We define π to satisfy clause 3 as follows:

Let $A = \{f \mid f : \mathbb{N} \mapsto Dom(s') \text{ such that } f(k) = 0 \text{ for all but finitely many } k\}$. A has the same cardinality as Dom(s'), so with each $x \in Dom(s')$ we can bijectively associate an $f_x \in A$.

If q is an integer in Dom(s') (i.e. if $s' \models N(q)$), define $\pi(x, q) = f_x(q)$. Then extend π arbitrarily to the rest of Dom(s'). It is easy to check that with this definition of π, clause 3 is satisfied by s'.

Finally, define T on Dom(s') such that $s' \models T(\mathbf{m}, a)$ iff m is the Gödel number of a formula $F(x_{i_1}, ..., x_{i_n})$ and $s \models F(\pi(a, \mathbf{i}_1), ..., \pi(a, \mathbf{i}_n))$. This implies that s' satisfies clauses 4 - 9. Finally, since $s \models \wedge_i F_i$, it follows that s' satisfies clause 10.

Thus $s' \models G$. □

Claims 1 - 4 complete the proof of Theorem 4.2. □

APPENDIX C

C.1 There are inequivalent recursively enumerable schemes *b*, *c* of infinite type such that PC(*b*) = PC(*c*).

Proof: Let *buzz* be the program scheme which always diverges:

while x = x **do** *NOP* **od**

Take *b* to be *NOP* and *c* to be the non-deterministic scheme of infinite type

$$\cup_{i\neq 1}\ (\textbf{if } x_i = x_1 \textbf{ then } NOP \textbf{ fi})$$

Clearly when *c* halts it behaves like *NOP*. Hence $R_c \subseteq R_{NOP}$, and PC(*NOP*) $\subseteq$ PC(*c*).

Suppose, in order to obtain a contradiction, (F, G) $\in$ PC(*c*) - PC(*NOP*). Thus, for some s with (s, s) $\in R_{NOP} - R_c$, $s \models F \wedge \neg G$. But note that $R_{NOP} - R_c = \{(s, s) \mid s \models \wedge_{i\neq 1}(x_i \neq x_1)\}$. Let $n_0 = \max\ \{n \mid x_n \in \tau(F) \cup \tau(G)\}$, and let $s' = R_{(x_{n_0+1} := x_1)}(s)$. Thus

$$s'(x_i) = \begin{cases} s(x_i) & \text{if } i \neq n_0 + 1 \\ s'(x_1) & \text{if } i = n_0 + 1 \end{cases}$$

Since $s' \models x_{n_0+1} = x_1$, we must have (s', s') $\in R_c$, but $s' \models F \wedge \neg G$, contradicting (F, G) $\in$ PC(*c*). □

C.2 There are inequivalent non-recursively enumerable schemes *b*, *c* of finite type such that PC(*b*) = PC(*c*).

Proof: Consider a language with a countable number of constants, and a countable number of functions and predicates of each arity. Let $F_1, F_2, F_3, \ldots$ be an enumeration of all satisfiable formulas in this language.

Let R, f, x, be respectively a unary predicate, unary function, and constant in the language. For each i define

$$R_i = \begin{cases} R(f^i(x)) & \text{if } F_i \wedge R(f^i(x)) \text{ is satisfiable.} \\ \neg R(f^i(x)) & \text{otherwise} \end{cases}$$

Note that by the construction, the formula $F_i \wedge R_i$ must be satisfiable for all i.

Take b to be *NOP* and let c be the (non-recursively enumerable) scheme

$$\cup_i \text{ (if } R_i \text{ then } \textit{NOP} \text{ else } \textit{buzz} \text{ fi)}.$$

Again, if c halts it behaves like *NOP*. Note that $R_b - R_c = \{(s, s) | s \vDash \wedge_i \neg R_i\}$. Thus $R_c \subset R_b$ (the containment being proper since $\{\neg R_i \mid i = 1,2,3,...\}$ is consistent) and $PC(b) \subseteq PC(c)$.

Now suppose $(F, G) \in PC(c) - PC(b)$. Thus $F \wedge \neg G$ must be satisfiable, so $F \wedge \neg G = F_{i_0}$ for some i_0. But $F_{i_0} \wedge R_{i_0}$ is satisfiable, say by s. Since $s \vDash R_{i_0}$, we must have $(s, s) \in R_c$, but $s \vDash F \wedge \neg G$, contradicting $(F, G) \in PC(c)$.

Hence b is not equivalent to c, but $PC(b) = PC(c)$. □

APPENDIX D

Proof that we can find a recursively enumerable set $\mathscr{P}$ which defines the semantics of **while**-program schemes with first order tests, and array and random assignments:

The proof of Theorem 4.1 shows it is sufficient to find a recursively enumerable $\mathscr{P}'$ such that $\mathscr{P}' \supseteq \{G_{H,b}\{b\}H \mid H$ is a first order formula, b is a **while**-program scheme$\} \cup \{ISO_s\{b\}\neg ISO_t \mid Dom(s) = Dom(t)$ is finite, $(s, t) \notin R_b\}$. That such a $\mathscr{P}'$ can be found is shown as follows:

Definition: A formula F is said to be k-bounded iff $s \models F$ implies $|Dom(s)| \leq k$. F is bounded if F is k-bounded for some k.

Remarks: 1. Note F is k-bounded iff $\models F \Rightarrow \exists y_1 \ldots \exists y_k \forall z(z = y_1 \vee \ldots \vee z = y_k)$. Thus it follows that the set of bounded formulas is recursively enumerable, since we can enumerate all the tautologies of first order predicate calculus with equality.

2. Note that if s has a finite domain, then ISO_s is bounded.

Lemma: For any **while**-program scheme b, the set $\{F\{b\}G \mid F, G$ are bounded $\wedge$ $F\{b\}G$ is true$\}$ is recursively enumerable. Moreover, it can be found uniformly from b.

Proof: If F is k-bounded, there is a G such that G is k-bounded and $G \equiv [b]F$. To see this, note that if τ is the similarity type of b, there are only finitely many distinct states with k elements and similarity type τ. Let $s_1, \ldots, s_m$ be those which model $[b]F$. Then take G to be $ISO_{s_1} \vee \ldots \vee ISO_{s_m}$. From this condition, often referred to as expressiveness, it is easy to see by induction on the structure of **while**-program schemes that the Floyd-Hoare axioms are complete for bounded predicates. [cf. Greif and Meyer, 1979; Cook, 1978; Harel, 1979.]

Hence $\{F\{b\}G \mid F,G$ bounded, b is a **while**-program scheme, and $F\{b\}G$ is true$\}$ is recursively enumerable and clearly can be found uniformly from (an index for) b. □

Returning now to the proof of the main theorem, just take $\mathscr{P}' = \{G_{H,b}\{b\}H \mid$ H is a first order formula, b a **while**-program scheme$\} \cup \{F\{b\}F' \mid F, F'$ are bounded, b a **while**-program scheme$\}$ and we are done. □.

APPENDIX E

Proof that $PC_{\mathcal{T}}(NOP) = PC_{\mathcal{T}}(c)$, from 4.7:

Let A_0 be $\forall x_1 \ldots \forall x_N(\text{Trivial}(x_1, \ldots, x_N))$ and let A_k be $\forall x(f^k(x) \neq x)$, for k = 1, 2, 3, ... Let $T = \{A_0, A_1, A_2, \ldots\}$. T axiomatizes the theory of the integers with successor and predecessor.

Since $R_c \subseteq R_{NOP}$, we know $PC(NOP) \subseteq PC(c)$. To obtain a contradiction, suppose $(F, G) \in PC(c) - PC(NOP)$. Note that $R_{NOP} - R_c = \{(s, s) | s \models T\}$. Thus there must be some state s_0 such that $(s_0, s_0) \in R_{NOP} - R_c$ and $s_0 \models F \wedge \neg G$. In fact, we must have

$$s_0 \models T \wedge F \wedge \neg G.$$

The theory of the integers under successor and predecessor is well known to admit elimination of quantifiers (cf. [Enderton, 1972, pp. 178 - 183]), so there is a quantifier-free formula H such that

$$T \vdash H \equiv (F \wedge \neg G).$$

Thus, for some n_0,

$$\{A_0, A_1, \ldots, A_{n_0}\} \vdash H \equiv (F \wedge \neg G). \quad (*)$$

Finally, note that $s_0 \models H$ since $s_0 \models T \wedge F \wedge \neg G$.

We want to find a structure s_1 with the following properties;

(i) $s_1 \models H$
(ii) $s_1 \models \{A_0, A_1, \ldots, A_{n_0}\}$
(iii) $s_1 \not\models T$

If we can find such an s_1, we will have our desired contradiction, since (i), (ii), and (*) will allow us to deduce $s_1 \models F \wedge \neg G$. But from (iii) it follows $(s_1, s_1) \in R_b$, and this contradicts $(F, G) \in PC(b)$.

Define s_1 to be the extension of s_0 such that $Dom(s_1) = Dom(s_0) \cup \{a_1, \ldots, a_{n_0}\}$, where $a_1, \ldots, a_{n_0}$ are new elements in the domain, all predicates are still trivial on $Dom(s_1)$, and all functions besides f and g are still projections on the first coordinate.

Then extend f so that $f(a_i) = a_{i+1}$ for $i = 1, \ldots, n_0\text{-}1$, and $f(a_{n_0}) = a_0$, and extend g so it is still the inverse of f.

It should now be clear that $s_1 \models \{A_0, A_1, \ldots, A_{n_0}\}$, but $s_1 \models \neg A_{n+1}$ (since $s_1 \models f^{n+1}(a_1) = a_1$) so $s_1 \not\models T$. Since $s_0 \models H$, H is quantifier-free, and s_1 is an extension of s_0, it must be the case that $s_1 \models H$. Thus (i), (ii), and (iii) are all satisfied, giving us the desired contradiction. □

APPENDIX F

Example of two inequivalent recursively enumerable schemes b, c with open tests and array assignments only, such that $T_{Open}(b) = T_{Open}(c)$, where *Open* is the set of true quantifier-free termination assertions:

Let b be the scheme $\cup_{m>n} b_{<m,n>}$, where $b_{<m,n>}$ is $f^m(x) := f^n(x)$. Let $c = b \cup NOP$.

$R_b \subseteq R_c$, so clearly $T_{Open}(b) \subseteq T_{Open}(c)$. To prove the converse containment, first note $R_c - R_b = \{(s, s) |\ s \models (x, f(x), f^2(x), \ldots \text{ are all distinct})\}$. So if $(F, G) \in T_{Open}(c) - T_{Open}(b)$, there must be some state s_0 such that $(s_0, s_0) \in R_c - R_b$ and $s_0 \models F \wedge G$.

As in Appendix A, let s_1 be a partial state with finite domain such that s_0 extends s_1 and $s_1 \models G$, and let $n_0 = \max\{n |\ f^n(x) \in \mathrm{Dom}(s_1)\}$. Let $t = b_{<n_0+1,\, n_0>}(s_0)$. Then $(s_0, t) \in R_b$, $s_0 \models F$ and $t \models G$ (since t extends s_1), and this contradicts $(F, G) \notin T_{Open}(b)$. □

Notes

1. Input-output semantics do not reflect the full operational behavior of programs, e.g., information about "looping" or "failing" is lost, but dealing with such extended semantics and the kind of *weakest pre-condition* assertions proposed by Dijkstra capable of defining this extended semantics is technically more complicated and does not appear to raise any new issues about the nature of axiomatic definitions of programs. (cf. [Greif and Meyer, 1979; Hoare, 1978; Harel, 1979].)

2. In the terminology of [Greif and Meyer,1979], $\mathscr{P}$ *determines* the standard relational semantics of $\mathscr{A}$. An equivalent definition is given in [Blikle, 1979].

3. Kleene's result for pure first order predicate calculus (without equality) was formulated in terms of finite axiomatizability, namely, the theory of any recursively enumerable set of axioms of finite type equals the theory restricted to the same finite type of some finite set of axioms. In general, however, the finite set of axioms must involve extra symbols.

Lemma 4.2 immediately implies Kleene's result, since without equality a first order formula is satisfiable iff it is satisfiable in an infinite domain. Conversely, Kleene's proof of his result is quite similar to the authors' independent proof given in Appendix B.

4. Given two types τ, τ' with $\tau \subseteq \tau'$, s a state of type τ, s' a state of type τ', then s' is an *expansion* of s if Dom(s) = Dom(s') and $s'|_\tau = s$.

5. If s and s' are two states (respectively partial states; cf. Appendix A) of the same type, then s' is an extension of s if Dom(s) $\subseteq$ Dom(s'), and for any function symbol f, if f_S is the function assigned by s to the symbol f, then $f_S \subseteq f_{S'}$ as relations, and for every predicate symbol P, $P_{S'}$ restricted to Dom(s) is equal to P_S.

6. It is not too difficult to show that the set of all true termination assertions defines the semantics of arbitrary deterministic schemes, even those of infinite type.

7. If we disallow array assignments, we can even handle the case where τ is not sufficiently rich, but then we must use a weak extension of τ in order to achieve the desired distinctions.

REFERENCES

Apt, K. R., J. A. Bergstra, L. G. L. T. Meertens. Recursive Assertions Are Not Enough - Or Are They? *Theoretical Computer Science*, vol. 8, pp. 73 - 87, 1979.

Bergstra, J. A., J. Tiuryn, and J. V. Tucker. Correctness Theories and Program Equivalence. Preprint. Stichting Mathematisch Centrum, Amsterdam, 1979.

Blikle, A. A Survey of Input-Output Semantics and Program Verification. Institute of Computer Science. Polish Academy of Sciences. 1979.

Cook, S. A. Soundness and Completeness of an Axiom System for Program Verification. *SIAM Journal on Computing*, vol. 7, no. 1, pp. 70 - 90, February, 1978.

Dijkstra, E. W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Comm. of the A.C.M.*, vol. 18, no. 8, pp. 453 - 457, 1975.

Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, 1976.

Enderton, H. B. *A Mathematical Introduction to Logic.* Academic Press, 1972.

Floyd, R. W. Assigning Meaning to Programs. In *Mathematical Aspects of Computer Science. Proceedings of Symposium in Applied Mathematics* (ed. J. T. Schwartz), pp. 19 - 33. American Math. Society, Providence, Rhode Island, 1967.

Friedman, H. Algorithmic Procedures, Generalized Turing Algorithms, and Elementary Recursion Theory. In *Logic colloquiuum*, 1969 (ed. R. O. Gandy and C. M. E. Yates), pp. 316 - 389. North Holland, Amsterdam, 1971.

Greif, I. and A. R. Meyer. Specifying the Semantics of **While**-Programs. M.I.T., Laboratory for Computer Science, TM-130. M.I.T., Cambridge, Mass. 02139. April, 1979.

Harel, D. *First-Order Dynamic Logic. Lecture Notes in Computer Science, 68.* Springer-Verlag, N.Y., 1979.

Harel, D., A. R. Meyer, and V. R. Pratt. Computability and Completeness in Logics of Programs. *Proceedings of Ninth Annual A.C.M. Symposium on Theory of Computing*, pp. 261 - 268, May, 1977.

Hoare, C. A. R. Some Properties of Predicate Transformers. *Journal of the A. C. M.*, vol. 25, no. 3, pp. 461 - 480, July, 1978.

Hoare, C. A. R. and P. Lauer. Consistent and Complementary Formal Theories of the Semantics of Programming Languages. *Acta Informatica 3*, pp. 135 - 155, 1974.

Hoare, C. A. R. and N. Wirth. An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica 2*, pp. 335 - 355, 1973.

Janssen, T. M. V. and P. van Emde Boas. The Expressive Power of Intensional Logic in the Semantics of Programming Languages. Preprint. Stichting Mathematisch Centrum, Amsterdam, May, 1977.

Kleene, S. C. *Two Papers on the Predicate Calculus. Memoirs of the American Math. Soc., No. 10*, pp. 27 - 66. American Math. Society Providence, Rhode Island, 1952.

London, Ralph L. Program Verification. In *Research Directions in Software Technology.* ed. Peter Wegner, pp. 302 - 315. M.I.T. Press, Cambridge, Mass., 1978.

Machtey, M. and P. Young. *An Introduction to the General Theory of Algorithms.* North Holland, 1978.

Meyer, A. R. and J. Y. Halpern. Axiomatic Definitions of Programming Languages: A Theoretical Assessment. *Proceedings of the 7th Annual Symposium on Principles of Programming Languages,* pp. 203 - 212, January, 1980.

Meyer, A. R. and K. Winklmann. On the Expressive Power of Dynamic Logic. *Proceedings of the 11th Annual Conference on Theory of Computing.* pp. 167 - 175. May, 1979.

Pratt, V. R. Semantical Considerations of Floyd-Hoare Logic. *17th Annual IEEE Symposium on the Foundations of Computer Science,* pp. 109 - 121, October, 1976.

Schwartz, R. L. An Axiomatic Semantic Definition of Algol 68. UCLA-ENG-7838; UCLA-34P214-75, University of California at Los Angeles, Los Angeles, Calif. 1978.

Schwartz, R. L. An Axiomatic Treatment of ALGOL 68 Routines. *Automata, Languages and Programming, Sixth Colloquium. Lecture Notes in Computer Science 71.* pp. 530 - 545. Springer-Verlag, N.Y., 1979.