MIT/LCS/TM-214

"TERMINATION ASSERTIONS FOR RECURSIVE PROGRAMS: COMPLETENESS AND AXIOMATIC DEFINABILITY"

Albert R. Meyer

John C. Mitchell

March 1982

Termination Assertions for Recursive Programs: Completeness and Axiomatic Definability

Albert R. Meyer and John C. Mitchell
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139, USA

3 March 1982

Keywords: Axiomatic Definitions of Programming Languages, Complete Axiomatization, Semantics of Programming Languages, Termination Assertions, Recursive Procedures

This work was supported in part by The National Science Foundation, Grant Nos. MCS 7719754 and MCS 8010707, and by a grant to the M.I.T. Laboratory for Computer Science from the IBM Corporation. The second author is also supported by a fellowship from the National Science Foundation.

Table of Contents

Abstract

The termination assertion p<S>q means that whenever the formula p is true, there is an execution of the possibly nondeterministic program S which terminates in a state in which q is true. A recursive program S may declare and use local variables and nondeterministic recursive procedures with call-by-address and call-by-value parameters, in addition to accessing undeclared variables and global procedures. Assertions p and q about calls to global procedures are first order formulas extended to express hypotheses about the termination of calls to undeclared global procedures. A complete, effective axiom system with axioms corresponding to the syntax of the programming language is given for the termination assertions valid over all interpretations. Termination assertions define the semantics of recursive programs in the following sense: if two programs have different input-output semantics, then there is a termination assertion that is valid for one program but not for the other. Thus the complete axiomatization of termination assertions constitutes an axiomatic definition of the semantics of recursive programs.

1. Introduction

Many formal systems for proving properties of programs consist of rules for deriving partial correctness assertions (cf. [2]). Although these assertions express many interesting and useful properties of programs, the assertions valid over all interpretations are technically more difficult to derive than assertions which state termination explicitly. The first order *partial correctness assertion* p{S}q means that if the first order formula p holds initially, and if the possibly nondeterministic program S halts, then the first order formula q holds after each possible halting computation of S. For even a very restricted set of p, S and q, the set of valid partial correctness assertions is not recursively enumerable [12, 15]. In contrast, the set of valid termination assertions about while programs has a simple axiomatization [16]. The *termination assertion* p<S>q means that if p is true initially, then there is a possible computation of S which terminates in a state in which q is true.[1] Since

[1] If S is deterministic, then the termination assertion p<S>q is equivalent to the partial correctness assertion p{S}q along with the assurance that S halts whenever p is true initially.

the valid termination assertions are effectively enumerable and the valid partial correctness assertions are not, it is possible that axiom systems for termination assertions may be more useful for proving properties of programs than partial correctness rules.

In order to integrate correctness proofs into program development, a proof system should allow the structures of proofs to correspond to the structures of programs. For a programming language whose basic structural unit is the procedure, a natural approach to software development is to specify the input-output behavior of all procedures. The correctness of a procedure P should then follow from the assumption that the procedures called by P meet their specifications; it should not depend on how these procedures might be written. Furthermore, once procedure implementations have been shown to satisfy their specifications, the proof rules should allow one to merge the proofs about procedures into a proof about the encompassing program. We consider recursive programs with calls to undeclared global (accessible everywhere) procedures. Since the meanings of global procedures are not determined by the calling programs, our preconditions must be able to express hypotheses about global procedures. To accomplish this, we extend the usual class of first order formulas to include termination assertions about global procedure calls. The resulting language, *first-order assertions about global procedures*, provides a convenient method for specifying the behavior of procedures and shares many essential properties with standard first-order logic. In addition, our proof rules show explicitly how to combine a proof about a declared procedure P with a proof about a calling program S to obtain a proof about the program which declares P and executes S.

We present a complete axiomatization of termination assertions about programs which include local variable declarations, calls to undeclared global procedures, and nondeterministic recursive procedures with call-by-address and call-by-value parameters. The axioms and rules of inference are sufficient to prove *all* termination assertions which are valid over *all* interpretations. In this respect, our completeness theorem contrasts with the usual "relative completeness" theorems for partial correctness assertions (e.g. [6]). That

is, we do not need to restrict ourselves to special structures and do not have to assume that the first order theory of any structure is given as axioms. As a result, our completeness theorem (Theorem 1) demonstrates that the valid termination assertions for recursive programs are recursively enumerable. This theorem extends the similar completeness theorem (Theorem 6.4) of Meyer and Halpern [16] for while programs without procedures.

The fact that the valid terminations assertions are so constructive suggests that they may suffer limitations in providing information about programs. The following example illustrates the uses and shortcomings of first order termination assertions. Let AX denote the conjunction of axioms for addition, proper subtraction and order for natural numbers using constants 0 and 1. Let n denote the numeral for n, i.e., $n=1+1+...+1$ where 1 appears n times. For any natural number n, the assertion

$$(AX \wedge x=n) \langle S_{square} \rangle (y=n^2)$$

is provable from the axioms of Theorem 1, where S_{square} is the recursive program

```
decl P ⇐ (val u, addr v): if u=0 then v:=0 else P(u-1,v); v:=v+u+u-1 fi
do
    P(x,y)
end.
```

However, the more general termination assertion stating that S_{square} computes the squaring function cannot be derived from these axioms and indeed is not valid under any set of first order hypotheses about natural numbers. That is, let p be the formula

$$(sq(0)=1) \wedge \forall x[sq(x+1)=sq(x)+x+x+1].$$

Then the termination assertion

$$(AX \wedge p) \langle S_{square} \rangle (y=sq(x))$$

is valid whenever the variables are interpreted as ranging over natural numbers and the arithmetic operations are given their usual meaning, as is easily proved by induction. But this assertion is not valid over *arbitrary* interpretations, even if AX is allowed to be any infinite set of first order formulas true about natural numbers. In particular, there is a

nonstandard interpretation that satisfies all first order formulas true of natural numbers, but in which x may be an "infinite" integer. In this interpretation, the program S_{square} will not terminate since it cannot produce 0 by subtracting 1 from x any finite number of times. Hence $(AX \wedge p) \langle S_{square} \rangle (y = sq(x))$ cannot be proved using inference rules that are sound for all interpretations. This inherent limitation of uninterpreted first order assertions is emphasized in the well-known paper of Hitchcock and Park [13].[2]

The fact that first order termination assertions are easily axiomatized depends heavily on the compactness of first order logic. Compactness ensures that whenever a first order assertion p implies that a program S halts, it is because p implies a fixed bound on the depth of the execution of procedure calls in S. As a consequence, there are many termination assertions $p\langle S \rangle q$ which are valid over specific interpretations such as the integers, but which cannot be proved from first order properties of the interpretation since S may not always terminate within a bounded number of calls (cf. [14]).

Despite their shortcomings, valid uninterpreted termination assertions express many useful properties of programs. In particular, Theorem 2 shows that termination assertions suffice to define the semantics of recursive programs in the sense of Meyer and Halpern [16]. This theorem extends Theorem 5.1 of [16] to programs with calls to global procedures and lends support to the thesis that practical programming languages may be defined axiomatically. Furthermore, the relative simplicity of our proofs when compared with proofs of analogous theorems for partial correctness assertions suggests that termination assertions are more

---

[2] It follows from this example that although induction over termination assertions is valid for the standard integers, it is not sound when the usual notion of termination is applied in nonstandard models. When a different view of termination is taken, however, induction *is* valid in nonstandard models. In this alternative setting, induction is a crucial rule of inference. For the first order language of arithmetic with termination assertions, the axiom system that is obtained from the Peano axioms with induction over arbitrary assertions in the language is adequate to prove termination of typical example programs [3]. In fact, this system can be proved complete for all formulas which are valid for a natural notion of nonstandard computation [1, 7]. In the present paper, however, we restrict our attention to the usual definition of termination and consider validity over arbitrary interpretations.

suitable than partial correctness assertions for axiomatic definitions of semantics.

2. Assertions About Global Procedure Calls

First order logic with global procedures will be seen to be a syntactic variant of standard first order logic. A *first order signature* is a set of function symbols $f_1,f_2,..$ and relation symbols $R_1,R_2,...$ with associated arities. A *signature for first order logic with global procedures*, or more simply *signature*, is a first order signature augmented with a disjoint set $P_1,P_2,...$ of *procedure variables*. Each P has an associated number of *value* parameters v_P and *address* parameters a_P. *First order assertions about global procedure calls* q are defined by the grammar

$$q ::= \textit{first-order-atomic-formula} \mid q_1 \vee q_2 \mid \neg q \mid \forall x[q_1] \mid \langle P(t,x)\rangle q$$

where P is a procedure variable, $t=t_1,...,t_{v_P}$ are first order terms, and $x=x_1,...,x_{a_P}$ are variables. The construct $\langle P(t,x)\rangle q$ is intended to express that formula q is true after calling global procedure P with value parameters t and address parameters x. Additional symbols such as $\wedge$, $\supset$, $\equiv$ and $\exists$ are considered abbreviations for the appropriate combinations of $\vee$, $\neg$ and $\forall$. To demonstrate the axiomatizability of termination assertions about programs with recursive procedures with parameters, we have chosen to consider recursive procedures with call-by-value and call-by-address parameters. To be consistent in the design of our programming language, we also consider call-by-value and call-by-address parameters in undeclared global procedures.

A few words of motivation are in order before presenting the formal semantics of assertions about procedure calls. The possible meanings of undeclared procedures should be as general as possible so as to include procedures written in any reasonable language. However, in order to prove common properties of procedures with value and address parameters, it is necessary to restrict the ways in which procedures may depend upon their parameters. It might be possible for a procedure in some language to test the length of an identifier passed as a parameter or to determine the lexicographical ordering between a pair

of actual parameter names. If a procedure can recognize the names of its parameters, then simple properties such as

$\forall x(\langle P(x)\rangle true) \supset \forall y(\langle P(y)\rangle true)$

may fail. Since capabilities such as testing the names of actual parameters go beyond the intentions of call-by-value and call-by-address, we prohibit them. For simplicity, we also insist that undeclared global procedures be *explicitly parameterized*, i.e. the meaning of an undeclared procedure call is independent of the values of any variables other than the actual parameters.

The behavior of a procedure depends on its address parameters quite differently from the way the procedure depends on its value parameters. For example, suppose that procedure P has two call-by-value parameters, and that the contents of x_1 and x_2 are equal to the contents of y_1 and y_2, respectively. Then the call $P(x_1,x_2)$ should have the same effect as $P(y_1,y_2)$. To see that call-by-address is different, consider a program which declares the following procedure

decl P ⇐ (**addr** u,v): (**if** u = v **then** v: = 0;v: = u) **do** ... **end**

with two address parameters. Suppose that x and y are variables with equal values prior to a call P(x,y). Because both parameters are passed by address, the call P(x,y) returns with x and y set to zero iff x and y share the same location; otherwise the procedure has no effect. This example shows how a procedure may detect, and hence may depend arbitrarily upon, which of its call-by-address actual parameters share locations. Therefore, we assume that the behavior of a procedure call depends only on the *values* of the actual *call-by-value* parameters and depends only on the *values* and *sharing* of addresses of the actual *call-by-address* parameters.

The possible behavior of a procedure with address parameters may be characterized using equivalence relations on finite sets of integers. For any vector of variables $\mathbf{x} = x_1,...x_k$ (not necessarily distinct), we define the *address sharing relation* E_x on {1,...,k} by

$i\ E_x\ j$ iff x_i and x_j are the same variable.

It is also useful to define the *congruence* of vectors of variables $\mathbf{x}=x_1,...,x_k$ and $\mathbf{y}=y_1,...,y_m$ by

$$\mathbf{x}\simeq\mathbf{y} \quad \text{iff} \quad k=m \text{ and } E_{\mathbf{x}}=E_{\mathbf{y}}.$$

The number of distinct variables in $\mathbf{x}$ is the number of equivalence classes of $E_{\mathbf{x}}$, i.e. the index of $E_{\mathbf{x}}$. If procedure P has k address parameters and $\mathbf{x}$ and $\mathbf{y}$ are both vectors of variables of length k, then $P(\mathbf{x})$ necessarily produces the same result as $P(\mathbf{y})$ iff $E_{\mathbf{x}}=E_{\mathbf{y}}$ and x_i has the same value as y_i for $1\leq i\leq k$. When $\mathbf{x} = x_1,...,x_{a_P}$ is a vector of distinct variables and $\mathbf{t} = t_1,...,t_{v_P}$ a vector of terms, the possible results of a call $P(\mathbf{t},\mathbf{x})$ to a nondeterministic procedure P may be characterized by an "input-output" relation. Informally, a tuple $\langle \mathbf{t},\mathbf{x},\mathbf{y}\rangle$, where $\mathbf{y}\simeq\mathbf{x}$, is in the input-output relation of P iff the call $P(\mathbf{t},\mathbf{x})$ can return with the address parameters equal to $\mathbf{y}$. Since a procedure P can distinguish between any pair of possible address sharing patterns, we use a set of input-output relations to describe the behavior of P, one for each possible address sharing relation among the address parameters.

The precise semantics of first order logic with global procedures is most easily defined by associating a first order signature with each signature that contains procedure names. Let P be a procedure name. The *associated set of input-output relations*

$$\mathfrak{R}_P = \{P_E \mid E \text{ is an equivalence relation on } \{1,...,a_P\}\}$$

is a set of first order relation symbols, one for each address sharing relation. The arity of each P_E is v_P+2a_P. If P_1 and P_2 are procedure names in signature $\mathcal{L}$, then $\mathfrak{R}_{P_1}$ and $\mathfrak{R}_{P_2}$ are assumed to be disjoint. Furthermore, each is disjoint from the set of first order relation symbols in $\mathcal{L}$. If $\mathcal{L}$ is a signature, then the *associated first order signature* $\mathcal{L}_P$ consists of all function and relation symbols of $\mathcal{L}$, together with all relation symbols in $\mathfrak{R}_P$ for each P in $\mathcal{L}$. Note that $\mathcal{L}_P$ contains no procedure names.

A *first order state* σ_1 for a first order signature is a domain $D^{\sigma 1}$ with functions $f^{\sigma 1}$ and relations $R^{\sigma 1}$ of appropriate arities on $D^{\sigma 1}$ corresponding to the function and relation symbols of the signature, and with element $x^{\sigma 1}$ for each variable symbol x. A *state with procedure environment* (or more simply *state*) σ for a signature $\mathcal{L}$ with procedure names is a

first order state for the associated first order signature $\mathcal{L}_P$, with the added restriction that $P_E(t,x,y)$ is false unless the values of x and y are consistent with E. More precisely, for all b $\in (D^\sigma)^{v_P}$, and c, d $\in (D^\sigma)^{a_P}$,

if $\langle b,c,d\rangle \in P_E{}^\sigma$ then, for $1 \le i,j \le a_P$, i E j implies $d_i = d_j$ and $c_i = c_j$.

We use $\sigma\{d/x\}$ to denote the state σ' that is identical to σ except possibly at x and such that $x^{\sigma'} = d$. Satisfaction of a first order assertion q about global procedure calls by a state σ is defined inductively exactly as for first order formulas, with one extra case for procedure calls. Namely,

$\sigma \models \langle P(t,x)\rangle q$ iff $\exists d \in (D^\sigma)^{a_P}$ such that $(t^\sigma, x^\sigma, d) \in P_{E_x}{}^\sigma$ and $\sigma\{d/x\} \models q$.

We interpret $\langle P(t,x)\rangle q$ to mean that q is true after calling P with t and x. The definition above forces the results of P(t,x) to depend only on the values of explicit parameters t,x and address sharing pattern E_x. Furthermore, a call P(t,x) may only alter the values of address parameters x.

As usual, we write $\Gamma \models p$ for a set of formulas Γ to mean that if $\sigma \models q$ for every $q \in \Gamma$, then $\sigma \models p$.

As for ordinary predicate calculus, the axiomatization of assertions about global procedure calls includes a universal instantiation axiom which involves substitution of terms for variables. The substitution of terms in first order assertions about global procedure calls raises a few extra complications. Since the construct $\langle P(t,x)\rangle q$ is well-formed only if x is a vector of variables, it is impossible to substitute terms for address parameters directly. In addition, substituting some address parameter x_i for another address parameter x_j may change the address sharing relation. As a consequence, *replacement* of one address parameter by another has a different semantic effect from first order *substitution*. For example,

$\forall x,y R(x,y) \supset \forall x R(x,x)$

is a valid first order sentence. But since a procedure P may detect sharing,

$\forall x,y\langle P(x,y)\rangle true \supset \forall x\langle P(x,x)\rangle true$

may not be true if both parameters are call-by-address. We circumvent this problem by defining a substitution on assertions with global procedure calls which differs from strict syntactic replacement but which has the same semantics as regular first order substitution.

We use *Free*(q) to denote the set of variables which occur free in an assertion q about global procedure calls and q[t/z] to denote the result of substituting the term t for free occurrences of the variable z in q. Formally, *Free*(q) and q[t/z] are defined by induction on the structure of assertions. These definitions are standard for all cases other than $\langle P(s,x)\rangle q$. We define $Free(\langle P(s,x)\rangle q) ::= Free(q) \cup Free(s) \cup \{x\}$. The definition of substitution for $\langle P(s,x)\rangle q$ is straightforward if t is a simple variable not among $x=x_1,...,x_k$ or if z is not among $x_1,...,x_k$, namely,

(S1) $(\langle P(s,x)\rangle q)[t/z] ::= \langle P(s[t/z],x[t/z])\rangle(q[t/z])$.

Otherwise, we define

(S2) $(\langle P(s,x)\rangle q)[t/z] ::= \forall w(t=w \supset (\langle P(s,x)\rangle q)[w/z])$

where w is a fresh variable which does not occur in t or $\langle P(s,x)\rangle q$. By choice of w, the recursive substitution in (S2) may be done according to rule (S1). Note that in general q[t/z] may have more connectives and quantifiers than q. However, if v is a variable which does not appear in q, then the assertion q[v/z] is the same length as the assertion q. This is critical to proofs by induction on the length of assertions. Furthermore, if v does not occur in q, then q[v/z][z/v] = q. A straightforward consequence of the definition of substitution is

> **Lemma 1:** (Substitution) Let q be a first order assertion about global procedures, t a term and z a variable. Then for any state σ, $\sigma \models q[t/z]$ iff $\sigma\{t^{\sigma}/x\} \models q$.

This lemma is critical in establishing the soundness of the instantiation and substitution axioms presented in Lemma 2 as well as the assignment axiom in Theorem 1.

The axioms for first order logic of Enderton [9], for example, may be augmented to a complete system for first order logic with global procedures.

Lemma 2: All generalizations of the following axioms, together with the inference rule *modus ponens*, form a deductively complete proof system for first order logic with global procedures. That is, $\Gamma \models p$ iff $\Gamma \vdash p$ for any set $\{p\} \cup \Gamma$ of first order assertions about global procedure calls.

P1. Propositional tautologies,

P2. $\forall x q \supset q[t/x]$,

P3. $\forall x(p \supset q) \supset (\forall x p \supset \forall x q)$,

P4. $q \supset \forall x q$ for x not free in q,

P5. $x = x$,

P6. $x = y \supset (r \supset s)$ where r is a first order *atomic* formula and s is the formula r with zero or more occurrences of some variable x replaced by another variable y,

P7. $(s = t \wedge x = y \wedge u = v \wedge \langle P(s,x)\rangle x = u) \supset \langle P(t,y)\rangle y = v$ where $x \simeq y$,

P8. $\langle P(t,x)\rangle q \equiv \exists y(\langle P(t,x)\rangle x = y \wedge q[y/x])$
where y is a vector of variables which are not free in t,x or q and $y \simeq x$.

The proof of Lemma 2, given in the Appendix, follows the usual Henkin-style construction of a state satisfying a given set of formulas.

3. Recursive Programs

Recursive programs have abstract syntax

$$S ::= \quad x := t \mid p? \mid P(t,v) \mid S_1;S_2 \mid S_1 \cup S_2 \mid \textbf{decl } D \textbf{ do } S \textbf{ end}$$

where declaration D is given by

$$D ::= \quad x \textbf{ init } t \mid P \Leftarrow B$$

and procedure abstract B has form

$$B ::= \quad \langle(\textbf{val } x, \textbf{addr } y) : S\rangle.$$

The statements are assignment, test, procedure call, concatenation, union and declaration. Union denotes nondeterministic choice of S_1 or S_2, i.e., execute S_1 or S_2. The test p? allows execution to continue iff p is true. In practice, one would require that p be effectively decidable, e. g. quantifier free, but this restriction is unnecessary for the results presented here. However, we do require that p be free of calls to procedures.[3] Informally, the declaration **decl** D **do** S **end** declares a local variable or recursive procedure with scope S. The variable declaration x **init** t defines a new local variable x with initial value t. The procedure declaration $P \Leftarrow \langle$(**val** x, **addr** y) : S$\rangle$ declares a possibly recursive procedure P with formal value parameters $\mathbf{x}=x_1,\ldots,x_{v_P}$ formal address parameters $\mathbf{y}=y_1,\ldots,y_{a_P}$ and body S. A procedure declaration P⇐B is considered syntactically well-formed only if B has exactly v_P value parameters and a_P address parameters. In order to use global procedure variables to reason formally about declared procedures, we also require that declared procedures, like global procedures, be explicitly parameterized. This requires that all variables which occur free in the body S of a procedure abstract B also appear in the parameter list (**val** x, **addr** y) of B.

Many statements common to Algol-like languages may be considered syntactic abbreviations for recursive programs, as is well known (cf. [8], [18]). For example, the statement **if..then..else..fi** may be written

$$\textbf{if } p \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \equiv (p?;S_1) \cup (\neg p?;S_2)$$

Thus the axioms of Theorem 1 may be considered complete for recursive programs with **if..then..else..fi** in addition to the statements listed in the grammar above. The **while** statement may be expressed using recursion by

---

[3] Without this requirement, it is possible to write programs without sensible semantics. Consider the following program which declares a parameterless procedure P.

decl P ⇐ ⟨(⟨P⟩*true*)?; P ∪ (⟨P⟩*false*)?⟩ **do** P **end**

The effect of P appears to be "keep calling P as long as it will halt, but halt without side effects otherwise." This procedure halts iff it does not halt!

while p **do** S **od** ≡

decl P ⇐ ⟨(**addr** x): p?;S;P(x) ∪ ¬p?⟩ **do** P(x) **end**

where x includes all free variables in S. Other iterative constructs such as **repeat** S **until** p may also be considered abbreviations for similar recursive programs. In addition, declarations may be nested as deeply as desired so that any number of local variables and procedures may be defined. A statement declaring variables $x_1,...,x_n$ with initial values $t_1,...,t_n$ may be considered an abbreviation for a sequence of nested declarations, i.e.

decl $x_1,...,x_n$ **init** $t_1,...,t_n$ **do** S **end** ≡

```
decl z1 init t1 do
   decl z2 init t2 do
      ...
      decl zn init tn do
            S[z/x]
      end
      ...
   end
end
```

where $\mathbf{z}=z_1,...,z_n$ are fresh variables not occurring in $t_1,...,t_n$ and S[z/x] is the result of the simultaneous replacement of z for x in S (cf. [2]).[4] In addition, mutually recursive procedures may also be declared as nested procedures. For example, the program with mutually recursive procedures

decl $P_1 \Leftarrow B_1$, $P_2 \Leftarrow B_2$ **do** S **end**

may be written as

---

[4] This definition is consistent with initialization in many Algol-like languages in that it uses the values of terms $t_1,...,t_n$ at block entry time for the variables $x_1,...,x_n$ rather than the least fixed-points of the set of possibly mutually recursive equations $\{x_i=t_i \mid 1\leq i\leq n\}$. Other definitions of simultaneous variable declaration and initialization may also be written as recursive programs.

```
decl P1 ⇐ (decl P2 ⇐ B2 do B1) do
    decl P2 ⇐ B2 do
        S
    end
end.
```

As a result, Theorems 1 and 2 apply to recursive programs with mutually recursive procedures.

The state semantics we define below are equivalent to the more usual environment and store semantics presented, e.g., in deBakker [8] and Apt [2]. However, the proof of Theorem 1 and, especially, the statement and proof of Theorem 2 are simplified by combining the notions of environments and stores into states.[5] The meaning m(S) of a nondeterministic program S is formally a mapping from "initial" states to sets of "final" states as in Harel [11]. We define the meaning of programs inductively. Assigning x to t in state σ produces a state whose value of x is the value of t in σ, i.e.

(i) $m(x := t)\sigma ::= \{ \sigma\{t^{\sigma}/x\} \}$.

Test p? aborts unless p is true, so that

(ii) $m(p?)\sigma ::= \{\sigma\}$ if $\sigma \models p$ and $\varnothing$ otherwise,

and choosing S_1 or S_2 in state σ gives the union of the meanings of S_1 and S_2 in σ, i.e.

(iii) $m(S_1 \cup S_2)\sigma ::= m(S_1)\sigma \cup m(S_2)\sigma$.

Sequencing (;) behaves like relational composition:

(iv) $m(S_1;S_2)\sigma ::= \cup_{\sigma_1 \in m(S_1)\sigma} m(S_2)\sigma_1$.

The meaning of a call to procedure P depends on the input-output relation P_E, where E is the address sharing relation of the actual address parameters. Formally, the meaning of a procedure call is

(v) $m(P(t,x))\sigma ::= \{\sigma\{a/x\} \mid (t^{\sigma}, x^{\sigma}, a) \in P_{E_x}{}^{\sigma}\}$

---

[5] More precisely, the meaning m(S) we assign to program S is the restriction of the environment-store meaning to the case when there is no sharing, viz., the environment is an injection from variables to locations.

The meaning of a statement with variable declaration is defined to be

(vi) $m(\textbf{decl } x \textbf{ init } t \textbf{ do } S \textbf{ end})\sigma ::= \{\sigma'\{x^{\sigma}/x\} \mid \sigma' \in m(S)(\sigma\{t^{\sigma}/x\})\}$.

Intuitively, the effect of **decl** x **init** t **do** S **end** is to set x to t, do S, then reset x to its original value.

Our approach to the definition of the meaning of a procedure declaration will be to use a formal version of an ALGOL 60 "copy rule". In effect, the rule replaces calls to a declared procedure by copies of the procedure body. The meaning of a program **decl** P⇐B **do** S_1 **end**, where B = ⟨(**val** x, **addr** y) : S_0⟩, will be approximated by programs without procedure declarations. We first recursively find approximations $S_0^{[i]}$ and $S_1^{[i]}$ to S_0 and S_1. Both $S_0^{[i]}$ and $S_1^{[i]}$ may contain calls to P but neither will contain procedure declarations. We replace all calls to P in $S_1^{[i]}$ by $S_0^{[i]}$. This produces a program in which each call to the recursive procedure P is approximated using "in-line code" for a single procedure call. To approximate recursive calls to depth 2, we again replace all calls to P by $S_0^{[i]}$. This process is iterated i times to approximate recursive calls to P up to depth i. Finally, all remaining calls to P are replaced by the divergent program *false*?. In each replacement of a procedure call, a simple parameter substitution operation is performed. The rules for parameter substitution will be in keeping with the convention for Algol-like languages, so that recursive programs will be defined to have statically scoped variables. We remark that dynamic scoping, as well as alternative parameter passing mechanisms such as call-by-value/result and call-by-name, may also be treated using variations of the definition of syntactic application and call replacement given below.

We require the routine definitions of free variables and substitution of variables in programs (cf. [2] and [8]). An occurrence of a simple variable x or procedure variable P in a program S may be free or bound, according to whether or not it is within the scope of a declaration x **init** t or P⇐B. The definition of program S with variable y substituted for free occurrences of x, written S[y/x], is routine (see deBakker [8]), as is *simultaneous substitution* of a vector of variables **y** for **x**, written S[**y**/**x**]. We also use S[Q/P] to denote the program S

with procedure variable Q substituted for free occurrences of P, where $v_P = v_Q$ and $a_P = a_Q$.

To define declaration-free approximations, we first discuss declarations **decl** P⇐B **do** S **end** that do not contain nested declarations. Let S_0 and S be statements without procedure declarations and let B denote a procedure abstract ⟨(**val** x, **addr** y): S_0⟩. We define the *replacement of calls to global procedure* P *in* S *using procedure abstract* B, written S[B/P], by induction on the structure of S as follows:

$(x := t)[B/P] \ ::= \ (x := t)$,

$p?[B/P] \ ::= \ p?$,

$P(t,v)[B/P] \ ::= \$ **decl** z **init** t **do** $S_0[z/x][v/y]$ **end**
where $z_1,...,z_{v_P}$ are distinct variables which do not appear in t, v or B.[6]

$P_1(t,v)[B/P] \ ::= \ P_1(t,v)$
when P_1 is a procedure name different from P,

$(S_1;S_2)[B/P] \ ::= \ S_1[B/P] ; S_2[B/P]$,

$(S_1 \cup S_2)\,[B/P] \ ::= \ S_1[B/P] \cup S_2[B/P]$,

decl x **init** t **do** S **end**$[B/P] \ ::= \$ **decl** z **init** t **do** $S[z/x][B/P]$ **end**
where z does not occur in S or B.

We remark that renaming of bound variables in the final clause gives our programs static scoping.

The i^{th} *iterated replacement of calls to* P *in* S *using procedure abstract* B, is defined inductively by

---

[6] This is the definition of the *syntactic application* of B to parameters t and v given in [2] and [8]. The declaration of a vector z of new variables with initial value t reflects the standard behavior of call-by-value parameters. Value/result, for example, may be obtained by resetting the actual parameters (which must then be variables) to z before block exit. Other mechanisms may also be handled by altering this definition (see Apt [2]).

$S[B/P]^0 ::= S,$
$S[B/P]^{i+1} ::= (S[B/P]^i)[B/P].$

Finally, we define the *i-th approximation*, $S^{[i]}$, of an arbitrary program S by induction on the structure of S. In general, S may contain procedure declarations but $S^{[i]}$ will *always* be free of procedure declarations. For S atomic, i.e., an assignment, test, or global procedure call P(t,v), we let $S^{[i]} ::= S$. The remaining cases are

$(S_1; S_2)^{[i]} ::= S_1^{[i]}; S_2^{[i]}$

$(S_1 \cup S_2)^{[i]} ::= S_1^{[i]} \cup S_2^{[i]}$

$(\textbf{decl } x \textbf{ init } t \textbf{ do } S \textbf{ end})^{[i]} ::= (\textbf{decl } x \textbf{ init } t \textbf{ do } S^{[i]} \textbf{ end})$

$(\textbf{decl } P \Leftarrow \langle(\textbf{val } x, \textbf{addr } y): S_0\rangle \textbf{ do } S \textbf{ end})^{[i]} ::=$
$(S^{[i]} [\langle(\textbf{val } x, \textbf{addr } y): S_0^{[i]}\rangle / P]^i) [\mathit{false}?/P].$

The meaning of a declared program is defined to be the union of the meanings of its approximations:

(vii) $m(\textbf{decl } P \Leftarrow B \textbf{ do } S \textbf{ end})\sigma ::= \cup_i m(\textbf{decl } P \Leftarrow B \textbf{ do } S \textbf{ end}^{[i]})\sigma.$

This concludes the inductive definition of the meanings of recursive programs.

It follows from our definition that every program is equivalent to a nondecreasing union of programs without procedure declarations.

Lemma 3: Let S be any recursive program. Then $m(S) = \cup_{i \geq 0} m(S^{[i]})$ and, whenever $i \leq j$, $m(S^{[i]}) \subseteq m(S^{[j]})$.

In addition, the meaning of a program does not depend on the names of its bound variables.

Lemma 4:
(a) $m(\textbf{decl } x \textbf{ init } t \textbf{ do } S \textbf{ end}) = m(\textbf{decl } y \textbf{ init } t \textbf{ do } S[y/x] \textbf{ end})$,
where y does not occur in S or t.
(b) $m(\textbf{decl } P \Leftarrow B \textbf{ do } S \textbf{ end}) = m(\textbf{decl } Q \Leftarrow B[Q/P] \textbf{ do } S[Q/P] \textbf{ end})$,
where $v_Q = v_P$, $a_Q = a_P$, and Q does not occur in B or S.

We note also that the semantic clauses above define input-output relations which are identical to those derived from the purely operational semantics of Gallier [10].[7]

4. Completeness

The termination assertion $p\langle S\rangle q$ means that if some state σ satisfies p, then some computation of S from σ halts in a state that satisfies q. More precisely, a state σ *satisfies* $p\langle S\rangle q$, written $\sigma \models p\langle S\rangle q$, if $\sigma \models p$ implies $\exists \sigma' \in m(S)\sigma$ such that $\sigma' \models q$. The termination assertion $p\langle S\rangle q$ is *valid*, written $\models p\langle S\rangle q$, if every state σ satisfies $p\langle S\rangle q$. The assertion $\langle S\rangle q$ abbreviates $true\langle S\rangle q$.[8]

> **Theorem 1:** The following axioms are sound and complete for proving termination assertions $p\langle S\rangle q$ where p, q are first order assertions about global procedure calls and S is a recursive program.

Axioms

A1. $q[t/x]\langle x := t\rangle q$,

A2. $(r \wedge q)\langle r? \rangle q$,

A3. $(\langle P(t,v)\rangle q)\langle P(t,v)\rangle q$,

Rules of Inference

A4. $p\langle S_1\rangle r, r\langle S_2\rangle q \vdash p\langle S_1;S_2\rangle q$,

---

[7] More specifically, Gallier defines the operational semantics of recursive flowcharts using execution trees. The input-output semantics defined by considering the set of "output" states at the leaves of an execution tree to be a function of the "input" state at the root is the same as the semantics we define above. As a consequence, our semantics are equivalent to the standard operational semantics.

[8] It is a straightforward consequence of the definitions that a state σ satisfies the termination assertion $\langle P(t,v)\rangle q$ iff σ satisfies the first order assertion about global procedure calls $\langle P(t,v)\rangle q$. As a result, there is no harm in failing to specify whether statements of this form are assertions about procedure calls or termination assertions.

A5. $p\langle S_1\rangle q,\ r\langle S_2\rangle q \vdash (p \vee r)\langle S_1 \cup S_2\rangle q,$

A6. $(p \wedge y=t)\langle S[y/x]\rangle q \vdash p\langle$**decl** x **init** t **do** S **end**$\rangle q$
where y does not occur in p,t,S or q,

A7. $p\langle$**decl** $P \Leftarrow B$ **do** $P(t,v)[B/P]$ **end**$\rangle q \vdash p\langle$**decl** $P \Leftarrow B$ **do** $P(t,v)$ **end**$\rangle q$

A8. $([\wedge_{i<n} \forall w(r_i \supset \langle P(u^{(i)},v^{(i)})\rangle t_i)] \wedge p)\langle S\rangle q$

$r_i \langle$**decl** $P \Leftarrow B$ **do** $P(u^{(i)},v^{(i)})$ **end**$\rangle t_i \qquad (i<n)$

$\vdash \quad p\langle$**decl** $P \Leftarrow B$ **do** S **end**$\rangle q$

where w, $u^{(i)}$ and $v^{(i)}$ are vectors of variables;
in each subformula $\forall w(r_i \supset \langle P(u^{(i)},v^{(i)})\rangle t_i)$,
w includes $u^{(i)}$, $v^{(i)}$ and all free variables in r_i and t_i;
and P does not appear in p or q.

A9. $p\langle$**decl** $Q \Leftarrow B[Q/P]$ **do** $S[Q/P]$ **end**$\rangle q \vdash p\langle$**decl** $P \Leftarrow B$ **do** S **end**$\rangle q$,
where $v_Q = v_P$, $a_Q = a_P$ and Q does not appear in B or S.

A10. $p\langle S\rangle q \vdash r\langle S\rangle q$ whenever $\vdash r \supset p$ by the rules of Lemma 2.

Most of the inference rules above are straightforward and are similar to many found in the literature [2, 8, 16]. The most complicated rule is A8. Intuitively, the rule states that if $p\langle S\rangle q$ holds under some finite set of hypotheses about calls to procedure P, and if each hypothesis can be proved for the declaration $P \Leftarrow B$, then conclude $p\langle$**decl** $P \Leftarrow B$ **do** S **end**$\rangle q$.[9] An important special case of A8 is when the conjunction of first-order assertions about calls to P is empty, i.e. $n=0$. Namely, if we can prove $p\langle S\rangle q$ and P does not appear in p, then A8 yields $p\langle$**decl** $P \Leftarrow B$ **do** S **end**q. This instance of A8 provides the base of an induction (using A7) showing the provability of assertions about declarations. The proof of

---

[9] A consequence of the completeness proof will be an upper bound on the number of hypotheses, i.e. conjuncts $r_i \supset \langle P(u,v)\rangle t_i$, needed to prove all valid assertions about programs calling a procedure P. The bound is an exponential function of the number of address parameters, a_P.

soundness of each rule is left to the reader.

Note that Theorem 1 is not a relative completeness theorem of the sort typical for partial correctness assertions (cf. [6], [5], [11]). If a termination assertion is valid, then it is provable from the axioms P1-8 of Lemma 2 and A1-10 above without appeal to further axioms. In particular, the valid termination assertions are recursively enumerable.[10] Harel, Meyer, and Pratt [12] previously observed that the valid termination assertions for a very general class of program schemes without global procedure calls are recursively enumerable. A complete axiomatization of termination assertions about **while**-programs without procedure calls was presented in Theorem 6.4 of [16]. The proofs in [12], [16] rested heavily on compactness properties of first order predicate calculus, but Lemma 2 provides corresponding properties for the logic of global procedure calls leading to the generalization we provide in Theorem 1.

The proof of Theorem 1 uses the fact that termination assertions about programs without procedure declarations can be "translated" into first-order assertions about global procedures.

> **Lemma 5:** (a) If S is a recursive program without procedure declarations and q is a first-order assertion about global procedure calls, then there is a first-order assertion r about global procedure calls such that $r \equiv \langle S \rangle q$.
> (b) For every recursive program S and first-order assertion about global procedure calls q, there is a set $\{q_i\}$ of first order assertions about procedure calls such that $\langle S \rangle q \equiv \vee_i q_i$.

---

[10] In contrast, the valid partial correctness assertions are not recursively enumerable. This follows from the observation that the set of nowhere terminating **while**-program schemes is not recursively enumerable [15], because the partial correctness assertion *true*{S}*false* expresses the fact that S fails to terminate. So the set of valid partial correctness assertions of this trivial form about only deterministic **while**-programs without procedure calls is not even recursively enumerable. The validity problem for all first order partial correctness assertions is shown to be Π_2^0 complete in [12]. This continues to be true whether the programs in the assertions are restricted to a single **while**-program without procedure calls or are allowed to vary over all recursive programs with global procedure calls.

Part (a) is easily proved by induction on programs [18]. Part (b) follows from (a) and the fact that any program with procedure declarations is equivalent to the union of its finite approximations without procedure declarations (Lemma 3). We omit the details. It is convenient to write $\vee_i\langle S^{[i]}\rangle q$ for the disjunction $\vee_i q_i$ with $q_i \equiv \langle S^{[i]}\rangle q$.

A second important property for the the proof of Theorem 1 is a compactness theorem for first order logic with global procedure calls. More specifically, if p implies some infinite disjunction $\vee_i q_i$ of first-order assertions about global procedures, then there is some integer j such that $p \supset \vee_{i\leq j} q_i$ is valid. This follows directly from Lemma 2 and the proof is omitted. As a consequence of compactness, whenever a termination assertion about a recursive program S is valid, the same assertion is also valid for some approximation to S.

Lemma 6: If $p\langle S\rangle q$ is valid, then for some j, $p\langle S^{[j]}\rangle q$ is also valid.

Proof: If $\models p\langle S\rangle q$, then by Lemma 5, $\models p \supset \vee_i \langle S^{[i]}\rangle q$. By the compactness of first-order logic with global procedure calls, there is some integer j such that $\models p \supset \vee_{i\leq j}\langle S^{[i]}\rangle q$. But by Lemma 3, $m(S^{[i]}) \subseteq m(S^{[j]})$ for $i\leq j$. Therefore $\models p\langle S^{[j]}\rangle q$. ∎

This lemma is critical to the completeness of axioms A1-10 for recursive programs.

Proof of Theorem 1: Suppose $\models p\langle S\rangle q$. We show that $p\langle S\rangle q$ is provable from A1-11 by induction on the structure of S.

(a) If $p\langle x := t\rangle q$ is valid, then $\sigma \models p$ implies $\sigma\{t^\sigma/x\} \models q$ by definition of $m(x := t)$. By the Substitution Lemma 1, $\models p \supset q[t/x]$, and therefore $\vdash p \supset q[t/x]$ by the rules of Lemma 2. By A1, we have $\vdash q[x/t]\langle x := t\rangle q$ and so $\vdash p\langle x := t\rangle q$ by A10.

(b) If $\models p\langle r?\rangle q$, then $\models p \supset (r \wedge q)$ and hence $\vdash p \supset (r \wedge q)$ by the rules of Lemma 2. Since $\vdash (r \wedge q)\langle r?\rangle q$ by A2, we have $\vdash p\langle r?\rangle q$ by A10.

(c) Assume $\models p\langle P(t,x)\rangle q$. Then the first order assertion about global procedure calls $p \supset \langle P(t,x)\rangle q$ is valid and hence provable by the rules of Lemma 2. Therefore $\vdash p\langle P(t,x)\rangle q$ by A3 and A10.

(d) Suppose $\models p\langle S_1;S_2\rangle q$. Then by Lemma 6, there is some j such that $p\langle S_1^{[j]};S_2^{[j]}\rangle q$ is valid. By Lemma 5, there exist first-order assertions r_1 and r_2

about global procedures such that

$$r_2 \equiv \langle S_2^{[j]} \rangle q \quad \text{and} \quad r_1 \equiv \langle S_1^{[j]} \rangle r_2 \equiv \langle S_1^{[j]}; S_2^{[j]} \rangle q.$$

By the inductive hypothesis, both termination assertions

$$r_2 \langle S_2 \rangle q \quad \text{and} \quad r_1 \langle S_1 \rangle r_2$$

are provable. Therefore, $r_1 \langle S_1; S_2 \rangle q$ may be proved using A4. Since $p \supset r_1$ is valid, and hence provable by the rules of Lemma 2, the termination assertion $p \langle S_1; S_2 \rangle q$ is provable by A10.

(e) Assume $\models p \langle S_1 \cup S_2 \rangle q$. By Lemma 6, we have $\models p \langle S_1^{[j]} \cup S_2^{[j]} \rangle q$ for some positive integer j. By Lemma 5, there are first-order assertions r_1 and r_2 about global procedure calls with

$$r_1 \equiv \langle S_1^{[j]} \rangle q \quad \text{and} \quad r_2 \equiv \langle S_2^{[j]} \rangle q.$$

By the inductive hypothesis, both

$$r_1 \langle S_1 \rangle q \quad \text{and} \quad r_2 \langle S_2 \rangle q$$

are provable. Therefore $(r_1 \vee r_2) \langle S_1 \cup S_2 \rangle q$ may be proved by A5. Since $\models p \supset (r_1 \vee r_2)$, it follows that $\vdash p \langle S_1 \cup S_2 \rangle q$ by A10.

(f) Suppose $\models p \langle$**decl** x **init** t **do** S **end**$\rangle q$ and let y be any variable that does not occur in p, t, S or q. We show that $(p \wedge y = t) \langle S[y/x] \rangle q$ is valid. By Lemma 4, if $p \langle$**decl** x **init** t **do** S **end**$\rangle q$ is valid, then $p \langle$**decl** y **init** t **do** S[y/x] **end**$\rangle q$ must be valid as well. Now suppose $\sigma \models (p \wedge y = t)$. Then there is some $\sigma' \in m(S[y/x])(\sigma\{t^\sigma/y\})$ such that $\sigma'\{y^\sigma/y\} \models q$. But because $\sigma \models (y = t)$, we have $\sigma\{t^\sigma/y\} = \sigma$ and $\sigma' \in m(S[y/x])\sigma$. Furthermore, since y does not occur in q, $\sigma' \models q$. Thus $(p \wedge y = t) \langle S[y/x] \rangle q$ is valid. By the induction hypothesis, this termination assertion is provable. Therefore $\vdash p \langle$**decl** x **init** t **do** S **end**$\rangle q$ by A6.

(g) Suppose S is of the form **decl** $P \Leftarrow B$ **do** S_1 **end** and suppose $\models p \langle S \rangle q$. By rule A9, we may assume that the procedure variable P does not appear in the assertions p and q. By Lemma 6, there is some positive integer j such that $p \langle S^{[j]} \rangle q$ is valid.

We fix j and define a set of first-order assertions about global procedure calls $\{r_{k,E}\}$ where k is an integer and E is an address sharing relation. The presentation is simplified by adopting two abbreviations. First, for each address sharing relation E on $\{1,...,a_P\}$, let v_E denote a vector of variables chosen from $v = v_1,...,v_{a_P}$ with address sharing relation E. Second, if B is $\langle$(**val** x, **addr** y):$S_0\rangle$,

then we define

$$B^k(u,v_E) \;::=\; (\textbf{decl } P \Leftarrow \langle(\textbf{val } x, \textbf{addr } y{:}S_0^{[j]}\rangle \textbf{ do } P(u,v_E) \textbf{ end})^{[k]}.$$

Intuitively, $B^k(u,v_E)$ is a program which behaves like an execution of $P(u,v_E)$ within the scope of the declaration $P \Leftarrow B$ when recursive calls to P are limited to depth k and recursive calls to other procedures are limited to depth j. In particular, note that

$$B^j(u,v_E) \;=\; (\textbf{decl } P \Leftarrow B \textbf{ do } P(u,v_E) \textbf{ end})^{[j]}.$$

This is a straightforward consequence of the definition of program approximation.

For each k, let $r_{k,E}$ denote a first-order assertion with

$$r_{k,E} \;\equiv\; (\langle B^k(u,v_E)\rangle\, v_E = w_E).$$

Since $B^k(u,v_E)$ does not contain any calls to P, we may assume that P does not appear in $r_{k,E}$. Let C_k denote the conjunction over all address sharing relations E for P of the universal closures of the of first-order assertions

$$r_{j,E} \supset \langle P(u,v_E)\rangle\, v_E = w_E.$$

By construction, $\sigma \models C_k$ forces

$$m(B^k(u,v_E))\sigma \;\subseteq\; m(P(u,v_E))\sigma$$

for every address sharing relation E. In particular, if $\sigma \models C_j$ for j as above, then

$$m(\textbf{decl } P \Leftarrow B \textbf{ do } P(u,v_E) \textbf{ end}^{[j]})\sigma \;\subseteq\; m(P(u,v_E))\sigma$$

for each E. Thus, whenever $\sigma \models C_j$, we have $m(S^{[j]})\sigma \subseteq m(S_1)\sigma$. Since $p\langle S^{[j]}\rangle q$ is valid, it follows that $(p \wedge C_j)\langle S_1\rangle q$ is valid. Therefore, by the inductive assumption, $(p \wedge C_j)\langle S_1\rangle q$ is provable from A1-10. In order to use inference rule A8 to prove $p\langle S\rangle q$, it now suffices to show that the termination assertions

$$r_{k,E}\, \langle \textbf{decl } P \Leftarrow B \textbf{ do } P(u,v_E) \textbf{ end}\rangle\, v_E = w_E$$

are provable from A1-A10 for all k and all E. We show this by induction on k.

To begin, note that $r_{0,E} \equiv \mathit{false}$ since $B^0(u,v_E) = \mathit{false}$?. Hence the termination assertion

$$r_{0,E}\, \langle P(u,v_E)[B/P]\rangle\, v_E = w_E$$

is vacuously valid. By the main inductive hypothesis, it is also provable. Since P does not occur in $r_{0,E}$, we can prove $r_{0,E}\, \langle \textbf{decl } P \Leftarrow B \textbf{ do } P(u,v_E)[B/P] \textbf{ end}\rangle\, v_E = w_E$

by A8 . Thus, by A7,

$$\vdash r_{0,E} \langle \textbf{decl } P \Leftarrow B \textbf{ do } P(u\ v_E) \textbf{ end} \rangle\ v_E = w_E.$$

This concludes the initial step of the induction.

For the inductive step, assume that all

$$(*) \quad r_{k,E} \langle \textbf{decl } P \Leftarrow B \textbf{ do } P(u,v_E) \textbf{ end} \rangle\ v_E = w_E$$

are provable for all E. Let E_0 be any address sharing relation for P. We wish to conclude (*) for E_0 and $k+1$, i.e. we must show that

$$r_{k+1,E_0} \langle \textbf{decl } P \Leftarrow B \textbf{ do } P(u,v_{E_0}) \textbf{ end} \rangle\ v_{E_0} = w_{E_0}$$

is provable. By the construction of C_k, we know that

$$(C_k \wedge r_{k+1,E_0}) \langle P(u,v_{E_0})[B/P] \rangle\ v_{E_0} = w_{E_0}$$

is valid. This is because $\sigma \models C_k$ ensures $m(P(u,v_E))\sigma \supseteq m(B^k(u,v_E))\sigma$ for all E and $\sigma \models r_{k+1,E}$ implies that some state σ' satisfying $v_{E_0} = w_{E_0}$ may be reached by running $B^{k+1}(u,v_E)$. Therefore σ' may be reached by running the program $B(u,v_{E_0})$ with global calls to P. By the main induction hypothesis, this termination assertion is provable. From this and the subinduction hypotheses (*) for all E, we may use A8 to derive

$$\vdash r_{k+1,E_0} \langle \textbf{decl } P \Leftarrow B \textbf{ do } P(u,v_{E_0})[B/P] \textbf{ end} \rangle\ v_{E_0} = w_{E_0}.$$

Thus, by A7,

$$\vdash r_{k+1,E_0} \langle \textbf{decl } P \Leftarrow B \textbf{ do } P(u,v_{E_0}) \textbf{ end} \rangle\ v_{E_0} = w_{E_0}.$$

This concludes the subinduction.

We now have

$$\vdash (C_j \wedge p) \langle S_1 \rangle q$$

and, for each conjunct $r_{j,E} \supset \langle P(u,v_E) \rangle\ v_E = w_E$ of C_j,

$$\vdash r_{j,E} \langle \textbf{decl } P \Leftarrow B \textbf{ do } P(u,v_E) \textbf{ end} \rangle\ v_E = w_E.$$

Thus, by A8, $\vdash p\langle S \rangle q$. This concludes case (g) of the main induction and shows that A1-10 are complete for all recursive programs. ∎

5. Axiomatic Semantics

Despite the limitations noted in the introduction, many useful properties of programs may be proved using uninterpreted termination assertions. In particular, termination assertions determine the semantics of programs in the sense discussed in Meyer and Halpern [16], i.e. the termination assertions valid for a program distinguish it from all inequivalent programs. To be more precise, we define the termination theory of a program S, written $\mathcal{T}(S)$, to be the set of all pairs (p,q) of first order assertions about procedure calls such that $p\langle S\rangle q$ is valid. Two programs have the same termination theory precisely when they are equivalent, i.e.

Theorem 2: (Semantical Determination) For any programs S and T, $\mathcal{T}(S) = \mathcal{T}(T)$ iff $m(S) = m(T)$.

Theorem 2 generalizes Theorem 5.1 of [16] to programs with calls to global procedures and the proof is a straightforward reformulation of that in [16]. The theorem holds, in fact, for any programs S and T which are equivalent to arbitrary unions of schemes, provided that for each scheme S_i in the union and every first order assertion about procedure calls q, the termination assertion $\langle S_i\rangle q$ is equivalent to some first order assertion about global procedure calls. In particular, Theorem 2 holds for any pair S and T of arbitrary, not necessarily even recursively enumerable, infinite flowcharts (see [16]).

Proof of Theorem 2: We may assume without loss of generality that $m(S)-m(T)\neq\emptyset$. Let $\sigma'\in m(S)\sigma - m(T)\sigma$ and let $x=x_1,\ldots,x_n$ include all free variables of S∪T. Since S is equivalent to a union of programs without procedure declarations, $\cup_i S_i$, there is some such program S_k with $\sigma'\in m(S_k)\sigma$. Let $x'=x_1',\ldots,x_n'$ be a vector of distinct variables not free in S∪T. By Lemma 5, there is a first order formula p equivalent to $\langle S_k\rangle x=x'$. Since $m(S_k)\subseteq m(S)$, we have $\models p\langle S\rangle x=x'$.

It remains to be shown that $p\langle T\rangle x=x'$ is not valid. First note that for any program T, the behavior of T depends only on its free variables. Formally, if $x'=x_1,\ldots,x_n$ are variables that are not free in T, then

$$\sigma' \in m(T)\sigma \text{ iff } \sigma'\{a/x'\} \in m(T)(\sigma\{a/x'\})$$

for all states σ,σ' and $a \in D^\sigma$.

Now let σ_0 and σ_0' be identical to σ and σ' on x, but let the values of x' in both σ_0 and σ_0' be the same as the values of x in σ'. That is, $\sigma_0 = \sigma\{x^{\sigma'}/x'\}$ and $\sigma_0' = \sigma'\{x^{\sigma}/x'\}$. Then $\sigma_0' \in m(S_k)\sigma_0$ since $\sigma' \in m(S_k)\sigma$, but σ_0' differs from each state in $m(T)\sigma_0$ on some variable in x. That is, $\sigma_0 \models \langle S_k \rangle x = x'$ but $\sigma_0 \not\models \langle T \rangle x = x'$. Therefore, $\sigma_0 \not\models p\langle T \rangle x = x'$. ∎

It seems worthwhile to mention a reasonable objection to our taking $m(S) = m(T)$ as synonymous with the *equivalence* of S and T. The problem with this notion of equivalence is that replacing one subprogram by an equivalent subprogram may not yield an equivalent program. For example, let S be the body of the procedure discussed in Section 2, i.e.

if u = v **then** v:=0;v:=u,

and let T be the program u:=u. Then $m(S) = m(T)$, but the program

decl P = ⟨(addr u,v): S⟩ **do** P(z,z) **end**

has the same effect as z:=0, whereas if S is replaced by T, the resulting program has no effects. The difficulty is that m(S) and m(T) only give the meanings of S and T when all variables are unshared, but their behavior as subprograms may depend on their behavior when sharing occurs. This problem does not arise with environment-store program semantics since the meanings of S and T differ in an environment that maps both u and v to the same location.

Unfortunately, ordinary first order logic cannot provide a suitable semantical determination theorem for environment-store meanings of programs. In the usual semantics of first-order formulas, the satisfiability of a formula p depends only upon first-order states which correspond to the composition of an environment and store. Therefore, our choice of a first-order assertion language that cannot detect sharing for expressing pre- and post-conditions leads naturally to the corresponding choice of unshared program semantics.

One straightforward way to extend Theorem 2 so that the environment-store meanings of programs match their termination theories is to extend the assertion language. A two-typed

first order language with a *location* type and a *contents* type as in [20] is a natural choice. The completeness results corresponding to Lemma 2 and Theorem 1 should extend without difficulty to this two typed first order logic, although we have not checked the details. Another variation on Theorem 2 may be obtained by adopting a different definition of termination theory. Define a *declaration body context* $\mathcal{C}$ to be a program fragment of the form

decl P ⇐ ((**val** x, **addr** y: ...) **do** P(u,v) **end**,

where all variables in y are assumed to be distinct but the variables of v need not be. Let $\mathcal{C}$[S] denote the result of inserting S for the ellipsis ... in $\mathcal{C}$. Then define the *contextual termination theory* of S to be the set of triples ($\mathcal{C}$,p,q) such that and $\mathcal{C}$[S] is a well-formed program (all free variables in S occur in x or y) and p<$\mathcal{C}$[S]>q is valid. Then it is a corollary of Theorem 2 that two programs are equivalent over all address sharing patterns of their free variables iff they have the same contextual termination theories. Furthermore, if S_1 is a subprogram of S, and S_2 has the same contextual termination theory as S_1, then S with S_1 replaced by S_2 remains equivalent to S.

6. Conclusion

Our completeness theorem shows that all valid termination assertions are provable. A stronger statement would be a *deductive completeness* theorem, i.e., if any set of termination assertions Γ semantically implies p<S>q, then p<S>q is provable from Γ. This holds if Γ contains only first order assertions by Lemma 2. However, even if we consider only sets Γ which are singletons, deductive completeness is not possible. This is because the assertion *true*<S>*true* semantically implies *false* iff the program S never halts. Since the set of totally divergent programs is not recursively enumerable [15], the set of termination assertions *true*<S>*true* such that {*true*<S>*true*}⊨*false* is not recursively enumerable (cf. footnote p. 19).

Two projects for further investigation are to enrich the programming language and to

expand the assertion language. Programs with procedures as parameters and with more complicated data objects are two possibilities. Our assumption that all undeclared global procedures are explicitly parameterized might also be relaxed by adding predicates to the assertion language which allow the global variables used by a procedure to be identified. For example, a zero-ary predicate $INDEP_{P,x}$ might be used to state that the behavior of procedure P is independent of the variable x. This is an adaptation of the "interference" concept discussed by Reynolds [19]. Another possibility noted in the previous section is to use a typed language which allows sharing of addresses to be treated explicitly following Trachtenbrot [20]. We do not foresee any fundamental difficulties in extending our results to handle variable dependence and sharing. Allowing procedure parameters, however, seems to lead to higher-order assertion languages. Insofar as our results depend crucially on the compactness property of first order logic, they will not generalize easily to programs with unrestricted procedure parameters. However, by considering the Henkin interpretation of type theory (cf. [17]), even this obstacle may be surmountable.

7. Appendix. Completeness for Global Procedure Calls

We show that axioms P1-8 are complete by showing that any consistent set of assertions is satisfiable. Two important preliminaries are

> **Lemma 7:** (Generalization) Let Γ be a set of assertions and q an assertion. If $\Gamma \vdash q$ and z is not free in Γ then $\Gamma \vdash \forall z q$.

and

> **Lemma 8:** (Deduction) If $\Gamma \cup \{p\} \vdash q$, then $\Gamma \vdash (p \supset q)$.

Both are proved by induction on proofs (cf. Enderton [9]).

Let Γ be a set of first order assertions about global procedure calls such that $x \neq x$ is not provable from Γ using P1-8 and modus ponens. Let $\mathfrak{L}$ denote the signature of Γ and let $\mathfrak{L}_P$ denote the associated first order signature. We construct a state satisfying Γ from constants following the usual Henkin-style procedure for first order logic (see Enderton [9] or Chang and Keisler [4]). The construction consists of the following five steps.

(1) Select an infinite set Υ of fresh variables. The state σ satisfying Γ will have equivalence classes of variables from Υ as its domain. Usually constants are used, but since constants may not occur as address parameters in procedure calls, variables work better for assertions about global procedures calls.

(2) Construct a set of formulas $\Gamma' \supseteq \Gamma$ such that for each formula q of the expanded language (with variables from Υ), Γ' contains formulas

(a) $\neg\forall x q \supset \neg q[v/x]$
(b) $\langle P(t,x)\rangle q \supset (\langle P(t,x)\rangle x = v) \wedge q[v/x]$

where v and $\mathbf{v} = v_1,...,v_k$ are new variables (from Step 1) and $\mathbf{v} \simeq x$. As each variable $v \in \Upsilon$ is added to Γ', an infinite set of formulas $\{v = v_j\}_{j\geq 0}$ for fresh $v_j \in \Upsilon$ is also added. This is done in such a way that Υ is not exhausted by any finite number of additions. The purpose of the formulas $\{v = v_j\}_{j\geq 0}$ are to provide infinite equivalence classes of variables from Υ, i.e. each equivalence class will have infinitely many representatives in the model we construct.

The construction proceeds in stages, starting from $\Gamma_0 = \Gamma$. Let Γ_i be the result of the i-th stage, and let q, x, P, t and x be the i-th formula, variable, procedure variable, vector of terms and vector of variables in some enumeration in which all necessary combinations appear. Then to construct Γ_{i+1}, pick variables v and $\mathbf{v} = v_1,...,v_k$ (with $\mathbf{v} \simeq x$) from Υ which do not occur in Γ_i, q, P or t. For each variable $w \in \{v, v_1,...,v_k\}$, also form a set of formulas $\mathcal{E}_w = \{w = w_j\}_{j\geq 0}$ such that $\mathcal{E}_w$ has infinitely many fresh variables $w_j \in \Upsilon$ and no w_j occurs in Γ_i, p, t, x, q or any previous $\mathcal{E}_w$. Let $\mathcal{E} = \cup_w \mathcal{E}_w$ and let

$$\Gamma_{i+1} = \Gamma_i \cup \{(a),(b)\} \cup \mathcal{E}.$$

Assuming that Γ_i is consistent, we prove that Γ_{i+1} is consistent as follows. Suppose $\Gamma_i \cup \{(b)\}$ is inconsistent. Then by the Deduction Lemma and propositional reasoning,

$$\Gamma_i \vdash \langle P(t,x)\rangle q$$

and

$$\Gamma_i \vdash \neg(\langle P(t,x)\rangle x = v \wedge q[v/x]).$$

But since $\mathbf{v}$ is a vector of variables which do not appear in Γ_i, it follows by Generalization (Lemma 7) that

$$\Gamma_i \vdash \neg\exists \mathbf{v}(\langle P(t,x)\rangle x = \mathbf{v} \wedge q[\mathbf{v}/x_j).$$

Therefore, by P8,

$$\Gamma_i \vdash \neg(\langle P(t,x)\rangle q),$$

which contradicts the assumption that Γ_i is consistent. By a similar argument (see [4]), the consistency of $\Gamma_i \cup \{(a),(b)\}$ may be reduced to that of $\Gamma_i \cup \{(b)\}$. Clearly adding sets of the form $\{w = w_j\}$ does not destroy consistency since none of the w_j's appear in $\Gamma_i \cup \{(a),(b)\}$. (If some set $\Gamma^* \cup \{w = w_k\}$ is inconsistent, then $\Gamma^* \vdash \neg(w = w_k)$ and so by Generalization $\Gamma^* \vdash \forall w_k \neg(w = w_k)$, i.e. Γ^* is inconsistent.) Thus if Γ is consistent, so are Γ_1, Γ_2,... and therefore $\Gamma'' = \cup_i \Gamma_i$ must be consistent.

(3) Extend Γ'' to a maximally consistent set Δ, i.e. for any formula q, either $q \in \Delta$ or $\neg q \in \Delta$. This is done in the usual manner [4].

(4) Define a state σ whose domain D^σ is the set of equivalence classes of variables from $\mathfrak{V}$. Define functions f^σ and relations R^σ, $P_E{}^\sigma$ according to the formulas in Δ.

For any terms t and t', define $t == t'$ iff $(t = t') \in \Delta$ and let [t] denote $\{t' \mid t == t'\}$. Let $D^\sigma = \{[v] \mid v \in \mathfrak{V}\}$. Define $v^\sigma = [v]$ and $t^\sigma = [t]$. Note that $t^\sigma \in D^\sigma$ since $\exists y(t = y)$ is provable from P1-6 and $(\exists y(t = y) \supset t = v) \in \Delta$ for some $v \in \mathfrak{V}$ by construction of Δ. For functions and relations, define

(a) $f^\sigma([v_1],....,[v_n]) = (fv_1...v_n)^\sigma$
(b) $\langle [v_1],....,[v_n]\rangle \in R^\sigma$ iff $Rv_1...v_n \in \Delta$ for each R in $\mathfrak{L}$
(c) $\langle \mathbf{b},\mathbf{c},\mathbf{d}\rangle \in P_E{}^\sigma$ iff there exist vectors of variables $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ from $\mathfrak{V}$ with $E_\mathbf{v} = E_\mathbf{w} = E$, $\mathbf{b} = [\mathbf{u}]$, $\mathbf{c} = [\mathbf{v}]$ and $\mathbf{d} = [\mathbf{w}]$ such that $(\langle P(\mathbf{u},\mathbf{v})\rangle \mathbf{v} = \mathbf{w}) \in \Delta$. Here $[\mathbf{u}]$ denotes $[u_1],....,[u_{v_P}]$.

It is straightforward to verify that f^σ and R^σ are well-defined by (a) and (b) as usual (see [4]). To see that (c) meets the restriction posed in Section 1, note that if $\langle \mathbf{b},\mathbf{c},\mathbf{d}\rangle \in P_E{}^\sigma$ then

$$i\, E\, j \Rightarrow v_i = v_j \text{ and } w_i = w_j$$

and so

$$i\,E\,j \Rightarrow c_i = c_j \text{ and } d_i = d_j$$

(5) Show that $\sigma \models q$ iff $q \in \Delta$ by induction on the length of formulas.

For first order atomic formulas, this is immediate from the definition of σ. The connective cases are also straightforward. For example, $\sigma \models \neg q$ iff $\sigma \not\models q$ iff $q \notin \Delta$ iff $\neg q \in \Delta$.

Consider $\forall x q$. Note that there is some formula $\neg\forall x q \supset \neg q[v_q/x]$ in Δ with $v_q \in \mathfrak{V}$ not appearing in q. If $\sigma \models \forall x q$ then certainly $\sigma\{[v_q]/x\} \models q$. By the Substitution Lemma, $\sigma \models q[v_q/x]$. Since v_q does not appear in q, the formula $q[v_q/x]$ has the same length as q and so by the inductive hypothesis $q[v_q/x] \in \Delta$. If $\forall x q$ is not in Δ, then $\neg\forall x q$ must be in Δ and hence $\neg q[v_q/x] \in \Delta$ by modus ponens. But since $q[v_q/x] \notin \Delta$, it follows that $\forall x q \in \Delta$.

For the converse, suppose $\sigma \not\models \forall x q$. Then for some $v \in \mathfrak{V}$, $\sigma\{[v]/x\} \not\models q$. Therefore, by the Substitution Lemma, $\sigma \not\models q[v/x]$. Since every equivalence class [v] is infinite by construction of Γ' (Step 2), it may be assumed that v does not occur in q and hence q[v/x] has the same length as q. Thus $q[v/x] \notin \Delta$ by the inductive hypothesis. Therefore $\neg q[v/x] \in \Delta$ and $\forall x q$ cannot be in Δ by P2.

The final case is $\langle P(t,x)\rangle q$. We first consider q of the form $x = y$ with $x \simeq y$. It follows from the definition of satisfaction that

$$\sigma \models \langle P(t,x)\rangle x = y \quad \text{iff} \quad (t^\sigma, x^\sigma, y^\sigma) \in P_{E_x}{}^\sigma.$$

By definition of σ, $(t^\sigma, x^\sigma, y^\sigma) \in P_{E_x}{}^\sigma$ iff

(*) There exist vectors of variables u, v and w from $\mathfrak{V}$ with $[u] = [t]$, $[v] = [x]$, $[w] = [y]$ and $E_v = E_w = E_x$ such that $(\langle P(u,v)\rangle v = w) \in \Delta$

It remains to be shown that (*) is equivalent to

(**) $\quad (\langle P(t,x)\rangle x = y) \in \Delta$

If (*) holds, then by definition of the equivalence classes [ ] of terms,

$\Delta \vdash u=t \wedge v=x \wedge w=y.$

Thus from P7,

$\Delta \vdash (\langle P(t,x)\rangle x=y)$,

which implies (**).

To see that (**) implies (*), assume that (**) holds. Since $\exists z(z=t)$ is provable from P1-6 for any term t, the construction of Δ ensures that for each term t there is a proof from Δ that t is equal to some variable in $\mathcal{V}$. Hence there exist vectors of variables u, v and w with $v \simeq x$ such that

$\Delta \vdash u=t \wedge v=x \wedge w=y.$

Therefore, from P7, we conclude (*). Thus

$\sigma \models (\langle P(t,x)\rangle x=y)$ iff $(\langle P(t,x)\rangle x=y) \in \Delta$.

In general, if $\sigma \models \langle P(t,x)\rangle q$, then by definition of satisfaction there is some $v \in \mathcal{V}$ with $v \simeq x$ such that

$(t^\sigma, x^\sigma, v^\sigma) \in P_{E_x}{}^\sigma$ and $\sigma\{v^\sigma/x\} \models q$.

Since each equivalence class of variables in $\mathcal{V}$ is infinite, each v_i may be chosen so as not to occur in q. By the Substitution Lemma, $\sigma \models q[v/x]$ and so by the inductive hypothesis, $q[v/x] \in \Delta$. Since $\sigma \models (\langle P(t,x)\rangle x=v)$, we have $(\langle P(t,x)\rangle x=v) \in \Delta$ and therefore $\Delta \vdash \langle P(t,x)\rangle q$ by P8. Since Δ is deductively closed, $(\langle P(t,x)\rangle q) \in \Delta$. This shows that if $\sigma \models \langle P(t,x)\rangle q$, then $(\langle P(t,x)\rangle q) \in \Delta$.

For the converse, assume $\langle P(t,x)\rangle q \in \Delta$. Then by the construction of Δ,

$\langle P(t,x)\rangle x=v \wedge q[v/x] \in \Delta$

for some $v \in \mathcal{V}$ not occurring in t, x or q and with $v \simeq x$. Therefore $\sigma \models \langle P(t,x)\rangle x=v$ and so

$(t^\sigma, x^\sigma, v^\sigma) \in P_{E_x}{}^\sigma$.

By the inductive hypothesis, $\sigma \models q[v/x]$ and so by the Substitution Lemma, $\sigma\{[v]/x\} \models q$. Thus $\sigma \models \langle P(t,x)\rangle q$. This concludes the proof of claim (5), i.e. for any first order assertion about global procedures q, $\sigma \models q$ iff $q \in \Delta$.

From (5) and $\Gamma \subseteq \Delta$ it follows that $\sigma \models \Gamma$. Thus every consistent set is satisfiable and the axiomatization is complete.∎

8. References

1. Andreka, H., I. Nemeti and I. Sain. A Complete First-Order Dynamic Logic. Tech. Rep. 801224, Math. Inst. Hungarian Academy of Sciences, May, 1981.

2. Apt.,K.R. Ten Years of Hoare's Logic, A Survey, Part I. Proceedings 5th Scandinavian Logic Symposium, 1979, pp. 1-44.

3. Cartwright, R. and J. McCarthy. First Order Programming Logic. Proc. 6-th Annual POPL Conf., January, 1979. pp 68-80

4. Chang, C.C. and H.J. Keisler. *Model Theory.* North-Holland, 1973.

5. Clarke, E.M Jr. Programming Language Constructs for Which it is Impossible To Obtain Good Hoare Axiom Systems. *JACM 26*, 1 (January 1979). pp 129-147

6. Cook, S.A. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Computing 7* (1978). pp 129-147.

7. Csirmaz, L. Programs and Program Verification in a General Setting. *Theoretical Computer Science 16*, 2 (November 1981). pp 199-210

8. deBakker, J. *Mathematical Theory of Program Correctness.* Prentice-Hall, 1980.

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

10. Gallier, J.H. Nondeterministic Flowchart Programs with Recursive Procedures: Semantics and Correctness I. *Theoretical Computer Science 13*, 2 (February 1981). pp 193-223

11. Harel, D. *Lecture Notes in Computer Science. Vol. 68: First-Order Dynamic Logic.* Springer-Verlag, 1979.

12. Harel, D., A.R. Meyer and V. Pratt. Computability and Completeness in Logics of Programs: Preliminary Report. 9-th ACM Symposium on Theory of Computing, Boulder, Colorado, May, 1977, pp. 261-268. Revised version, M.I.T. Lab. for Computer Science

TM-97, (Feb. 1978) 16 pp.

13. Hitchcock, P. and D. Park. Induction Rules and Termination Proofs. In M. Nivat, Ed., *Automata, Languages and Programming*, American Elsevier, New York, 1973, pp. 225-251.

14. Kfoury, D.J. Comparing Algebraic Structures up to Algorithmic Equivalence. In M. Nivat, Ed., *Automata, Languages and Programming*, American Elsevier, New York, 1973, pp. 253-263.

15. Luckham, D.C., D.M. Park and M.S. Paterson. On Formalized Computer Programs. *J. Computer System Sciences*, 4 (1970). pp 220-249.

16. Meyer, A.R. and Halpern, J.Y. Axiomatic Definitions of Programming Languages: A Theoretical Assessment (Preliminary Report). Proc. 7-th Annual POPL Conf., January, 1980. Massachusetts Institute of Technology Tech. Report MIT/LCS/TM-163 (April 1980); to appear JACM (1982).

17. Monk, J. D. *Graduate Texts in Mathematics. Vol. 37: Mathematical Logic.* Springer-Verlag, 1976.

18. Pratt, V. Semantical Considerations on Floyd-Hoare Logic. Proc. 17-th Symp. on Foundations of Computer Science, Houston, TX, October, 1976, pp. 109-121.

19. Reynolds, J.C. Idealized Algol and its Specification Logic. Tech. Rep. 1-81, School of Computer and Information Science, Syracuse University, 1981.

20. Trachtenbrot, B.A. On Denotational Semantics and Axiomatization of Partial Correctness for Languages with Procedures as Parameters and with Aliasing. Unpublished Manuscript (1981).