Axiomatic Definability and Completeness for Recursive Programs

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

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. Termination assertions are more tractable technically than the partial correctness assertions usually treated in the literature. Termination assertions are studied for a programming language which includes local variable declarations, calls to undeclared global procedures, and nondeterministic recursive procedures with call-by-address and call-by-value parameters. By allowing formulas p and q to place conditions on global procedures, we provide a method for reasoning about programs with calls to global procedures based on hypotheses about procedure input-output behavior. The set of first-order termination assertions valid over all interpretations is completely axiomatizable without reference to the theory of any interpretation. Although uninterpreted assertions have limited expressive power, the set of valid termination assertions defines the semantics of recursive programs in the sense of Meyer and Halpern [10]. Thus the axiomatization constitutes an axiomatic definition of the semantics of recursive programs.

Introduction

Many formal systems for proving properties of programs consist of rules for deriving partial correctness assertions. 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 computations of S. For even a very simple set of programs, the set of valid partial correctness assertions is not recursively enumerable [7]. In contrast, the set of valid termination assertions about **while** programs has a simple axiomatization [10]. 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. 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. Termination assertions are statements of *total correctness* for deterministic programs.

A natural approach to the modular design of software 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. We consider recursive programs with calls to undeclared global (accessible everywhere) procedures. We propose as a reasonable "specification language" for proving first-order termination assertions about recursive programs the class of first-order formulas extended to include termination assertions about global procedure calls.

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. [3]). We consider only assertions true of all structures and do not assume that the first-order theory of any structure is given as axioms. As a result, Theorem 1 (completeness) 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 [10] 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 suggests the uses and shortcomings of first-order termination assertions. Let AX denote the conjunction of axioms for addition, proper subtraction and multiplication for natural numbers using constants 0 and 1. Let $\underline{n}$ denote the numeral for n, i.e. $\underline{n} = 1 + 1 + \ldots + 1$ where 1 appears n times. For any natural number n, the assertion

$$(AX \wedge x = \underline{n}) \langle S_{fact} \rangle (y = \underline{n!})$$

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

```
proc P(u,v)  =  (if u = 0 then v: = 1
                       else P(u-1,v); v: = v*u fi);
Call P(x,y).
```

However, the more general termination assertion stating that S_{fact} computes the factorial 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 $f(0) = 1 \wedge \forall x[f(x+1) = (x+1)*f(x)]$. Then the termination assertion $(AX \wedge p) \langle S_{fact} \rangle (y = f(x))$ is valid whenever the variables are interpreted as ranging over natural numbers and the arithmetic operations are given their usual meaning. 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_{fact} will not terminate since it cannot produce 0 by subtracting 1 from x any finite number of times. Hence $(AX \wedge p) \langle S_{fact} \rangle (y = f(x))$ cannot be proved using sound inference rules. This inherent limitation of uninterpreted first-order assertions is emphasized in the well-known paper of Hitchcock and Park [8].

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 [10]. This theorem extends Theorem 5.1 of [10] 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 suitable than partial correctness assertions for axiomatic definitions of semantics.

1. Global Procedures

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 $f_1, f_2, \ldots$ and relation symbols $R_1, R_2, \ldots$ each of some specific arity. A *signature for first-order logic with global procedures*, or more simply *signature*, is a first-order signature augmented with a disjoint set $P_1, P_2, \ldots$ 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

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

where P is a procedure variable, $\mathbf{t} = t_1, \ldots, t_{v_P}$ are first-order terms, and $\mathbf{x} = x_1, \ldots, x_{a_P}$ are variables. The construct $\langle P(\mathbf{t},\mathbf{x}) \rangle q$ is intended to express that formula q is true after calling global procedure P with value parameters $\mathbf{t}$ and address parameters $\mathbf{x}$. Additional symbols such as $\wedge$, $\supset$, $\equiv$ and $\exists$ are considered abbreviations for the appropriate combinations of $\vee$, $\neg$ and $\forall$.

A few words of motivation are in order before presenting the formal semantics of assertions about procedure calls. 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. Axioms for assertions about programs which declare procedures with other parameter passing mechanisms such as call-by-value/result and call-by-name may be obtained by making slight changes in the "macro-expansion" rule presented in Section 2. To be consistent in the design of our programming language, we also provide call-by-value and call-by-address parameters to undeclared global procedures.

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

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

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

The way the behavior of a procedure may depend on its address parameters is different from the way it may depend on its value parameters. For example, suppose procedure P has two call-by-value parameters, and 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 the procedure

proc P(u,v) = (u: = 0; v: = 1; **if** u = 1 **then** v: = 0)

If both parameters are passed by address, then the call P(x,y) returns with y = 0 iff x and y share the same location. This shows that a procedure may detect and hence may depend arbitrarily upon which of the call-by-address actual parameters share locations. In summary, we assume that the behavior of a procedure call depends only on the values of the actual call-by-value parameters and the values and sharing of addresses of the actual call-by-address parameters.

The possible behavior of procedures 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), define the *address sharing relation* $E_\mathbf{x}$ on $\{1,...,k\}$ by

$$i\, E_\mathbf{x}\, j \quad \text{iff} \quad x_i \text{ and } x_j \text{ are the same variable.}$$

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 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 x and y are both vectors of variables of length k, then P(x) necessarily produces the same result as P(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(t,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(t,x) can return with the address parameters equal to **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*

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

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 $\mathcal{R}_{P_1}$ and $\mathcal{R}_{P_2}$ are assumed to be disjoint. Furthermore, each is disjoint from the first-order relations 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 relations in $\mathcal{R}_P$ 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^{σ_1} with functions f^{σ_1} and relations R^{σ_1} of appropriate arities on D^{σ_1} corresponding to each function and relation symbol of the signature, and with elements x^{σ_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

$$\text{If } \langle \mathbf{b},\mathbf{c},\mathbf{d}\rangle \in P_E{}^\sigma \text{ with } |\mathbf{c}| = |\mathbf{d}| = a_P,$$
$$\text{then } i\,E\,j \Rightarrow d_i = d_j \text{ and } c_i = c_j.$$

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

$$\sigma \models \langle P(\mathbf{t},\mathbf{x})\rangle q \quad \text{iff} \quad \exists \mathbf{d} = d_1,...,d_{a_P} \in D^\sigma \text{ such that}$$
$$(\mathbf{t}^\sigma,\mathbf{x}^\sigma,\mathbf{d}) \in P_{E_\mathbf{x}}{}^\sigma \text{ and } \sigma\{\mathbf{d}/\mathbf{x}\} \models q$$

Note that if $\langle P(\mathbf{t},\mathbf{x})\rangle q$ is interpreted as meaning that q is true after calling P with t and x, then this definition forces the output of P(t,x) to depend only on the values of explicit parameters t,x and address sharing pattern $E_\mathbf{x}$ and only allows P(t,x) to alter the values of address parameters x.

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, 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* does not imply $\forall x\langle P(x,x)\rangle$*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.

The substitution of a term t for a variable z in an assertion q about global procedure calls, written q[t/z], is defined inductively on assertions as usual for all cases other than $\langle P(s,x)\rangle q$. If t is a simple variable not among $x = x_1,\ldots,x_k$ or z is not among $x_1,\ldots,x_k$ then substitution is straightforward replacement, i.e.

(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 substitution $(\langle P(s.x)\rangle q)[w/z]$ 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 $q[v/z][z/v] = q$. This is easily proved by induction on formulas. 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 to 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 [5], 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 inference rule modus ponens, form a sound and complete proof system for first-order logic with global procedures.

P1. Propositional Tautologies.

P2. $\forall xq \supset q[t/x]$ whenever no variable of t becomes bound in q[t/x]

P3. $\forall x(p \supset q) \supset (\forall xp \supset \forall xq)$

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

P5. $x = x$

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

2. Recursive Programs

Recursive programs have abstract syntax

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

where declaration D is given by

$D ::= \; (x \textbf{ init } t) \mid P \Leftarrow B$

and procedure abstract B has form

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

The statements are assignment, test, procedure call, concatenation, union and declaration. Union denotes nondeterministic choice of S_1 or S_2 (i.e. do S_1 or S_2 but not both) and p? denotes the test which allows execution to continue iff p is true. Many statements common to Algol-like languages may be considered "syntactic sugar" for recursive programs. For example, the statement **if..then..else..fi** may be written

if p **then** S_1 **else** S_2 **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 declaration **decl** D **do** S **end** declares a local variable or recursive procedure with scope S. Variable declaration (x **init** t) defines a new local variable x with initial value t. Procedure declaration $P \Leftarrow \langle$(**val x, addr y**) : S$\rangle$ declares a possibly recursive procedure P with formal value parameters $\mathbf{x} = x_1,...,x_{v_P}$ formal address parameters $\mathbf{y} = y_1,...,y_{a_P}$ 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. The **while** statement may be expressed using recursion by

```
while p do S od  ≡
    decl P ⇐ ⟨(p?;S;P) ∪ ¬p⟩ do P end
```

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 (z_1 init t_1) do
          decl (z_2 init t_2) do
            ...
            decl (z_n  init t_n ) do
                S
            end
            ...
          end
        end.
```

Although somewhat more involved, mutually recursive procedures may also be declared as nested procedures. For example, the program with mutually recursive procedures

```
decl P_1⇐⟨S_1⟩,P_2⇐⟨S_2⟩ do S end
```

may be written as

```
decl P_1 ⇐ ⟨decl P_2 ⇐ ⟨S_2⟩ do S_1⟩ do
    decl P_2 ⇐ ⟨decl P_1 ⇐ ⟨S_1⟩ do S_2⟩ do
        S
    end
end.
```

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

The meaning m(S) of a nondeterministic program S is a mapping from "initial" states to sets of "final" states as in Harel [6]. We define the meaning of programs inductively as usual, with

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

and

$$m(\mathbf{decl}\ (x\ \mathbf{init}\ t)\ \mathbf{do}\ S\ \mathbf{end})\sigma = (m(S)\sigma\{t^\sigma/x\})\{x^\sigma/x\}.$$

The right hand side of the variable declaration definition denotes the set of states $m(S)\sigma\{t^\sigma/x\}$ with the replacement $\{x^\sigma/x\}$ applied to each one. It should be noted that our state semantics are entirely consistent with the more usual environment and store semantics as in deBakker [4] and Apt [1]. However, the proof of Theorem 1 and, especially, the statement and proof of Theorem 2 are simplified by choosing state semantics.

Following Apt [1], we treat procedure calls by syntactic means. In keeping with the convention for Algol-like languages, recursive programs have statically scoped variables. Dynamic scoping, as well as alternative parameter mechanisms such as call-by-value/result and call-by-name may also be handled using variations of the macro-expansion definition below. The definitions of program S with variable x substituted for y and with procedure name P_1 substituted for P_2, written S[y/x] and $S[P_1/P_2]$ respectively, are straightforward (see deBakker [4]). We define the *macro-expansion of calls to* P *in* S using procedure abstract B = $\langle$(**val x, addr y**) : $S_0\rangle$, written S≪B/P≫, by induction on program structure:

(i) (x := t)≪B/P≫ = (x := t). p? is similar.

(ii) P(t,v)≪B/P≫ = **decl (z init t) do** S_0[z/x][v/y] **end**
where $z_1,...,z_{v_P}$ do not appear in **t**, **v** or B.

(iii) $S_1;S_2$≪B/P≫ = S_1≪B/P≫ ; S_2≪B/P≫. $S_1 \cup S_2$ is similar.

(iv) **decl** (x **init** t) **do** S **end**≪B/P≫ =
decl (z **init** t) **do** S[z/x]≪B/P≫ **end**
where z is not free in S or B.

(v) **decl** $P_1 \Leftarrow S_1$ **do** S **end** ≪B/P≫ =
decl $P_2 \Leftarrow S_1$ **do** $S[P_2/P_1]$≪B/P≫ **end**
where P_2 is not free in S or B.

The declaration of a vector **z** of new variables with initial value **t** in (ii) produces call-by-value. Value/result 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 (ii) (see Apt [1]). Renaming of bound variables and procedure names in (iv) and (v) give static scoping. Dynamic scoping in recursive programs can be treated by changing (iv) to get dynamically scoped variables or (v) to get dynamically scoped procedure names. The *syntactic expansions* of a procedure abstract B = ⟨**val** x, **addr** y : S⟩ with respect to P are defined inductively by

$B^0 = \langle$**val** x, **addr** y : *false*?$\rangle$
$B^{i+1} = \langle$**val** x, **addr** y : $S \ll B^i/P \gg \rangle$

The (i+1)-st expansion of B with respect to P is the procedure abstract whose body is the macro-expansion of P in S using B^i. The meaning of a program with procedure declaration is then defined using macro-expansions by

m(**decl** (P⇐B) **do** S **end**)σ = $\cup_i$ m($S \ll B^i/P \gg$)σ

This syntactic definition suggests inference rule A7 used in Theorem 1.

3. Completeness

The termination assertion p<S>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<S>q, written $\sigma \models$p<S>q, if $\sigma \models$p implies $\exists \sigma' \in m(S)\sigma$ such that $\sigma' \models$q. The termination assertion p<S>q is *valid*, written $\models$p<S>q, if every state σ satisfies p<S>q. It is a straightforward consequence of the definitions that a state σ satisfies the termination assertion <P(t,v)>q iff σ satisfies the first-order assertion about global procedure calls <P(t,v)>q. Thus the use of identical syntax for first-order assertions about procedure calls and termination assertions about procedure calls should not be confusing. All valid termination assertions about recursive programs are provable using the axioms and rules of inference of Theorem 1 below.

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

Axioms

A1. q[t/x] <x := t> q

A2. $(r \wedge q)$ <r?> q

A3. (<P(t,v)>q) <P(t,v)> q

Rules of Inference

A4. p<S_1>r, r<S_2>q $\vdash$ p<S_1;S_2>q

A5. (a) p<S_1>q $\vdash$ p<$S_1 \cup S_2$>q
(b) p<S_1>q $\vdash$ p<$S_2 \cup S_1$>q

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

A7. p<$S \ll B^i/P \gg$>q $\vdash$ p<**decl** (P⇐B) **do** S **end**>q

A8. p<S>q $\vdash$ r<S>q whenever $\vdash r \supset p$ by the rules of Lemma 2.

A9. p<S>q, r<S>q $\vdash$ $(p \vee r)$ <S>q

Note that Theorem 1 is not a relative completeness theorem of the sort typical for partial correctness assertions. If a termination assertion is valid, i.e. true under all interpretations of the function, predicate and global procedure variables, then it is provable from axioms P1-8 of Lemma 2 and A1-9 above without appeal to further axioms. In particular, the valid termination assertions are recursively enumerable, whereas the valid partial correctness assertions are not. Theorem 1 generalizes Theorem 6.4 of Meyer and Halpern [10].

The proof of Theorem 1 uses

Lemma 3: For every recursive program S and formula q, there is a set $\{q_i\}$ of first-order assertions about procedure calls such that <S>q$\equiv \vee_i q_i$. Furthermore, if S has no procedure declarations, then there is a single such assertion q' with <S>q≡q'.

The lemma is proved by induction on programs, using the fact that **decl** (P⇐B) **do** S **end** is equivalent to the union of programs $S \ll B^i/P \gg$ and hence to the disjunction of the corresponding formulas. As a consequence of Lemma 2, we also have a compactness theorem for first-order logic with global procedure calls. Thus if p<S>q is valid, and <S>q$\equiv \vee_i q_i$, then there is some finite disjunction $q' = q_1 \vee ... \vee q_m$ such that $p \supset q'$ is valid. This fact is critical to the proof below.

Proof of Theorem 1: Suppose $\models$p<S>q. We show that p<S>q is provable from A1-9 by induction on the structure of S.

(a) If p<x := t>q is valid, then $\sigma \models p$ implies $\sigma\{t^\sigma/x\} \models q$. By the Substitution Lemma, $\models p \supset q[t/x]$ and therefore $\vdash p \supset q[t/x]$ by the rules of Lemma 2. By A1, we have $\vdash$q[x/t]<x := t>q and so $\vdash$p<x := t>q by A8.

(b) If $\models$p<r?>q, then $\models p \supset (r \wedge q)$ and hence $\vdash p \supset (r \wedge q)$. Since $\vdash (r \wedge q)$<r?>q by A2, we have $\vdash$p<r?>q by A8.

(c) Assume $\models$p<P(t,x)>q. Then the first-order assertion about global procedure calls $p \supset$<P(t,x)>q is valid and hence provable by the rules of Lemma 2. Therefore $\vdash$p<P(t,x)>q by A3 and A8.

(d) Suppose $\models$p<S_1;S_2>q. There exist sets of formulas $\{p_j\}$ and $\{q_{i,j}\}$ such that $\models$<S_2>q$\equiv \vee_j p_j$ and $\models$<S_1>$p_j \equiv \vee_i q_{i,j}$. Since $\models p \supset \vee_{i,j} q_{i,j}$, it follows from the compactness theorem of first-order logic (with global procedure calls) that there is some finite set $M = M_i \times M_j$ such that $\models p \supset \vee_{(i,j) \in M} q_{i,j}$. Let r denote $\vee_{j \in M_j} p_j$. Then since for any j,

$$\models \bigvee_{i \in M_j} q_{i,j} \supset \bigvee_i q_{i,j} \text{ and } \models (\bigvee_i q_{i,j})\langle S_1\rangle p_j$$

we have

$$\models (\bigvee_{(i,j)\in M} q_{i,j})\langle S_1\rangle r$$

and hence $\models p\langle S_1\rangle r$. By similar reasoning we obtain $\models r\langle S_2\rangle q$. From the inductive assumption we have $\vdash p\langle S_1\rangle r$ and $\vdash r\langle S_2\rangle q$ which allow us to conclude $\vdash p\langle S_1;S_2\rangle q$ by A4.

(e) Union is similar to composition. Assume $\models p\langle S_1 \cup S_2\rangle q$. As above, there is a set of formulas $\{q_{i,j}\}_{j\geq 0, i=1,2}$ with $\models \langle S_i\rangle q \equiv \bigvee_j q_{i,j}$. Then $\models p \supset \bigvee_{i,j} q_{i,j}$ and so by compactness there is a finite set M with $\models p \supset \bigvee_{j\in M,\ i=1,2} q_{i,j}$. Since $\models q_{i,j}\langle S_i\rangle q$, we have $\vdash q_{i,j}\langle S_i\rangle q$ and so by A9 and A5, $\vdash \bigvee_{j\in M,\ i=1,2} q_{i,j}\langle S_1 \cup S_2\rangle q$. Thus by A8, $\vdash p\langle S_1 \cup S_2\rangle q$.

(f) Assume $\models p\langle$**decl** (x **init** t) **do** S **end**$\rangle q$. A simple induction on programs shows that for any variable y that does not occur in p, t, S or q,

$\models p\langle$**decl** (x **init** t) **do** S **end**$\rangle q$
iff
$\models p\langle y := t; S[y/x]\rangle q$

Therefore $\vdash p\langle y := t; S[y/x]\rangle q$ by the inductive hypothesis. Thus $\vdash p\langle$**decl** (x **init** t) **do** S **end**$\rangle q$ by A6.

(g) Suppose $\models p\langle$**decl** ($P \Leftarrow B$) **do** S **end**$\rangle q$. Then by definition of m(**decl** ($P \Leftarrow B$) **do** S **end**), we have $\models p \supset \bigvee_i \langle S\langle\langle B^i/P\rangle\rangle\rangle q$. As above, there is a set of formulas $\{q_{i,j}\}$ with $\langle S\langle\langle B^i/P\rangle\rangle\rangle q \equiv \bigvee_j q_{i,j}$ for each i. From compactness, we know that there must be a finite set M with $\models p \supset \bigvee_{(i,j)\in M} q_{i,j}$ and so $\vdash p \supset \bigvee_{(i,j)\in M} q_{i,j}$ from P1-8. Since $\models q_{i,j}\langle S\langle\langle B^i/P\rangle\rangle\rangle q$ for any i and j, it follows from the inductive hypothesis that $\vdash q_{i,j}\langle S\langle\langle B^i/P\rangle\rangle\rangle q$. By A7, $\vdash q_{i,j}\langle$**decl** ($P \Leftarrow B$) **do** S **end**$\rangle q$ and so from A9, $\vdash \bigvee_{(i,j)\in M} q_{i,j}\langle$**decl** ($P \Leftarrow B$) **do** S **end**$\rangle q$. Therefore, by A8, $\vdash p\langle$**decl** ($P \Leftarrow B$) **do** S **end**$\rangle q$. ∎

4. Axiomatic Semantics

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 [10]. Namely, the termination assertions valid for a program distinguish it from all inequivalent programs. More precisely, for any program S, the termination theory of S, written $\mathcal{T}(S)$, is 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 [10] to programs with calls to global procedures and the proof is a straightforward reformulation of that in [10]. 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 and first-order assertion about procedure calls q, $\langle S_i\rangle q$ is equivalent to another such assertion. In particular, Theorem 2 holds for any set of arbitrary, not even recursively enumerable, infinite flowcharts (see [10]).

The main idea of the proof of Theorem 2 is that if $\sigma' \in m(S)\sigma - m(T)\sigma$, then we can find some pair of formulas p and q with the property that for any program T^0, $p\langle T^0\rangle q$ is valid iff there is some $\sigma'' \in m(T^0)\sigma$ which is identical to σ' on the free variables of SUT. It follows that $p\langle S\rangle q$ is valid but $p\langle T\rangle q$ is not. Therefore S and T have different termination theories.

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 SUT. Since S is equivalent to a union of programs without procedure declarations, $\bigcup_i S_i$, 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 fresh variables. By Lemma 3, there is a formula p with

$$p \equiv \langle S_k\rangle\, x = x'.$$

Since $m(S_k) \subseteq m(S)$, we have $\models p\langle S\rangle x = x'$.

It remains to be shown that $\not\models p\langle T\rangle x = x'$. Let $\sigma_0 = \sigma\{x^{\sigma'}/y\}$ and note that since the variables of x' do not appear in S or T, $m(S)\sigma_0 = (m(S)\sigma)\{x^{\sigma'}/x'\}$ and similarly for T. In particular, $\sigma_0' = \sigma'\{x^{\sigma'}/x'\}$ is in $m(S_k)\sigma_0$ but differs from each state in $m(T)\sigma_0$ on some variable of x. By choice of $(x')^{\sigma_0}$, we have $\sigma_0 \models p$ but $\sigma_0 \not\models \langle T\rangle x = x'$. Therefore $p\langle T\rangle x = x'$ fails at σ_0 and the theorem is proved. ∎

5. Conclusion

We have shown that the semantics of uninterpreted recursive programs may be defined axiomatically using first-order termination assertions. The set of valid termination assertions defines program semantics, and all valid termination assertions are derivable from axioms. This provides support for the general thesis that practical programming languages may be defined axiomatically.

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 recursive procedure calls in S in all interpretations. As a consequence, there are many termination assertions p<S>q which are valid over specific interpretations such as the integers, but which cannot be proved in general since S may not always terminate. Nonetheless, uninterpreted first-order termination assertions provide enough information about programs to distinguish between any pair of inequivalent programs.

The extension of first-order logic to include calls to global procedures has several applications. In addition to providing a convenient "specification language" for procedures called by recursive programs, first-order logic with global procedures may be used as a starting point for stronger logics of programs such as full Dynamic Logic [6]. Lemma 2 suggests that first-order reasoning about programs with calls to undeclared global procedures is essentially no more difficult than for programs without global procedures. A possible direction for further work might be to extend our system to include specifications for more complicated "black-box" modules such as abstract data types.

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 assertions Γ semantically implies p<S>q, then p<S>q is provable from Γ. As a consequence of compactness, this is possible if Γ contains only first-order assertions. More precisely, if $\Gamma \models$p<S>q for any set Γ of first-order assertions about global procedure calls, then there is a single first-order assertion r which is a conjunction of assertions from Γ such that $\vdash$(p$\wedge$r)<S>q by the axioms presented in Theorem 1. However, if Γ is a set of termination assertions, then it will not in general be possible to prove all consequences of Γ from any recursively enumerable set of axioms. In fact, even if we consider only single termination assertions, 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 [9], the set of termination assertions p<S>q such that (p<S>q)$\models$*false* is not recursively enumerable.

Two directions for further investigation are to enrich the programming language and to expand the assertion language. Programs with procedures as parameters or 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 predicate $INDEP_P(x)$ might be used to state that the behavior of procedure P is independent of x. This is an adaptation of the "interference" concept discussed by Reynolds [11]. Another possibility, following the direction of Trachtenbrot [12]. is to add relations to the language which would allow sharing of addresses to be treated explicitly. We do not foresee any fundamental difficulties arising from these possible extensions.

References

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

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

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

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

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

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

7. 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.

8. 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.

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

10. 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 (1981).

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

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

Appendix I. 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 4:** (Generalization) Let Γ be a set of assertions and q an assertion. If $\Gamma \vdash q$ and z is not free in Γ then $\Gamma \vdash \forall zq$.

and

> **Lemma 5:** (Deduction) If $\Gamma \cup \{p\} \vdash q$, then $\Gamma \vdash (p \supset q)$.

Both are proved by induction on proofs (cf. Enderton [5]).

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 $\mathcal{L}$ denote the signature of Γ and $\mathcal{L}_P$ the associated first-order signature. We can construct a state satisfying Γ from constants following the usual Henkin-style procedure for first-order logic (see Enderton [5] or Chang and Keisler [2]). The construction consists of the following five steps.

(1) Select an infinite set $\mathcal{V}$ of fresh variables. The state σ satisfying Γ will have equivalence classes of variables from $\mathcal{V}$ 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 $\mathcal{V}$), Γ' contains formulas

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

where v and $\mathbf{v} = v_1,\ldots,v_k$ are new variables (from Step 1) and $\mathbf{v} \simeq \mathbf{x}$. As each variable $v \in \mathcal{V}$ is added to Γ', an infinite set of formulas $\{v = v_j\}_{j \geq 0}$ for fresh $v_j \in \mathcal{V}$ is also added. This is done in such a way that $\mathcal{V}$ 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 $\mathcal{V}$, 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 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 $\mathbf{v} = v_1,\ldots,v_k$ (with $\mathbf{v} \simeq \mathbf{x}$) from $\mathcal{V}$ which do not occur in Γ_i or q, P or t. For each variable $w \in \{v, v_1,\ldots,v_k\}$, 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 $w_j \in \mathcal{V}$ and no w_j occurs in Γ_i, p, t, x, q or any previous $\mathcal{E}_w$. Let $\mathcal{E} = \bigcup_w \mathcal{E}_w$ and let

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

If Γ_i is consistent, then so is Γ_{i+1} 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

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

But since **v** is a vector of variables which do not appear in Γ_i, it follows by Generalization (Lemma 4) that

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

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 [2]), 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 $\Gamma' = \bigcup_i \Gamma_i$ 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 [2].

(4) Define a state σ whose domain D^σ is the set of equivalence classes of variables from $\mathcal{V}$. Define functions f^σ and relations R^σ 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 $D^\sigma = \{[v] \mid v \in \mathcal{V}\}$. 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 $v \in \mathcal{V}$ by construction of Δ. For functions and relations, define

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

It is straightforward to verify that f^σ and R^σ are well-defined by (a) and (b) as usual (see [2]). To see that (c) meets the restriction posed in Section 1, note that if $\langle b,c,d\rangle \in P_E{}^\sigma$ 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 $\forall xq$. Note that there is some formula $\neg\forall xq \supset \neg q[v_q/x]$ in Δ with $v_q \in \Upsilon$ not appearing in q. If $\sigma \models \forall xq$ 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 $q[v_q/x] \in \Delta$. It follows that $\forall xq$ must be in Δ since otherwise $\neg\forall xq$ would be in Δ and hence $\neg q[v_q/x]$.

For the converse, suppose $\sigma \not\models \forall xq$. Then for some $v \in \Upsilon$, $\sigma\{[v]/x\} \not\models q$. 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 $\forall xq$ cannot be in Δ by P2.

The final case is $\langle P(t,x)\rangle q$. We first consider q of the form $\mathbf{x} = \mathbf{y}$ with $\mathbf{x} \simeq \mathbf{y}$. 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 with [**u**] = [t], [v] = [x], [w] = [y]
and $E_v = E_w = E_x$ such that $(\langle P(\mathbf{u},v)\rangle v = w) \in \Delta$

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

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

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

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

Thus from P7,

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

which implies (**). Conversely, if (**), then since $\exists z(z = t)$ is provable from P1-6 for any term t, the construction of Δ ensures that there exist vectors of variables **u**, v and w with $v \simeq x$ such that

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

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

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

In general, if $\sigma \models \langle P(t,x)\rangle q$, then $\exists v \in \Upsilon$ with $v \simeq x$ and

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

Since each equivalence class of variables in Υ 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, $\langle P(t,x)\rangle q \in \Delta$.

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 $v \in \Upsilon$ 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 shows that 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.∎