JACM

Effective Axiomatizations of Hoare Logics

Edmund M. Clarke, Jr.[1], Steven M. German[1,3], and Joseph Y. Halpern[1,2]

1. Aiken Computation Laboratory, Harvard University, Cambridge, MA02138
2. Laboratory for Computer Science, M.I.T., Cambridge, MA02139
3. Computer Systems Laboratory, Stanford University, Stanford, CA94305

Abstract: For a wide class of programming languages **P** and expressive interpretations I, we show that there exist sound and relatively complete Hoare logics for both partial correctness and termination assertions. In fact, under mild assumptions on **P** and I we show that the assertions true in I are uniformly decidable in the theory of I (Th(I)) iff the halting problem for **P** is decidable for finite interpretations. Moreover the set of true termination assertions is uniformly r.e. in Th(I) even if the halting problem for **P** is not decidable for finite interpretations. Since total correctness assertions coincide with termination assertions for deterministic programming languages, this last result unexpectedly suggests that good axiom systems for total correctness may exist for a wider spectrum of languages than is the case for partial correctness.

The present paper is an expanded version of a paper with the same title [CGH82] given at the Ninth Annual ACM Symposium on Principles of Programming Languages at Albuquerque, New Mexico, in January, 1982. This research was supported in part by NSF Grants MCS79-08365 and MCS80-10707, Advanced Research Projects Agency contract N0039-82-C-0250, Rome Air Development Center contract F30602-80-C-0022, and a grant from the National Science and Engineering Research Council of Canada.

Table of Contents

List of Figures

1. Introduction

1.1. Background

Because Hoare Logic, or axiomatic semantics, is one of the most widely used approaches to defining programming language semantics and proving properties of programs, it is important to understand its limitations and their causes. The question of the existence of good Hoare Axiom systems for programming languages was first raised by Clarke in [Cl76/79], where it was shown that languages with certain features cannot have axiom systems that are sound and relatively complete in the sense of Cook [Co78]; natural examples of such features include: call by name parameter passing in the presence of recursive procedures, functions, and global variables, and coroutines with local recursive procedures that can access global variables.

The incompleteness results are established by observing that if a programming language **P** has a sound and relatively complete proof system for all expressive interpretations, then the halting problem for **P** must be decidable for finite interpretations. Lipton [Li77] considered a form of converse: If **P** is an *acceptable* programming language and the halting problem is decidable for finite interpretations, then **P** has a sound and relatively complete Hoare logic for expressive and effectively presented interpretations. The acceptability of the programming language is a mild technical assumption which ensures that the language is closed under certain reasonable programming constructs, and that given a program, it is possible to effectively ascertain its step-by-step computation in interpretation I by asking some quantifier-free questions about I.

Lipton actually proved a partial form of the converse. He showed that given a program P and the effective presentation of I, it is possible to enumerate all the partial correctness assertions of the form *true*{P}*false* which are true in I. From this it easily follows that we can enumerate all true quantifier-free partial correctness assertions, since we can encode quantifier-free tests into the programs. But it does *not* follow that we can enumerate all

first-order partial correctness assertions, since an acceptable programming language will not in general allow first-order tests (cf. Section 2).

A number of other researchers ([Me78], [La80]) have since attempted to clarify Lipton's proof and to extend it to handle first-order pre- and post-conditions and a wider range of acceptable programming languages.

1.2. New Results of This Paper

We consider acceptable programming languages which permit recursive procedure calls. We also require, for technical reasons, that every element of the domain of I correspond to some term in the assertion language. (These requirements seem quite reasonable; cf. Sections 2 and 4.) Under these assumptions we are able to significantly extend the results of [Cl76/79] and [Li77]:

1. We are able to eliminate the requirement that pre- and post-conditions be quantifier-free and that the interpretation be effectively presented. Under the assumption that the halting problem for **P** is decidable for finite interpretations, we show that, for all expressive interpretations, **P** has a sound and relatively complete Hoare axiom system for partial correctness assertions with arbitrary first-order pre- and post-conditions.

2. We show, in fact, that the set of partial correctness assertions true in I is actually (uniformly) decidable in the theory of I (Th(I)) provided that the halting problem for **P** is decidable for finite interpretations. Lipton's proof, on the other hand, produces an enumeration procedure for partial correctness assertions and, thus, shows only that the set of true partial correctness assertions is r.e. in Th(I).

3. We extend the decidability result to termination assertions (which coincide with total correctness assertions for deterministic programming languages). Here even stronger results can be obtained. The set of true termination assertions is (uniformly) decidable in Th(I) iff the halting problem for **P** is decidable for finite interpretations. Moreover, the set of true termination assertions is (uniformly) r.e. in Th(I) even if the halting problem for **P** is not decidable for finite interpretations.

This last result unexpectedly suggests that good axiom systems for total correctness may exist for a wider spectrum of languages than is the case for partial correctness. In particular, it may be possible to find a sound and relatively complete total correctness proof system for a language with call by name parameter passing, recursive procedures, functions, and global variables, even though no corresponding partial correctness proof system can exist.

1.3. Outline

The paper is organized as follows. In section 2 we give precise definitions for all our terms; in particular, we carefully specify the conditions that a programming language must satisfy in order to be acceptable. In section 3 we state and prove our main results, contrasting them with those of Lipton. As in Lipton's paper, our results split into two cases depending on whether there is for every program $P \in \mathbf{P}$ a number M such that P never accesses more than M elements of the domain on any input. In case such a bound exists we show that it is possible to enumerate the true termination assertions even if the halting problem for **P** is not decidable. For partial correctness our proof in this case is similar to Lipton's.

In case some program can access an unbounded number of different program states, our approach is different from that of Lipton. We show that if the interpretation I is expressive, then it is possible to effectively find formulas which make I into a standard model of arithmetic. (Lipton is able to prove the existence of a standard model of arithmetic embedded within the interpretation, but is not able to find it effectively.) We use the standard model of arithmetic to encode partial and total correctness formulas as first-order formulas over I. The oracle for Th(I) is then used to determine the truth of the encoded partial correctness assertions (resp. termination assertions).

The paper concludes in section 4 with a statement of some open problems and a discussion of the philosophical implications of our results.

2. Basic Definitions

2.1. Interpretations and Valuations

A *type* or *signature* is a set of function and predicate symbols, each with an associated arity. (Constants are just function symbols of arity zero.) An *interpretation* I (over a type Σ) consists of a domain, dom(I), and an assignment to each function (resp. predicate) symbol of Σ a function (resp. predicate) over dom(I) of the appropriate arity. Th(I) is the set of all first-order sentences (over Σ) true in I.

In all that follows, we assume we are working over a fixed finite type Σ, which contains the binary predicate $=$ (equality). Equality is always given its standard meaning in any interpretation. For technical reasons, we also assume the constant a is an element of Σ. Throughout this paper, for ease of exposition we will take $\Sigma = \{a,b,f,g,A_0,=\}$, where a and b are constants, f is a unary function symbol, g is a binary function symbol, and A_0 is a binary predicate symbol. We also assume a fixed set of variables, $var = \{x_0,x_1,...\}$. For a term t, let $var(t) = \{y \in var |\ y \text{ appears in } t\}$. Similarly, for a quantifier-free formula A, let $var(A) = \{y \in var |\ y \text{ appears in } A\}$.

A *literal* is a formula involving only a predicate symbol or its negation. Thus, the literals of type Σ are of the form $t_1 = t_2$, $\neg(t_1 = t_2)$, $A_0(t_1,t_2)$, or $\neg A_0(t_1,t_2)$.

For each interpretation I, a *valuation over I* is a mapping σ: var $\rightarrow$ dom(I). We can extend a valuation to a mapping σ: Terms $\rightarrow$ dom(I) in the obvious way. To represent a diverging computation we introduce one special valuation, $\perp$, such that $\perp(x)$ is undefined for all variables x. The valuation $\sigma[x/a]$ is identical to σ on all variables except x, and $\sigma[x/a](x) = a$.

2.2. Acceptable Programming Languages with Recursion

An *acceptable* programming language **P** must satisfy the four criteria given below.

1. For each program $P \in \mathbf{P}$ we can effectively find finite subsets $cv(P) \subseteq var(P) \subseteq var$ satisfying certain constraints given below. Intuitively, cv(P) corresponds to those variables whose values may get changed as we run program P, while var(P) also includes input variables, output variables, and any additional variables (such as those that appear in tests) upon whose values the behavior of P depends. In each interpretation I we can also associate with each $P \in \mathbf{P}$ a set of *trajectories*, $\mathcal{T}_I(P)$, where each trajectory $\tau \in \mathcal{T}_I(P)$ is a finite sequence of valuations $(\sigma_0,\sigma_1,...)$ such that $\perp$, if it appears at all, only appears as the last valuation. There is no trajectory of the form $(\perp)$. These trajectories must also satisfy:

   a. If $y \notin cv(P)$, then for all i, $\sigma_i(y) = \sigma_0(y)$. (This corresponds to our intuition that the only variables which get changed as we run program P are those in cv(P).)

   b. If $y \in cv(P)$, then for $i > 0$, $\sigma_i(y) = a$, b, $\sigma_j(x)$, $f(\sigma_j(x))$, or $g(\sigma_j(x),\sigma_k(z))$, for some $j, k < i$ and $x, z \in var(P)$.

   c. If $\sigma_0(var(P)) = \sigma_0'(var(P))$ then there is a trajectory $\tau' = (\sigma_0',\sigma_1',...) \in \mathcal{T}_I(P)$ such that $\sigma_i(var(P)) = \sigma_i'(var(P))$ for all i. This confirms the intuition that the computation of P depends only on the variables in var(P).

2. The set of (codes of) programs in **P** is recursive, and we can effectively compute the possible i^{th} steps of running a program $P \in \mathbf{P}$ on any input by asking a finite number of quantifier-free questions about I. (Note we are allowing boundedly nondeterministic computations here). Intuitively, we can think of a tree which defines all the possible computations of a program P, whose nodes are labelled by literal formulas. By knowing which of the literals are true in I, we can determine which computation path (or paths, in the nondeterministic case) were taken by P. In particular, we can determine the values of the program variables at the i^{th} step of the computation in terms of their initial values.

   More formally, given a (code for) program P and i, we can effectively find some finite set of literals, say $A_1, ..., A_k$ with $var(A_j) \subseteq var(P) = \{y_1,...y_n\}$ such that by knowing the truth value of A_j in I,σ_0, we can effectively compute a finite number of sets of terms $\{\{t_{m1},...,t_{mn}\} | m = 1,2,...\}$ over $\{a,b,f,g,y_1,...y_n\}$ which represent the possible values of the variables in var(P) at the i^{th} step of any trajectory in $\mathcal{T}_I(P)$ starting with σ_0. That is, σ is the i^{th} step of such a trajectory

iff, for some m, $\sigma(y_j) = \sigma_0(t_{mj})$ for j = 1,...n, and $\sigma(x) = \sigma_0(x)$ for $x \notin var(P)$. We can also effectively compute which (if any) of the sets $\{t_{m1},...,t_{mn}\}$ represent output values; i.e. whether there is some trajectory $(\sigma_0,...,\sigma_i)$ in $\mathcal{T}_I(P)$ with $\sigma_i(y_j) = \sigma_0(t_{mj})$ for j = 1,...,n.

3. **P** is *effectively closed under variable substitutions*; that is, given $P \in \mathbf{P}$ with $var(P) = \{x_{i_1},...,x_{i_m}\}$ and any set of m variables $\{y_1,...,y_m\}$ we can effectively find a program $P' \in \mathbf{P}$ such that $var(P') = \{y_1,...,y_m\}$ and $(\sigma_0,\sigma_1,...) \in \mathcal{T}_I(P)$ iff for some $(\sigma_0',\sigma_1',...) \in \mathcal{T}_I(P')$ we have $\sigma_j(x_{i_k}) = \sigma_j'(y_k)$ for k = 1,...,m.

4. **P** is *effectively closed under flowchart operations, subroutine calls, and runtime checks.*

To make this last notion precise, let **P'** be the least set of programs containing **P** such that if $P, Q \in \mathbf{P}'$ and A is a quantifier-free formula, then the following programs are all in **P'**. (Note that the programs in **P'** will not necessarily be in **P**. There will just be programs in **P** which simulate them.)

1. basic assignments x := a, x := b, x := y, x := f(y), x := g(y,z),

2. P;Q,

3. **if** A **then** P **else** Q,

4. **while** A **do** P,

5. **run** P **until** A,

6. **after each step of** P **do** Q.

7. **begin local** $x_{i_1},...,x_{i_m}$; P **end**

The flowchart operations work in the standard fashion. Intuitively, **run** P **until** A inserts a test for A before every statement of P, halting as soon as the test is satisfied, while **after each step of** P **do** Q inserts the whole computation of Q between successive steps of P. When we run **begin local** $x_{i_1},...,x_{i_m}$; P **end**, we first save the values of $x_{i_1},...,x_{i_m}$ (as given

in the current state), initialize them all to a, run P, and then restore the original values of $x_{i_1},...,x_{i_m}$. We extend $\mathcal{T}$, cv, and var to **P'** in a straightforward way, thus giving the programs of **P'** their formal semantics as trajectories. We leave details to Appendix 1.

Now we formally define **P** to be effectively closed under flowchart operations, subroutine calls, and runtime checks if for all $P \in \mathbf{P'}$ and all interpretations I, we can effectively find a $Q \in \mathbf{P}$ which *simulates* P in I. That is, $cv(P) \subseteq cv(Q)$, $var(P) \subseteq var(Q)$, and for all $\tau \in \mathcal{T}_I(P)$ (resp. $\mathcal{T}_I(Q)$) with $last(\tau) \neq \perp$ there exists a $\tau' \in \mathcal{T}_I(Q)$ (resp. $\mathcal{T}_I(P)$), such that $first(\tau)(var(P)) = first(\tau')(var(P))$ and $last(\tau)(var(P)) = last(\tau')(var(P))$ (where given a trajectory $\tau = (\sigma_0,...,\sigma_k)$, we define $first(\tau) = \sigma_0$ and $last(\tau) = \sigma_k$).

Thus we only require of a program like **after each step of** P **do** Q that it can be simulated by a program in **P**, possibly using some extra variables as flags. It is easy to see that flowcharts, PASCAL, ALGOL, and almost any ALGOL-like language will all constitute acceptable programming languages.

Our definition of acceptable programming language seems to coincide with the rather vague definition given in Lipton [Li77]. In any case, as we shall see below, it certainly gives us languages which are sufficiently rich to contain all the programs required by Lipton to prove his results. But for our stronger results, we seem to require that our programming languages be *acceptable with recursion*, which we define to mean acceptable and *effectively closed under (possibly recursive) procedure calls.*

To make this precise, we use semantics similar to those of [Mi81]. Let plab = $\{Z_0, Z_1, ...\}$ be some set of *program labels* and let **P"** be the smallest language containing **P**, plab, and closed under the programming constructs described above, such that if $P \in \mathbf{P"}$ and $Z \in$ plab, then $\mu Z[P]$ is a program in **P"**. Essentially $\mu Z[P]$ acts as a least fixed operator and allows us to program recursive calls. We extend $\mathcal{T}$, cv, and var to **P"** in Appendix 1.

Finally, we define **P** to be effectively closed under recursive calls, (as well as flowchart

operations, subroutine calls, and runtime checks) if for every program $P \in \mathbf{P}$" and interpretation I, there is a program $Q \in \mathbf{P}$ which simulates P in I in the sense defined above. (The observant reader will have noticed that we have not dealt with issues such as the copy rule and naming conflicts between global and local variables. But since we only require that every program $P \in \mathbf{P}$" with the semantics that we have given can be simulated by some program in **P** whatever the semantics of **P** are, such problems will not concern us here.)

A program P is *deterministic* iff for all valuations σ there is at most one trajectory $\tau \in \mathcal{T}_I(P)$ with $\text{first}(\tau) = \sigma$ and $\text{last}(\tau) \neq \perp$. The programming language **P** is deterministic iff all programs $P \in \mathbf{P}$ are.

2.3. Partial Correctness and Termination

We expand the type Σ to Σ^P by adding, for each $P \in \mathbf{P}$, a predicate symbol A_P of arity 2k, where $k = |\text{var}(P)|$. In any interpretation I, $I \models A_P(u,v)$ iff for some trajectory $(\sigma_0,\ldots,\sigma_k) \in \mathcal{T}_I(P)$ with $\sigma_k \neq \perp$, we have $\sigma_0(\text{var}(P)) = u$ and $\sigma_k(\text{var}(P)) = v$. (Note we use the letters u, v, w to indicate domain elements, while x, y, z denote variables. We use italics for vectors. Thus u indicates a vector of domain elements, and x indicates a vector of variables.) A_P defines the input-output semantics of program P. We say P *halts* on input u (in interpretation I) if there is a trajectory $\tau \in \mathcal{T}_I(P)$ such that $\text{first}(\tau)(\text{var}(P)) = u$ and $\text{last}(\tau) \neq \perp$. Otherwise we say P *diverges* on input u.

A *(first-order) partial correctness* (resp. *termination*) *assertion* is a triple $U\{P\}V$ (resp. $U\langle P\rangle V$) where U and V are first-order formulas (over Σ) and $P \in \mathbf{P}$. By definition

$$I \models U\{P\}V \text{ iff } I \models \forall x,y(U(x) \wedge A_P(x,y) \Rightarrow V(y))$$
$$I \models U\langle P\rangle V \text{ iff } I \models \forall x \exists y(U(x) \Rightarrow A_P(x,y) \wedge V(y))$$

Thus $I \models U\{P\}V$ (resp. $U\langle P\rangle V$) iff, if $U(u)$ then for all (resp. some) v which are possible outputs of P on input u, we have $I \models V(v)$. Note that in the case of deterministic programs, total correctness and termination coincide.

2.4. Expressiveness

An interpretation I is *weakly expressive* for **P** iff for every $P \in \mathbf{P}$ there is a formula B_P (of type Σ) such that

$$I \models B_P(u) \text{ iff } I \models \exists y(A_P(u,y))$$

Thus $I \models B_P(u)$ iff there is a halting computation of P on input u. Note that we do not assume we can effectively find such a B_P; only that it exists.

In Dijkstra's terminology [Di76], B_P corresponds to the weakest precondition of P with respect to *true*, or the negation of the weakest liberal precondition of P with respect to *false*.

2.5. Expressive-Herbrand and Expressive-Effective Interpretations

An interpretation I of type Σ is *effectively presented* if there is a tuple of integers pres(I) $= \langle n_{dom}, n_a, n_b, n_f, n_g, n_{A_0} \rangle$, where n_{dom} is a code for dom(I), a recursive subset of $\mathcal{N}$ (the integers), n_a, $n_b \in$ dom(I) are the interpretations of a and b, and n_f, n_g, and n_{A_0} are codes for recursive functions and predicates of the right arity which interpret f, g, and A_0 respectively.

I is *Herbrand definable* iff for all $i \in$ dom(I), there is a term t in the Herbrand Universe of {a,b,f,g} such that $I \models t = i$.

Finally, we say an interpretation I is *expressive-Herbrand* with respect to programming language **P** iff it is weakly expressive for **P** and either Herbrand definable or finite. I is *expressive-effective* if it is weakly expressive and either effectively presented or finite.

2.6. Strongly and Weakly Arithmetic Interpretations

I is said to be *strongly arithmetic* if there exist first-order formulas Z(x), S(x,y), A(x,y,z), and M(x,y,z), and a bijection φ: dom(I) $\rightarrow \mathcal{N}$ such that

1. $I \models Z(u)$ iff $\varphi(u) = 0$

2. $I \models S(u,v)$ iff $\varphi(u) + 1 = \varphi(v)$

3. $I \models A(u,v,w)$ iff $\varphi(u) + \varphi(v) = \varphi(w)$

4. $I \models M(u,v,w)$ iff $\varphi(u) \times \varphi(v) = \varphi(w)$

Note we do not assume that we can find Z, S, A, M, or φ effectively.

I is *weakly arithmetic* if there exist first-order formulas $N(x)$, $E(x,y)$, $Z(x)$, $S(x,y)$, $A(x,y,z)$, and $M(x,y,z)$ (with, respectively, k, 2k, k, 2k, 3k, and 3k free variables for some k) such that E defines an equivalence relation on $dom(I)^k$, and if $[u] = \{v \in dom(I)^k | I \models E(u,v)\}$, there is a bijection φ: $\{[u] | I \models N(u)\} \rightarrow \mathcal{N}$ such that conditions 1-4 above hold (when restricted to N) with $[u]$ replacing u as the argument to φ. (Thus, for example, condition 2 becomes

$I \models N(u) \wedge N(v) \wedge S(u,v)$ iff $\varphi([u]) + 1 = \varphi([v])$.)

Thus the natural numbers are embedded in a weakly arithmetic interpretation as equivalence classes of domain elements, while in a strongly arithmetic interpretation, every natural number corresponds to some distinct domain element.

3. Main Results

3.1. Statements of Theorems

With all these definitions in hand, we can now state our main theorems precisely:

Theorem 1: Let **P** be a deterministic, acceptable programming language with recursion. Then the following are equivalent:

1. There is an effective procedure, which, for expressive-Herbrand interpretations I, will decide which first-order partial correctness (resp. termination) assertions are true in I when given an oracle for Th(I). Thus the set of first-order partial correctness (resp. termination) assertions true in I is *uniformly* recursive in Th(I) for expressive-Herbrand interpretations I.

2. **P** has a decidable halting problem for finite interpretations; (i.e. there is an effective procedure which, when given I with dom(I) finite, a program

$P \in \mathbf{P}$ with $|var(P)| = k$, and $u \in dom(I)^k$, decides if P halts on input u in domain I.)

Moreover, even *without* the assumption that **P** has a decidable halting problem for finite interpretations, we can show that the set of first-order termination assertions true in I is uniformly r.e. in Th(I) for expressive-Herbrand I.

Similar techniques allow us to prove a variant of this theorem. By exchanging Herbrand definability for effective presentation, we can drop the assumption that the programming language allows recursive calls, but at the price of losing uniformity. We no longer get one algorithm which works as soon as it is given an oracle for Th(I), but a different algorithm for each interpretation.

Theorem 2: Let **P** be a deterministic, acceptable programming language. Then the following are equivalent:

1. The set of first-order partial correctness (resp. termination) assertions true in I is recursive in $\langle pres(I), Th(I)\rangle$ if I is expressive-effective.

2. **P** has a decidable halting problem for finite interpretations.

Moreover, the set of first-order termination assertions true in I is r.e. in $\langle pres(I), Th(I)\rangle$ for expressive-effective interpretations I.

By way of contrast, Lipton showed (in [Li77]):

Theorem (Lipton): Let **P** be a deterministic, acceptable programming language. Then the following are equivalent:

1. The true quantifier-free partial correctness assertions are uniformly r.e. in $\langle pres(I), Th(I)\rangle$ for expressive-effective interpretations I.

2. **P** has a decidable halting problem for finite interpretations.

Lipton's proof only showed how to enumerate the true partial correctness assertions of the form $true\{P\}false$. However, note that

$I \models A\{P\}B$ iff $I \models true\{\textbf{if } \neg A \textbf{ then } \omega;\ P;\ \textbf{if } B \textbf{ then } \omega\}false$

(where ω is the program which always diverges). Moreover, if A and B are quantifier-free, this modified program (or one that simulates it) is in **P**. Thus it is easy to extend Lipton's proof to quantifier-free partial correctness assertions. But this trick does not extend to first-order formulas. If A is first-order, then the program (**if** $\neg$A **then** ω) *cannot* in general be simulated by a program in an acceptable programming language, since the simulating program would violate condition 2 of Definition 2.2.

Theorem 1 uses the following lemma, which is interesting in its own right and again generalizes one of Lipton's results:

Lemma 1: If **P** is acceptable with recursion and I is expressive-Herbrand with respect to **P** then either:

1. I is strongly arithmetic, or

2. $\forall P \in \mathbf{P} \exists n$(P reaches at most n distinct valuations in any computation) (i.e. for all $\tau \in \mathcal{T}_I(P)$, $\{\sigma_i | \sigma_i \in \tau\}$ has $\leq$ n elements).

We will abbreviate condition 2 of the lemma by (†) since we refer to it so often below.

Lipton proved the same result with "acceptable with recursion" replaced by "acceptable", "expressive-Herbrand" replaced by "expressive-effective", and "strongly arithmetic" replaced by "weakly arithmetic". However we can actually get a stronger result. As a corollary to the proof of Theorem 1, we will show that if I is strongly arithmetic and expressive-Herbrand, we can *effectively find* the formulas which make I strongly arithmetic. We will rederive Lipton's result in the course of our proof of Lemma 1, and use it in proving Theorem 2.

3.2. Proof of Theorem 1

The fact that (1) $\Rightarrow$ (2) in the first half of Theorem 1 was proved by Clarke [Cl76/79]. The proof in fact goes through under much weaker hypotheses: **P** does not have to be acceptable or deterministic. To prove the remainder of Theorem 1, we will describe five

effective procedures, M_1, ..., M_5. When given an oracle for Th(I) of an expressive-Herbrand interpretation I each of them outputs first-order partial correctness or termination assertions, or their negations. They are all *sound*; that is, any assertion which is output is true in I. If I is strongly arithmetic, then M_1 is *complete* for partial correctness assertions; that is, it outputs U{P}V or ¬U{P}V for each partial correctness triple, depending on whether it is true or false in I. Similarly, M_2 is complete for termination assertions if I is strongly arithmetic. If **P** has a decidable halting problem for finite interpretations and (†) holds, then M_3 (resp. M_4) is complete for partial correctness (resp. termination) assertions. Finally, M_5 is similar to M_4, but it just enumerates all the true termination assertions U<P>V if (†) holds (but not the negations of the false ones), and does not require the assumption that **P** has a decidable halting problem for finite interpretations.

Theorem 1 then follows from Lemma 1 (which we will prove below). To decide first-order partial correctness assertions we run M_1 and M_3 in parallel. To decide first-order termination assertions we run M_2 and M_4 in parallel. To enumerate first-order termination assertions without the assumption that **P** has a decidable halting problem for finite interpretations, we run M_2 and M_5 in parallel.

3.2.1. Construction of M_1 and M_2

Consider the following set of axioms for arithmetic:

AX1. $\neg(S(x) = 0)$
AX2. $S(x) = S(y) \Rightarrow x = y$
AX3. $x+0 = x$
AX4. $x+S(y) = S(x+y)$
AX5. $x\times 0 = 0$
AX6. $x\times S(y) = x\times y + x$
AX7. $\neg(x<0)$
AX8. $x<S(y) \equiv (x<y \vee x=y)$
AX9. $x<y \vee x=y \vee y<x$

Of course, these do not constitute a complete set of axioms for arithmetic. However, an interpretation which satisfies these axioms has a "standard part" (cf. [SH67]), consisting of those elements in the domain of the form $S^k(0)$ for some integer k. In general there is no first-order formula which defines the standard part, but under certain stronger hypotheses, we will show that it can be defined.

First we inductively define an encoding of Herbrand terms of type Σ:

$\ulcorner a \urcorner = 0$
$\ulcorner b \urcorner = 1$
$\ulcorner f \urcorner = 2$
$\ulcorner g \urcorner = 3$
$\ulcorner f(t) \urcorner = \langle \ulcorner f \urcorner, \ulcorner t \urcorner \rangle$
$\ulcorner g(t,u) \urcorner = \langle \ulcorner g \urcorner, \langle \ulcorner t \urcorner, \ulcorner u \urcorner \rangle \rangle$

where $\langle\rangle$ denotes the pairing function $\langle x,y \rangle = \frac{1}{2}(x+y)(x+y+1) + x$.

Let H be a binary predicate symbol (whose intended meaning is H(u,d) iff u is the encoding of a Herbrand term equal to d) and consider the following encoding axiom:

(Enc) $\forall x,d[H(x,d) \equiv (x = \ulcorner a \urcorner \wedge d = a) \vee (x = \ulcorner b \urcorner \wedge d = b) \vee$
$(\exists y,d'(Pr(x,\ulcorner f \urcorner,y) \wedge H(y,d') \wedge d = f(d')) \vee$
$(\exists y,d_1,d_2,z_1,z_2(Pr(x,\ulcorner g \urcorner,y) \wedge (Pr(y,z_1,z_2) \wedge H(z_1,d_1)$
$\wedge H(z_2,d_2) \wedge d = g(d_1,d_2))]$
where $Pr(z,x,y) \equiv y \leq z \wedge x \leq z \wedge (z = \frac{1}{2}(x+y)(x+y+1) + x)$

We now show H "works right" on standard elements:

Lemma 2: If I satisfies AX1-9 and Enc, then $I \models H(S^k(0),d)$ iff k is the encoding of a Herbrand term whose value in I is d.

Proof: We begin by showing that the nonstandard elements, if there are any, come after all of the standard elements in the ordering $<$. That is, if u is standard and v nonstandard, $I \models u<v$. This in turn is proved using induction on k to show that if v is nonstandard, then $I \models \neg(v<S^k(0))$. The desired result then follows immediately by AX9. The base case of the

induction is just AX7, and the inductive step follows using AX8, the inductive hypothesis, and the fact that we cannot have $v = S^k(0)$ since v is nonstandard.

Now we prove our result by induction on k; we write k^* as an abbreviation for $S^k(0)$.

The base cases $k=0$ or $k=1$ follow directly from the definition. (Note the last two disjuncts in Enc cannot hold if $k=0$ or 1, since if $k=0$ or 1 and $m=\ulcorner f \urcorner$ or $\ulcorner g \urcorner$, then for all $u \in dom(I)$ we must have $I \models \neg Pr(k^*,m^*,u)$).

Inductive case: Let us assume that the lemma is true for all $k'<k$. Assume $k=\ulcorner f(t_1) \urcorner$ for some term t_1, and let $k_1=\ulcorner t_1 \urcorner$, $k_1<k$. (The case where k encodes a term of the form $g(t_1,t_2)$ is similar and will be left to the reader.) By our encoding of Herbrand terms, $k=\langle \ulcorner f \urcorner, k_1 \rangle$, and so $I \models Pr(k^*,2^*,k_1^*)$. By the inductive assumption, $I \models H(k_1^*,t_1)$. Hence, by axiom Enc it follows that $I \models H(k^*,f(t_1))$ as required.

Conversely, assume that $I \models H(k^*,d)$. By Enc, we can assume without loss of generality that there are $v,d_1 \in dom(I)$ such that $I \models Pr(k^*,2^*,v) \wedge H(v,d_1) \wedge d=f(d_1)$. (The other case is similar). Since $I \models Pr(k^*,2^*,v)$ implies $I \models v<k^*$, v must be a standard element, say k_1^*, where $k_1<k$. It then follows from the inductive assumption that k_1 encodes a Herbrand term t_1 such that $I \models t_1=d_1$. Since $k=\langle 2,k_1 \rangle$ and $I \models d=f(d_1)$, we conclude that k encodes the term $f(t_1)$ which has value d. ∎

Now we show how to use H to define the standard part in a nonstandard model of arithmetic.

Lemma 3: If I satisfies AX1-9, Enc, and is Herbrand definable, then $Std(x) \equiv \exists d \forall z(H(z,d) \Rightarrow x < z)$ defines the standard part of I.

Proof: Suppose u is standard. Because dom(I) is infinite (this is forced by AX1 and AX2) and I is Herbrand definable, there exists an element d all of whose encodings are greater than u. For this d, $I \models \forall z(H(z,d) \Rightarrow u<z)$, because if H(w,d), either w is standard, in

which case by Lemma 2 it encodes d, or it is nonstandard. In either case, w must be greater than u. Thus $I \models Std(u)$. On the other hand, if u is nonstandard, then for every $d \in dom(I)$, there exists a standard encoding w of d such that $I \models H(w,d) \wedge \neg(u<w)$. Therefore, $I \models \neg Std(u)$. ∎

Finally we need

Lemma 4: Suppose we can effectively find formulas Z'(x), S'(x,y), A'(x,y,z), and M'(x,y,z) (of type Σ) which make I strongly arithmetic. Then, for each $P \in \mathbf{P}$, we can effectively find a formula A_P' of type Σ which is equivalent to A_P in I.

Proof: See Appendix 2. ∎

Now we can define M_I to decide partial correctness assertions. It systematically guesses formulas Z'(x), S'(x,y), L'(x,y), A'(x,y,z), M'(x,y,z), and H'(x,y) and checks (by consulting its oracle for Th(I)) that Z' defines a unique element of I (i.e. $I \models \exists x(Z'(x) \wedge \forall y(Z'(y) \Rightarrow y=x))$, S', A' and M' define functions (i.e. $I \models \forall x \exists y(S'(x,y) \wedge \forall z(S'(x,z) \Rightarrow y=z))$, etc.), and that AX1-9 and Enc hold in I when written in terms of these formulas. (For example, AX2 becomes $(S'(x,z) \wedge S'(y,z)) \Rightarrow x=y$.) Now using these formulas, we can define Std(x) as in Lemma 3, and check if $I \models \forall x(Std(x))$. If not, then M_I continues guessing. But if $\forall x(Std(x))$ does hold in I, then we have *effectively* found the formulas which make I strongly arithmetic, and the hypotheses of Lemma 4 are satisfied. Then for every pair of first-order formulas U, V and every program $P \in \mathbf{P}$, M_I constructs the formula $PC_{U,P,V}$:

$$\forall x,y(U(x) \wedge A_P'(x,y) \Rightarrow V(y))$$

By consulting the oracle for Th(I), M_I can tell if this formula is true in I. If so, M_I outputs $U\{P\}V$; otherwise it outputs $\neg U\{P\}V$.

From Lemma 4, it follows immediately that M_I is sound. And if I is strongly arithmetic, M_I will eventually find first-order formulas Z', S', L', A', M', and H' which satisfy all the conditions, and hence will also be complete. (Here we are using the fact that the formula H

is definable in strongly arithmetic domains. The construction is straightforward but technical, using encoding of sequences much as in the proof of Lemma 4, and is omitted here.)

For total correctness assertions, M_2 proceeds just as M_1, but instead of using $PC_{U,P,V}$, it uses $T_{U,P,V}$:

$$\forall x \exists y (U(x) \Rightarrow A_P'(x,y) \wedge V(y)) \quad \blacksquare$$

Note that in constructing M_1 and M_2 we did not need the full strength of the assumption that I is strongly arithmetic. We could have weakened it to "I is weakly arithmetic and there is a formula H which satisfies (Enc)". In this case, we would also have to guess a formula $N(x)$ for natural number, and formula $E(x,y)$ for equivalence. AX1-9 would also have to be appropriately modified to restrict everything to N. For example, AX2 would read:

$$N(x) \wedge N(y) \wedge N(z) \Rightarrow [S(x,y) \wedge S(x,z) \Rightarrow E(y,z)]$$

We also would also have to include axioms to check that E is an equivalence relation, and that N, S, and Z interact correctly. Thus we would also have to check that the following two formulas held in I:

$$E(x,x) \wedge (E(x,y) \Rightarrow E(y,x)) \wedge (E(x,y) \wedge E(y,z) \Rightarrow E(x,z)),$$
$$(Z(x) \Rightarrow N(x)) \wedge ((N(x) \wedge S(x,y)) \Rightarrow N(y)).$$

With these new hypotheses, we could prove slight variants of Lemmas 2, 3, and 4 which would suffice to prove our theorem. We omit details here.

3.2.2. Construction of M_3, M_4, and M_5

We extend the techniques of [Li77] to the first-order case.

Given an interpretation I, an integer M, a program $P \in \mathbf{P}$ with $var(P) = x = \langle x_{i_1},\ldots,x_{i_k}\rangle$, and $u = \langle u_1,\ldots,u_k\rangle \in dom(I)^k$, we make the following definitions:

1. $U_M(x) =$ {terms of depth $\leq M$ over $\{f,g,a,b,x\}$}.

2. $I_M(u)$ = {values obtained by substituting u_j for x_{i_j} in the terms of $U_M(x)$}.

3. K_M = {K| K is an interpretation of type Σ, dom(K) has size $\leq N$ where $N=1+|U_M(x)|$, and there is a distinguished element $\lambda \in$ dom(K)}

4. P_M is the program which acts just like P except that on input u it halts at any valuation σ such that $\sigma(y) \notin I_M(u)$ for some $y \in cv(P)$. P_M is just

run P **until** $\neg[\wedge_{y\in cv(P)}(\vee_{t\in U_M(x)} y=t)]$.

If $y \in var(P)$, $\tau = (\sigma_0,\sigma_1,...) \in \mathcal{T}_I(P)$, and $\sigma_n(y)$ is the k^{th} distinct valuation in τ, then it is straightforward to show using condition 1 on acceptable programming languages and induction on k that $\sigma_n(y) \in I_k(\sigma_0(x))$. From this observation we get

Lemma 5: (Lipton [Li77]) If (†) holds in I, then there exists an M such that for all $y \in$ var(P), all $\tau \in \mathcal{T}_I(P)$, and all n, we have $\sigma_n(y) \in I_M(\sigma_0(x))$.

We say that I is *isomorphic to* $\langle K,c\rangle$ *on* $I_M(u)$ (where $K \in K_M$ and $c \in dom(K)^k$) iff there exists a bijection ψ: $I_M(u) \rightarrow dom(K) - \{\lambda\}$ such that

1. $\psi(u_i) = c_i$, for i = 1,...,k.

2. $I \models A_0(t_1,t_2)$ for $t_1, t_2 \in I_M(u)$ iff $K \models A_0(\psi(t_1),\psi(t_2))$.

3. If $t_1 \in I_M(u)$ and $f(t_1) \notin I_M(u)$, then $K \models f(\psi(t_1))=\lambda$. Similarly for g.

4. If $t_1, f(t_1) \in I_M(u)$, then $K \models f(\psi(t_1)) = \psi(f(t_1))$. Similarly for g.

Note that there are only finitely many non-isomorphic pairs $\langle K,c\rangle$ for a given M. Moreover, for each such pair we can find a first-order formula $\Delta_{\langle K,c\rangle}(x)$ such that

$I \models \Delta_{\langle K,c\rangle}(u)$ iff I is isomorphic to $\langle K,c\rangle$ on $I_M(u)$

Call a pair $\langle K,c\rangle$ *diverging* if P_M diverges when run in interpretation K on input *c*. Call a pair *cleanly halting* if P_M halts with output *d* when run in interpretation K on input *c*, and no $d_i=\lambda$. Let $x_{\langle K,c\rangle}$ be the term in $U_M(x)$ corresponding to *d* (i.e. when we substitute *c* in

for x in $x_{\langle K,c\rangle}$ we get d).

It is easy to check that if $\langle K,c\rangle$ is diverging and $I \models \Delta_{\langle K,c\rangle}(u)$, then P diverges in I on input u. If $\langle K,c\rangle$ is cleanly halting and $I \models \Delta_{\langle K,c\rangle}(u)$ then $I \models A_P(u,u_{\langle K,c\rangle})$ (where $u_{\langle K,c\rangle}$ is the result of substituting u for x in the term $x_{\langle K,c\rangle}$). Thus we define two first-order sentences, the first of which says U{P}V is true, while the second says U{P}V is false:

$$PC'_{M,U,P,V}: \forall x[U(x) \Rightarrow (\vee_{\langle K,c\rangle \text{ diverging}} \Delta_{\langle K,c\rangle}(x) \vee \vee_{\langle K,c\rangle \text{ cleanly halting}} (\Delta_{\langle K,c\rangle}(x) \wedge V(x_{\langle K,c\rangle})))]$$

$$FPC'_{M,P,U,V}: \exists x[U(x) \wedge \vee_{\langle K,c\rangle \text{ cleanly halting}} (\Delta_{\langle K,c\rangle}(x) \wedge \neg V(x_{\langle K,c\rangle}))]$$

M_3 proceeds as follows. For each M, U, P, and V, it constructs the sentences $PC'_{M,U,P,V}$ and $FPC'_{M,P,U,V}$. This can be done effectively. By assumption the halting problem is decidable for finite interpretations so we can effectively find all the diverging pairs $\langle K,c\rangle$. We remark that we are using the fact that proagrams are deterministic here: a program either diverges or halts on a given input (but cf. Remark 3.2.4). (Note we do *not* need the halting problem to be decidable to recursively enumerate the cleanly halting pairs. By condition 2 of acceptable programming language we can simply simulate P_M on input c in interpretation K simultaneously for each pair $\langle K,c\rangle$. Eventually we will find all the cleanly halting pairs, although we will not know *when* we have found all of them.) If (by consulting its oracle for Th(I)) M_3 discovers that $PC'_{M,U,P,V}$ (resp. $FPC'_{M,P,U,V}$) holds in I for any M, it outputs U{P}V (resp. $\neg$U{P}V). The procedure is sound by the comments above, and complete if (†) holds for I by Lemma 5.

M_4 is identical to M_3 but replaces $PC'_{M,U,P,V}$ and $FPC'_{M,P,U,V}$ by

$$T'_{M,U,P,V}: \forall x[U(x) \Rightarrow \vee_{\langle K,c\rangle \text{ cleanly halting}} (\Delta_{\langle K,c\rangle}(x) \wedge V(x_{\langle K,c\rangle}))]$$

$$FT'_{M,U,P,V}: \exists x[U(x) \wedge (\vee_{\langle K,c\rangle \text{ diverging}} \Delta_{\langle K,c\rangle}(x) \vee \vee_{\langle K,c\rangle \text{ cleanly halting}} (\Delta_{\langle K,c\rangle}(x) \wedge \neg V(x_{\langle K,c\rangle})))]$$

Finally, for M_5, note that we do not need the assumption that the halting problem is decidable for finite interpretations to compute $T'_{M,U,P,V}$, since we only need the cleanly halting pairs $\langle K,c\rangle$ and not the diverging pairs. Thus M_5 starts simulating P_M on input c in

interpretation K simultaneously for each pair <K,*c*>. Every so often it discovers that another pair <K,*c*> is cleanly halting. Let J be those pairs which it has so far discovered to be cleanly halting. M_5 checks if

$$I \models \forall x[U(x) \Rightarrow \vee_{<K,c>\in J}(\Delta_{<K,c>}(x) \wedge V(x_{<K,c>}))]$$

If so, it outputs U<P>V. By the same arguments as above M_5 is sound, and it is complete if (†) holds in I. Note that we cannot effectively find all the pairs <K,*c*> which are diverging, but we do not need them to enumerate the true termination assertions.

3.2.3. Proof of Lemma 1

Assume that (†) does not hold for I. Then there is some program $P \in \mathbf{P}$ which has no bound on the number of distinct valuations it goes through in any computation; i.e. for all M there exists $\tau \in \mathcal{T}_I(P)$, $\tau = (\sigma_0, \sigma_1, ...)$ such that $\{\sigma_i | \sigma_i \in \tau\}$ has at least M distinct elements. We show how to define programs whose weakest preconditions (the B_P of Definition 2.4) define the formulas necessary to make I arithmetic. Our initial steps are much like those of Lipton. We use his technique for representing integers in I and show how to write programs that perform arithmetic operations on this notion of integer. However, we go much further than Lipton in that we use these primitive programs to write more complicated programs, and ultimately to construct a program which translates the encoding of a Herbrand term into its corresponding value.

The programming details are themselves interesting. It turns out that under this representation of integers we can compute a predecessor function, but no successor function. But we can compute a *bounded* successor function, and that is sufficient for our needs.

In the constructions below, we assume for ease of exposition that $\mathbf{P} = \mathbf{P}''$, so that programs like **after each step of** P **do** Q really are in **P**. In general, of course, we would have to replace the programs below by the programs in **P** which simulate them.

Suppose var(P) = x. As a notational convenience we will write P(x) to indicate this.

We then use P(x') to denote the result of substituting x for x' in P. Occasionally when we write R(y) for some program R, we may omit some of the variables in var(R) which are just local variables; thus in general y will just consist of those variables on whose value the program R depends.

We first construct a program Q(x) such that if we run Q(x) on any input, x takes on the same values as when we run P(x) on the same input, but without repetition; i.e. if $\tau = (\sigma_0, \sigma_1, ...) \in \mathcal{T}_I(P)$ and $\tau' = (\sigma_0', \sigma_1', ...) \in \mathcal{T}_I(Q)$ with $\sigma_0 = \sigma_0'$ then $\{\sigma_i'(x) \mid i \geq 0\} = \{\sigma_i(x) \mid i \geq 0\}$ and if $\sigma_i'(x) = \sigma_j'(x)$ for $i<j$, then $\sigma_k'(x) = \sigma_i'(x)$ for all k, $i \leq k \leq j$. Essentially this is done by keeping track of the initial and current values of x, and then running a copy P with input the initial value and looking for the next new value it reaches after the current value (see [Li77] for more details). The code for Q(x) is given in Figure 3-1.

```
begin local init, x', y;
    init := x;
    x' := x;
    after each step of P(x') do R(x,x',y,init);
end
```

where R(x,x',y,*init*) is the program

```
if x ≠ x' then begin
    y := init;
    run P(y) until (y = x' ∨ y = x);
    if y = x then x := x';
end
```

Figure 3-1: The program Q(x).

The pair $u = (u_1,u_2)$ will represent the integer k iff u_2 is the k^{th} distinct value reached by Q on input u_1. We write $[u] = k$ to indicate that the pair $u = (u_1,u_2)$ represents k.

Choose two Herbrand terms tt and ff which get distinct values in I, to represent *true* and *false* respectively. Then using Q it is straightforward to write programs which meet the

following specifications.

(1) CHECKINT(x): halts with x unchanged if x represents an integer; otherwise CHECKINT will diverge.

(2) EQ(x,y,ans): if x and y do not both represent integers, EQ will diverge. Otherwise EQ will terminate with x, y unchanged and

ans = tt if $[x] = [y]$
ff otherwise.

(3) LESS(x,y,ans): if x and y do not both represent integers, LESS will diverge. Otherwise LESS will terminate with x, y unchanged and

ans = tt if $[x] < [y]$
ff otherwise

(4) NUM_k(x,ans): if x does not correspond to the integer k, NUM_k will diverge. Otherwise, NUM_k will terminate with x unchanged and

ans = tt if $[x] = k$
ff otherwise

The idea for computing EQ(x,y,ans) is to compute the successive values reached by Q starting from x_1 and y_1 and check that we reach x_2 and y_2 at the same time. (Recall that we assume x is of the form x_1, x_2 and likewise y.) We give the code in Figure 3-2; the codes for CHECKINT, LESS, and NUM_k are similar and will not be given.

In more detail, the program works as follows. The initial calls to CHECKINT check that x and y are integers, and diverge otherwise. The while loop then uses ONEMORESTEP(x,y,z) to compute the successive values reached by Q; z is the next value reached by Q after it reaches y when started on x. We get ONEMORESTEPQ(x,y,z) by running Q starting from x until we reach y. At this point Q*(y,z,flag) sets flag to tt; we then continue running Q for one more step.

```
CHECKINT(x);
CHECKINT(y);
begin local u, v, u', v';
    u := x_1;
    v := y_1;
    while u ≠ x_2 ∨ v ≠ y_2 do begin
        u' := u;
        v' := v;
        ONEMORESTEPQ(x_1,u',u);
        ONEMORESTEPQ(y_1,v',v);
    end;
    if u = x_2 ∧ v = y_2 then ans := tt else ans := ff;
end
```

ONEMORESTEPQ(x,y,z) computes z such that [x,z] = [x,y] + 1:

```
begin local flag;
    flag := ff;
    z := x;
    run Q*(y,z,flag) until (flag = tt ∧ y ≠ z);
end
```

where Q*(y,z,flag) is **after each step of** Q(z) **do** (**if** $z = y$ **then** flag := tt).

Figure 3-2: The program EQ(x,y,ans).

In general, it does not seem possible to construct a program SUC(x,y) which will compute a y such that [y] = [x] + 1. If [x] = k, it may be the case that only k distinct elements of dom(I) are reachable from x_1 by the program Q. The program ONEMORESTEPQ above only worked because at the point when it was called we were guaranteed that a "next" element existed. However, it is possible to generalize this idea and construct a "bounded" successor program, as well as the bounded addition and multiplication programs described below.

(4) SUC(b,x,y,ofl): if b, x, and y do not all initially represent integers, SUC will diverge. Otherwise SUC will terminate with b, x unchanged and

$[y] = [x] + 1$, ofl = ff if $[x] < [b]$
ofl = tt if $[b] \leq [x]$

(6) ADD(b,x,y,z,ofl): if b, x, y do not all initially represent integers, ADD will diverge. Otherwise, ADD will terminate with b, x, y unchanged and

$[z] = [x] + [y]$, ofl = ff if $[x] + [y] \leq [b]$;
ofl = tt if $[b] < [x] + [y]$

(7) MULT(b,x,y,z,ofl): similar to (6) above except that

$[z] = [x] \times [y]$, ofl = ff if $[x] \times [y] \leq [b]$
ofl = tt if $[b] < [x] \times [y]$.

The code for SUC(b,x,y,ofl) is given in Figure 3-3. The idea is to initialize y to b and then increase y (using ONEMORESTEPQ) until $x < y$. The code for ADD and MULT is straightforward to write using SUC and is omitted here. It is, however, important to ensure that no intermediate integer value ever exceed the value determined by b.

```
begin local ans, y';
    LESS(x,b,ans);
    if ans = ff then ofl := tt else begin
        y1 := b1;  /*recall y = (y1,y2)*/
        y2 := b1;
        while ans = ff do begin
            y' := y2;
            ONEMORESTEPQ(y1,y',y2);
            LESS(x,y,ans);
        end;
    end;
end
```

Figure 3-3: The program SUC(b,x,y,ofl).

By slightly modifying the programs written above so that they compute predicates instead of functions (e.g. we would modify ADD so that it halts on input x,y,z iff $[z] = [x] + [y]$) and taking weakest preconditions we could already define formulas N, Z,

E, S, L, A, and M which satisfy Definition 2.6. We note that none of the above programs required recursive calls. Thus it follows that if (†) does not hold, **P** is an acceptable programming language (but not necessarily acceptable with recursion), and I is expressive-Herbrand or expressive-effective with respect to **P**, then I is weakly arithmetic. This is exactly Lipton's result. But we require more; we need a formula H which satisfies the axiom (Enc).

We get H by using the programs defined above to construct a program HRBD which relates the encoding of a Herbrand term as an integer to its corresponding value. We use the encoding of Herbrand terms described in 3.2.1. The formal specification for HRBD is given below.

(8) HRBD(x,enc,d): if x does not represent an integer, HRBD will fail to terminate. Otherwise, HRBD will terminate with x unchanged and

enc = tt, d = h (in I)	if $[x]$ encodes Herbrand term h,
enc = ff	if $[x]$ does not encode a Herbrand term

Thus, for example, if $[x] = \ulcorner f(a) \urcorner$ ($= \langle 2,0 \rangle = 5$), then after the execution of HRBD(x,enc,d), we will have enc = tt and d = f(a).

Note that a true pairing function cannot be programmed using the above techniques. Given only x and y, it is not in general possible to compute z with $[z] = \langle [u],[y] \rangle$, since the value to be computed will be larger than both of the input values. The corresponding projection function, on the other hand, is relatively easy to compute and is sufficient for programming HRBD. Thus we need a program PR which satisfies

(9) PR(z,x,y): if z does not represent an integer, then PR diverges. Otherwise, PR will terminate with the final value of z unchanged and the final values of x and y will satisfy the relationship

$$[z] = \tfrac{1}{2}([x] + [y])([x] + [y] + 1) + [x].$$

The program for PR simply tests all $[x]$, $[y] \leq [z]$ until it finds $[x]$, $[y]$ which satisfy this relationship. It uses the identity $1+2+...+(n+m) = \frac{1}{2}(n+m)(n+m+1)$ to ensure that no intermediate value for the right choice $[x]$ and $[y]$ exceeds the initial value of $[z]$. The code for PR will not be given; the code for HRBD is given in Figure 3-4.

A straightforward modification of HRBD(x,enc,d) gives us HRBD'(x,d) which halts iff d is equal to the Herbrand term encoded by x. Now, by taking weakest preconditions, we can already show that I is weakly arithmetic and has a formula H which satisfies Enc. As we remarked in 3.2.1, this would already be enough to enable us to define the procedures M_1 and M_2 and prove our main theorem. However, with a little more work, we can show that I is strongly arithmetic.

List the terms in the Herbrand universe (of {a,b,f,g}) in order of increasing encoding: a, b, f(a), f(b), g(a,a), ...

Using this list, we can define a bijection φ: dom(I) $\rightarrow \mathcal{N}$. For u $\in$ dom(I), φ(u) = m iff, if t is the first term on the list such that I $\models$ t = u, then m different values are taken on by the terms on the list before t. For example, suppose that in I we have a = f(a), but a, b, and f(b) are all distinct. Then φ(a) = 0, φ(b) = 1, φ(f(b)) = 2.

Since I is Herbrand definable (by assumption) and has an infinite domain (otherwise (†) would hold), φ is indeed a bijection.

We can also use this list and our old way of looking at tuples in dom(I) as integers, to define a new way of looking at tuples in dom(I) as integers. We want $[\![v]\!]$ = m iff, for some u, φ(u) = m and $[v]$ encodes u. We use the notation $[\![v]\!]$ to contrast with the $[v]$ used before.

Define $[\![v]\!]$ = m, if, for some k:

1. $[v]$ = k,

2. k is the encoding of some Herbrand term t (i.e. $\ulcorner t \urcorner$ = k),

```
μH[begin
    local init, ans, fid, arg, arg', arg'', d';
    init := x;
    enc := tt;
    NUM⌜a⌝(x,ans);
    if ans = tt then d := a
    else begin
        NUM⌜b⌝(x,ans);
        if ans = tt then d := b
        else begin
            PR(x,fid,arg);
            NUM⌜f⌝(fid, ans);
            if ans = tt then begin
                x := arg;
                H;
                d := f(d);
                x := init;
            end;
            else begin
                NUM⌜g⌝(fid,ans);
                if ans = tt then begin
                    PR(arg, arg', arg'');
                    x := arg';
                    H;
                    d' := d;
                    if enc = tt then begin
                        x := arg'';
                        H;
                        d := g(d', d);
                    end;
                    x := init;
                end;
                else enc := ff;
            end;
        end;
    end;
end]
```

Figure 3-4: The program HRBD(x,enc,d).

3. there is no term t' with $\ulcorner t' \urcorner < k$ such that $I \models t' = t$,

4. the Herbrand terms t' with $\ulcorner t' \urcorner < k$ take on m distinct values in I.

If the conditions above do not hold, then $[\![v]\!]$ is undefined.

Consider the example give above where we have a = f(a), but a, b, and f(b) are all distinct. It is easy to check that $\ulcorner a \urcorner = 0$, $\ulcorner b \urcorner = 1$, $\ulcorner f(a) \urcorner = 5$, $\ulcorner f(b) \urcorner = 8$. Thus, if $[v_0] = 0$, $[v_1] = 1$, $[v_2] = 2$, $[v_3] = 5$, $[v_4] = 8$, then $[\![v_0]\!] = 0$, $[\![v_1]\!] = 1$, $[\![v_2]\!]$ is undefined, $[\![v_3]\!]$ is undefined (since there is a Herbrand term, namely a, with $\ulcorner a \urcorner < \ulcorner f(a) \urcorner$ but $I \models a = f(a)$ by assumption), and $[\![v_4]\!] = 2$.

We would like to define programs SUC', ADD', and MULT' which meet the following specifications:

1. SUC'(v_1, v_2) halts iff $[\![v_1]\!] + 1 = [\![v_2]\!]$.

2. ADD'(v_1, v_2, v_3) halts iff $[\![v_1]\!] + [\![v_2]\!] = [\![v_3]\!]$.

3. MULT'(v_1, v_2, v_3) halts iff $[\![v_1]\!] \times [\![v_2]\!] = [\![v_3]\!]$.

Suppose we could define these programs. Note that indeed $\varphi(u) = m$ iff for some v, $[v]$ encodes a Herbrand term equal to u, and $[\![v]\!] = m$. Thus it follows that $\varphi(u_1) + \varphi(u_2) = \varphi(u_3)$ iff there exists v_1, v_2, and v_3 such that $[v_i]$ encodes u_i for i = 1,2,3 and $[\![v_1]\!] + [\![v_2]\!] = [\![v_3]\!]$. Similar statements hold for successor and multiplication. Thus we can define Z(x) via x = a (since $\varphi(a) = 0$) and formulas S, A, and M which make I strongly arithmetic by using weakest preconditions. For example, if $W_{ADD'}$ and $W_{HRBD'}$ are the weakest preconditions with respect to *true* of ADD' and HRBD' respectively, then we define A as

$$A(u_1,u_2,u_3) \text{ iff } \exists v_1,v_2,v_3\,(\wedge_{i=1,2,3} W_{HRBD'}(v_i,u_i) \wedge W_{ADD'}(v_1,v_2,v_3))$$

Similar definitions can be made for S and M, and thus I is strongly arithmetic.

All that remains is to show how to define SUC', ADD', and MULT'. In order to do this,

we need a program Q'(x) such that on input v_0, if x successively takes values v_0, v_1, v_2, ..., then $[\![v_0,v_i]\!]$ = i. If we look at the definition of $[\![\ ]\!]$, we see that this means that x goes through the encodings of Herbrand terms without repeating values. Thus, the definition of Q' bears much the same relationship to Q as Q does to P. We give the code for Q'(x) in Figure 3-5.

```
begin local init, x';
    init := x;
    x' := x;
    after each step of Q(x') do R'(x,x',init);
end
```

where R'(x,x',*init*) is

```
begin local y1,y2,y',enc,enc',d,d',e,e';
    if x ≠ x' then begin
        /*check if x' encodes a Herbrand term*/
        HRBD(x',enc',d');
        if enc = tt ∧ d ≠ d' then begin
            HRBD(x,enc,d);
            y1 := init;
            y2 := init;
            e := a;
            /*check if d', the Herbrand term encoded by x', has
            occurred previously in the list of Herbrand terms*/
            while e ≠ d ∨ e ≠ d' do begin
                y' := y2;
                ONEMORESTEPQ(y1,y',y2);
                HRBD(y,enc,e');
                if enc = tt then e := e';
            end;
            if y = x then x := x';
        end;
    end;
end
```

Figure 3-5: The program Q'(x)

Now by simply replacing the Q in SUC, ADD, and MULT above by Q', and making

very minor modifications, we can easily define SUC', ADD', and MULT'.

This completes the proof of Lemma 1, and with it the proof of Theorem 1. ∎

3.2.4. Remarks

We have used two of our hypotheses on P -- that P is deterministic and that **P** allows recursive procedure calls -- in a weak way. In particular, note that as long as Lemma 1 holds, the construction of M_1, M_2, and M_5 is unaffected even if **P** has nondeterministic programs. For M_3 and M_4 to work in the presence of nondeterministic programs, we need to strengthen the hypothesis that "**P** has a decidable halting problem for finite interpretations" to "**P** has a decidable input-output relation for finite interpretations"; i.e. if I is finite, then for all P ∈ **P**, we can decide for which $u, v \in$ dom(I) we have $I \models A_P(u,v)$. Note that the two hypotheses are equivalent if **P** is deterministic.

It is only in the proof of Lemma 1 that we really needed determinism, because we needed to know that if (†) does not hold, then there is a *deterministic* program P which goes through unboundedly many valuations on any given input. But once the presence of *one* such program is guaranteed, the programming language could certainly have other nondeterministic programs.

Similarly, the only place in which we used recursive calls was in the construction of the program HRBD of the previous section, which in turn was necessary to show that I was strongly arithmetic. We could remove this condition by insisting, for example, that there be some program P ∈ **P** and some x ∈ var(P) such that if we run P on some input u, x takes on every value in dom(I). In particular, under our assumption that I is Herbrand definable, having a deterministic program which would generate all the Herbrand terms would be a sufficient condition to remove both of these hypotheses on **P**.

It is also worth noting that our decision procedures for partial correctness and termination also extend to decision procedures for the full first-order dynamic logic (cf.

[Pr76,Ha79]) of any acceptable programming language with recursion.

3.3. Proof of Theorem 2

By the comments made in the proof of Lemma 1, since I is expressive-effective, I is either weakly arithmetic or (†) holds. If (†) holds, then the procedures M_3, M_4, and M_5 defined above work perfectly well in this case too. If I is weakly arithmetic, we show below that given the formulas N, Z, S, A, M, and E which make I weakly arithmetic, we can use them to effectively check if a given partial correctness or termination assertion holds for I.

Since I is expressive-effective, and hence effectively presented, it follows that A_P defines an r.e. subset of $dom(I)^{2k} \subseteq \mathcal{N}^{2k}$ (where $k = |var(P)|$). Thus by a well-known result of recursive function theory (cf. [Sh67]), given (the code for) P, we can effectively find a first-order formula A_P^* of number theory (i.e. over the type $\{0,+,\times,S\}$) such that $\mathcal{N} \models A_P^*(u,v)$ iff $I \models A_P(u,v)$. Similarly, we can effectively translate any formula U of type Σ to a formula U* of number theory such that $\mathcal{N} \models U^*(u,v)$ iff $I \models U(u,v)$. To see this, note that since all the functions and predicates of I are recursive by assumption, (and thus definable by a first order formula of number theory by the result in [Sh67] referred to above) we can easily translate any quantifier-free formula. Using the fact that dom(I) is recursive and thus also definable, we can restrict the quantificatiers to range over dom(I) in $\mathcal{N}$ and thereby translate any first-order formula. For example, a formula such as $\forall x T$ of type Σ would be translated to $\forall x(x \in dom(I) \Rightarrow T^*)$.

Now note that

$$I \models U\{P\}V \quad \text{iff} \quad \mathcal{N} \models \forall u,v(U^*(u) \wedge A_P^*(u,v) \Rightarrow V^*(v))$$
$$I \models U\langle P\rangle V \quad \text{iff} \quad \mathcal{N} \models \forall u(U^*(u) \Rightarrow \exists v(A_P^*(u,v) \wedge V^*(v)))$$

Thus we can effectively translate partial correctness and termination assertions to sentences of number theory. But by a straightforward syntactic translation using N, Z, S, A, M, and E, we can effectively translate *any* sentence B of number theory to a sentence B' of type Σ such that

$I \models B'$ iff $\mathcal{N} \models B$.

The translation is a generalization of that used in 3.2.1 above to translate AX1-9. By using the N predicate we can essentially view I as a standard model of arithmetic.

Thus, to decide if $I \models U\{P\}V$ (resp. $I \models U\langle P\rangle V$) we translate $U\{P\}V$ (resp. $U\langle P\rangle V$) to a sentence of first-order number theory as described above, translate this sentence back to a first-order sentence of type Σ, and then consult the oracle for Th(I) to see if this sentence is true. Note this procedure is not uniform in I. In contrast to Theorem 1, we have no effective way of finding the formulas N, Z, S, A, M, and E; all we know is that they exist. ∎

4. Conclusions and Open Problems

We believe that this paper raises a number of open questions of both technical and philosophical interest. Perhaps the most important technical question concerns to what extent the various hypotheses that we used in Theorem 1 can be eliminated or replaced by weaker conditions. In particular, the hypotheses that the programming language be deterministic and allow recursive calls do not appear to be essential (cf. 3.2.4), and we conjecture that our results can be extended to a wider class of languages. On the other hand, the assumption of Herbrand definability, or something like it, does appear to be necessary. (We shall return to this point below.) In any case, both Herbrand definability and effective presentability seem to be very natural conditions. The first limits the values of the domain to those which can be effectively described, while the second limits the interpretations to those which can be effectively described.

A second open question concerns the relationship between an axiomatization of the kind given by Floyd and Hoare (consisting of a finite number of axiom schemes), and a decision procedure of the sort provided by Theorems 1 and 2; i.e. a nondeterministic, recursive computation which checks the truth of certain formulas in the interpretation. One trivial observation is that *any* such computation can be axiomatized by a formal system in which formulas describe the state of the computation and the "rules of inference"

correspond to state transitions. Clearly such a formal axiom system would not compare with the familiar syntactic axiom systems in elegance or understandability.

On the other hand, there are many applications in which the form of the axiomatization is of secondary importance. For example, the quantifier-free theory of linear inequalities (i.e. the quantifier-free theory of the symbols $\{0,1,+,-,<,=\}$) has many applications in program verification. A first order axiom system is a useful representation of this theory for purposes of logical analysis, but for deciding the truth of formulas in an automatic theorem prover, a representation based on an efficient algorithm for deciding linear inequalities (such as the Simplex Algorithm) may be preferred.

Of course, it would be important and interesting to understand if and when the existence of a decision procedure such as we have given implies the existence of a more syntactic, "natural" axiomatization.

Recall that our proof procedure splits into two cases. In the first case, where our interpretations were essentially finite (or, more precisely, satisfied (†)), we simulated the program's behavior on finite interpretations. In the second case we used the power of expressiveness to enable us to find formulas which made the interpretation strongly arithmetic, and then made use of the encoding power of arithmetic. Our main use of arithmetic was in encoding the behavior of the program as a first-order formula (cf. Lemma 4). This was necessary in our model because of our abstract notion of acceptable programming language. For many concrete programming languages, we believe that an axiomatic description of the control features could replace this use of arithmetic, leaving us with a system that is more natural.

However, for sufficiently rich programming languages, some encoding seems to be necessary. The various assumptions that we and other researchers have made -- in our case that the interpretation is expressive and Herbrand definable or effectively presented; in the case of, say, [Ha], that is (weakly) arithmetic and that there is a definable predicate which

encodes finite sequences of domain elements as on element -- can all be viewed as ways of ensuring that the interpretation has sufficient encoding power to admit an axiomatization. Indeed, our results show that these hypotheses are equivalent in many cases, and the results of [Cl76/79] and [BCT82] suggest that they are often necessary. In contrast, the work of Cook and Gorelick [Co75/78, Go75] shows that encoding of control structures is unnecessary when reasoning about programming languages which include recursive procedures.

If a programming language requires an unnatural use of encoding in order to get an axiomatization, then it is perhaps too powerful to reason about effectively. The arguments given in Section 3 can also be used to show that a sound and relatively complete partial correctness proof system can be obtained for any acceptable programming language *provided* that we consider only unbounded expressive interpretations. This is precisely because of the encoding power available in such interpretations under the hypotheses of expressiveness and Herbrand definability. Although it is difficult to define the notion of "natural" axiomatization precisely, it seems that a natural axiomatization should not make use of such encoding. We believe the incompleteness results of [Cl76/79], which depend only on finite interpretations, show that certain programming language features cannot have natural axiomatizations. In fact, we would argue that finite interpretations are often more useful than infinite interpretations for judging whether an axiomatization is natural, since they preclude the possibility that domain elements can be used to encode complicated run-time data structures such as the run-time stack or linked lists of activation records. Moreover, all of the standard partial correctness rules (e.g. the assignment axiom, the **while** statement rule, etc.) work just as well for finite interpretations as for infinite ones.

Our results show that expressiveness is a very powerful hypothesis. Although expressiveness has been assumed by many previous researchers (cf. [Co75/78], [Cl76/79],...) to get a complete axiomatization, the use they have made of this assumption (e.g. to guarantee the existence of loop invariant) seems more natural than the way we have

used it here. This suggests that perhaps the hypothesis of expressiveness could be weakened or restricted in some way. We note that such a weakening would not affect the incompleteness results of [Cl76/79].

Yet a third question concerns the relationship between the uninterpreted case considered in [MH80] and [MM82], and the interpreted case discussed here. In [MH80] and [MM82], complete axiomatizations for termination assertions are given which do not require that the interpretations satisfy any restrictions such as expressiveness. However, the termination assertions provable in these systems are exactly those which are valid in all interpretations. Thus, in these systems we cannot prove that a given program terminates in a given interpretation, if its termination depends on facts about the interpretation. It is interesting to note that termination assertions were shown (in [MH80]) to be somewhat more tractable than partial correctness assertions in the uninterpreted case.

This leads us to our last point: the relationship between partial correctness and termination, and our ability to find good axiom systems for complicated programming languages. One conclusion we can draw is that under the assumption that the halting problem is decidable for finite interpretations, partial correctness and termination seem to have essentially the same complexity. However, for more complicated deterministic programming languages such as those discussed in [Cl76/79] which do not have a decidable halting problem for finite interpretations, termination assertions, and hence total correctness assertions, are effectively axiomatizable, while partial correctness assertions are not. This suggests the possibility of a total correctness proof system which, unlike most currently available, does not require the establishment of partial correctness as an essential first step.

Acknowledgment: We would like to thank an anonymous referee for his careful reading of the paper, and for finding some technical errors in the proof of Theorem 2.

Appendix 1

The semantics of **P'**:

Given a trajectory $\tau = (\sigma_0,...,\sigma_k)$, define $first(\tau) = \sigma_0$ and $last(\tau) = \sigma_k$; and for trajectories τ_0 and τ_1, define

$\tau_0 \circ \tau_1 = (\sigma_0,...,\sigma_k,\sigma_1',...)$ if $\tau_0 = (\sigma_0,...,\sigma_k)$, $\tau_1 = (\sigma_0',\sigma_1',...)$, and $\sigma_k = \sigma_0'$,
undefined, otherwise.

1. If t is a term, $cv(x := t) = \{x\}$; $var(x := t) = \{x\} \cup var(t)$;
   $\mathcal{T}_I(x := t) = \{(\sigma,\sigma[x/u]) | \sigma \neq \perp, \sigma(t) = u \in dom(I)\}$.

2. $cv(P;Q) = cv(P) \cup cv(Q)$; $var(P;Q) = var(P) \cup var(Q)$;
   $\mathcal{T}_I(P;Q) = \{\tau_0 \circ \tau_1 | \tau_0 \in \mathcal{T}_I(P), \tau_1 \in \mathcal{T}_I(Q)\}$

3. $cv($**if** A **then** P **else** Q$) = cv(P) \cup cv(Q)$;
   $var($**if** A **then** P **else** Q$) = var(P) \cup var(Q) \cup var(A)$;
   $\mathcal{T}_I($**if** A **then** P **else** Q$) =$
   $\{\tau | (I,first(\tau) \models A, \tau \in \mathcal{T}_I(P))$, or $(I,first(\tau) \models \neg A, \tau \in \mathcal{T}_I(Q))\}$

4. $cv($**while** A **do** P$) = cv(P)$; $var($**while** A **do** P$) = var(P) \cup var(A)$;
   $\mathcal{T}_I($**while** A **do** P$) = \cup_{i \geq 1} \mathcal{T}_I(W^i)$; where $W^0 = \omega$, $W^{i+1} =$ **if** A **then** P; W^i **else** SKIP. SKIP is the program which has no effect: $\mathcal{T}_I(SKIP) = \{(\sigma) | \sigma \neq \perp\}$, and ω is the diverging program: $\mathcal{T}_I(\omega) = \{(\sigma, \perp) | \sigma \neq \perp\}$;

5. $cv($**run** P **until** A$) = cv(P)$; $var($**run** P **until** A$) = var(P) \cup var(A)$;
   $\mathcal{T}_I($**run** P **until** A$) =$
   $\{\tau \in \mathcal{T}_I(P) | \tau = (\sigma_0,\sigma_1,...)$, and for all i, $I,\sigma_i \models \neg A\} \cup \{\tau | \tau = (\sigma_0,...,\sigma_k)$, τ is a prefix of some $\tau' \in \mathcal{T}_I(P)$, if $i < k$ then $I,\sigma_i \models \neg A$, and $\sigma_k = \perp$ or $I,\sigma_k \models A\}$.

6. $cv($**after each step of** P **do** Q$) = cv(P) \cup cv(Q)$;
   $var($**after each step of** P **do** Q$) = var(P) \cup var(Q)$;
   If $var(P) \cap cv(Q) \neq \varnothing$, then $\mathcal{T}_I($**after each step of** P **do** Q$) = \varnothing$. (We consider **after each step of** P **do** Q syntactically incorrect unless $var(P) \cap cv(Q) = \varnothing$; thus we do not allow the computation of Q to affect the variables of P.) If $var(P) \cap cv(Q) = \varnothing$, $\mathcal{T}_I($**after each step of** P **do** Q$) = \{\tau | \tau = (\sigma_0,\sigma_1,...)$ such that for some subsequence $\sigma_{i_0} < \sigma_{i_1} < ... < \sigma_{i_k}$ we have

a. $\sigma_0 = \sigma_{i_0}$

b. $\text{last}(\tau) = \sigma_{i_k}$

c. if $\sigma_{i_j+1} \neq \perp$, $(\sigma_{i_j+1},...,\sigma_{i_{j+1}}) \in \mathcal{T}_I(Q)$

d. for some $(\sigma_0',\sigma_1',...,\sigma_{k'}') \in \mathcal{T}_I(P)$, we have either $k=k'$ or ($k \leq k'$ and $\sigma_{i_k} = \perp$), and $\sigma_j'(\text{var}(P)) = \sigma_{i_j}(\text{var}(P))$ for all $j \leq k$.

7. cv(**begin local** $x_{i_1},...,x_{i_m}$; P **end**) = cv(P);
var(**begin local** $x_{i_1},...,x_{i_m}$; P **end**) = var(P);
$\mathcal{T}_I$(**begin local** $x_{i_1},...,x_{i_m}$; P **end**) = $\{(\sigma_0,\sigma_1)\circ\tau\circ(\text{last}(\tau),\sigma_2) | \sigma_1 = \sigma_0[x_{i_1}/a,...,x_{i_m}/a], \tau \in \mathcal{T}_I(P)$, and $\sigma_2 = \text{last}(\tau)[x_{i_1}/\sigma_0(x_{i_1}),...,x_{i_m}/\sigma_0(x_{i_m})]\}$. (Thus the local variables $x_{i_1},...,x_{i_m}$ are set to the constant value a when the block is entered, and reset to their previous values when the block is exited.)

Note that the programs in **P'** still satisfy constraints 1 and 2 above.

The semantics of **P"**:

1. cv(Z) = var(Z) = $\varnothing$ for all Z $\in$ plab;
$\mathcal{T}_I(Z) = \mathcal{T}_I(\omega) = \{(\sigma, \perp) | \sigma \neq \perp\}$ for all Z $\in$ plab.

2. cv(μZ[P]) = var(μZ[P]) = var(P);
$\mathcal{T}_I(\mu Z[P]) = \cup_{i \geq 0} \mathcal{T}_I(P^i)$, where $P^0 = \omega$, and $P^{i+1} = P[Z/P^i]$ (i.e. we syntactically replace all *free* occurrences (where free and bound occurrence have the familiar meaning) of Z in P by P^i). Essentially, μZ[P] acts as a least fixed point operator. Note that $\mathcal{T}_I$(**while** A **do** P **od**) = $\mathcal{T}_I(\mu Z$[**if** A **then** P;Z **else** SKIP])

Appendix 2

Proof of Lemma 4: Because I is strongly arithmetic, we may use the formulas Z', S', A' and M' to treat the elements of dom(I) as integers. Thus we will be able to encode formulas of type Σ and sequences of domain elements by one domain element. The formula $A_P'(u,v)$ will be of the form

$$\exists c,t\,[\mathrm{Traj}_P(c,t) \wedge \mathrm{Trseq}(c,u) \wedge \mathrm{Term}(u,t_1,v_1) \wedge \ldots \wedge \mathrm{Term}(u,t_n,v_n)]$$

where Traj_P, Trseq, and Term are formulas of type Σ. Our intention is that $\mathrm{Traj}_P(c,t)$ will be true in I iff P has a trajectory such that t encodes terms for the values of variables in the final valuation under the assumption that c encodes a set of literal formulas true in I and the initial valuation (cf. part 2 of the definition of acceptable programming language). $\mathrm{Trseq}(c,u)$ will be true iff c encodes a finite set of literal formulas true in I and the valuation defined by $x_i = u_i$. Henceforth, we will simply call this the valuation u. $\mathrm{Term}(u,t_i,v_i)$ will be true iff the value of the term encoded by t_i in I and the valuation u is v_i. By these comments and the assumptions that P is an acceptable programming language and that Z', S', A', and M' make I strongly arithmetic, one can see that $A_P'(u,v)$ will define the relation $A_P(u,v)$, provided we can construct the necessary subformulas.

By part 2 of the definition of an acceptable language, if we are given a code for a program P, we may assume that we can effectively find a recursive relation which represents the trajectories of P. Thus if we have a set of formulas of type Σ which makes I strongly arithmetic, we can effectively construct the formula $\mathrm{Traj}_P(c,t)$ of type Σ.

For the definition of Term and Trseq, we have to discuss some techniques for encoding formulas of Σ as integers. In a strongly arithmetic domain, we can represent finite sequences of integers by using the predicates Length(s,x), which holds iff s represents a sequence of length x, and Select(s,i,y), which holds iff y is the i^{th} element of the sequence represented by s. (Further details are left as an exercise to the interested reader.) For ease of notation, we introduce the terms |x| to stand for the length of x, and x[i] to stand for the

i^{th} element of x (provided that x is a sequence with at least i elements). As above, we will write $\langle u,v\rangle$ for the value which encodes a pair of elements u and v; we also use $\langle u,v,w\rangle$ as an abbreviation for $\langle u,\langle v,w\rangle\rangle$. It is clear that formulas with these terms can be rewritten over Σ by the addition of new bound variables. We will encode a term t as a sequence $n_1,...,n_q$, such that each element n_k is either an integer $\ulcorner a \urcorner$ or $\ulcorner b \urcorner$ corresponding to one of the constant symbols of Σ, an encoding of a pair $\langle \ulcorner var \urcorner, i\rangle$ where $\ulcorner var \urcorner = 4$, corresponding to the variable x_i in var(P), a pair $\langle \ulcorner f \urcorner, i\rangle$ $i<k$, or a triple $\langle \ulcorner g \urcorner, i,j\rangle$ $i,j<k$. In the case that n_k is a pair $\langle \ulcorner f \urcorner, i\rangle$, the subsequence $n_1,...,n_k$ encodes the term $f(t_i)$ where t_i is the term encoded by $n_1,...,n_i$; similarly for g. Thus the complete sequence $n_1,...,n_q$ encodes the term t.

With these conventions, we can define the formula Term(u,t,y), which checks whether y is equal to the value of the term encoded by t in valuation u. The idea is to check whether there is a sequence w ending with y such that for all $i \leq |w|$, w[i] is equal to the value of the term encoded by the subsequence t[1],..,t[i]. (Note this is not the same encoding of terms that we used in Lemmas 1-3; this lemma does not depend on I being Herbrand definable.)

$$
\begin{aligned}
&\text{Term}(u,t,y) \equiv \\
&\quad \exists w(|w| = |t| \wedge \forall i(i \leq |w| \Rightarrow \\
&\qquad ((t[i] = \ulcorner a \urcorner \wedge w[i] = a) \\
&\qquad \vee (t[i] = \ulcorner b \urcorner \wedge w[i] = b) \\
&\qquad \vee (t[i] = \langle \ulcorner var \urcorner, 1\rangle \wedge w[i] = u_1) \\
&\qquad \vee \ ... \\
&\qquad \vee (t[i] = \langle \ulcorner var \urcorner, n\rangle \wedge w[i] = u_n) \\
&\qquad \vee \exists j\ (j<i \wedge t[i] = \langle \ulcorner f \urcorner, j\rangle \wedge w[i] = f(w[j])) \\
&\qquad \vee \exists j,k\ (j<i \wedge k<i \wedge t[i] = \langle \ulcorner g \urcorner, j,k\rangle \wedge w[i] = g(w[j],w[k])))) \\
&\quad \wedge w[|w|] = y)
\end{aligned}
$$

Trseq(c,u) must check that every element of the sequence c of literal formulas is true in I and u. Thus, we define Trseq in terms of Trlit(lit,u), where lit is the encoding of a single literal:

$$\text{Trseq}(c,u) \equiv \forall\ i\ (i \leq |c| \Rightarrow \text{Trlit}(c[i],u))$$

Trlit uses Term to find the values of the terms appearing in the (encoding of the) literal

formula, and then checks the truth value of the appropriate predicate symbol. In order to do the encoding, we first assign an integer to each predicate symbol appearing in Σ and to its negation: $\ulcorner = \urcorner = 0$, $\ulcorner \neg = \urcorner = 1$, $\ulcorner A_0 \urcorner = 3$, $\ulcorner \neg A_0 \urcorner = 4$. Since both A_0 and $=$ are binary predicates, we can encode a literal formula by a triple representing the code for the predicate symbol (which will also indicate whether or not it is negated), and the encodings of the two terms which are its arguments. For example, the literal $\neg(a = g(a,b))$ is encoded as $\langle \ulcorner \neg = \urcorner, m_a, m_{g(a,b)} \rangle$, where m_a and $m_{g(a,b)}$ are encodings of a and $g(a,b)$ as described above; i.e. m_a is an integer representing the sequence $(\ulcorner a \urcorner)$ and $m_{g(a,b)}$ represents the sequence $(\ulcorner a \urcorner, \ulcorner b \urcorner, \langle \ulcorner g \urcorner, 1, 2 \rangle)$. Thus

$$
\begin{aligned}
&\mathrm{Trlit}(\mathrm{lit},u) \equiv \\
&\quad \exists v_1,v_2,w_1,w_2\, (\mathrm{Term}(u,v_1,w_1) \wedge \mathrm{Term}(u,v_2,w_2) \\
&\qquad \wedge ((\mathrm{lit} = \langle \ulcorner = \urcorner, v_1, v_2 \rangle \wedge w_1 = w_2) \\
&\qquad\quad \vee (\mathrm{lit} = \langle \ulcorner \neg = \urcorner, v_1, v_2 \rangle \wedge \neg(w_1 = w_2)) \\
&\qquad\quad \vee (\mathrm{lit} = \langle \ulcorner A_0 \urcorner, v_1, v_2 \rangle \wedge A_0(w_1,w_2)) \\
&\qquad\quad \vee (\mathrm{lit} = \langle \ulcorner \neg A_0 \urcorner, v_1, v_2 \rangle \wedge \neg A_0(w_1,w_2))))
\end{aligned}
$$

This completes our encoding. ∎

References

[BCT82] J. Bergstra, A. Chmielinska, J. Tiuryn, Hoare's logic is incomplete when it does not have to be, in "IBM Workshop on Logics of Programs" (ed. D. Kozen), Lecture Notes in Computer Science, 131, Springer-Verlag, N.Y., 1982, pp. 9-23.

[Cl76/79] E. M. Clarke, Programming language constructs for which it is impossible to obtain good Hoare axiom systems, *JACM* 26:1, January, 1979, pp. 129-147. Ph.D. Thesis, Cornell, 1976.

[CGH82] E. M. Clarke, S. M. German, and J. Y. Halpern, On effective axiomatizations of Hoare Logics, in "Proceedings of the Ninth Annual ACM Symposium on Principles of Programming", 1982, pp. 309-321.

[Co78] S. A. Cook, Soundness and completeness of an axiom system for program verification, *SIAM Journal on Computing* 7:1, pp. 70-90, February, 1978.

[Di76] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[Go75] G. A. Gorelick, A complete axiom system for proving assertions about recursive and nonrecursive programs, University of Toronto TR-75, 1975.

[Ha79] D. Harel, *First-Order Dynamic Logic*, Lecture Notes in Computer Science, 68, Springer-Verlag, N.Y., 1979.

[Ho69] C. A. R. Hoare, An axiomatic approach to computer programming. *CACM* 12:10, October, 1969, pp. 322-329.

[La80] H. Langmaack, Proof of a theorem of Lipton on Hoare Logic and applications. Institut fur Informatik und Praktische Mathematik bericht 8003, June, 1980.

[LO80] H. Langmaack and E. R. Olderog. Present day Hoare-like systems for programming languages with procedures: power, limits, and most likely extensions, in "Proceedings of the 7th Conference on Automata, Languages, and Programming", Nordwijkerhout 1980, Eds: J. W. de Bakker, J. van Leeuwen, LNCS 25, pp. 363-373, June, 1980.

[Li77] R. J. Lipton, A necessary and sufficient condition for the existence of Hoare logics, in "Proceedings of the 18th IEEE Symposium on Foundations of Computer Science", pp. 1-6, October, 1977.

[Me78] A. R. Meyer, Notes on Lipton's generalization of the theorems of Cook and Clarke on expressiveness, privately circulated notes.

[MH80] A. R. Meyer and J. Y. Halpern, Axiomatic definitions of programming languages: a theoretical assessment, in "Proceedings of the 7th ACM Symposium on Principles of Programming Languages", pp. 202-212, January, 1980 (to appear in *JACM*).

[MM82] A. R. Meyer and J. C. Mitchell, Axiomatic definability and completeness for Recursive Programs, in "Proceedings of the 9th ACM Symposium on the Principles of Programming Languages", pp. 337-346, January, 1980.

[Pr76] V. R. Pratt, Semantical considerations of Floyd-Hoare logic, in "Proceedings of the 17th IEEE Symposium on Foundations of Computer Science", pp. 109-121, October, 1976.

[Sh67] J. R. Shoenfield, *Mathematical logic*, Addison Wesley, 1967.