On 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-like logics for both partial correctness and termination assertions. In fact, under mild assumptions on P and I, we show that the assertions true for P in I are uniformly decidable in the theory of I (Th(I)) iff the halting problem for P is decidable for finite interpretations. Moreover termination assertions are 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 the class of languages with good axiom systems for total correctness may be wider than for partial correctness.**

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.

This research was supported in part by NSF Grants MCS79-08365 and MCS80-10707, Advanced Research Projects Agency contract MDA903-80-C-0159, and a grant from the National Science and Engineering Research Council of Canada.

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 $P \in$ **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. For technical reasons, we always assume the constant a is an element of Σ. Throughout this paper we will assume for ease of exposition that $\Sigma = \{a,b,f,g,A_0\}$, 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) $= \{y \in \text{var} \mid y \text{ appears in } t\}$. Similarly, for a quantifier-free formula A, let var(A) $= \{y \in \text{var} \mid y \text{ appears in } A\}$. 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 $P \in$ **P** we can effectively find finite subsets cv(P), dep(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 dep(P) includes input variables, output variables, and any additional variables (such as those that appear in tests) upon whose values the behavior of P depends.

Define var(P) = cv(P) ∪ dep(P). In each interpretation I we can also associate with each P ∈ **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 $\sigma_0(dep(P)) = \sigma_0'(dep(P))$ then there is a trajectory $\tau' = (\sigma_0',\sigma_1',...) \in \mathcal{T}_I(P)$ such that $\sigma_i(dep(P)) = \sigma_i'(dep(P))$ for all i. This confirms the intuition that the computation of P depends only on the variables in dep(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 ∈ **P** on any input by asking a finite number of atomic questions about I. (Note we are allowing boundedly nondeterministic computations here). More formally, given a (code for) program P and i, we can effectively find a finite set of quantifier-free formulas $A_1, ..., A_k$ with $var(A_j) \subsetneq var(P) = \{y_1,...y_n\}$ 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 $\{\{t_{m_1},...,t_{m_n}\}| m = 1,2,...\}$ 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, $\sigma(y_j) = \sigma_0(t_{m_j})$ 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 $\{t_{m_1},...,t_{m_n}\}$ represent output values; i.e. whether there is some trajectory $(\sigma_0,...,\sigma_i)$ in $\mathcal{T}_I(P)$ with $\sigma_i(y_j) = \sigma_0(t_{m_j})$ for $j = 1,...,n$.

3. **P** is *effectively closed under variable substitutions*; that is, given P ∈ **P** with $dep(P) = \{x_{i_1},...,x_{i_m}\}$ and any set of m variables $\{y_1,...,y_m\}$ we can effectively find a program P' ∈ **P** such that $dep(P') = \{y_1,...,y_m\}$ 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 ∈ **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 all of** Q.

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

We extend $\mathcal{T}$, cv, and dep to **P'** below. 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

$$\tau_0 \circ \tau_1 = \begin{cases} (\sigma_0,...,\sigma_k,\sigma_1',...) & \text{if } \tau_0 = (\sigma_0,...,\sigma_k), \\ & \tau_1 = (\sigma_0',\sigma_1',...), \text{ and } \sigma_k = \sigma_0', \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

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

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

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

5. cv(**run** P **until** A) = cv(P);
dep(**run** P **until** A) = dep(P) ∪ var(A);
$\mathcal{T}_I(\textbf{run } P \textbf{ until } A) =$
$\{\tau \in \mathcal{T}_I(P)| \tau = (\sigma_0,\sigma_1,...)$, 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 $I,\sigma_k \models A\}$. Essentially, we can think of **run** P **until** A as inserting a test for A before every statement of P. As soon as the test is satisfied, the computation halts.

6. cv(**after each step of** P **do all of** Q) = cv(P) ∪ cv(Q);
dep(**after each step of** P **do all of** Q) = dep(P) ∪ dep(Q);
If var(P) ∩ cv(Q) ≠ ∅, then $\mathcal{T}_I$(**after each step of** P **do all of** Q) = ∅. (We consider **after each step of** P **do all of** Q syntactically incorrect unless var(P) ∩ cv(Q) = ∅; thus we do not allow the computation of Q to affect the variables of P.) If var(P) ∩ cv(Q) = ∅, $\mathcal{T}_I$(**after each step of** P **do all of** 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. $\mathrm{last}(\tau) = \sigma_{i_k}$

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 $\sigma_j'(\mathrm{dep}(P)) = \sigma_{i_j}(\mathrm{dep}(P))$ for all $j \leq k$.

7. cv(**begin local** $x_{i_1},...,x_{i_m}$; P **end**) = cv(P);
dep(**begin local** $x_{i_1},...,x_{i_m}$; P **end**) = dep(P) - $\{x_{i_1},...,x_{i_m}\}$;
$\mathcal{T}_I$(**begin local** $x_{i_1},...,x_{i_m}$; P **end**) = $\{(\sigma_0,\sigma_1)\circ\tau\circ(\mathrm{last}(\tau),\sigma_2) \mid \sigma_1 = \sigma_0[x_{i_1}/a,...,x_{i_m}/a],\ \tau \in \mathcal{T}_I(P)$, and $\sigma_2 = \mathrm{last}(\tau)[x_{i_1}/\sigma_0(x_{i_1}),...,x_{i_m}/\sigma_0(x_{i_m})]\}$. (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.

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

Thus we only require of a program like **after each step** of P **do all of** 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 all the programs described above, such that if P ∈ **P"** and Z ∈ plab, then $\mu Z[P]$ is a program in **P"**. We extend $\mathcal{T}$, cv, and dep to **P"** as follows:

1. cv(Z) = dep(Z) = ∅ for all Z ∈ plab;
$\mathcal{T}_I(Z) = \mathcal{T}_I(\omega) = \{(\sigma,\perp) \mid \sigma \neq \perp\}$ for all Z ∈ plab.

2. $\mathrm{cv}(\mu Z[P]) = \mathrm{dep}(\mu Z[P]) = \mathrm{dep}(P)$;
$\mathcal{T}_I(\mu Z[P]) = \cup_{i \geq 0}\mathcal{T}_I(P^i)$, where $P^0 = P$, 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, $\mu Z[P]$ 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** NOOP])

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 ∈ **P"** and interpretation I, there is a program Q ∈ 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 ∈ **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 $\mathrm{first}(\tau) = \sigma$ and $\mathrm{last}(\tau) \neq \perp$. The programming language **P** is deterministic if all programs P ∈ **P** are.

2.3. Partial Correctness and Termination

We expand the type Σ to $\Sigma^{\mathbf{P}}$ by adding, for each P ∈ **P**, a predicate symbol A_P of arity 2k, where k = |dep(P)|. In any interpretation I, $I \models A_P(u,v)$ iff for some trajectory $(\sigma_0,...,\sigma_k) \in \mathcal{T}_I(P)$ with $\sigma_k \neq \perp$, we have $\sigma_0(\mathrm{dep}(P)) = u$ and $\sigma_k(\mathrm{dep}(P)) = v$. (Note we use italics to indicate a vector of variables.) Thus 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 $\mathrm{first}(\tau)(\mathrm{dep}(P)) = u$ and $\mathrm{last}(\tau) \neq \perp$. Otherwise we say P *diverges* on input *u*.

A *(first-order) partial correctness* (resp. *termination*) *assertion* is a triple U{P}V (resp. U<P>V) where U and V are first-order formulas (over Σ) and P ∈ **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<P>V) iff, if U(*x*) then for all (resp. some) *y* which are possible outputs of P on input *x*, we have $I \models V(y)$. 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 ∈ **P** there is a formula B_P (of type Σ) such that

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

Thus I $\models B_P(x)$ iff there is a halting computation of P on input x. 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 recursively 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(x) iff $\varphi(x) = 0$
2. I $\models$ S(x,y) iff $\varphi(x) + 1 = \varphi(y)$
3. I $\models$ A(x,y,z) iff $\varphi(x) + \varphi(y) = \varphi(z)$
4. I $\models$ M(x,y,z) iff $\varphi(x) \times \varphi(y) = \varphi(z)$

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

I is *weakly arithmetic* if we can find 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 $\mathrm{dom}(I)^k$, and if $[x] = \{y \in \mathrm{dom}(I)^k \mid I \models E(x,y)\}$, there is a bijection φ: $\{[x] \mid I \models N(x)\} \rightarrow \mathcal{N}$ such that conditions 1-4 above hold (when restricted to N) with $[x]$ replacing x as the argument to φ. (Thus, for example, condition 2 becomes

$$I \models N(x) \wedge N(y) \wedge S(x,y) \quad \text{iff} \quad \varphi([x]) + 1 = \varphi([y]).)$$

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

2. 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.

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. Thus we get

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

1. P has a decidable halting problem for finite interpretations.

2. 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.

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. P has a decidable halting problem for finite interpretations.

2. The true quantifier-free partial correctness assertions are uniformly r.e. in <pres(I),Th(I)> for expressive-effective interpretations I.

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*{if $\neg$A **then** ω; P; **if** B **then** ω}*false*

(recall 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 (2) $\Rightarrow$ (1) 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 $\neg$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(x,d) iff x is the encoding of a Herbrand term equal to d) and consider the following encoding axiom, which we abbreviate by Enc:

$$\begin{aligned} \forall x,d[H(x,d) \equiv (x = \ulcorner a \urcorner \wedge d = a) \vee (x = \ulcorner b \urcorner \wedge d = b) \\ \vee (\exists y,e(Pr(x,\ulcorner f \urcorner,y) \wedge H(y,e) \wedge d = f(e)) \\ \vee (\exists y,d_1,d_2,z_1,z_2(Pr(x,\ulcorner g \urcorner,y) \wedge (Pr(y,z_1,z_2) \\ \wedge H(z_1,d_1) \wedge H(z_2,d_2) \wedge d = g(d_1,d_2)] \end{aligned}$$

where $Pr(z,x,y) \equiv y<z \wedge x<z \wedge z = \frac{1}{2}(x+y)(x+y+1)+x$

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: By induction on k. Details appear in the final paper.

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 and Enc, then $Std(x) \equiv \exists d \forall z(H(z,d) \Rightarrow x < z)$ defines the standard part of I.

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 x is standard and y nonstandard, $I \models x<y$. This in turn is proved using induction on k to show that if y is nonstandard, then $I \models \neg(y<S^k(0))$. 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 $y=S^k(0)$ since y is nonstandard.

We will now show that $I \models Std(x)$ iff x is a standard element. If x is standard, lemma 2 implies that Std(x) holds. Because dom(I) is infinite, for any standard x there exists an element d all of whose encodings are greater than x. For this d, $I \models \forall z(H(z,d) \Rightarrow x<z)$, because if z is either a standard value encoding d or a nonstandard value, it must be greater than x. Thus $I \models Std(x)$. On the other hand, if x is nonstandard, then for every $d \in dom(I)$, there exists a standard encoding z of d such that $I \models H(z,d) \wedge \neg(x<z)$. Therefore, $I \models \neg Std(x)$.

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: Deferred to the final paper.

Now we can define M_1 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_1 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_1 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_1 can tell if this formula is true in I. If so, M_1 outputs U{P}V; otherwise it outputs $\neg$U{P}V.

From Lemma 4, it follows immediately that M_1 is sound. And if I is strongly arithmetic, M_1 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 coding of sequences, 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:

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

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, $M \in \mathcal{N}$, a program $P \in \mathbf{P}$ with $\mathrm{dep}(P) = x = \langle x_{i_1},\ldots,x_{i_k}\rangle$, and $u = \langle u_1,\ldots,u_k\rangle \in \mathrm{dom}(I)^k$, 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. $\mathbf{K}_M = \{$domains of size $\leq N$, where $N = 1 + |U_M(x)|\}$ We also assume each $K \in \mathbf{K}_M$ has one distinguished element λ.

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

$$\textbf{run } P(x) \textbf{ until } \neg[\textstyle\bigwedge_{y \in \mathrm{cv}(P)}(\bigvee_{t \in U_M(x)} y = t)].$$

If $y \in \mathrm{cv}(P)$, $\tau = (\sigma_0,\sigma_1,\ldots) \in \mathcal{T}_I(P)$, 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 $y \in \mathrm{cv}(P)$, 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 $K \in \mathbf{K}_M$ and $c \in \mathrm{dom}(K)^k$) iff there exists a map ψ: $I_M(u) \to \mathrm{dom}(K) - \{\lambda\}$ such that

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

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 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

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

Call a pair $\langle K,c\rangle$ *diverging* if $P_M(x)$ diverges when run in interpretation K on input c. Call a pair *cleanly halting* if $P_M(x)$ halts with output d when run in interpretation K on input c, and no $d_i = \lambda$. Let $u_{\langle K,c\rangle}$ be the vector in $I_M(u)$ corresponding to 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 $I \models A_P(u,u_{\langle K,c\rangle})$. Thus we define the two first-order sentences

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

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

M_3 proceeds as follows. For each M, U, P, and V, it constructs the sentences $\mathrm{PC'}_{M,U,P,V}$ and $\mathrm{FPC'}_{M,P,U,V}$. This can be done effectively. By assumption the halting problem is decidable for finite interpretations so we can effectively find all the diverging pairs $\langle K,c\rangle$. (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 $\mathrm{PC'}_{M,U,P,V}$ (resp. $\mathrm{FPC'}_{M,P,U,V}$) holds in I for any M, it outputs U{P}V (resp. ¬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 $\mathrm{PC'}_{M,U,P,V}$ and $\mathrm{FPC'}_{M,P,U,V}$ by

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

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

Finally, for M_5, note that we do not need the assumption that the halting problem is decidable for finite interpretations to compute $\mathrm{T'}_{M,U,P,V}$, 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 $\langle K,c\rangle$. Every so often it discovers that another pair $\langle K,c\rangle$ is cleanly halting. Let J be those pairs which it has so far discovered to be cleanly halting. M_5 checks if

$$\mathrm{I} \models \forall x[\mathrm{U}(x) \Rightarrow \bigvee_{\langle K,c\rangle \in J}(\Delta_{\langle K,c\rangle}(x) \wedge \mathrm{V}(x_{\langle K,c\rangle}))]$$

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 $\langle K,c\rangle$ 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 $P \in \mathbf{P}$ with dep(P) $= x$ such that card(<P(x)>) is unbounded; i.e. for all M there exists $\tau \in \mathcal{T}_I(P)$, $\tau = (\sigma_0, \sigma_1, ...)$ such that $\{\sigma_i(x) | \sigma_i \in \tau\}$ 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 **P** = **P**", so that programs like **after each step of** P **do all of** 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. We write P(x) to indicate dep(P) $= x$. P(x') is just P with the variable x' substituted for x.

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 $\{\sigma_i'(x) | i \geq 0\} = \{\sigma_i(x) | i \geq 0\}$ 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 all of 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 $x = (x_1,x_2)$ will represent the integer k iff x_2 is the k^{th} distinct value reached by Q on input x_1. We write $[x] = k$ to indicate that the pair $x = (x_1,x_2)$ 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

$$\text{ans} = \begin{cases} \text{tt} & \text{if } [x] = [y] \\ \text{ff} & \text{otherwise} \end{cases}$$

(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

$$\text{ans} = \begin{cases} \text{tt} & \text{if } [x] < [y] \\ \text{ff} & \text{otherwise} \end{cases}$$

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

$$\text{ans} = \begin{cases} \text{tt} & \text{if } [x] = k \\ \text{ff} & \text{otherwise} \end{cases}$$

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.

```
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 all of (if $z = y$ then flag := tt).

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

In more detail, the program works as follows. The initial calls to CHECKINT check that x and y are integers, and diverge otherwise. We get ONEMORESTEPQ(x,y,z) by using Q*(y,z,flag) to compute successive values taken on by z when we run Q(z) starting with x, setting flag to tt when $y = z$, and then continuing the computation one more step.

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 alue determined by b.

```
begin local ans, y';
    LESS(x,b,ans);
    if ans = ff then ofl := tt else begin
        y_1 := b_1;
        y_2 := b_1;
        while ans = ff do begin
            y' := y_2;
            ONEMORESTEPQ(y_1,y',y_2);
            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	otherwise

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 $[z] = \langle [x],[y] \rangle$, 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 $1+2+\ldots+(n+m) = \tfrac{1}{2}(n+m)(n+m+1)$ 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 encoding and our old way of looking at tuples in dom(I) as integers, we can define a new way of looking at tuples in dom(I) as integers. 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. $\ulcorner t \urcorner = k$),
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.

```
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).

For example, suppose that in I 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 can use this listing of Herbrand terms to define a bijection φ: dom(I) $\rightarrow \mathcal{N}$. For u ∈ dom(I), $\varphi(u) = m$ 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. So in our example above, $\varphi(a) = 0$, $\varphi(b) = 1$, $\varphi((f(b)) = 2$.

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

It is not hard to define programs similar to the ones above which do arithmetic on this notion of integers. We can then use their weakest preconditions to show I is strongly arithmetic. We leave details to the full paper.

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 the construction of M_1, M_2, and M_5 are unaffected 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 $\in$ 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 such that card(<P(x)>) is unbounded. 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 $\in$ P and some x $\in$ 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 effectively find a formula A_P" of type Σ (analogous to the formula A_P' of Lemma 4) which is equivalent to A_P in I.

Since I is expressive-effective, and hence effectively presented, it follows that A_P defines an r.e. subset of $\mathrm{dom(I)}^{2k} \subseteq \mathcal{N}^{2k}$ (where k = |dep(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 $\mathcal{N} \models A_P^*(x,y)$ iff I $\models A_P(x,y)$. But by a straightforward syntactic translation using N, Z, S, A, M, and E, for *any* formula of number theory we can find a formula B' of type Σ such that I $\models$ B'(z) iff $\mathcal{N} \models$ B(z). Applying this syntactic translation to the A_P^*, we get the desired formula A_P".

Now we can construct M_1' and M_2' which are identical to M_1 and M_2, except they replace the A_P' of Lemma 4 by the A_P" constructed above. 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 questions concern 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 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 (perhaps the existence of a pairing function, so that sequences of values can be coded up by one value) does seem necessary. Moreover, both Herbrand definability and effective presentability (used in Theorem 2) 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. In order for a decision procedure to be a realistic analogue of a Floyd-Hoare axiom system, it should, in some sense, be uniform; i.e. independent of the particular interpretation I that is being used. For this reason, Theorem 2 is perhaps not all that useful. But it still might have application when we have a fixed interpretation in mind.

A third question concerns the relationship between the uninterpreted case considered in [MH80] and the interpreted case discussed here. 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 use of a total correctness proof system which, unlike most currently available, does not require the establishment of partial correctness as an essential first step.

References

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

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

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

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

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

[LO80] Langmaack, H.A. and E.R. Olderog. Present day Hoare-like systems for programming languages with procedures: Power, limits, and most likely extensions. In: Proceed. 7th Conf. Automata, Languages, and Programming, Nordwijkerhout 1980, Eds: J. W. de Bakker, J. van Leeuwen, LNCS 25, pp. 363-373, June, 1980.

[Li77] Lipton, R.J. A necessary and sufficient condition for the existence of Hoare logics. In: 18th IEEE Symp. on Found. Comp. Science, pp. 1-6, October, 1977.

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

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

[Mi81] Mitchell, J.C. Axiomatic Definability and Completeness for Recursive Programs. S.M. Thesis. M.I.T., 1981.

[Pr76] Pratt, V.R. Semantical Considerations of Floyd-Hoare Logic. In: 17th IEEE Symp. on Found. Comp. Science, pp. 109-121, October, 1976.

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