Effective Axiomatizations of Hoare Logics

EDMUND M. CLARKE, JR.

Harvard University, Cambridge, Massachusetts

STEVEN M. GERMAN

Harvard University, Cambridge, Massachusetts
Stanford University, Stanford, California

AND

JOSEPH Y. HALPERN

Harvard University, Cambridge, Massachusetts
Massachusetts Institute of Technology, Cambridge, Massachusetts

Abstract. For a wide class of programming languages P and expressive interpretations I, it is shown 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 it is shown 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 recursively enumerable 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.

Categories and Subject Descriptors: B.3.1 [Memory Structures]: General; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—assertions, logics of programs, pre- and postconditions; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—control primitives

General Terms: Languages, Verification, Theory

Additional Key Words and Phrases: Partial-correctness assertions, total-correctness assertions, termination assertions, sound and relatively complete axiomatization expressiveness, Lipton's theorem

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

This paper is an expanded version of a paper with the same title [3] given at the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, N.M., January 1982.

This research was supported in part by National Science Foundation Grants MCS 79-08365 and MCS 80-10707, Advanced Research Projects Agency Contract N00035-82-C-0250, Rome Air Development Center Contract F30602-80-C-0022, and a grant from the National Science and Engineering Research Council of Canada.

Authors' present addresses: E. M. Clarke, Jr., Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213; S. M. German, Aiken Computation Laboratories, Harvard University, Cambridge, MA 02138; J. Y. Halpern, IBM San Jose Laboratory, 5600 Cottle Road, San Jose, CA 95193.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM 0004-5411/83/0700-0612 $00.75

causes. The question of the existence of good Hoare axiom systems for programming languages was first raised in [2], where it was shown that languages with certain features cannot have axiom systems that are sound and relatively complete in the sense of [4]; 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 [11] 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 [9, 12] have since attempted to clarify Lipton's proof and extend it to handle first-order pre- and postconditions 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 [2] and [11]:

1. We are able to eliminate the requirement that pre- and postconditions 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 postconditions.

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 recursively enumerable 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)
recursively enumerable in \( \text{Th}(I) \) 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 \( \text{Th}(I) \) is then used to determine the truth of the encoded partial-correctness assertions (respectively, 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 \( \Sigma \)) consists of a domain, \( \text{dom}(I) \), and an assignment to each function (respectively, predicate) symbol of \( \Sigma \) a function (respectively, predicate) over \( \text{dom}(I) \) of the appropriate arity. \( \text{Th}(I) \) is the set of all first-order sentences (over \( \Sigma \)) true in \( I \).

In all that follows we assume we are working over a fixed finite type \( \Sigma \) 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 \( \Sigma \). 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, \( \text{var} = \{ x_0, x_1, \ldots \} \). For a term \( t \), let \( \text{var}(t) = \{ y \in \text{var} \mid y \text{ appears in } t \} \). Similarly, for a quantifier-free formula \( A \), let \( \text{var}(A) = \{ y \in \text{var} \mid y \text{ appears in } A \} \).

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

For each interpretation \( I \), a valuation over \( I \) is a mapping \( \sigma : \text{var} \to \text{dom}(I) \). We can extend a valuation to a mapping \( \sigma : \text{Terms} \to \text{dom}(I) \) in the obvious way. To represent
a diverging computation, we introduce one special valuation, $\bot$, such that $\bot(x)$ is undefined for all variables $x$. The valuation $\sigma[x/a]$ is identical to $\sigma$ 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 following criteria:

(1) For each program $P \in P$ we can effectively find finite subsets $cv(P) \subseteq var(P)$
    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 $P \in P$ a set of trajectories, $T_I(P)$, where each trajectory $\tau \in T_I(P)$ is a finite sequence of valuations $(\sigma_0, \sigma_1, \ldots)$ such that $\bot$, if it appears at all, only appears as the last valuation. There is no trajectory of the form $(\bot)$. These trajectories must also satisfy:

(a) If $y \notin cv(P)$, then for all $i$, $\sigma_i = \bot$ or $\sigma_i(y) = \sigma_0(y)$. (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 $i > 0$, $\sigma_i = \bot$ or $\sigma_i(y) = a, b, \sigma_i(x), f(\sigma_i(x))$, or $g(\sigma_i(x), \sigma_i(z))$ 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 $\tau' = (\sigma_0, \sigma_1', \ldots) \in T_I(P)$ 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 $P \in P$ on any input by asking a finite number of quantifier-free questions about $I$. (Note that 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 labeled 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 $A_1, \ldots, A_k$ with $var(A_i) \subseteq var(P) = \{y_1, \ldots, y_n\}$ such that by knowing the truth value of $A_i$ in $I$, $\sigma_0$, we can effectively compute a finite number of sets of terms $\{(l_{m_1}, \ldots, l_{m_n}) | m = 1, 2, \ldots\}$ over $(a, b, f, g, y_1, \ldots, y_n)$ which represent the possible values of the variables in $var(P)$ at the $i$th step of any trajectory in $T_I(P)$ starting with $\sigma_0$. That is, $\sigma$ is the $i$th step of such a trajectory iff, for some $m$, $\sigma(y_j) = \sigma_0(l_{m_j})$ for $j = 1, \ldots, n$ and $\sigma(x) = \sigma_0(x)$ for $x \notin var(P)$. We can also effectively compute which (if any) of the sets $\{l_{m_1}, \ldots, l_{m_n}\}$ represent output values, that is, whether there is some trajectory $(\sigma_0, \ldots, \sigma_i)$ in $T_I(P)$ with $\sigma_i(y_j) = \sigma_0(l_{m_j})$ for $j = 1, \ldots, n$.

(3) $P$ is effectively closed under variable substitutions; that is, given $P \in P$ with $var(P) = \{x_1, \ldots, x_m\}$ and any set of $m$ variables $\{y_1, \ldots, y_m\}$, we can effectively find a program $P' \in P$ such that $var(P') = \{y_1, \ldots, y_m\}$ and $(\sigma_0, \sigma_1, \ldots) \in T_I(P)$ iff for some $(\sigma_0, \sigma_1, \ldots) \in T_I(P')$ we have $\sigma_i(x_{ik}) = \sigma_i(y_k)$ for $k = 1, \ldots, m$. 
(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 $P, Q \in P'$ 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 $x_{i_1}, \ldots, x_{i_n}; P$ 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 $x_{i_1}, \ldots, x_{i_n}; P$ end, we first save the values of $x_{i_1}, \ldots, x_{i_n}$ (as given in the current state), initialize them all to $a$, run $P$, and then restore the original values of $x_{i_1}, \ldots, x_{i_n}$. We extend $T, cv$, and var to $P'$ in a straightforward way, thus giving the programs of $P'$ their formal semantics as trajectories. We leave details to Appendix A.

Now we formally define $P$ to be effectively closed under flowchart operations, subroutine calls, and run-time checks if for all $P \in P'$ and all interpretations $I$ we can effectively find a $Q \in P$ which simulates $P$ in $I$. That is, $cv(P) \subseteq cv(Q)$, var($P$) \subseteq var($Q$), and for all $\tau \in T_I(P)$ (respectively, $T_I(Q)$) with last($\tau') \neq \bot$ there exists a $\tau'' \in T_I(Q)$ (respectively, $T_I(P)$) such that first($\tau)(\text{var}(P)) = \text{first}(\tau')(\text{var}(P))$ and last($\tau)(\text{var}(P)) = \text{last}(\tau')(\text{var}(P))$ (where given a trajectory $\tau = (\sigma_0, \ldots, \sigma_n)$, we define first($\tau$) = $\sigma_0$ and last($\tau$) = $\sigma_n$).

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 [11]. 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 [14]. Let plab = ($Z_0, Z_1, \ldots$) be some set of program labels, and let $P''$ be the smallest language containing $P$, lab, and closed under the programming constructs described above, such that if $P \in P''$ and $Z \in \text{plab}$, 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 $T, cv$, and var to $P''$ in Appendix A.

Finally, we define $P$ to be effectively closed under recursive calls (as well as flowchart operations, subroutine calls, and run-time checks) if for every program $P \in P''$ and interpretation $I$ there is a program $Q \in P$ 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 $P \in \mathcal{P}$ with the semantics that we have given can be simulated by some program in $\mathcal{P}$ whatever the semantics of $\mathcal{P}$ are, such problems will not concern us here.)

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

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

A (first-order) partial-correctness (respectively, termination) assertion is a triple $U(P) \forall V$ (respectively, $U(P) \exists V$) where $U$ and $V$ are first-order formulas (over $\Sigma$) and $P \in \mathcal{P}$. By definition,

$I \models U(P) \forall V$ iff $I \models \forall \vec{x} \exists \vec{y} (U(\vec{x}) \land A_P(\vec{x}, \vec{y}) \implies V(\vec{y}))$,

$I \models U(P) \exists V$ iff $I \models \forall \vec{x} \exists \vec{y} (U(\vec{x}) \land A_P(\vec{x}, \vec{y}) \land V(\vec{y}))$.

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

2.4. EXPRESSIVENESS. An interpretation $I$ is weakly expressible for $\mathcal{P}$ iff for every $P \in \mathcal{P}$ there is a formula $B_P$ (of type $\Sigma$) such that

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

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

In Dijkstra’s terminology [5], $B_P$ 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 $\Sigma$ is effectively presented if there is a tuple of integers $\text{pres}(I) = \langle n_{\text{card}}, n_{\text{dom}}, n_a, n_b, n_f, n_g, n_{\text{af}}, n_{\text{gf}} \rangle$, where $n_{\text{card}}$ encodes the size of $\text{dom}(I)$ ($n_{\text{card}} = 0$ if $\text{dom}(I)$ is infinite, $n_{\text{card}} = N + 1$ if $|\text{dom}(I)| = N$); $n_{\text{dom}}$ is a code for $\text{dom}(I)$, a recursive subset of $\mathcal{N}$ (the integers); $n_a, n_b \in \text{dom}(I)$ are the interpretations of $a$ and $b$; and $n_f, n_g$, and $n_{\text{af}}, n_{\text{gf}}$ 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 $i \in \text{dom}(I)$ 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 $\mathcal{P}$ iff it is either (weakly expressible for $\mathcal{P}$ and Herbrand definable) or finite. $I$ is expressive-effective iff it is either (weakly expressible and 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 $\phi: \text{dom}(I) \rightarrow \mathcal{N}$ such that

1. $I \models Z(u) \iff \phi(u) = 0$,
2. $I \models S(u, v) \iff \phi(u) + 1 = \phi(v)$,
3. $I \models A(u, v, w) \iff \phi(u) + \phi(v) = \phi(w)$,
4. $I \models M(u, v, w) \iff \phi(u) \times \phi(v) = \phi(w)$.

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

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

$I \models N(\bar{u}) \land N(\bar{v}) \land S(\bar{u}, \bar{v}) \iff \phi([\bar{u}]) + 1 = \phi([\bar{v}]).$

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 (respectively, termination) assertions are true in $I$ when given an oracle for $\text{Th}(I)$. Thus the set of first-order partial-correctness (respectively, termination) assertions true in $I$ is uniformly recursive in $\text{Th}(I)$ 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 $\text{dom}(I)$ finite, a program $P \in P$ with $|\text{var}(P)| = k$, and $\bar{u} \in \text{dom}(I)^k$, decides if $P$ halts on input $\bar{u}$ 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 recursively enumerable in $\text{Th}(I)$ 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 $\text{Th}(I)$, 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 (respectively, termination) assertions true in $I$ is recursive in $\langle \text{pres}(I), \text{Th}(I) \rangle$ if $I$ is expressive-effective and is uniformly recursive in $\text{Th}(I)$ if $I$ is infinite.
2. $P$ has a decidable halting problem for finite interpretations.
Moreover, the set of first-order termination assertions true in \( I \) is recursively enumerable in \( \langle \text{pres}(I), \text{Th}(I) \rangle \) for expressive-effective interpretations \( I \).

By way of contrast, Lipton showed:

**Theorem [11]:** Let \( \mathcal{P} \) be a deterministic acceptable programming language. Then the following are equivalent:

1. The true quantifier-free partial-correctness assertions are uniformly recursively enumerable in \( \langle \text{pres}(I), \text{Th}(I) \rangle \) for expressive-effective interpretations \( I \).
2. \( \mathcal{P} \) has a decidable halting problem for finite interpretations.

Lipton’s proof only showed how to enumerate the true partial-correctness assertions of the form \( \text{true}(P) \) false. However, note that

\[
I \vdash A(P) B \quad \text{iff} \quad I \vdash \text{true}(\neg A \text{ then } \omega; P; \text{if } B \text{ then } \omega) \text{ false}
\]

(where \( \omega \) is the program which always diverges). Moreover, if \( A \) and \( B \) are quantifier-free, this modified program (or one that simulates it) is in \( \mathcal{P} \). 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 \( \neg A \text{ then } \omega \)) cannot in general be simulated by a program in an acceptable programming language, since the simulating program would violate condition (2) of acceptable programming languages as defined in Section 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 \( \mathcal{P} \) is acceptable with recursion and \( I \) is expressive-Herbrand with respect to \( \mathcal{P} \) then either

1. \( I \) is strongly arithmetic, or
2. \( \forall P \in \mathcal{P} \exists n (P \text{ reaches at most } n \text{ distinct valuations in any computation}) \) (i.e., for all \( \tau \in \mathcal{J}_I(P) \), \( \sigma_i \sigma_i \in \tau \) has \( \leq n \) elements).

We will abbreviate condition (2) of the lemma by \((\dagger)\) 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 [2]. The proof in fact goes through under much weaker hypotheses: \( \mathcal{P} \) does not have to be acceptable or deterministic. To prove the remainder of Theorem 1, we will describe five effective procedures, \( M_1, \ldots, M_5 \). When given an oracle for \( \text{Th}(I) \) 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 \( M_3 \) 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, \( M_3 \) is complete for termination assertions if \( I \) is strongly arithmetic. If \( \mathcal{P} \) has a decidable halting problem for finite interpretations and \((\dagger)\) holds, then \( M_3 \)
(respectively, $M_4$) is complete for partial-correctness (respectively, 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 $M_5$ 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:

$$\begin{align*}
\text{AX1:} & \quad \neg(S(x) = 0), \\
\text{AX2:} & \quad S(x) = S(y) \Rightarrow x = y, \\
\text{AX3:} & \quad x + 0 = x, \\
\text{AX4:} & \quad x + S(y) = S(x + y), \\
\text{AX5:} & \quad x \times 0 = 0, \\
\text{AX6:} & \quad x \times S(y) = x \times y + x, \\
\text{AX7:} & \quad \neg(x < 0), \\
\text{AX8:} & \quad x < S(y) = (x < y \lor x = y), \\
\text{AX9:} & \quad x < y \lor x = y \lor y < x.
\end{align*}$$

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. [16]), 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 $\Sigma$:

$$\begin{align*}
[a] & = 0, \\
[b] & = 1, \\
[f] & = 2, \\
g & = 3, \\
[f(t)] & = \langle [f], [t] \rangle, \\
g(t, u) & = \langle [g], \langle [t], [u] \rangle \rangle,
\end{align*}$$

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

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:

$$\begin{align*}
\text{Enc:} & \quad \forall x, d[H(x, d) = (x = [a] \land d = a) \lor (x = [b] \land d = b) \\
& \lor \exists y, d'(Pr(x, [f], y) \land H(y, d') \land d = f(d')) \\
& \lor \exists y, z_1, z_2, z_3Pr(x, [g], y) \land Pr(y, z_1, z_2) \land H(z_1, d) \\
& \land H(z_2, z_3) \land d = g(d_1, d_2),
\end{align*}$$

where $Pr(x, y, z) = y \leq z \land x \leq z \land (z = \frac{1}{2}(x + y)(x + y + 1) + x)$.

We now show that $H$ "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 ⊨ u < v. This in turn is proved using induction on k to show that if v is nonstandard, then I ⊨ ¬(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 that the last two disjunctions in Enc cannot hold if k = 0 or 1, since if k = 0 or 1 and m = [f'] or [g'], then for all u ∈ dom(I) we must have I ⊨ ¬Pr(k*, m*, u).)

**Inductive case.** Let us assume that the lemma is true for all k' < k. Assume k = [f(t)] for some term t, and let k_1 = [t_1], 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 = ([f'], k_1), and so I ⊨ Pr(k*, 2*, k*_1). By the inductive assumption, I ⊨ H(k*, t_1). Hence, by axiom Enc it follows that I ⊨ H(k*, f(t_1)) as required.

Conversely, assume that I ⊨ H(k*, d). By Enc we can assume without loss of generality that there are v, d ∈ dom(I) such that I ⊨ Pr(k*, 2*, v) ∧ H(v, d_1) ∧ d = f(d_1). (The other case is similar.) Since I ⊨ Pr(k*, 2*, v) implies I ⊨ 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 ⊨ t_1 = d_1. Since k = (2, k_1) and I ⊨ 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) = \exists d \forall y(H(z, d) ⇒ 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 ⊨ \forall z(H(z, d) ⇒ 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 ⊨ Std(u). On the other hand, if u is nonstandard, then for every d ∈ dom(I) there exists a standard encoding w of d such that I ⊨ H(w, d) ∧ ¬(u < w). Therefore, I ⊨ ¬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 ∈ P we can effectively find a formula A_P of type Σ which is equivalent to A_P in I.

**Proof.** See Appendix B. □

Now we can define M; 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 ⊨ ∃x(Z' (x) ∧ ∀y(Z' (y) ⇒ y = x)), that S', A', and M' define functions (i.e., I ⊨ ∀x∃y(S' (x, y) ∧ ∀z(S' (x, z) ⇒ 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) ∧ S' (y, z) ⇒ x = y.) Now, using these formulas, we can define Std(x) as
in Lemma 3 and check if \( I \models \forall x (\text{Std}(x)) \). If not, then \( M_1 \) continues guessing. But if \( \forall x (\text{Std}(x)) \) 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 \( P \in P \), \( M_1 \) constructs the formula \( PC_{U,P,V} \):

\[
\forall \bar{x}, \bar{y}(U(\bar{x}) \land A'\rho(\bar{x}, \bar{y}) \Rightarrow V(\bar{y})).
\]

By consulting the oracle for \( \text{Th}(I) \), \( M_1 \) can tell if this formula is true in \( I \). If so, \( M_1 \) outputs \( U(P) \lor \); otherwise it outputs \( \neg U(P) \lor \).

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 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 \( T_{U,P,V} \):

\[
\forall \bar{x} \exists \bar{y}(U(\bar{x}) \Rightarrow A'\rho(\bar{x}, \bar{y}) \land V(\bar{y})).
\]

\( \square \)

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 \( N(\bar{x}) \) for natural number and formula \( E(\bar{x}, \bar{y}) \) for equivalence. AX1–9 would also have to be appropriately modified to restrict everything to \( N \). For example, AX2 would read

\[
N(\bar{x}) \land N(\bar{y}) \land N(\bar{z}) \Rightarrow [S(\bar{x}, \bar{y}) \land S(\bar{x}, \bar{z}) \Rightarrow E(\bar{y}, \bar{z})].
\]

We 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 have to check that the following two formulas hold in \( I \):

\[
E(\bar{x}, \bar{x}) \land (E(\bar{x}, \bar{y}) \Rightarrow E(\bar{y}, \bar{x})) \land (E(\bar{x}, \bar{y}) \land E(\bar{y}, \bar{z}) \Rightarrow E(\bar{x}, \bar{z})),
\]

\[
(Z(\bar{x}) \Rightarrow N(\bar{x})) \land ((N(\bar{x}) \land S(\bar{x}, \bar{y}) \Rightarrow N(\bar{y})).
\]

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

3.2.2. Construction of \( M_3, M_4, \) and \( M_6 \). We extend the techniques of [11] to the first-order case.

Given an interpretation \( I \), an integer \( M \), a program \( P \in P \) with \( \text{var}(P) = \bar{x} = \langle x_i, \ldots, x_a, \rangle \), and \( \bar{u} = \langle u_0, \ldots, u_b \rangle \in \text{dom}(I)^a \), we make the following definitions:

1. \( U_M(\bar{x}) = \{ \text{terms of depth } \leq M \text{ over } \{ f, g, a, b, \bar{x} \} \} \).
2. \( I_M(\bar{u}) = \{ \text{values obtained by substituting } u_j \text{ for } x_j \text{ in the terms of } U_M(\bar{x}) \} \).
3. \( K_M = \{ K | K \text{ is an interpretation of type } \Sigma, \text{dom}(K) \text{ has size } \leq N \text{ where } N = 1 + | U_M(\bar{x}) | \text{, and there is a distinguished element } \lambda \in \text{dom}(K) \} \).
4. \( P_M \) is the program which acts just like \( P \) except that on input \( \bar{u} \) it halts at any valuation \( \sigma \) such that \( \sigma(\bar{y}) \notin I_M(\bar{u}) \) for some \( y \in \text{cv}(P) \). \( P_M \) is just

\[
\text{run } P \text{ until } \neg [\forall \sigma \in \text{cv}(P) \forall \bar{x} (\sigma(\bar{x}) = \bar{u})].
\]

If \( y \in \text{var}(P) \), \( \tau = (\sigma_0, \sigma_1, \ldots) \in \mathcal{T}_I(P) \), and \( \sigma_n(\bar{y}) \) is the \( m \)-th distinct valuation in \( \tau \), then it is straightforward to show using condition (1) on acceptable programming languages and induction on \( m \) that \( \sigma_n(\bar{y}) \in I_m(\sigma_0(\bar{x})) \). From this observation we get
LEMMA 5 [11]. If (†) holds in I, then there exists an M such that for all y ∈ var(P), all τ ∈ ℱτ, and all n, we have σ_n(y) ∈ I_M(σ_n(τ)).

We say that I is isomorphic to ⟨K, c⟩ on I_M(υ) (where K ∈ K_M and c ∈ dom(K)^k) if for there exists a bijection Ψ: I_M(υ) → dom(K) - {λ} such that
1. Ψ(υ_i) = c_i for i = 1, ..., k.
2. I ⊨ A_0(υ_1, υ_2) for υ_1, υ_2 ∈ I_M(υ) if K ⊨ A_0(Ψ(υ_1), Ψ(υ_2)).
3. If υ_1 ∈ I_M(υ) and f(υ_1) /∈ I_M(υ), then K ⊨ f(Ψ(υ_1)) = λ. Similarly for g.
4. If υ_1, f(υ_1) ∈ I_M(υ), then K ⊨ f(Ψ(υ_1)) = Ψ(f(υ_1)). Similarly for g.

Note that there are only finitely many nonisomorphic pairs ⟨K, c⟩ for a given M. Moreover, for each such pair we can find a first-order formula Δ_{K,c}(υ) such that
I ⊨ Δ_{K,c}(υ) if and only if I is isomorphic to ⟨K, c⟩ on I_M(υ).

Call a pair ⟨K, c⟩ 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 = λ. Let x_{i(K,c)} be the term in U_M(υ) corresponding to d (i.e., when we substitute c in for υ in x_{i(K,c)} we get d).

It is easy to check that if ⟨K, c⟩ is diverging and I ⊨ Δ_{K,c}(υ), then P diverges in I on input υ. If ⟨K, c⟩ is cleanly halting and I ⊨ Δ_{K,c}(υ), then I ⊨ A_P(υ, u_{i(K,c)}) (where u_{i(K,c)} is the result of substituting υ for υ in the term x_{i(K,c)}). 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}: ∀x[U(υ) ⇒ (∨_{i(K,c)}Δ_{K,c}(υ) ∧ V(x_{i(K,c)}))]
FPC_{M,U,P,V}: ∃x[U(υ) ∧ (∨_{i(K,c)}Δ_{K,c}(υ) ∧ ¬V(x_{i(K,c)}))]

M_5 proceeds as follows. For each M, U, P, and V it constructs the sentences PC_{M,U,P,V} and FPC_{M,U,P,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 ⟨K, c⟩. We remark that we are using the fact that programs are deterministic here: a program either diverges or halts on a given input (but cf. Section 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 ⟨K, c⟩. 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(⟨⟩)) M_5 discovers that PC_{M,U,P,V} (respectively, FPC_{M,U,P,V}) holds in I for any M, it outputs U(P) V (respectively, ¬U(P) V). The procedure is sound by the comments above and complete if (†) holds for I by Lemma 5.

M_5 is identical to M_5 but replaces PC_{M,U,P,V} and FPC_{M,U,P,V} by

T_{M,U,P,V}: ∀x[U(υ) ⇒ (∨_{i(K,c)}Δ_{K,c}(υ) ∧ V(x_{i(K,c)}))]
FT_{M,U,P,V}: ∃x[U(υ) ∧ (∨_{i(K,c)}Δ_{K,c}(υ) ∧ ¬V(x_{i(K,c)}))]

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 ⟨K, c⟩ 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 \( (K, \bar{c}) \) is cleanly halting. Let \( J \) be those pairs which it has so far discovered to be cleanly halting. \( M_b \) checks if

\[
I = \forall \bar{x} [U(\bar{x}) = \bigvee_{(K, \bar{c}) \in I} (\Delta_{(K, \bar{c})}(\bar{x}) \land V(\bar{x}(K, \bar{c})))].
\]

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

3.2.3. Proof of Lemma 1. Assume that \( (\dagger) \) does not hold for \( I \). Then there is some program \( P \in P \) which has no bound on the number of distinct valuations it goes through in any computation; that is, for all \( M \) there exists \( \tau \in \mathcal{F}_t(P) \), \( \tau = (\sigma_0, \sigma_1, \ldots) \), such that \( \{\sigma_i | \sigma_i \in \tau\} \) has at least \( M \) distinct elements. We show how to define programs whose weakest preconditions (the \( \mathcal{B}_P \) of Section 2.4) define the formulas necessary to make \( I \) arithmetic. Our initial steps are much like those of Lipton [11]. 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 \( Q \) 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 \( \text{var}(P) = \bar{x} \). As a notational convenience we will write \( P(\bar{x}) \) to indicate this. We then use \( P(\bar{x}') \) to denote the result of substituting \( \bar{x}' \) for \( \bar{x} \) in \( P \). Occasionally when we write \( R(\bar{y}) \) for some program \( R \), we may omit some of the variables in \( \text{var}(R) \) which are just local variables; thus in general \( \bar{y} \) will just consist of those variables on whose value the program \( R \) depends.

We first construct a program \( Q(\bar{x}) \) such that if we run \( Q(\bar{x}) \) on any input, \( \bar{x} \) takes on the same values as when we run \( P(\bar{x}) \) on the same input, but without repetition; that is, if \( \tau = (\sigma_0, \sigma_1, \ldots) \in \mathcal{F}_t(P) \) and \( \tau' = (\sigma'_0, \sigma'_1, \ldots) \in \mathcal{F}_t(Q) \) with \( \sigma_0 = \sigma'_0 \), then \( \{\sigma'_i(\bar{x}) | i \geq 0\} = \{\sigma_i(\bar{x}) | i \geq 0\} \), and if \( \sigma'_i(\bar{x}) = \sigma'_j(\bar{x}) \) for \( i < j \), then \( \sigma'_i(\bar{x}) = \sigma'_j(\bar{x}) \) for all \( k, i \leq k \leq j \). Essentially this is done by keeping track of the initial and current values of \( \bar{x} \) 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 [11] for more details). The code for \( Q(\bar{x}) \) is given in Figure 1.

The pair \( \bar{u} = (\bar{u}_1, \bar{u}_2) \) will represent the integer \( k \) iff \( \bar{u}_2 \) is the \( k \)th distinct value reached by \( Q \) on input \( \bar{u}_1 \). We write \( [\bar{u}] = k \) to indicate that the pair \( \bar{u} = (\bar{u}_1, \bar{u}_2) \) represents \( k \).

Choose two Herbrand terms \( tt \) and \( ff \) which get distinct values in \( I \) to represent \emph{true} and \emph{false}, respectively. Then, using \( Q \), it is straightforward to write programs which meet the following specifications.

(1) CHECKINT(\( \bar{x} \)) \: Halts with \( \bar{x} \) unchanged if \( \bar{x} \) represents an integer; otherwise CHECKINT will diverge.
begin local \( \overline{\text{init}}, x', y' \);
\( \text{init} := \overline{x} \);
\( x' := \overline{x} \);
after each step of \( P(x') \) do \( R(x, x', y', \overline{\text{init}}) \);
end

where \( R(x, x', y', \overline{\text{init}}) \) is the program

if \( \overline{x} \neq \overline{x}' \) then begin
\( y := \overline{\text{init}} \);
run \( P(f) \) until (\( f = \overline{x}' \lor y = \overline{x} \));
if \( y = \overline{x} \) then \( x := x' \);
end

Fig. 1. The program \( Q(\overline{x}) \).

(2) EQ(\( \overline{x}, \overline{y}, \text{ans} \)): If \( \overline{x} \) and \( \overline{y} \) do not both represent integers, EQ will diverge.
Otherwise EQ will terminate with \( \overline{x}, \overline{y} \) unchanged and
\[
\text{ans} = \begin{cases} 
  \text{tt} & \text{if } [\overline{x}] = [\overline{y}], \\
  \text{ff} & \text{otherwise}.
\end{cases}
\]

(3) LESS(\( \overline{x}, \overline{y}, \text{ans} \)): If \( \overline{x} \) and \( \overline{y} \) do not both represent integers, LESS will diverge.
Otherwise LESS will terminate with \( \overline{x}, \overline{y} \) unchanged and
\[
\text{ans} = \begin{cases} 
  \text{tt} & \text{if } [\overline{x}] < [\overline{y}], \\
  \text{ff} & \text{otherwise}.
\end{cases}
\]

(4) NUM(\( \overline{x} \), \( \text{ans} \)): If \( \overline{x} \) does not correspond to the integer \( k \), NUM will diverge.
Otherwise, NUM will terminate with \( \overline{x} \) unchanged and
\[
\text{ans} = \begin{cases} 
  \text{tt} & \text{if } [\overline{x}] = k, \\
  \text{ff} & \text{otherwise}.
\end{cases}
\]

The idea for computing EQ(\( \overline{x}, \overline{y}, \text{ans} \)) is to compute the successive values reached
by \( Q \) starting from \( \overline{x} \) and \( \overline{y} \), and check that we reach \( \overline{x} \) and \( \overline{y} \) at the same time.
(Recall that we assume \( \overline{x} \) is of the form \( \overline{x}_1, \overline{x}_0 \) and likewise \( \overline{y} \).) We give the code in
Figure 2; the codes for CHECKINT, LESS, and NUM are similar and will not be given.

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

In general, it does not seem possible to construct a program SUC(\( \overline{x}, \overline{y} \)) which will
calculate \( \overline{y} \) such that \( [\overline{y}] = [\overline{x}] + 1 \). If \( [\overline{x}] = k \), it may be the case that only \( k \)
distinct elements of \( \text{dom}(f) \) are reachable from \( \overline{x} \) 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.

(5) SUC(\( \overline{b}, \overline{x}, \overline{y}, \text{off} \)): If \( \overline{b}, \overline{x} \), and \( \overline{y} \) do not all initially represent integers, SUC will
diverge. Otherwise SUC will terminate with \( \overline{b}, \overline{x} \) unchanged and
\[
[\overline{y}] = [\overline{x}] + 1, \quad \text{off} = \text{tt} \quad \text{if } [\overline{x}] < [\overline{b}],
\]
\[
\text{off} = \text{ff} \quad \text{if } [\overline{b}] < [\overline{x}].
\]
CHECKINT(\(\vec{x}\));
CHECKINT(\(\vec{y}\));
begin local \(\vec{u}, \vec{v}, \vec{u}', \vec{v}'\);
\(\vec{u} := \vec{x};\)
\(\vec{v} := \vec{y};\)
while \(\vec{u} \neq \vec{x} \lor \vec{v} \neq \vec{y}\) do begin
\(\vec{u}' := \vec{u};\)
\(\vec{v}' := \vec{v};\)
ONEMORESTEPQ(\(\vec{x}, \vec{u}', \vec{u}\));
ONEMORESTEPQ(\(\vec{y}, \vec{v}', \vec{v}\));
end;
if \(\vec{u} = \vec{x} \land \vec{v} = \vec{y}\) then \(\text{ans} := \text{tt}\) else \(\text{ans} := \text{ff}\);
end

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

begin local \text{flag};
\(\text{flag} := \text{ff};\)
\(\vec{z} := \vec{x};\)
run \(Q^*(\vec{y}, \vec{z}, \text{flag})\) until \((\text{flag} = \text{tt} \land \vec{y} \neq \vec{z})\);
end

where \(Q^*(\vec{y}, \vec{z}, \text{flag})\) is after each step of \(Q(\vec{z})\) do (if \(\vec{z} = \vec{y}\) then \(\text{flag} := \text{tt}\)).

FIG. 2. The program \(\text{EQ}(\vec{x}, \vec{y}, \text{ans})\).

(6) ADD(\(\vec{b}, \vec{x}, \vec{y}, \vec{z}, \text{ofl}\)): If \(\vec{b}, \vec{x}, \vec{y}\) do not all initially represent integers, ADD will diverge. Otherwise, ADD will terminate with \(\vec{b}, \vec{x}, \vec{y}\) unchanged and

\([\vec{z}] = [\vec{x}] + [\vec{y}], \text{ ofl} = \text{ff}\) if \([\vec{x}] + [\vec{y}] \leq [\vec{b}]\),
\(\text{ofl} = \text{tt}\) if \([\vec{b}] < [\vec{x}] + [\vec{y}]\).

(7) MULT(\(\vec{b}, \vec{x}, \vec{y}, \vec{z}, \text{ofl}\)): Similar to (6) above except that

\([\vec{z}] = [\vec{x}] \times [\vec{y}], \text{ ofl} = \text{ff}\) if \([\vec{x}] \times [\vec{y}] \leq [\vec{b}]\),
\(\text{ofl} = \text{tt}\) if \([\vec{b}] < [\vec{x}] \times [\vec{y}]\).

The code for \(\text{SUC}(\vec{b}, \vec{x}, \vec{y}, \text{ofl})\) is given in Figure 3. The idea is to initialize \(\vec{y}\) to \(\vec{b}\) and then increase \(\vec{y}\) (using ONEMORESTEPQ) until \(\vec{x} < \vec{y}\). 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 \(\vec{b}\).

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 \(\vec{x}, \vec{y}, \vec{z}\) iff \([\vec{z}] = [\vec{x}] + [\vec{y}]\)) and taking weakest preconditions we could already define formulas \(N, Z, E, S, L, A,\) and \(M\) which satisfy the definition of Section 2.6. We note that none of the above programs required recursive calls. Thus it follows that if \((\dagger)\) 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.
begin local ans, y';
  LESS(x, b, ans);
  if ans = ff then of! := it else begin
    y' := b1; /* recall y' = (y1, y2)*/
    y := b2;
  while ans = ff do begin
    y' := y;
    ONEMORESTEPQ(b3, y', y); 
    LESS(x, y, ans);
  end;
end;

Fig. 3. The program $SUC(\delta, \tilde{x}, y, of!)$.

(8) HRBD($\tilde{x}$, enc, d): If $\tilde{x}$ does not represent an integer, HRBD will fail to terminate. Otherwise, HRBD will terminate with $\tilde{x}$ unchanged and

$$
\text{enc} = tt, \quad d = h \quad \text{(in I)} \quad \text{if \ } [\tilde{x}] \text{ encodes Herbrand term } h,
$$

$$
\text{enc} = ff \quad \text{if \ } [\tilde{x}] \text{ does not encode a Herbrand term.}
$$

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

Note that a true pairing function cannot be programmed using the above techniques. Given only $\tilde{x}$ and $\tilde{y}$, it is not in general possible to compute $\tilde{z}$ with $[\tilde{z}] = [\tilde{u}]$, 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($\tilde{z}$, $\tilde{x}$, $\tilde{y}$): If $\tilde{z}$ does not represent an integer, then PR diverges. Otherwise, PR will terminate with the final value of $\tilde{z}$ unchanged, and the final values of $\tilde{x}$ and $\tilde{y}$ will satisfy the relationship

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

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

A straightforward modification of HRBD($\tilde{x}$, enc, d) gives us HRBD($\tilde{x}$, d) which halts iff $d$ is equal to the Herbrand term encoded by $\tilde{x}$. Now, by taking weakest preconditions we can already show that $t$ is weakly arithmetic and has a formula $H$ which satisfies Enc. As we remarked in 3.2.1, this would 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 $t$ is strongly arithmetic.

List the terms in the Herbrand universe (of $(a, b, f, g)$) in order of increasing encoding: $a, b, f(a), f(b), g(a, a), \ldots$.

Using this list, we can define a bijection $\phi: \text{dom}(I) \rightarrow N$. For $u \in \text{dom}(I)$, $\phi(u) = m$ iff, if $t$ is the first term on the list such that $I \vdash t = u$, 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 $\phi(a) = 0$, $\phi(b) = 1$, and $\phi(f(b)) = 2$.

Since $I$ is Herbrand definable (by assumption) and has an infinite domain (otherwise $\uparrow$ would hold), $\phi$ is indeed a bijection.
\[ \mu H \begin{align*}
&\text{begin} \\
&\text{local } \text{init}, \text{ans}, \overline{\text{fd}}, \overline{\text{ar}}, \overline{\text{arg}}, \overline{\text{arg}}', \overline{\text{arg}}'', d'; \\
&\text{init} := \overline{\text{x}}; \\
&\text{enc} := tt; \\
&\text{NUM}_{\text{enc}}(\overline{\text{x}}, \text{ans}); \\
&\text{if } \text{ans} = tt \text{ then } d := a \\
&\text{else} \begin{align*}
&\text{NUM}_{\text{enc}}(\overline{\text{x}}, \text{ans}); \\
&\text{if } \text{ans} = tt \text{ then } d := b \\
&\text{else begin} \\
&\text{PR}(\overline{\text{x}}, \overline{\text{fd}}, \overline{\text{ar}}); \\
&\text{NUM}_{\text{enc}}(\overline{\text{x}}, \overline{\text{fd}}, \overline{\text{ans}}); \\
&\text{if } \text{ans} = tt \text{ then begin} \\
&\text{x} := \overline{\text{arg}}'; \\
&H; \\
&d := f(d'); \\
&\text{x} := \text{init}; \\
&\text{end}; \\
&\text{else begin} \\
&\text{NUM}_{\text{enc}}(\overline{\text{fd}}, \overline{\text{ans}}); \\
&\text{if } \text{ans} = tt \text{ then begin} \\
&\text{PR}(\overline{\text{ar}}', \overline{\text{ar}}'', \overline{\text{arg}}'''); \\
&\text{x} := \overline{\text{arg}}'''; \\
&H; \\
&d' := d; \\
&\text{if } \text{enc} = tt \text{ then begin} \\
&\text{x} := \overline{\text{arg}}'''; \\
&H; \\
&d := g(d', d); \\
&\text{end}; \\
&\text{x} := \text{init}; \\
&\text{end}; \\
&\text{else enc} := ff; \\
&\text{end}; \\
&\text{end}; \\
&\text{end}; \\
&\text{end}.
\end{align*}
\end{align*} \]

Fig. 4. The program HRBD(\overline{x}, enc, d).

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 \([\overline{\text{v}}] = m \iff \text{for some } u, \phi(u) = m \text{ and } [\overline{\text{v}}] \text{ encodes } u. \) We use the notation \([\overline{\text{v}}]\) to contrast with the \([\overline{\text{v}}]\) used before.

Define \([\overline{\text{v}}] = m, \) if for some \(k,\)

1. \([\overline{\text{v}}] = k,\)
2. \(k\) is the encoding of some Herbrand term \(t\) (i.e., \([t] = k,\)
3. there is no term \(t'\) with \([t'] < k\) such that \(I \models t = t',\)
4. the Herbrand terms \(t'\) with \([t'] < k\) take on \(m\) distinct values in \(I.\)

If the conditions above do not hold, then \([\overline{\text{v}}]\) is undefined.

Consider the example given above, where we have \(a = f(a)\) but \(a, b,\) and \(f(b)\) are all distinct. It is easy to check that \([a] = 0, [b] = 1, [f(a)] = 5, [f(b)] = 8.\) Thus, if \([\overline{\text{v}}_0] = 0, [\overline{\text{v}}_1] = 1, [\overline{\text{v}}_2] = 2, [\overline{\text{v}}_3] = 5,\) and \([\overline{\text{v}}_8] = 8,\) then \([\overline{\text{v}}_0] = 0, [\overline{\text{v}}_1] = 1, [\overline{\text{v}}_2]\)

is undefined, \([\overline{\text{v}}_8]\) is undefined (since there is a Herbrand term, namely \(a,\) with \([a] < [f(a)]\) but \(I \models a = f(a)\) by assumption), and \([\overline{\text{v}}_2] = 2.\)
begin local \( \text{init, } x' \);
\( \text{init} := x; \)
\( x' := x; \)

after each step of \( Q(x') \) do \( R'(\overline{x}, x', \text{init}) \);
end

where \( R'(\overline{x}, x', \text{init}) \) is

begin local \( \overline{y}_1, \overline{y}_2, \overline{y}', \text{enc, enc', d, d', e, e'} \);
if \( x \not= x' \) then begin
  /* check if \( x' \) encodes a Herbrand term */
  HRBD(\( x', \text{enc', d'} \));
  if \( \text{enc} = \text{tt} \land d \not= d' \) then begin
    HRBD(\( x, \text{enc, d} \));
    \( \overline{y}_1 := \text{init} \); \\
    \( \overline{y}_2 := \text{init} \); \\
    \( e := a; \)
  end;
end;

/* check if \( d' \), the Herbrand term encoded by \( x' \), has occurred previously in the list of Herbrand terms */
while \( e \not= d \lor e \not= d' \) do begin
  \( \overline{y'} := \overline{y}_1 \); \\
  ONEMORESTEPQ(\( \overline{y}, \overline{y}', \overline{y}_2 \)); \\
  HRBD(\( \overline{y}, \text{enc, e'} \));
  if \( \text{enc} = \text{tt} \) then \( e := e' \);
end;
if \( y = x \) then \( x := x' \);
end;
end;
\)

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

1. SUC'(\( \overline{v}_1, \overline{v}_2 \)) halts iff \[ [\overline{v}_1] + 1 = [\overline{v}_2]. \]
2. ADD'(\( \overline{v}_1, \overline{v}_2, \overline{v}_3 \)) halts iff \[ [\overline{v}_1] + [\overline{v}_2] = [\overline{v}_3]. \]
3. MULT'(\( \overline{v}_1, \overline{v}_2, \overline{v}_3 \)) halts iff \[ [\overline{v}_1] \times [\overline{v}_2] = [\overline{v}_3]. \]

Suppose we could define these programs. Note that indeed \( \phi(u) = m \) iff for some \( \overline{v}, [\overline{v}] \) encodes a Herbrand term equal to \( u \), and \([\overline{v}] = m \). Thus it follows that \( \phi(u_1) + \phi(u_2) = \phi(u_3) \) iff there exist \( \overline{v}_1, \overline{v}_2, \) and \( \overline{v}_3 \) such that \([\overline{v}_1] + [\overline{v}_2] = [\overline{v}_3] \). Similar statements hold for successor and multiplication. Thus we can define \( Z(x) \) via \( x = a \) (since \( \phi(a) = 0 \)) 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 \text{true} \ of ADD' and HRBD', respectively, then we define \( A \) as
\[
A(u_1, u_2, u_3) \iff \exists \overline{v}_1, \overline{v}_2, \overline{v}_3 (\bigwedge_{i=1,2,3} W_{HRBD}(\overline{v}_i, u_i) \land W_{ADD}(\overline{v}_1, \overline{v}_2, \overline{v}_3)).
\]

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 \( Q'(\overline{x}) \) such that on input \( \overline{v}_0 \) if \( \overline{x} \) successively takes values \( \overline{v}_0, \overline{v}_1, \overline{v}_2, \ldots \), then \[ [\overline{v}_i, \overline{v}_2] = i. \] If we look at the definition of \([\overline{v}] \), 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 \( Q'(\overline{x}) \) in Figure 5.

---

**Fig. 5.** The program \( Q'(\overline{x}) \).
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 \( M_3 \) 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”; that is, if \( I \) is finite, then for all \( P \in I \) we can decide for which \( u, v \in \text{dom}(I) \) 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 \((\dagger)\) 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 \( P \in I \) and some \( x \in \text{var}(P) \) such that if we run \( P \) on some input \( \bar{u}, x \) takes on every value in \( \text{dom}(I) \). In particular, under our assumption that \( I \) is Herbrand definable, having a deterministic program which would generate all the Herbrand terms would be sufficient to remove both of these hypotheses on \( I \).

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. [7, 15]) 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 \((\dagger)\) holds. If \((\dagger)\) holds, then the procedures \( M_3, M_4, \) and \( M_5 \) defined above work perfectly well in this case. If \( I \) is weakly arithmetic, then below we show 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 \( \text{dom}(I)^{2k} \subseteq N^{2k} \) (where \( k = |\text{var}(P)| \)). Thus, by a well-known result of recursive function theory (cf. [16]), given (the code for) \( P \), we can effectively find a first-order formula \( A_P^* \) of number theory (i.e., over the type \( \{0, +, \times, \ast, \} \) such that \( N \models A_P^*(\bar{u}, \bar{v}) \iff I \models A_P(u, v) \). Similarly, we can effectively translate any formula \( U \) of type \( \Sigma \) to a formula \( U^* \) of number theory such that \( N \models U^*(\bar{u}, \bar{v}) \iff I \models U(u, v) \). 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 [16] referred to above), we can easily translate any quantifier-free formula. Using the fact that \( \text{dom}(I) \) is recursive and thus also definable, we can restrict the quantifiers to range over \( \text{dom}(I) \) in \( N \) and thereby translate any first-order formula. For example, a formula such as \( \forall x T \) of type \( \Sigma \) would be translated to \( \forall x (x \in \text{dom}(I) \Rightarrow T^*) \).

Now note that
\[
I \models U(P)V \iff \forall \bar{v}: \forall \bar{u}, \bar{v}, \bar{v}: (U^*(\bar{u}) \land A_P^*(\bar{u}, \bar{v}) \land V^*(\bar{v})),
\]
\[
I \models U(V)P \iff \forall \bar{u}: \exists \bar{v}: (A_P^*(\bar{u}, \bar{v}) \land V^*(\bar{v})).
\]
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 $\Sigma$ such that

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

The translation is a generalization of that used in Section 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 $I \vdash U\langle P \rangle V$ (respectively, $I \vdash U\langle P \rangle V$) we translate $U\langle P \rangle V$ (respectively, $U\langle P \rangle V$) to a sentence of first-order number theory as described above, translate this sentence back to a first-order sentence of type $\Sigma$, and then consult the oracle for $\text{Th}(I)$ 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. Section 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, that is, 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 ($\dagger)$), 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, [7], that it is (weakly) arithmetic and that there is a
definable predicate which encodes finite sequences of domain elements as one
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 [1] and [2] suggest that
they are often necessary. In contrast, the work of Cook [4] and Gorelick [6] shows
that encoding of control structures is unnecessary when reasoning about certain
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 pro-
gramming language provided that we consider only unbounded expressive interpre-
tations. This is precisely because of the encoding power available in such interpre-
tations 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 [2], 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. [2, 4], . . .) 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 [2].

Yet a third question concerns the relationship between the uninterpreted case
considered in [13] and [14] and the interpreted case discussed here. In [13] and [14],
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 [13]) 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 [2], 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.

Appendix A

The semantics of P. Given a trajectory τ = (σ₀, . . . , σₖ), define first(τ) = σ₀ and last(τ) = σₖ, and for trajectories τ₀ and τ₁, define

\[ τ₀ \circ τ₁ = \begin{cases} 
(σ₀, . . . , σₖ, σ'; . . .) & \text{if } τ₀ = (σ₀, . . . , σₖ), \ τ₁ = (σ₀', σ₁', . . .), \\
\text{undefined} & \text{otherwise.} 
\end{cases} \]

(1) If t is a term, cv(x := t) = \{x\}; var(x := t) = \{x\} \cup var(t);
\[ T_I(x := t) = \{(σ, σ[x/u]) | σ \neq ⊥, σ(t) = u \in \text{dom}(I)\}. \]

(2) cv(P, Q) = cv(P) \cup cv(Q); var(P, Q) = var(P) \cup var(Q);
\[ T_I(P, Q) = (τ₀ \circ τ₁ | τ₀ ∈ T_I(P), τ₁ ∈ T_I(Q)). \]

(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);
\[ T_I(if A then P else Q) = \{τ | (I, first(τ) \models A, τ ∈ T_I(P)), \text{ or } (I, first(τ) \models \neg A, τ ∈ T_I(Q)). \}

(4) cv(while A do P) = cv(P); var(while A do P) = var(P) \cup var(A);
\[ T_I(while A do P) = \bigcup_{i \geq 0} T_I(W^i); \]
where
\[ W^0 = \omega, W^{i+1} = \text{if } A \text{ then } P; W^i \text{ else SKIP}. \text{ SKIP is the program which has no effect: } T_I(\text{SKIP}) = \{(σ) | σ \neq ⊥\}, \text{ and } ω \text{ is the diverging program: } T_I(ω) = \{(σ, ⊥) | σ \neq ⊥\}; \]

(5) cv(run P until A) = cv(P); var(run P until A) = var(P) \cup var(A);
\[ T_I(\text{run P until A}) = \{τ | τ ∈ T_I(P) \mid τ = (σ₀, σ₁, . . . ), \text{ and for all } i, I, σ_i \models \neg A \} \cup \{τ | τ = (σ₀, . . . , σₖ), τ \text{ is a prefix of some } τ' ∈ T_I(P), \text{ if } i < k \text{ then } I, σ_i \models \neg A, \text{ and } σ_k = ⊥ \text{ or } I, σ_k \models A\}. \]

(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) ≠ ∅, then T_I(after each step of P do Q) = ∅. (We consider after each step of P do Q syntactically incorrect unless var(P) \cap cv(Q) = ∅; thus we do not allow the computation of Q to affect the variables of P.)
If var(P) \cap cv(Q) = ∅, T_I(after each step of P do Q) = \{τ | τ = (σ₀, σ₁, . . . ) such that for some subsequence σ₀ < σ₁ < . . . < σₖ and all j < k we have (a) σ₀ = σ₀; (b) last(τ) = σₖ; (c) if σ_{j+1} ≠ ⊥, (σ_{j+1}, . . . , σₖ) ∈ T_I(Q); (d) if y \notin var(P), σ_{j+1} ≠ ⊥, then σ_j(y) = σ_{j+1}(y); and (e) for some (σ₀', σ₁', . . . , σₖ') ∈
\( \mathcal{T}_i(P) \), we have either \( k = k' \) or \( k \leq k' \) and \( \sigma_k = \bot \), \( \sigma_0 = \sigma_0 (= \sigma_0) \), and \( \sigma_{k+1}(\operatorname{var}(P)) = \sigma_{k+1}(\operatorname{var}(P)) \).

(7) \( \operatorname{cv}(\text{begin local } x_1, \ldots, x_n; P \text{ end}) = \operatorname{cv}(P) \)
\[ \operatorname{var}(\text{begin local } x_1, \ldots, x_n; P \text{ end}) = \operatorname{var}(P); \]
\[ \mathcal{T}_i(\text{begin local } x_1, \ldots, x_n; P \text{ end}) = \{(\sigma_0, \sigma_1) \; : \tau \in \mathcal{T}_i(P) \}, \text{and } \sigma_2 = \text{last}(\tau)_i[\sigma_0(x_1), \ldots, x_n/\sigma_0(x_n)]. \]
(Thus the local variables \( x_1, \ldots, x_n \) 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) \( \operatorname{cv}(Z) = \operatorname{var}(Z) = \emptyset \) for all \( Z \in \mathsf{plab} \);
\[ \mathcal{T}_i(Z) = \mathcal{T}_i(\omega) = \{(\omega, \bot) \; : \sigma \neq \bot \} \] for all \( Z \in \mathsf{plab} \).

(2) \( \operatorname{cv}(\mu Z[P]) = \operatorname{var}(\mu Z[P]) = \operatorname{var}(P) \);
\[ \mathcal{T}_i(\mu Z[P]) = \bigcup_{i \in \omega} \mathcal{T}_i(P'), \text{where } P^0 = \omega, \text{ and } P^{i+1} = P[Z/P'] (i.e., we syntactically replace all free occurrences (where free and bound occurrence have the familiar meaning) of } Z \text{ in } P \text{ by } P'). \text{ Essentially, } \mu Z[P] \text{ acts as a least fixed point operator.} \]
Note that \( \mathcal{T}_i(\text{while } A \text{ do } P \text{ od}) = \mathcal{T}_i(\mu Z[\text{if } A \text{ then } P; Z \text{ else } \text{SKIP}]) \).

Appendix B. Proof of Lemma 4

Because \( I \) is strongly arithmetic, we may use the formulas \( Z', S', A' \) and \( M' \) to treat the elements of \( \operatorname{dom}(I) \) as integers. Thus we will be able to encode formulas of type \( \Sigma \) and sequences of domain elements by one domain element. The formula \( A_I(\hat{u}, \hat{v}) \) will be of the form

\[ \exists \, \hat{c}, \hat{t} \left[ \operatorname{Traj}_P(c, i) \land \operatorname{Trseq}(c, \hat{u}) \land \operatorname{Term}(\hat{u}, t_1, v_1) \land \cdots \land \operatorname{Term}(\hat{u}, t_n, v_n) \right], \]

where \( \operatorname{Traj}_P \), \( \operatorname{Trseq} \), and \( \operatorname{Term} \) are formulas of type \( \Sigma \). Our intention is that \( \operatorname{Traj}_P(c, i) \) will be true in \( I \) iff \( P \) has a trajectory such that \( i \) encodes terms for the values of \( c \) 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 in Section 2.2). \( \operatorname{Trseq}(c, \hat{u}) \) 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 \( \hat{u} \). \( \operatorname{Term}(\hat{u}, t_i, v_i) \) will be true iff the value of the term encoded by \( t_i \) in \( I \) and the valuation \( \hat{u} \) 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 \( A_I(\hat{u}, \hat{v}) \) will define the relation \( A_P(\hat{u}, \hat{v}) \), 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 \( \Sigma \) which makes \( I \) strongly arithmetic, we can effectively construct the formula \( \operatorname{Traj}_P(c, i) \) of type \( \Sigma \).

For the definition of \( \operatorname{Term} \) and \( \operatorname{Trseq} \), we have to discuss some techniques for encoding formulas of \( \Sigma \) as integers. In a strongly arithmetic domain we can represent finite sequences of integers by using the predicates \( \operatorname{Length}(s, x) \), which holds iff \( s \) represents a sequence of length \( x \), and \( \operatorname{Select}(s, i, y) \), 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 \( \lfloor x \rfloor \) 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 \( \Sigma \) by the addition of new bound variables. We will encode a term \( t \) as a sequence \( n_1, \ldots, n_q \), such that each element \( n_k \) is either an integer \([a] \) or \([b] \) corresponding to one of the constant symbols of \( \Sigma \), an encoding of a pair \( \langle \langle \text{var}, i \rangle, \rangle \) where \([\text{var}] = 4\), corresponding to the variable \( x_i \) in \( \text{var}(P) \), a pair \( \langle \langle f \rangle, i \rangle \), \( i < k \), or a triple \( \langle \langle g \rangle, i, j \rangle \), \( i < j < k \). In the case that \( n_k \) is a pair \( \langle \langle f \rangle, i \rangle \), the subsequence \( n_1, \ldots, n_k \) encodes the term \( f(t_i) \) where \( t_i \) is the term encoded by \( n_1, \ldots, n_i \); similarly for \( g \). Thus the complete sequence \( n_1, \ldots, n_q \) encodes the term \( t \).

With these conventions, we can define the formula \( \text{Term}(\langle u, t, y \rangle) \), 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 \( t[1], \ldots, t[i] \). (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.)

\[
\text{Term}(\langle u, t, y \rangle) = \exists w( |w| = |t| \wedge w[|w|] = y \wedge \forall i (i \leq |w| \Rightarrow (t[i] = [a] \wedge w[i] = a) \vee (t[i] = [b] \wedge w[i] = b) \wedge (t[i] = \langle \langle \text{var}, i \rangle \wedge w[i] = u_i) \wedge \ldots \wedge (t[i] = \langle \langle \text{var}, n \rangle \wedge w[i] = u_n) \wedge \exists j (j < i \wedge t[j] = \langle \langle f \rangle, j \rangle \wedge w[j] = f(w[j])) \wedge \exists k (j < k < i \wedge \wedge t[k] = \langle \langle g \rangle, j, k \rangle \wedge w[k] = g(w[j], w[k]) \))
\]

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

\[
\text{Trseq}(c, \langle u \rangle) = \forall i (i \leq |c| \Rightarrow \text{Trlit}(\langle \text{lit}(i), \langle u \rangle \rangle))
\]

\( \text{Trlit} \) uses \( \text{Term} \) 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 \( \Sigma \) and to its negation: \([\neg] = 0\), \([\neg \neg] = 1\), \([\neg A_0] = 3\), \([\neg A_0] = 4\). 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 \( \langle \neg, m_a, m_{g(a,b)} \rangle \), where \( m_a \) and \( m_{g(a,b)} \) are encodings of \( a \) and \( g(a, b) \) as described above; that is, \( m_a \) is an integer representing the sequence \( \langle i_1 \rangle \) and \( m_{g(a,b)} \) represents the sequence \( \langle [a], [b], \langle f, 1, 2 \rangle \rangle \).

\[
\text{Trlit}(\langle \text{lit}, \langle u \rangle \rangle) = \exists v_1, v_2, w_1, w_2 (\text{Term}(\langle u, v_1, w_1 \rangle) \wedge \text{Term}(\langle u, v_2, w_2 \rangle) \wedge (\text{lit} = \langle [=], v_1, v_2 \rangle \wedge w_1 = w_2) \vee (\text{lit} = \langle [\neg =], v_1, v_2 \rangle \wedge \neg(w_1 = w_2)) \vee (\text{lit} = \langle [A_0], v_1, v_2 \rangle \wedge A_0(w_1, w_2)) \vee (\text{lit} = \langle [\neg A_0], v_1, v_2 \rangle \wedge \neg A_0(w_1, w_2)))
\]

This completes our encoding. \( \square \)
ACKNOWLEDGMENT. We would like to thank H. Langmaack for his careful reading of the paper and for finding some technical errors in the proof of Theorem 2.

REFERENCES
(Note. References [8, 10] are not cited in the text.)
1. BERGEStRA, J., CHMIELINSKA, A., AND TiuRYN, J. Hoare’s logic is incomplete when it does not have to be. In IBM Workshop on Logics of Programs, Lecture Notes in Computer Science 131, D. Kozen, Ed., Springer-Verlag, New York, 1982, pp. 9–23.
2. CLARKE, E.M. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. J. ACM 26, 1 (Jan. 1979), 129–147.
12. MEYER, A.R. Notes on Lipton’s generalization of the theorems of Cock and Clarke on expressiveness. Privately circulated notes.

RECEIVED DECEMBER 1981; REVISED JUNE 1982; ACCEPTED JULY 1982