Inadequacy of Computable Loop Invariants

ANDREAS BLASS
University of Michigan
and
YURI GUREVICH
Microsoft Research

Hoare logic is a widely recommended verification tool. There is, however, a problem of finding easily checkable loop invariants; it is known that decidable assertions do not suffice to verify `while` programs, even when the pre- and postconditions are decidable. We show here a stronger result: decidable invariants do not suffice to verify single-loop programs. We also show that this problem arises even in extremely simple contexts. Let $\mathcal{N}$ be the structure consisting of the set of natural numbers together with the functions $S(x) = x+1, D(x) = 2x, H(x) = \lfloor x/2 \rfloor$. There is a single-loop program Π using only three variables x, y, z such that the asserted program $x = y = z = 0$ Π `false` is partially correct on $\mathcal{N}$ but any loop invariant $I(x, y, z)$ for this asserted program is undecidable.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Assertion, Hoare logic, automated deduction, automated reasoning, loop invariants, precondition, postcondition, computable, uncomputable, recursive inseparability

1. INTRODUCTION

Hoare logic [Hoare 1969] is a system for proving *asserted programs*, i.e., expressions of the form $\phi\ \Pi\ \psi$ where Π is a program and ϕ and ψ are assertions about possible states of Π. The meaning of the asserted program $\phi\ \Pi\ \psi$ (under the so-called partial correctness interpretation, the only interpretation we shall consider in this paper) is that, if Π is started in a state satisfying ϕ and if this computation of Π terminates, then the final state satisfies ψ. Among the deductive rules of Hoare logic, the most important one for this paper is the *iteration rule* for deducing properties of a `while`-loop:

$$\frac{(\phi \wedge g)\ \Pi\ \phi}{\phi\ (\texttt{while}\ g\ \texttt{do}\ \Pi)\ (\phi \wedge \neg g)}.$$

A. Blass's research was partially supported by a grant from Microsoft Corporation.
Authors' addresses: A. Blass, Mathematics Department, University of Michigan, Ann Arbor, MI 48109–1109; email: ablass@umich.edu; Y. Gurevich, Microsoft Research, One Microsoft Way, Redmond, WA 98052; email: gurevich@microsoft.com.

Here the assertion ϕ serves as a *loop invariant*; that is, if true initially then it remains true after each execution of the body of the loop.

We shall also have occasion to refer to the *composition rule* for programs $\Pi_0; \Pi_1$ obtained by sequential composition

$$\frac{\phi\ \Pi_0\ \theta, \qquad \theta\ \Pi_1\ \psi}{\phi\ \Pi_0; \Pi_1\ \psi}$$

and the *consequence rule*

$$\frac{\phi \to \phi' \qquad \phi'\ \Pi\ \psi' \qquad \psi' \to \psi}{\phi\ \Pi\ \psi}.$$

It is possible for the proof of an asserted program $\phi\ \Pi\ \psi$ to make use of assertions considerably more complicated than ϕ and ψ. These more complicated assertions could be used as the loop invariant in the iteration rule, as the intermediate formula θ in the composition rule, or as the formulas ϕ' and ψ' in the consequence rule. Apt et al. [1979] proved that such use of more complicated intermediate assertions cannot always be avoided. Specifically, they proved the existence of two programs Π_0 and Π_1 such that the asserted program `true` $\Pi_0; \Pi_1$ `false` is correct (i.e., $\Pi_0; \Pi_1$ has no terminating computations) but any proof of it in Hoare logic requires an application of the composition rule with an intermediate assertion θ that defines a nonrecursive set.

It is reasonable to expect that, if such complications arise in the use of the composition rule, then they should also arise in the use of the iteration rule, simply because `while` is a more complicated concept than sequential composition. We shall show that this expectation is realized in considerably simpler situations than the one described in Apt et al. [1979]. Specifically, our construction uses a program of a very simple syntactic form[1], operating in a structure that includes only a weak (and decidable) fragment of arithmetic. Furthermore, the precondition in our construction uniquely specifies the initial state. (In contrast, the example in Apt et al. [1979] depended crucially on the availability of many states satisfying the precondition.)

Before stating our result precisely, we need some preliminary definitions and observations.

We refer to the original papers [Hoare 1969; Cook 1978] or to Apt's survey [Apt 1981] for background material on Hoare logic and Cook's completeness theorem, mentioning here only the facts that we shall specifically need later. We consider Hoare logic for `while` programs, often referring to them simply as programs. These programs Π are built from assignment statements $x := t$ (where x is a variable and t a term) by means of sequential composition $\Pi_0; \Pi_1$, conditional branching `if` g `then` Π_0 `else` Π_1, and iteration `while` g `do` Π_0. The guards g are assumed to be quantifier-free formulas of first-order logic. A program Π works in the context of a structure (in the usual sense of first-order logic) for a vocabulary that includes the function and predicate symbols occurring in the terms in Π's assignment statements and the guards in Π's conditional and iteration statements. The operation of Π

[1] Actually the motivation for this work came from the theory of abstract state machines (ASM) [Gurevich 2000]. A run of a sequential ASM can be regarded as a loop whose body contains no loops. This corresponds to the simple syntactic form involved in our main theorem.

consists of changing the values of the variables that occur in Π, so a state can be regarded as simply an assignment of values to these variables.

In asserted programs $\phi\ \Pi\ \psi$, the pre- and postconditions ϕ and ψ are from some logical system; usually one uses first-order logic here, but we have advocated in Blass and Gurevich [1987; 2000] the use of existential fixed-point logic instead. The choice of logic will be immaterial in this paper, as long as very simple quantifier-free formulas are included.

As indicated above, an asserted program $\phi\ \Pi\ \psi$ is *correct* for a class of structures if, for every structure in this class and every initial state making ϕ true, if the run of Π terminates then the final state makes ψ true. The choice of a class of structures affects the deductive apparatus of Hoare logic only in the consequence rule, where the premises $\phi \to \phi'$ and $\psi' \to \psi$ are to be admitted if and only if they are valid in the chosen class of structures.

In this paper, we shall be concerned with the class containing just the following version of the standard model of arithmetic, $\mathcal{N} = \langle \mathbb{N}, 0, S, D, H, \dots \rangle$, where $\mathbb{N}$ is the set $\{0, 1, 2, \dots\}$ of natural numbers and the functions S, D, H are defined on $\mathbb{N}$ by[2]

$$S(x) = x + 1, \qquad D(x) = 2x, \qquad H(x) = \left\lfloor \frac{x}{2} \right\rfloor,$$

and where the $\dots$ in $\mathcal{N}$ refers to unspecified additional functions and predicates. There need not be any of these additional functions and predicates, so our results will apply in particular to the very weak structure $\mathcal{N}^- = \langle \mathbb{N}, 0, S, D, H \rangle$; see Remark 2 below for more about the choice of structure.

We shall also deal only with programs of an unusually simple form. By a *single-loop* program, we mean a program of the form `while` g `do` Π_0 where the body Π_0 of the loop does not contain `while`. Notice that the definition does not require merely that the `while` construct occurs only once but also that it is the program's outermost construct.

THEOREM 1. *There is a single-loop program* Π *using only three variables* x, y, z *and the vocabulary of* $\mathcal{N}^-$ *such that*

(1) *the asserted program* $x = y = z = 0\ \Pi$ `false` *is correct in* $\mathcal{N}$ *and*

(2) *any proof of* $x = y = z = 0\ \Pi$ `false` *in Hoare logic for* $\mathcal{N}$ *uses a loop invariant that is undecidable.*

The proof of the theorem involves the following fact (Lemma 5) which may be of interest in its own right. There exists a recursive function f on strings such that the following two sets are disjoint but recursively inseparable: the set of strings reachable from a certain initial string by iterated application of f, and the set of strings from which a certain final string is reachable by iterated application of f.

To place the theorem in its proper context, we should point out several things about loop invariants. First, they always exist as relations. One can simply take the set of states reachable by repeated execution of the loop's body starting with a state satisfying the precondition. Second, for sufficiently powerful vocabulary and logic, loop invariants are expressible by formulas and therefore usable in Hoare

[2] The letters S, D, H refer to successor, double, and half.

logic proofs. Indeed, this is the point of the expressivity hypothesis in Cook's completeness theorem [Cook 1978]. Third, if the precondition is recursive (or only recursively enumerable) then the loop invariant can be taken to be recursively enumerable. This is clear from the "reachability" formulation of the loop invariant. It is also a consequence of the facts that expressivity is automatic (for any vocabulary, language, and structures) if one works with existential fixed-point logic [Blass and Gurevich 2000] and that, in the standard model of Peano arithmetic, existential fixed-point formulas define exactly the recursively enumerable relations.

Our theorem is independent of the choice of logical system, for it refers to the intrinsic complexity of the loop invariant, not to its expressibility in any specific formal system. It tells us that, although the loop invariant can be taken to be recursively enumerable, it cannot be taken to be recursive, even in very simple situations.

Remark 2. The particular functions in $\mathcal{N}^-$ were chosen because they arise naturally from our proof and because they constitute a very weak fragment of arithmetic. Their role is just to provide the vocabulary used by the Π of the theorem.

As an indication of the weakness of this fragment of arithmetic, we mention that its first-order theory is decidable. Indeed, 0, S, D, and H are all first-order definable in the structure $\langle \mathbb{N}, + \rangle$, whose first-order theory, called Presburger arithmetic, is well known to be decidable.

The proofs of $x = y = z = 0$ Π `false` in the theorem are allowed to use much richer vocabularies. For example, they could use the vocabulary of Peano arithmetic, which contains 0, S, addition, multiplication, and the order relation $<$. With this vocabulary, $\mathcal{N}$ satisfies the expressivity hypothesis in Cook's completeness theorem, and so this theorem guarantees the existence of proofs of $x = y = z = 0$ Π `false`.

The usual formulation of Peano arithmetic does not include symbols for the functions D and H that we have included in $\mathcal{N}^-$. In the case of D, this does not matter, since it is definable by a term in Peano arithmetic: $D(x) = x + x$. In the case of H, we still have definability, but not by a term. As a result, the program Π of our theorem could be translated into the language of Peano arithmetic, but at the cost of including a subroutine to compute H. Although writing such a subroutine is trivial, its presence in Π would mean that this is no longer a single-loop program. We have not attempted to avoid H and thus to prove the theorem with a single-loop program in the vocabulary of Peano arithmetic. One of the referees suggested that, by using two-counter automata instead of Turing machines in Section 4 below, one could replace D and H with the predecessor function. But note that the predecessor function is also not given by a term in the standard formulation of Peano arithmetic.

The rest of this paper is devoted to the proof of the theorem.

2. PROOFS OF `WHILE` STATEMENTS

Before beginning the construction of the program Π promised by the theorem, we consider in general terms what a proof P of an asserted program of the form

$$\phi \ (\texttt{while}\ g\ \texttt{do}\ \Pi_0)\ \psi$$

can look like. This will suggest what sort of Π can satisfy conclusion (2) of the theorem.

Of all the rules of Hoare logic for `while` programs, as listed in Hoare [1969], Cook [1978], and Apt [1981], only the iteration rule and the consequence rule have conclusions of the required form. The others have conclusions of the form $\phi\ \Theta\ \psi$ where Θ is an assignment or a sequential composition or a conditional statement. So our proof P must end with an application of either the iteration rule or the consequence rule.

Suppose, for a moment, that it ends with an application of the consequence rule. So this last step has the form

$$\frac{\phi \to \phi' \qquad \phi'\ (\texttt{while}\ g\ \texttt{do}\ \Pi_0)\ \psi' \qquad \psi' \to \psi}{\phi\ (\texttt{while}\ g\ \texttt{do}\ \Pi_0)\ \psi}.$$

Then the immediately preceding step must again be a use of the iteration rule or the consequence rule. If it is a use of the consequence rule, then the same argument can be applied again, and we can continue this way until we finally reach (because the proof P is finite) a use of the iteration rule. Thus, the proof P must have a final segment consisting of a use of the iteration rule followed by a number (possibly zero) of uses of the consequence rule.

If there are two or more uses of the consequence rule, they can be reduced to one by repeated use of the observation that

$$\frac{\phi' \to \phi'' \qquad \phi''\ \Pi\ \psi'' \qquad \psi'' \to \psi'}{\phi'\ \Pi\ \psi'}$$

followed by

$$\frac{\phi \to \phi' \qquad \phi'\ \Pi\ \psi' \qquad \psi' \to \psi}{\phi\ \Pi\ \psi}$$

can be combined into a single step

$$\frac{\phi \to \phi'' \qquad \phi''\ \Pi\ \psi'' \qquad \psi'' \to \psi}{\phi\ \Pi\ \psi}.$$

Thus, we may assume that P ends with a use of the iteration rule followed by at most one use of the consequence rule.

In fact, we may assume "exactly one" rather than "at most one," since we can append a trivial application of the consequence rule where ϕ' and ψ' coincide with ϕ and ψ. Notice also that the preceding simplifications of P do not change the loop invariant used in the iteration rule. Thus, as far as Theorem 1 is concerned, we may assume without loss of generality that proofs of $x = y = z = 0\ (\texttt{while}\ g\ \texttt{do}\ \Pi_0)\ \texttt{false}$ have, as the last two steps,

$$\frac{(\phi \wedge g)\ \Pi_0\ \phi}{\phi\ (\texttt{while}\ g\ \texttt{do}\ \Pi_0)\ (\phi \wedge \neg g)}$$

followed by

$$\frac{x = y = z = 0 \to \phi \qquad \phi\ (\texttt{while}\ g\ \texttt{do}\ \Pi_0)\ (\phi \wedge \neg g) \qquad (\phi \wedge \neg g) \to \texttt{false}}{x = y = z = 0\ (\texttt{while}\ g\ \texttt{do}\ \Pi_0)\ \texttt{false}}.$$

This use of the consequence rule requires that the two implications occurring in it as premises are true in $\mathcal{N}$. In other words, the loop invariant ϕ must be true in the initial state, i.e., it is satisfied in $\mathcal{N}$ when $x = y = z = 0$, and ϕ must imply the guard formula g in $\mathcal{N}$.

Furthermore, the use of the iteration rule requires that its premise, $(\phi \wedge g)\ \Pi_0\ \phi$, be correct in $\mathcal{N}$. Since we have just seen that ϕ must imply g, we can infer that $\phi\ \Pi_0\ \phi$ must be correct. (Here, as always in this paper, "correct" refers to the partial correctness interpretation, but in the present context total correctness would work as well. Since Π_0 does not involve `while`, it will certainly terminate.)

Thus, ϕ must be true in the initial state $x = y = z = 0$ and in every state reachable from it by repeated application of Π_0; we call such states *positive*. In addition, ϕ must be false in any state where g fails (i.e., any state where Π would halt) and therefore also in any state from which a $\neg g$ state could be reached by repeated use of Π_0; we call such states *negative*. Thus, the loop invariant ϕ must be true in all positive states and false in all negative ones. Since a state can be identified with a triple of numbers (the values of x, y, z), the positive states and the negative states constitute two ternary relations on $\mathbb{N}$, which we designate as $+(\Pi_0)$ and $-(\Pi_0, g)$, respectively. (The notation reflects the fact that the definition of negative states involves the guard g but the definition of positive states does not.)

LEMMA 3. *Suppose Π_0 is a program and g is a guard, both using only the variables x, y, z and the vocabulary of $\mathcal{N}^-$. Suppose further that Π_0 does not use* `while`. *Finally, suppose that the relations $+(\Pi_0)$ and $-(\Pi_0, g)$ are disjoint but recursively inseparable*[3]. *Then the program* `while` g `do` Π_0 *serves as the Π required in Theorem 1.*

PROOF. Clearly, `while` g `do` Π_0 is a single-loop program using only the variables and vocabulary allowed in Theorem 1. The disjointness of the sets of positive and negative states means that Π, started with $x = y = z = 0$, will never halt, so the asserted program $x = y = z = 0\ \Pi$ `false` is correct. Finally, the discussion preceding the lemma shows that any proof of this asserted program in Hoare logic must use a loop invariant ϕ that is true in all positive states and false in all negative ones. By the final hypothesis of the lemma, the relation defined by ϕ cannot be recursive. □

From now on, our goal will be to find Π_0 and g satisfying the hypotheses of the lemma.

Remark 4. For Π_0 and g as in the hypotheses of the lemma, or indeed for any program defining a recursive function and any guard defining a recursive relation, both $+(\Pi_0)$ and $-(\Pi_0, g)$ are recursively enumerable. So we shall be dealing with recursively enumerable but recursively inseparable sets.

3. SOME RECURSIVELY INSEPARABLE EXAMPLES

In this section, we construct an approximation to what Lemma 3 requires. We shall not (yet) be concerned about the exact nature of the program and the guard, nor even about the underlying structure. But apart from these points we shall obtain the essential structure required in Lemma 3.

[3]Two r-ary relations A and B on the set of natural numbers are *recursively inseparable* if there is no recursive r-ary relation C which includes A and is disjoint from B. This concept was introduced in Trakhtenbrot [1953] and widely used in undecidability proofs in logic; see for example Börger et al. [1997]. It has been applied to Hoare logic in the already mentioned paper Apt et al. [1979] and in Bergstra and Tucker [1982], but not in a way directly relevant to our theorem.

LEMMA 5. *There exists a (total) recursive function f on strings over a certain finite alphabet, such that the following two sets are disjoint but recursively inseparable:*

—*the set $+(f)$ of strings reachable from a certain initial string by iterated application of f*

—*the set $-(f)$ of strings from which a certain final string is reachable by iterated application of f.*

PROOF. We use the well-known fact that the set A of sentences provable in Peano Arithmetic and the set B of sentences refutable in Peano arithmetic are recursively enumerable and recursively inseparable; see for example Shoenfield [1967, Chapter 6, Exercise 13]. The only information we shall use about A and B is that they are recursively enumerable, recursively inseparable sets of sentences in a finite vocabulary, closed under logical equivalence. (In fact, we need only a little bit of closure under logical equivalence, namely that either of these sets contains a sentence θ if and only if it contains $\neg\neg\theta$.) Thus, we could equally well have used Trakhtenbrot's theorem [Trakhtenbrot 1953] that, for nontrivial vocabularies, the set of logically valid sentences is recursively inseparable from the set of sentences falsifiable in finite structures.

We regard sentences as strings over a finite alphabet. The use of infinitely many variables in first-order logic causes no problem here, as we can regard a variable as the letter v followed by a number written in binary notation.

For A and B as above, fix an algorithm enumerating A, an algorithm enumerating B, and an algorithm enumerating all strings of symbols in our alphabet. We assume that each of these algorithms enumerates at most one new element at any stage. We also assume that, whenever any one of them enumerates a sentence of the form $\neg\neg\theta$, it has already enumerated θ. This is easily arranged; just modify the algorithms so that, whenever they are about to enumerate $\neg\neg\theta$ they first enumerate θ if they haven't already done so. Let a_0 and b_0 be the first strings in the enumerations of A and B, respectively. These will be the initial string and the final string in the statement of the lemma. We shall define a recursive function f so that the strings reachable from a_0 by iterated application of f will be exactly those in A, while the strings from which b_0 is reachable will be exactly those in B. This will clearly suffice to establish the lemma.

We give an algorithm defining f in stages. At each stage, the value of $f(x)$ will have been defined for only finitely many strings x. In this situation, to "iterate f as long as possible starting at x" means to form the sequence $x, f(x), f^2(x) = f(f(x)), \ldots$, until reaching either a value not yet in the domain of f or b_0. This makes sense because of the first of the following four inductive hypotheses, which we shall ensure at each stage of our construction.

(1) The only cycle in f is at $f(b_0) = b_0$. In other words, for any positive integer n, $f^n(x) = x$ implies $x = b_0$.

(2) Only elements of A ever become reachable from a_0. Conversely, any element enumerated into A at any stage becomes reachable from a_0 at the same stage if not earlier.

(3) The only elements from which b_0 ever becomes reachable are those of B. Conversely, if b is enumerated into B at some stage, then b_0 becomes reachable from b at the same stage if not earlier.

(4) If a string x is not (yet) reachable from a_0, if b_0 is not (yet) reachable from x, and if $f(x)$ is defined, then $f(x)$ is $\neg\neg x$.

Begin the definition of f by setting $f(b_0) = b_0$. Then do the following at each stage n, for $n = 0, 1, 2, \ldots$.

First, see whether a new element has been enumerated into A at stage n. If so, if this element is θ, and if θ is not yet reachable from a_0, iterate f as long as possible starting at a_0, and let x be the final element in the resulting sequence. (Note that (1) is crucial here.) Being reachable from a_0, x will be in A, so $x \neq b_0$, and therefore $f(x)$ is not yet defined. Set $f(x) = \theta$. Thus, we ensure that the element θ of A is reachable from a_0.

Second, see whether a new element has been enumerated into B at stage n. If so, and if this element is ζ, iterate f as long as possible starting at ζ, and let y be the final element in the resulting sequence. (Again, (1) is used.) If $y = b_0$, do nothing (as b_0 is already reachable from ζ). Otherwise, every element of this iteration, from ζ to y, is, by inductive hypothesis (4), a sentence logically equivalent to ζ and therefore in B. Set $f(y) = b_0$. Thus, we ensure that b_0 is reachable from the element ζ of B.

Third, see whether a new string has been enumerated at stage n by the algorithm that lists all the strings. If so, if this string is z, and if $f(z)$ is not yet defined, then set $f(z) = \neg\neg z$.

This completes the description of stage n in the construction of f. It is easy to check that our four inductive hypotheses are preserved at every step. Of the three parts of stage n, the first ensures that each element of A eventually becomes reachable from a_0; the second ensures that b_0 eventually becomes reachable from each element of B; and the third ensures that f is a total function. Since the entire construction is algorithmic, f is a recursive function, and the lemma is proved. □

The preceding proof could be modified so that a_0 and b_0 are any two specified strings and so that f is one-to-one except that two strings (b_0 and one other one) map to b_0.

4. CONVERSION TO ARITHMETIC

In this section, we convert Lemma 5 into a form that supplies the hypotheses of Lemma 3 and thus completes the proof of Theorem 1. The conversion uses two familiar ideas: convert an arbitrary algorithm into a Turing machine, and describe the operation of a Turing machine in arithmetic.

LEMMA 6. *There is a Turing machine M, operating on a two-way infinite tape, using only two symbols, 0 (the blank) and 1, with two distinguished control states q_0 and q_1, and having the following properties:*

(1) In state q_1, the machine halts.

(2) If started on a blank tape, in state q_0, the machine M computes forever (therefore without ever entering state q_1).

(3) *The set of configurations reached by M during the computation in property (2) is recursively inseparable from the set of configurations from which M would eventually halt.*

PROOF. Consider the function f of Lemma 5. It acts on strings over a certain finite alphabet. Transform it to act on binary strings via a coding function. Specifically, assign to each symbol from the finite alphabet a code that is a positive integer (different integers for different symbols). Then encode strings of symbols by writing the code of each symbol in unary notation (a string of 1's) and separating these by single 0's. Via this coding, f becomes a recursive function on binary strings. (It should be defined arbitrarily but recursively on binary strings that do not code strings over the original alphabet.)

Now let M be a Turing machine that does the following. When started in control state q_0, it writes the binary code of a_0 on its tape and then goes into control state q_2, scanning the left end of this code. (Since 0 is the blank symbol, there are also infinitely many 0's covering the unused part of the tape.) In control state q_1, M halts. In control state q_2, it begins computing f repeatedly. That is, it computes f of the current tape contents, but from the halting state of this subcomputation it proceeds as follows: if the current tape contents are essentially b_0 (see below for "essentially") then it goes to state q_1; otherwise, it returns to state q_2 scanning the essential left end of the tape contents, for another round of f. Here "essentially" b_0 means the following. M should check, starting from its presently scanned square and going both to the left and to the right until encountering two consecutive 0's, whether this segment of the tape contains the binary code of b_0. If so, then we say the tape contents are essentially b_0. They may not be exactly b_0, because there may be some 1's beyond the double-zeros, but then the tape contents are not the binary code of anything; so we need not worry about this situation. Similarly, the "essential" left end of the tape means the point just to the right of the first double-zeros on the left.

The first of the lemma's three conclusions is clear by definition, and the second follows immediately from the fact that b_0 is not reachable from a_0 by iteration of f. It remains to prove the third conclusion. So suppose we had a recursive set separating the configurations reachable from the initial configuration (state q_0 and empty tape) from the configurations that eventually lead to halting. Then we would have, by a one-to-one reduction to this set, a recursive set separating $+(f)$ from $-(f)$. Indeed, if g is the recursive function assigning to each string x over the original alphabet (used by f) the configuration consisting of the binary code of x, the control state q_2, and the scanned square at the left end of the code, then

—if $x \in +(f)$ then $g(x)$ is reachable by M from the initial configuration
—if $x \in -(f)$ then a halting state is reachable by M from $g(x)$.

Because $+(f)$ and $-(f)$ are, by assumption, recursively inseparable, we have a contradiction, and the proof of the lemma is complete. □

Finally, we apply a standard procedure for arithmetically encoding Turing machine configurations and instructions. Number the states of M with natural numbers, so that q_0 and q_1 are represented by 0 and 1, respectively. Then any configuration of M can be represented by a triple of natural numbers (x, y, z). Here x is the number of the state, and y is the contents of the tape to the left of the

scanned square and including this square; the contents are interpreted as the binary notation for a natural number, the scanned square being the least significant digit. (Thus the infinitely many blanks stretching to the left have no effect on y.) Finally z is the contents of the tape strictly to the right of the scanned square, read, from right to left, as the binary notation for a natural number, so that the square to the right of the scanned square is the least significant digit. With these conventions, the initial configuration of M, where the tape is empty and the control state is q_0, is represented by $x = y = z = 0$.

The instructions of M can easily be written as a program Π_0 of the sort required in Lemma 3. For example, an instruction saying "if the control state is 3 and the scanned square contains 1 then change the 1 to 0, move to the left, and enter control state 4" would become

$$\begin{array}{l}\texttt{if } x = SSS0 \wedge \neg(y = DHy)\\ \texttt{then } x := SSSS0; y := Hy; z := Dz\\ \texttt{else } \ldots .\end{array}$$

Here S, D, and H are as in $\mathcal{N}^-$ in Theorem 1, and the ... at the end is where M's other instructions would be filled in. It should be clear that all Turing machine instructions admit similar translations, so that the operation of a single step of M can be expressed by a program Π_0. This program and the guard $x \neq 1$ (expressing that M has not entered the halting state q_1) satisfy the hypotheses of Lemma 3. By that lemma, this completes the proof of the theorem.

REFERENCES

Apt, K. R., "Ten years of Hoare's logic: A survey — Part 1," ACM Transactions on Programming Languages and Systems 3:4 (October 1981), 431–483.

Apt, K. R., Bergstra, J. A., and Meertens, L. G. L. T., "Recursive assertions are not enough — or are they?," Theoret. Comp. Sci. 8 (1979), 73–87.

Apt, K. R. and Olderog, E.-R., "Verification of sequential and concurrent programs," Springer-Verlag, 1991.

Bergstra, J. A. and J. V. Tucker, J. V., "Some natural structures which fail to possess a sound and decidable Hoare-like logic for their while-programs," Theoret. Comput. Sci. 17 (1982) 303–315.

Blass, A. and Gurevich, Y., "Existential fixed-point logic," In "Logic and Complexity" (ed. E. Börger), Springer Lecture Notes in Computer Science 270 (1987), 20–36.

Blass, A. and Gurevich, Y., "The underlying logic of Hoare logic," Bull. European Assoc. Theoret. Comp. Sci. To appear.

Börger, E., Grädel, E., and Gurevich, Y., "The classical decision problem," Springer-Verlag, 1997.

Cook, S. A., "Soundness and completeness of an axiom system for program verification," SIAM Journal of Computing 7:1 (1978), 70–90.

Gurevich, Y., "For every sequential algorithm there is an equivalent sequential abstract state machine," ACM Transactions on Computational Logic 1:1 (July 2000).

Hoare, C. A. R., "An axiomatic basis for computer programming," Communications of ACM 12:10 (October 1969), 576–580, 583.

Shoenfield, J. R., "Mathematical Logic," Addison-Wesley, 1967.

Trakhtenbrot, B. A., "On Recursive Separability," Doklady Acad. Nauk SSSR 88 (1953), 953–955, in Russian.

Received March 2000; revised June 2000; accepted June 2000