First Order Programming Logic

Robert Cartwright
Department of Computer Science
Cornell University

John McCarthy
Department of Computer Science
Stanford University

January, 1979

Abstract

First Order Programming Logic is a simple, yet powerful formal system for reasoning about recursive programs. In its simplest form, it has one major limitation: it cannot establish any property of the least fixed point of a recursive program which is false for some other fixed point. To rectify this weakness, we present two intuitively distinct approaches to strengthening First Order Programming Logic and prove that either extension makes the logic relatively complete. In the process, we prove that the two approaches are deductively equivalent; they prove exactly the same set of theorems. The relative completeness of the extended logic is significant because it suggests it can establish all "ordinary" properties (obviously we cannot escape the Godelian incompleteness inherent in any programming logic) of recursive programs including those which compute partial functions.

1 Introduction

Many theoretical computer scientists (*e.g.*, Hitchcock and Park [HP 73]) have dismissed first order logic as too weak a formalism for reasoning about recursively defined functions. Nevertheless, the authors [Cart 76a; Cart 76b; McC 78] have demonstrated that first order logic can serve as a powerful, yet convenient formal system for establishing properties of recursively defined functions.

The key idea underlying our formal system is that recursive definitions of partial functions can be interpreted as equations extending a first order theory of the program data domain. The resulting programming logic (henceforth called First Order Programming Logic) is very simple and convenient to use; yet, it is sufficiently powerful to prove most theorems of practical interest about recursively defined functions. In fact, First Order Programming Logic seems to be the logic of choice for reasoning about recursive programs. The most recent incarnation of the Boyer-Moore LISP verifier [BM 75] is based on a restricted variant of First Order Programming Logic.

From a theoretical viewpoint, the major weakness of ordinary First Order Programming Logic is its inability to prove true statements about the least fixed point solution of a recursive program which are false for some other fixed point solution. For recursive programs that compute total functions this limitation is irrelevant, since the least fixed point is the only fixed point. For example, proving that a particular recursive program computes a total function (e.g. Ackermann's function) is remarkably simple in most cases. Nevertheless, the few recursive programs encountered in practice which do not compute total functions (e.g. interpreters) may have simple properties which cannot be established within ordinary First Order Programming Logic. To remove this limitation, the authors [Cart 78; McC 78] have independently developed extensions to First Order Programming Logic which use different techniques to capture the concept of least fixed-point.

In this paper, we prove that either extension to First Order Programming Logic makes the logic relatively complete with respect to the theory of the underlying data domain. In the process, we prove that the two extensions are interderivable, i.e. that either extension can be formally derived from the other. The relative completeness of the extended logic is significant because it indicates that the logic can prove all "ordinary" properties of partial functions computed by recursive programs. Of course, the extended logic cannot escape the essential Goedelian incompleteness of any non-trivial programming logic. But the fact that the extended logic is relatively complete demonstrates that the logic completely captures the semantics of recursive programs—that the only source of incompleteness in the logic lies in the first order axiomatization of the underlying data domain. Fortunately, we can overcome the incompleteness of the data domain axiomatization when necessary by following the same approach that Gentzen [Gent 36] used to prove the consistency of Peano arithmetic (Peano's Axioms); we simply augment the axiomatization of the data domain when necessary by the appropriate transfinite induction axiom.

An interesting consequence of the inter-derivability of the two extensions to ordinary First Order Programming Logic is that both extensions can be added to the unadorned logic without interference. In practice, First Order Programming Logic with both extensions is probably more convenient to use than it is with either extension alone.

The second contribution of this paper is to show that First Order Programming Logic is applicable to iterative programs (e.g. PASCAL programs) as well. Recently, the intermittent assertions method developed by Burstall [Bur 74] and Manna and Waldinger [MaWa 78] has attracted considerable attention as a possible alternative to the standard inductive assertions method for reasoning about iterative programs. However, all of the published descriptions of the method have been very informal. In this paper, we formalize the intermittent assertions method by showing that it can be interpreted simply as sugared First Order Programming Logic applied to the recursive translations of iterative programs.

2 Recursive Programs

An adequate background for this paper is contained in Enderton's text on mathematical logic [End 72] and Manna's text on the mathematical theory of computation [Manna 1974]. Before we can formally define the concept of a recursive program, we must introduce some preliminary definitions. A *data domain* D consists of a set $|D|$ of data objects (called the *universe*) together with a set of primitive operations (functions) $g_1, \ldots, g_k$ on $|D|$. Constants are treated as nullary operations. While all our results easily generalize to typed (sorted) data domains, we will restrict ourselves to typeless (unsorted) data domains for the sake of notational and conceptual simplicity. Consequently, we impose the restriction that all primitive operations must be total functions on the universe $|D|$ (of the appropriate arity).

To accommodate Boolean operations within D, we can represent the Boolean values **TRUE** and **FALSE** by ordinary data objects in $|D|$. Since all operations must be total on $|D|$, we must associate a Boolean value with every data object in $|D|$. The programming language LISP follows this convention. In LISP, the data domain consists of the single type S-expression. The atom NIL represents the Boolean value **FALSE**; every other S-expression represents the Boolean value **TRUE**. The atom T serves as the nominal representative of the set of objects denoting **TRUE**; most Boolean operations (e.g. **equal**) always use T to represent a **TRUE** result.

To enforce these conventions, we assume that every data domain D has the following properties:

1. The universe $|D|$ is partitioned into two non-empty disjoint sets $|D|_T$ and $|D|_F$. Primitive operations requiring Boolean arguments must interpret objects in $|D|_T(|D|_F)$ as the Boolean value **TRUE** (**FALSE**). For example, in LISP $|D|_T$ and $|D|_F$ are $\{x|x \neq NIL\}$ and $\{NIL\}$ respectively.

2. The set of primitive operations includes the constant symbols **true** and **false** denoting the nominal representatives of the Boolean values **TRUE** and **FALSE** respectively. Obviously, **true** and **false** must belong to the sets $|D|_T$ and $|D|_F$ respectively. Boolean valued operations such as isD_T must return either **true** or **false**.

3. The primitive operations include the standard binary equality function **equal** and the Boolean characteristic functions isD_T and isD_F for the sets $|D|_T$ and $|D|_F$ respectively.

To facilitate writing recursive definitions, we also assume that D has the following properties:

1. The primitive operations include the standard ternary conditional expressions operator $if - then - else$ defined as follows:

$$\text{if } p \text{ then } x \text{ else } y = \begin{cases} x & \text{if } p \in |D|_T \\ y & \text{if } p \in |D|_F \end{cases}$$

2. The universe $|D|$ forms a well-founded partial ordering under the primitive binary Boolean operation $\prec$.

Let L be a first order language with equality for the program data domain D, and let $\vec{x}$ denote a sequence of distinct variables $x_1, \ldots, x_m$ of L. A *recursive program* on the data domain D consists of a set of function definitions of the form

$$f_i(\vec{x}_i) \leftarrow t_i(\vec{x}_i),\ i = 1, \ldots, n$$

where $f_1, ..., f_n$ are function symbols not in L and $t_i(\vec{x}_i)$ is a term in L (extended to include the new function symbols $f_1, ..., f_n$) with free variables $\vec{x}_i$.

For example, a recursive program on the natural numbers N which computes the factorial function is

$$fact(n) \leftarrow \text{if } n \textbf{ equal } 0 \text{ then } 1 \text{ else } n \times fact(n-1) \tag{1}$$

where we have used infix notation for the binary operators **equal** and $\times$.

In the sequel, we assume without loss of generality that every recursive program P consists of a single recursive definition. All of our results easily generalize to recursive programs with an arbitrary number of mutually recursive definitions.

3 The Functional Equation for a Recursive Program

Assume we are given a first order axiomatization A_D for the data domain D such that A_D includes a structural induction axiom schema (*e.g.*, Peano's axioms for the natural numbers N). To formally state and prove theorems about the function f defined by the recursive program

$$f(\vec{x}) \leftarrow t[f](\vec{x}) \tag{2}$$

we would like to extend D and its corresponding axiomatization A_D to include the defined operation f by converting the recursive definition (2) into the defining axiom

$$\forall \vec{x}\ [f(\vec{x}) = t[f](\vec{x})] \tag{3}$$

and interpreting f in D as the function computed by evaluating (2). Unfortunately, this naive approach does not work, because the function defined by (2) may not be total. In many cases, the proposed axiom (3) is inconsistent with A_D. A simple example illustrating this problem is the recursive program

$$f(x) \leftarrow f(x) + 1$$

over the natural numbers N. The corresponding first order axiom is obviously inconsistent with Peano's axioms.

We can salvage our basic approach to axiomatizing recursive programs by enlarging the data domain D to explicitly include the undefined element $\perp$ ("bottom"). We construct the augmented data domain D^+ from D by adding $\perp$ to the universe $|D|$ and appropriately extending all the operations of D. With the exception of *if-then-else*, we extend every operation of D to the corresponding *strict* function on $|D^+| = |D| \cup \{\perp\}$ (a function f on $|D^+|$ is strict iff f has the value $\perp$ if any of its arguments is $\perp$). We extend *if-then-else* to $|D^+|$ as follows:

$$\text{if } p \text{ then } x \text{ else } y = \begin{cases} \perp & \text{if } p = \perp \\ x & \text{if } p \in D_T \\ y & \text{if } p \in D_F \end{cases}$$

The corresponding modification to A_D required to form the axiomatization A_{D^+} for D^+ is a simple mechanical construction. The details are left to the reader.

For notational convenience, we let $x{:}D$ abbreviate the formula $x \neq \perp$. Intuitively, $x{:}D$ means that x is defined, *i.e.*that it belongs to the proper universe $|D|$.

Given the augmented domain D^+ and the corresponding axiomation A_{D^+}, we can successfully convert recursive programs on D into first order axioms augmenting A_{D^+} as suggested above. However, before proceeding further, we need to clarify what we mean by "evaluating" recursive programs.

Which partial function is defined by a recursive program P (such as (1)) depends on whether a *call-by-name* or *call-by-value* computation rule (in the terminology of Manna, Ness and Vuillemin [MNV 73]) is used to evaluate the program. From the standpoint of "denotational" semantics, the function computed by applying a call-by-name (call-by-value) functional for P [deB 75]. The call-by-name functional corresponding to the recursive program (1) is simply:

$$\lambda g\ .\ \lambda \vec{x}\ .\ t[g](\vec{x}).$$

The call-by-value functional is slightly more complex syntactically; it has the form:

$$\lambda g\ .\ \lambda \vec{x}\ .\ \text{if } \delta(\vec{x}) \text{ then } t[g](\vec{x}) \text{ else } \perp .$$

Fortunately, we can capture either meaning within first order logic by converting the recursive program into the appropriate first order axiom. The call-by-name (call-by-value) interpretation for the partial functin defined by an arbitrary recursive program (2) satisfies the first order sentence

$$\forall \vec{x}\ [f(\vec{x}) = t[f](\vec{x})] \tag{4}$$

where

$$\lambda g\ .\ \lambda \vec{x}\ .\ t[g](\vec{x})$$

is the call-by-name (call-by-value) functional corresponding to P. A formal proof that the call-by-value interpretation satisfies equation (4) appears in [Cart 76b]. The corresponding proof for the call-by-name interpretation is nearly identical.

In the call-by-value case, sentence (4) is equivalent to the conjunction of the two simpler sentences:

$$\forall \vec{x}{:}D\ [f(\vec{x}) = t[f](\vec{x})$$

$$\forall \vec{x}\ [\neg \vec{x}{:}D \rightarrow f(\vec{x}) = \perp].$$

In subsequent call-by-value examples, we will use these axioms in preference to equation (4).

As an illustration of our approach to axiomatizing recursive programs, consider the factorial program (1) presented as an example in the previous section. The call-by-name axiom (4) corresponding to the program is:

$$\forall n\ [fact(n) = \text{if } n \textbf{ equal } 0 \text{ then } 1 \text{ else } n \times fact(n-1)].$$

The corresponding call-by-value axioms for the same program are:

$$\forall n{:}N\ [fact(n) = \text{if } n \textbf{ equal } 0 \text{ then } 1 \text{ else } n \times fact(n-1)]$$

$$\forall n\ [\neg n{:}N \rightarrow fact(n) = \perp].$$

In practice, the second call-by-value axiom is rarely used, because it describes the behavior of f when it is applied to an undefined (divergent) argument. In the sequel, we will explicitly state which type of computation rule — call-by-name or call-by-value — that each program uses.

4 Reasoning about Recursive Programs

To reason about a recursive program P, we simply addend the first order sentence(s) characterizing P to the axiomatization A_{D+} of the augmented data domain D^+ and apply standard first order deduction. We call the resulting formal system Weak First Order Programming Logic. For example, assume A_{D+} is an augmented Peano axiomatization for the augmented natural numbers N^+ including the primitive functions $+, -, \times$, **equal**, and $<$. Let the Ackermann function ack be defined by the call-by-name recursive program on N

$$ack(x,y) \leftarrow \begin{cases} \text{if } x \textbf{ equal } 0 \text{ then } y \\ \text{else if } y \textbf{ equal } 0 \text{ then } ack(x-1,x) \\ \text{else } ack(x-1, ack(x,y-1)). \end{cases}$$

The first order sentence corresponding to the preceding program is

$$\forall x,y\, [ack(x,y) = \begin{cases} \text{if } x \textbf{ equal } 0 \text{ then } y \\ \text{else if } y \textbf{ equal } 0 \text{ then } ack(x-1,x) \\ \text{else } ack(x-1, ack(x,y-1))]. \end{cases}$$

We can prove that ack is total by proving the theorem

$$\forall x,y\, [x{:}N \wedge y{:}N \rightarrow ack(x,y){:}N].$$

The proof proceeds by structural induction on the pair (x,y) under the standard lexicographic ordering. The base case, $x = 0$ is trivial. For the induction step, we assume the induction hypothesis

$$\forall x',y'[x' < x \;\vee\; x' = x \wedge y' < y \;\rightarrow\; ack(x',y'){:}N].$$

Since $y{:}N$ by hypothesis, we can partition the proof of the induction step into two subcases: $y = 0$ and $y > 0$. In the first subcase $(y = 0)$, $ack(x,y) = ack(x-1, ack(x,x))$. By the induction hypothesis, $ack(x,x){:}N$. By a second application of the induction hypothesis, $ack(x-1, ack(x,x)){:}N$, proving that $ack(x,y){:}N$. The proof of the second subcase is analogous. By the induction hypothesis, $ack(x,y-1){:}N$. By a second application of the induction hypothesis, $ack(x-1, ack(x,y-1)){:}N$, proving that $ack(x,y){:}N$. Q.E.D.

5 Capturing the Concept of Least Fixed Point

The functional equation corresponding to a recursive program does not completely characterize it in some cases. For example, consider the call-by-name program

$$\Omega(x) \leftarrow \Omega(x) \tag{5}$$

on the natural numbers and the corresponding first order sentence

$$\forall x\, [\Omega(x) = \Omega(x)]. \tag{6}$$

Although the program (5) clearly does not terminate for any x, the sentence (6) is satisfied by any interpretation for the function Ω — not just the everywhere undefined function.

The problem is that the first order sentence corresponding to an arbitrary call-by-name (call-by-value) program P is satisfied by any fixed-point of the call-by-name (call-by-value) functional for P. If the function computed by P is total, this ambiguity does not arise because the functional for P has unique fixed point. For some programs which compute partial functions, however, the presence of extraneous fixed points prevents Weak First Order Programming Logic from proving any property of a function which is not true for all fixed points of the defining functional. The authors [Cart 78; McC 78] have proposed different ways to augment weak first order programming logic to solve this roblem.

Cartwright [Cart 78] extends Weak First Order Programming Logic by introducing the concept of a *complete recursive program* — a program whose functional has a unique fixed point. Given an arbitrary program P (computing f) on the data domain D, it is possible to mechanically construct a corresponding program P' (computing f') on the extended doman $Sexp(D)$ (where $Sexp(D)$ denotes the set of S-expressions over D) with the following properties:

1. P' computes the computation sequence for P.

2. P' is a complete recursive program.

The construction is described in detail in [Cart 78].

Let *last* denote the strict unary function on $Sexp(D)$ which extracts the last element of a list (represented as an S-expression). Then f' satisfies the sentence

$$\forall x{:}D\ [last(f'(x)) = f(x)]. \tag{7}$$

Obviously, this sentence is not provable in general Weak First Order Programming Logic, since it forces f to have a unique fixed point interpretation (with respect to a particular data domain model). Consequently, to fully characterize the recursive program P on the data domain D, [Cart 78] augments the axiomatization of $Sexp(D)^+$ (constructed from A_D) by the following three axioms describing P:

1. The recursion equation corresponding to P.

2. The recursion equation corresponding to P'.

3. The *equivalence axiom* (7) asserting the equivalence of $last \circ f'$ and f on D.

It is a straightforward exercise to construct Peano-like axiomatizations for $Sexp(D)$ and $Sexp(D)^+$; for one possible aproach see [Cart 76a; 76b]. Moreover, under suitable assumptions (the hypothesis of the relative completeness theorem presented in the next section) which any plausible data domain satisfies, it is possible to implement (encode) S-expressions over D as objects in D (using pairing functions) and avoid extending the data domain D altogether.

In contrast to the complete recursive program approach, McCarthy [McC 78] extends Weak First Order Programming Logic by adding an axiom schema (called the *minimization schema*) for each recursive program P. The schema asserts that the function computed by the program P is less-defined or equal to any function which is fixed point of the same functional. Let $\lambda g.\lambda x.t[g](x)$ be the functional corresponding to the program defining the function f. The minimization schema has the form

$$\forall x{:}D\ [t[G](x){:}D \rightarrow G(x) = t[G](x)] \ \rightarrow\ \forall x{:}D\ [f(x){:}D \rightarrow G(x) = f(x)].$$

where G is an arbitrary function symbol. The schema can be stated more succintly by introducing Scott's partial ordering on D^+ [Scott 70]. We define

$$\forall x, y\ [x \sqsubseteq y \leftrightarrow (x =\perp \vee x = y)].$$

Using this notation, the minimization schema becomes

$$\forall x{:}D\ [t[G](x) \sqsubseteq G(X)] \ \rightarrow\ \forall x{:}D\ [f(x) \sqsubseteq G(x)].$$

Let us illustrate and compare these two extensions to Weak First Order Programming Logic by considering the following call-by-value program over the natural numbers N:

$$loop(x) \leftarrow loop(x+1). \tag{8}$$

The function loop defined by (8) clearly diverges (equals $\perp$) everywhere, yet this fact is not provable within Weak First Order Programming Logic since any constant function is a fixed point of the corresponding call-by-value functional. However, the divergence of loop is provable from either the equivalence axiom (asserting the equivalence between (8) and the corresponding complete recursive program) or the minimization schema. The proof from the equivalence axiom proceeds as follows. The complete recursive program corresponding to (8) is:

$$loop'(x) \leftarrow cons(x+1, loop'(x+1)). \tag{9}$$

By the equivalence axiom (7) and the strictness of *last*, proving that loop diverges reduces to showing

$$\forall x{:}N[loop'(x) =\perp]. \tag{10}$$

We can prove (10) by structural induction on the value of $loop'(x)$. Assume (10) holds for all x' such that x' is a proper tail of x. If $x =\perp$, then the theorem holds. Otherwise,

$$loop'(x) = cons(x+1, loop'(x+1))$$

where

$$loop'(x+1) \neq\perp$$

(since *cons* is strict). But, by the induction hypothesis,

$$loop'(x+1) =\perp,$$

generating a contradiction. Q.E.D.

The derivation of the same theorem from the minimization schema is trivial. Let the function g on N^+ be defined by the call-by-value program

$$g(x) \leftarrow\perp .$$

The function g clearly satisfies the hypothesis of the minimization schema. Hence,

$$loop(x) \sqsubseteq g(x) = \perp$$

implying that

$$\forall x{:}N\ [loop(x) = \perp].$$

For simple theorems involving partial functions where suitable comparison functions like g are easily constructed, the minimization schema tends to produce shorter, simpler proofs than the equivalence axiom. For more complicated examples, however, (such as interpreters) the equivalence axiom may be more useful, because the complete recursive program construction automatically generates the complex comparison function required to successfully utilize the minimization schema.

The close relationship between complete recursive programs and suitable minimization comparison functions G suggests the following theorem:

Theorem The equivalence axiom and the minimization schema corresponding to a call-by-value (call-by-name) recursive program P are inter-derivable.

Proof In each direction, the proof is a routine but tedious induction on the structure of the body of the recursive program P. An outline of the proof appears in an Appendix.

Since either extension to Weak First Order Programming Logic can be derived from the other, we can safely extend the weak logic to include both extensions. We call the resulting system Strong First Order Programming Logic.

6 Relative Completeness of First Order Programming Logic

In Weak First Order Programming Logic, there are simple properties of recursive programs which we can expresss but cannot prove. Does the strong logic suffer from the same weakness? Fortunately, the answer is no. We can prove a relative completeness theorem which suggests that the strong logic is, for all practical purposes, as powerful a deductive system for reasoning about recursive programs (excluding programs which take functions as arguments) as we can hope to devise. Informally, the theorem asserts that any sentence in the strong logic for a recursive program P logically reduces to a sentence in the pure logic of the data domain. In other words, the inevitable Godelian incompleteness of the strong logic lies entirely in the data domain axiomatization, not in the axiomatization of the recursive program P.

Before we can precisely state the theorem, we need to introduce the following definitions.

1. An *interpretation* σ of a first order theory T in the language L into a theory T' in a (possible different) language L' is a function which maps the formulas of L into the corresponding formulas of L' such that $\sigma(T) \equiv T'$. A more precise definition of the concept appears in [Enderton 72, p. 162].

2. Let S be an arbitrary set of sentence in a first order language L. The set of *consequences* of T (denoted CnS) is the set of formulas logically implied by T.

Theorem Let P be an arbitrary call-by-value (call-by-name) recursive program

$$f(\vec{x}) \leftarrow t[f](\vec{x})$$

on the data domain D. If there is an interpretation of $CnA_{Sexp(D)}$ into CnA_D, then any formula ϕ in the language $L_{Sexp(D)^+} \cup f$ (*i.e.*the language of the strong logic for P) is provably equivalent (within the strong logic) to a formula in the language L_D.

Proof From the standpoint of mathematical logic, First Order Programming Logic provides a method for systematically introducing partial functions definitions on a data domain D into an axiomatic theory Cn A_D for D. In the terminology of [Enderton 72], this theorem simply asserts that, under suitable assumptions, these definitions are *eliminable*.

The key step in the proof is to construct a formula $\theta(\vec{x}, y)$ in the language of $Sexp(D)$ such that the sentence

$$\forall \vec{x}{:}D, y{:}D[\theta(\vec{x}, y) \leftrightarrow f(\vec{x}) = y]. \tag{11}$$

is provable in the strong logic for P. The construction for call-by-value programs in described in [McC 78]; the corresponding construction for call-by-name programs is similar but more complicated.

Given an arbitrary formula γ in the language $L_{Sexp(D)^+} \cup \{f\}$, we can logically reduce it to an equivalent formula γ' in the language $L_{Sexp(D)}$ by replacing references to f by references to θ and subsequently performing a simple case analysis to eliminate references to $\perp$. Then we can use σ to convert γ' to an equivalent formula γ in L_D. Q.E.D.

7 Reasoning About Iterative Programs

The intermittent assertions method developed by Burstall [Bur 74] and Manna and Waldinger [MW 78] has been widely promoted as a better proof method than the standard inductive assertions method for reasoning about iterative programs. However, the descriptions of the method appearing in the literature have all been informal, making the relative merits of the method difficult to assess. In this paper we provide a simple formilization of the intermittent method within First Order Programming Logic. In fact, we will demonstrate that intermittent assertions method can be viewed simply as convenient notation for proving theorems in first order programming logic about the recursive translations of iterative programs.

The fundamental idea in the formilization is the interpretation of the statement

$$\text{Sometime at L}, Q(\vec{x}, \vec{y}) \tag{12}$$

where L is a node in the flowchart program (a directed graph with assignment states and boolean tests attacted to the edges) and $Q(\vec{x}, \vec{y})$ is an arbitrary first order formula in the language of the data domain with the free variable lists $\vec{x}$, $\vec{y}$ ranging over the program data domain D. The variable lists $\vec{x}$ and $\vec{y}$ denote lists of program variables and arbitrary free variables, respectively. We formalize statement (12) within First Order Programming Logic for the program data domain D as

$$\exists \vec{x}{:}D\ [Q(\vec{x}, \vec{y}) \ \wedge\ f_L(\vec{x}) = f_{Start}(\vec{x}_0))]. \tag{13}$$

where $\vec{x}_0$ denotes the initial program state and f_L and f_{Start} are recursive programs equivalent to executing the flowchart starting at nodes L and Start respectively.

Of course, more comlex formalizations which explicitly mention labeled execution traces possible. Our translation seems to be the simplest one which works. Informally, (13) asserts that there is a program state $\vec{x}$ such that executing the program from L in state $\vec{x}$ produces the same output as executing the program from Start in state $\vec{x}_0$. This statement is weaker than the obvious (but more complex) translation involving the labeled execution trace for $f_{Start}(\vec{x}_0)$, because the state $\vec{x}$ may not occur at L during the execution of the program from Start in state $\vec{x}_0$. In fact, L may be a node which is never reached by any execution path beginning at Start. However, this apparent weakness does not diminish the deductive power of the intermittent assertions method; the logic is still relatively complete with respect to the theory of the underlying data domain. Moreover, all of the arguments used in Manna and Waldinger's sample proofs hold for our interpretation of "sometime" statements.

Our interpretation of "sometime" assertions does retain the crucial intuitive property that an assertion of the form

$$\text{Sometime } Q(\vec{x}, \vec{y}) \text{ at Finish}$$

(where Finish is the termination node of the flowchart) implies that the program terminates for input state $\vec{x}_0$. Otherwise, the condition,

$$f_{Finish}(\vec{x}) = f_{Start}(\vec{x}_0)$$

would be false since

$$f_{Finish}(\vec{x}) = \vec{x}_{Out}$$

where the output variable list $\vec{x}_{Out}$ is a non-empty subset of the program variables $\vec{x}$. Manna and Waldinger describe the input- output specifications for a program using "sometime" formulas by stating

$$\text{Sometime } In(\vec{x}, \vec{y}) \text{ at Start } \rightarrow \text{ Sometime } Out(\vec{x}, \vec{y}) \text{ at Finish.} \tag{14}$$

Using our interpretation of "sometime" formulas, (14) becomes

$$\begin{aligned} \forall \vec{x}_0{:}D, \vec{y}{:}D \quad & [\exists \vec{x}{:}D(In(\vec{x}, \vec{y}) \wedge f_{Start}(\vec{x}) = f_{Start}(\vec{x}_0)) \rightarrow \\ & \exists \vec{x}{:}D(Out(\vec{x}, \vec{y}) \wedge f_{Finish}(\vec{x}) = f_{Start}(\vec{x}_0))] \end{aligned}$$

which is equivalent to

$$\exists \vec{x}{:}D \vec{y}{:}D[In(\vec{x}, \vec{y}) \rightarrow f_{Start}(\vec{x}){:}D \wedge Out(f_{Start}(\vec{x}))] \tag{15}$$

Given our translation of "sometime" formulas into First Order Programming Logic, all of the lemmas and theorems appearing Manna and Waldinger's examples can be interpreted simply as sentences in a first order programming logic, for the program data domain. Similarly, their informal arguments can be translated into ordinary first order programming logic proofs. As an illustration, we will show how to formalize the tip-counting example described in [MW 78]. The recursive translation of the tip-counting program is:

$$f_{Start}(tree{:}TREE){:}INTEGER \quad \leftarrow \quad f_{More}(< tree >, 0)$$

$$f_{More}(stack{:}LIST, count{:}INTEGER){:}INTEGER \quad \leftarrow \quad \begin{array}{l} \text{if } stack \ \mathbf{equal} <> \text{ then} \\ \quad f_{Finish}(stack, count) \\ \text{else if } istip head stack \text{ then} \\ \quad f_{More}(tail stack, count+1) \\ \text{else} \\ \quad f_{More}([left\ head\ stack].[[right\ head\ stack].[tail\ stack]], count) \end{array}$$

$$f_{Finish}(stack{:}LIST, count{:}INTEGER){:}INTEGER \quad \leftarrow \quad count$$

where we have used the same primitive operation names and variable names as the original program. TREE denotes the set of binary trees, INTEGER denotes the set of integers, istip is the Boolean characteristic function for leaves, and the operations $\{., head, tail\}$ and the operations analogous to $\{cons, car, cdr\}$ in Lisp for manipulating lists of trees. The constructor . is used as an infix operator. In the interest of clarity, we have changed the notation slightly. First, we have omitted the parentheses enclosing unary function argument lists. Second, we have used the notation $< e_1, \ldots, e_n >$ instead of $(e_1, \ldots, e_n)$ to denote the list consisting of the elements $e_1, ..., e_n$.

To precisely define the number of tips in a tree, Manna and Waldinger provide the following recursive program:

$$tips(tree{:}TREE){:}INTEGER \leftarrow \text{if } istip(tree) \text{ then } 1 \text{ else } tips(lefttree) + tips(righttree).$$

Given the definition of tips, they prove the following input-output specifications for the tip-counting program:

$$\text{Sometime } tree = t \text{ at Start } \rightarrow \text{ Sometime } count = tips(t) \text{ at Finish.} \tag{16}$$

In our formilization, the statement (16) immediately reduces (by (15) and the substitution of $tree$ for t) to

$$\exists tree{:}TREE[f_{Start}(tree){:}INTEGER \wedge f_{Start}(tree) = tips(tree)] \tag{17}$$

where $:TREE$ and $:INTEGER$ are postfix operators denoting characteristic predicates for trees and integers respectively.

The key step in Manna and Waldinger's proof is proving the following lemma:

$$\begin{array}{c} \text{Sometime } count = c \wedge stack = t.s \text{ at More } \rightarrow \\ \text{Sometime } count = c + tips(t) \wedge stack = s \text{ at More.} \end{array} \tag{18}$$

where . is an infix operator (analogous to Lisp $cons$) mapping TREE$\times$LIST into LIST.

The formal translation of (18) reduces to

$$\exists t{:}TREE,\ s{:}LIST,\ c{:}INTEGER\,[f_{More}(t.s, c) = f_{More}(s, c + tips(t))] \tag{19}$$

where $:LIST$ is a postfix operator denoting the characteristic predicate for lists of trees. If we assume that $tips(t)$ is defined for any tree t, it is very easy to prove (19) by structural induction on the value of $[t.s]$. Moreover, we can trivially prove within First Order Programming Logic that $tips(t)$ is defined for any tree t, *i.e.*

$$\exists t{:}TREE\,[tips(t){:}INTEGER], \tag{20}$$

by induction on t. Manna and Waldinger implicitly utilize this lemma in their informal proof without mentioning it.

Given the lemmas (19) and (20), and the recursion equations corresponding to the tip-counting program, the input-output theorem (17) immediately follows.

Note that we formalized the tip-counting proof within Weak First Order Programming Logic; the extra power of the strong logic was not necessary. In fact, all of Manna and Waldinger's sample correctness proofs can be formalized within the weak logic. The weak logic suffices because totally correct iterative programs correspond to total recursive programs.

On the other hand, formalizing intermittent assertions proofs of partial correctness requires the strong logic. While Manna and Waldinger do not present any sample partial correctness proofs, they do show how to transform arbitrary inductive assertions (partial correctness) proofs into intermittent assertions proofs. The resulting proofs cannot be formalized within the weak logic because they utilize induction on the length of the program computation sequence rather than induction on the structure of the program data.

We believe our formalization of the intermittent assertions method sheds some light on why the method produces surprisingly simple correctness proofs for certain program. If an iterative program has a particularly simple recursive translation, then the intermittent assertions method, in essence, applies structural induction to the corresponding recursive program. As Boyer and Moore [BM 75] and Cartwright [Cart 76a] have observed, it is much easier to reason about naturally recursive programs using structural induction it is using other methods (e.g. inductive assertions, fixed point induction).

On the other hand, we doubt the intermittent assertions method will fare as well when applied to typical iterative programs. Manna and Waldinger's choice of examples tends to support this conjecture; all of their sample programs have natural recursive translations. It hardly seems advantageous to uniformly convert nonrecursive programs into corresponding recursive programs in order to reason about them; yet this is precisely the course that the intermittent assertions method follows.

8 References

References

[BM 75] R. Boyer and J Moore. Proving Theorems about Lisp Functions. *J. ACM* **22**(1) (January) 1975, 120–144.

[Bur 74] R. Burstall. Program Proving as Hand Simulation With a Little Induction. *Information Processing 74*, North-Holland, Amsterdam, 1974, 308–312.

[Cart 76a] R. Cartwright. User-Defined Data Types as Aid to Verifying LISP Programs. In S. Michaelson and R. Milner (eds.), *Automata Languages, and Programming.* Edinburgh Press, Edinburgh, 228-256.

[Cart 76b] R. Cartwright. *A Practical Formal Semantic Definition and Verification System for TYPED LISP*, Stanford Artificial Intelligence Laboratory Memo AIM-296, Stanford University, 1976.

[Cart 78] R. Cartwright. First Order Semantics: A Natural Programming Logic for Recursively Defined Functions. Cornell University Computer Science Dept. Tech. Report TR 78-339, Ithaca, New York, 1978.

[deB 75] J.W. DeBakker. *The Fixed Point Approach in Semantics: Theory and Applications.* Mathematical Centre Tracts 63, Free University, Amsterdam, 1975.

[End 72] H. B. Enderton. A Mathematical Introduction to Logic. Academic Press, New York, 1972.

[Floyd 67] R. Floyd. Assigning Meaning to Programs. In J. T. Schwarz (ed.), *Proc. Symp. in Applied Math. Vol. 19*, Amer. Math. Soc., Providence, Rhode Island, 1967, 19–32.

[Gent 36] G. Gentzen. Die Widerspruchsfreheit der reinen Zahlentheorie. *Math. Annalen* **112**, 1936, 493–565, English translation in M. E. Szabo (ed.), *The Collected Works of Gerhard Gentzen*, North Holland, Amsterdam, 1969, 132–213.

[HP 73] P. Hitchcock and D. Park. Induction Rules and Proofs of Program Termination. In M. Nivat (ed.), *Automata, Languages and Programming*, North Holland, Amsterdam, 1973, 225–251.

[Hoare 69] C.A.R. Hoare. An Axiomatic Basis of Computer Programming. *Comm. ACM* **12**(10), (October) 1969, 576–580.

[Manna 74] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[MNV 73] Z. Manna, S. Ness, and J. Vuillemin. Inductive Methods for Proving Properties of Programs. *Comm. ACM* **16**(8) (August) 1973 491–502.

[MV 72] Manna, Z., and J. Vuillemin Z. Manna and J. Vuillemin. Fix-point Approach to the Theory of Computation. *Comm. ACM* **15**, 1972, 528–536.

[MW 78] Z. Manna and R. Waldinger. Is "Sometime" Sometimes Better than "Always"?. *Comm. ACM* **21**(2), 1978, 159–171.

[McC 63] J. McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems.* North-Holland Publishing Company, Amsterdam, 1963, 33–70.

[McC 78] J. McCarthy. Representation of Recursive Programs in First Order Logic. Unpublished draft, Computer Science Department, Stanford University, Stanford, California, 1978.

[MoWe 77] J.H. Morris and B. Wegbreit. Program Verification by Subgoal Induction, *Comm. ACM* **20**(4), (April) 1977, 209–222.

[Park 69] D. Park. Fixpoint Induction and Proofs of Program Properties. In *Machine Intelligence 5*, Edinburgh University Press, Edinburgh, 1969, 59–78.

[Scott 70] D. Scott. Outline of a Mathematical Theory of Computation. *Proceedings of Fourth Annual Princeton Conference on Information Science and Systems* Princeton, 1970, 169–176.

9 Appendix: The Equivalence Axiom and the Minimization Schema

To establish that the equivalence axiom for a recursive program P is provable from the minimization schema for P we:

1. Construct the complete recursive program P' corresponding to P, generating the first order sentence characterizing P'.

2. Show that the sentence
$$\exists{:}D(last(f'(x)) = f(x))$$
is provable by structural induction on f'(x) within Weak First Order Programming Logic. The metaproof proceeds by structural induction on the right hand side of the recursive definition of f in P.

3. Show that the sentence
$$\exists x{:}D(t[last \circ f'](x) \sqsubseteq last \circ f'(x))$$
is provable by structural induction on $f'(x)$ within the weak logic. As in 2) above, the metaproof proceeds by structural induction on the right hand side of P.

4. Deduce
$$\exists x{:}D(f(x) \sqsubseteq last(f'(x)))$$
from the minimization schema and 3) above.

5. Combine the conclusions of 2) and 4) to finally prove
$$\exists x{:}D(f(x) = last(f'(x)).$$

To prove that the minimization schema is derivable from the equivalence axiom, we must show that given an arbitrary program P

$$f(x) \leftarrow t(x)$$

and an arbitrary function g

$$\exists x{:}D(t[g](x) \sqsubseteq g(x)) \rightarrow \exists x{:}D(f(x) \sqsubseteq g(x))$$

which reduces (by the equivalence axiom) to the formula

$$\exists x{:}D(t[g](x) \sqsubseteq g(x)) \rightarrow \exists x{:}D(last(f'(x)) \sqsubseteq g(x)).$$

But this formula is an immediate consequence of the lemma

$$\exists x{:}D(t[g](x) \subseteq g(x)) \rightarrow \exists x{:}D(last(f'(x)) \subseteq t[g](x))$$

which we can prove (by induction on the structure of t) is deducible within the weak logic augmented by the equivalence axiom.