FIRST ORDER PROGRAMMING LOGIC

Robert Cartwright
Computer Science Department
Cornell University

John McCarthy
Computer Science Department
Stanford University

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 formally equivalent. 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.

The second contribution of this paper is to establish that First Order Programming Logic is applicable to iterative programs as well. In particular, we show that the intermittent assertions method--an informal proof method for iterative programs which has not been formalized--is conveniently formalized simply as sugared First Order Programming Logic applied to the recursive translations of iterative programs.

1. Introduction.

Many theoretical computer scientists (e.g. Hitchcock and Park [1973]) have dismissed first order logic as too weak a formalism for reasoning about recursively defined functions. Nevertheless, the authors [Cartwright 76a; Cartwright 76b; McCarthy 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 [Boyer and Moore 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 which 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 [Cartwright 78; McCarthy 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 the partial functions computed by recursive programs. Of course, the extended logic cannot escape the essential Goedelian incompleteness of any non-trival 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 36, 38] 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 1974] and [Manna and Waldinger 1978] 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 1972] and either [Manna 1974] or [Manna, Ness and Vuillemin 1973]. Before we can formally define the concept of a recursive program, we must introduce some preliminary definitions. A <u>data domain</u> D consists of a set $|D|$ of data objects (called the <u>universe</u>) together with a set of primitive operations (functions) $g_1, \ldots, g_k$ on $|D|$. Constants are treated as 0-ary 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 accomodate 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 operation 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 $\{x|x\neq\text{NIL}\}$ and $\{\text{NIL}\}$ 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 expression operator if-then-else defined as follows:

$$\underline{\text{if}}\ x\ \underline{\text{then}}\ y\ \underline{\text{else}}\ z = \begin{cases} y \text{ if } \text{isD}_t(x) \\ z \text{ if } \text{isD}_f(x) \end{cases} .$$

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

Let L be a first order language with equality for the program data domain D, and let $\overline{x}$ 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

$$f_i(\overline{x}_i) \leftarrow t_i[f_1,\ldots,f_n](\overline{x}_i),\quad i=1,\ldots,n,$$

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

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

(1) $\text{fact}(n) \leftarrow \underline{\text{if}}\ n\ \underline{\text{equal}}\ 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ n*\text{fact}(n-1)$.

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

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

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

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 the defining axiom

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

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 $\bot$ ("bottom"). We construct the augmented data domain D+ from D by adding $\bot$ to the universe $|D|$ and appropriately extending all the operations of D. With the exception of <u>if-then-else</u>, we extend every operation of D to the corresponding <u>strict</u> function on $|D+| = |D| \cup \bot$ (a function f on $|D+|$ is strict iff f has the value $\bot$ if any of its arguments is $\bot$). We extend <u>if-then-else</u> to $|D+|$ as follows:

$$\underline{\text{if}}\ \underline{p}\ \underline{\text{then}}\ d_1\ \underline{\text{else}}\ d_2 = \begin{cases} \bot \\ d_1 \text{ if } \underline{isD}_t(p) \\ d_2 \text{ if } \underline{isD}_f(p) \end{cases}$$

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 $x \neq \bot$. 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 axiomatization 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 <u>call-by-name</u> or <u>call-by-value</u> computation rule (in the terminology of Manna, Ness and Vuillemin [1973]) is used to evaluate the program. From the standpoint of "denotational" semantics, the function computed by applying a call-by-name (call-by-value) computation rule to P is the least fixed point of the call-by-name (call-by-value) functional for P [deBakker 75]. The call-by-name functional corresponding to the recursive program (1) is simply:

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

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

$$\lambda g \,.\, \lambda \bar{x} \,.\, \underline{\text{if}}\ \delta(\bar{x})\ \underline{\text{then}}\ t[g](\bar{x})\ \underline{\text{else}}\ \bot \,.$$

where

$$\delta(\bar{x}) = \begin{cases} \underline{\text{true}} \text{ if } \bar{x}{:}D \\ \bot \text{ otherwise} \end{cases}.$$

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 function defined by an arbitrary recursive program (2) satisfies the first order sentence

(4) $\forall\bar{x}\ [f(\bar{x})=\alpha[f](\bar{x})$

where

$$\lambda\ g\ .\ \lambda\ \bar{x}\ .\ \alpha[g](\bar{x})$$

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 [Cartwright 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:

(4a) $\forall\bar{x}{:}D\ [f(\bar{x})=t[f](\bar{x})$

(4b) $\forall\bar{x}[\neg\ \bar{x}{:}D \rightarrow f(\bar{x})=\bot]$.

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:

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

The corresponding call-by-value axioms [(4a),(4b)] for the same program are:

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

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

In practice, the call-by-value axiom (4b) 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 in First Order Programming Logic.

To reason about a recursive program P, we simply append 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 +, -, *, equal, and less. Let the Ackermann function ack be defined by the call-by-name recursive program on N

```
ack(x,y) <- if x equal 0 then y
            else ack(x-1,ack(x,y-1)) .
```

The first order sentence corresponding to the preceding program is

$$\forall x\ y\ [\text{ack}(x,y) = \underline{\text{if}}\ x\ \underline{\text{equal}}\ 0\ \underline{\text{then}}\ y\ \underline{\text{else}}\ \text{ack}(x-1,\text{ack}(x,y-1)))\ .$$

We can prove that ack is total by proving the theorem

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

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

$$\forall x'\ y'\ [x'\ \underline{less}\ x\ \mid\ x'=x\ \&\ y'\ \underline{less}\ y \rightarrow ack(x',y'):N].$$

Since $x \neq 0$, ack(x,y)=ack(x-1,ack(x,y-1)). By hypothesis, ack(x,y-1):N. Applying the hypothesis a second time, we deduce ack(x-1,ack(x,y-1)):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 on the natural numbers

(5) $\underline{Q}(x) \leftarrow \underline{Q}(x)$

and the corresponding first order sentence

(6) $\forall x\ \underline{Q}(x) = \underline{Q}(x)$.

Although the program (5) clearly does not terminate for any x, the sentence (6) is satisfied by any interpretation for the function $\underline{Q}$--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 a unique fixed point. For some programs which compute partial functions, however, the presence of "non-standard" 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 [Cartwright 1978; McCarthy 1978] have proposed different ways to augment weak first order programming logic to solve this problem.

[Cartwright 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 domain 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 [Cartwright 78].

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

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

Obviously, this sentence is not provable in general in 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, [Cartwright 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 $\underline{last} \circ f'$ and f on D.

It is a straightforward exercise to construct Peano-like axiomatizations for Sexp(D) and Sexp(D)+; for one possible approach see [Cartwright 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 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 a 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 $\sqsubseteq$ on D+ [Scott 70]. We define

$$\forall x\ y\ (x \sqsubseteq y \leftrightarrow x=\perp \mid x=y).$$

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:

(8) loop(x) <- loop(x+1).

The function loop defined by (8) is 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:

(9) loop'(x) <- cons(x+1,loop'(x+1)) .

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

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

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,

$$\text{loop}'(x) = \text{cons}(x+1,\text{loop}'(x+1))$$

where

$$\text{loop}'(x+1) \neq \perp$$

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

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

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,

$\text{loop}(x) \sqsubseteq g(x) \sqsubseteq \perp$

implying that

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

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 Strong First Order Programming Logic.

In Weak First Order Programming Logic, there are simple properties of recursive programs which we can express 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_1 in a (possible different) language L_1 is a function which maps the formulas of L into corresponding formulas of L_1 such that $\sigma[T]$ T_1. 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 Cn S) is the set of formulas logically implied by T.

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

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

on the data domain D. If there is an interpretation of Cn $A_{Sexp(D)}$ into Cn A_D, then any formula θ 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 $\theta(\overline{x},y)$ in the language of Sexp(D) such that the sentence

(11) $\forall \overline{x}:D\ y:D\ [\theta(\overline{x},y) <-> f(\overline{x})=y]$.

is provable in the strong logic for P. The construction for call-by-value programs is described in [McCarthy 78]; the corresponding construction for call-by-name programs is similar but slightly 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 in First Order Programming Logic.

The intermittent assertions method developed by Burstall [1974] and Manna and Waldinger [1978] 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 very informal, making the relative merits of the method difficult to assess. In this paper we provide a simple formalization 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 formalization is the interpretation of the statement

(12) Sometime at L, $Q(\overline{x},\overline{y})$

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

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

where $\overline{x}_0$ denotes the initial progam 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 complex formalizations which explicitly mention labeled execution traces are possible. Our translation seems to be the simplest one which works. Informally, (13) asserts that there is a program state $\overline{x}$ such that executing the program from L in state $\overline{x}$ produces the same output as executing the program from Start in state $\overline{x}_0$. This statement is weaker than the obvious (but more complex) translation involving the labeled execution trace for $f_{Start}(\overline{x}_0)$, because the state may not occur at L during the execution of the program from Start in state $\overline{x}_0$. 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 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. In intuitive terms, our formalization of intermittent asserts permits non-standard program executions, but forces them to produce standard results.

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

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

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

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

would be false since

$$f_{\text{Finish}}(\overline{x}) = \overline{x}_{\text{out}}$$

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

(14) Sometime $In(\overline{x},\overline{y})$ at Start ->
  Sometime $Out(\overline{x},\overline{y})$ at Finish.

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

$$\forall \overline{x}_0{:}D\ \ \overline{y}{:}D\ [\exists \overline{x}{:}D\ (In(\overline{x},\overline{y})\ \&\ f_{\text{Start}}(\overline{x})=f_{\text{Start}}(\overline{x}_0)) \rightarrow \exists \overline{x}{:}D\ (Out(\overline{x},\overline{y})\ \&\ f_{\text{Finish}}(\overline{x})=f_{\text{Start}}(\overline{x}_0))]$$

which is equivalent to

$$(15)\quad \forall \overline{x}{:}D\ \overline{y}{:}D\ [In(\overline{x},\overline{y}) \rightarrow f_{\text{Start}}(\overline{x}){:}D\ \&\ Out(f_{\text{Start}}(\overline{x}))]\ .$$

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 [Manna and Waldinger 78]. The recursive translation of the tip-counting program is:

```
f_Start(tree) <- f_More(<tree>,0)

f_More(stack,count) <-
    if stack equal <> then
        f_Finish(stack,count)
    else if istip head stack then
        f_More(tail stack,count+1)
    else
        f_More([left head stack] .
            [[right head stack] .
             [tail stack]])

f_Finish(stack,count) <- count
```

where we have used the same primitive operation names and variable names as the original program. 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 $e_1,\ldots,e_n$.

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

```
tips(tree) <-
    if istip(tree) then 1
    else tips(left tree) +
         tips(right tree) .
```

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

(16) If Sometime tree = t at Start, then Sometime count = tips(t) at Finish .

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

(17) $\forall$ tree:TREE $[f_{Start}(\text{tree})\text{:INTEGER} \;\&\; f_{Start}(\text{tree})=\text{tips}(\text{tree})]$

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:

(18) If Sometime count = c & stack = t . s at More, then Sometime count = c + tips(t) & stack = s at More .

The formal translation of (18) reduces to

(19) $\forall$ t:TREE s:LIST c:INTEGER $f_{More}(\text{t.s},\text{c}) = f_{More}(\text{s},\text{c+tips(t)})$

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.

(20) $\forall$ t:TREE tips(t):INTEGER ,

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 programs. 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 75] and [Cartwright 76a] have observed, it is much easier to reason about naturally recursive programs using structural induction than 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 non-recursive programs into corresponding recursive programs in order to reason about them; yet this is precisely the course that the intermittent assertions method follows.

References

Boyer, R. and J Moore. (1975): Proving Theorems about LISP Functions, J. ACM 22(1): 129-144 (January).

Burstall, R. (1974): Program Proving as Hand Simulation With a Little Induction, in Information Processing 74, pp. 308-312, North-Holland, Amsterdam.

Cartwright, R. (1976a): User-Defined Data Types as Aid to Verifying LISP Programs, in S. Michaelson and R. Milner (eds.), Automata Languages, and Programming, pp. 228-256, Edinburgh Press, Edinburgh.

Cartwright, R. (1976b): A Practical Formal Semantic Definition and Verification System for TYPED LISP, Stanford A. I. Lab. Memo AIM-296, Stanford University, Stanford, California.

Cartwright, R. (1978): First Order Semantics: A Natural Programming Logic for Recursively Defined Functions, Cornell University Computer Science Dept. Tech. Report TR78-339, Ithaca, New York.

DeBakker, J. W. (1975): The Fixed Point Approach in Semantics: Theory and Applications, Mathematical Centre Tracts 63, Free University, Amsterdam.

Floyd, R. (1967): Assigning Meaning to Programs, in J. T. Schwarz (ed.), Proc. Symp. in Applied Math. Vol. 19., pp. 19-32, Amer. Math. Soc., Providence, Rhode Island.

Gentzen, Gerhard. (1936): Die Widerspruchsfreiheit der reinen Zahlentheorie, Math. Annalen 112: 493-565; English translation in M. E. Szabo (ed.), The Collected Works of Gerhard Gentzen, pp. 132-213, North Holland, Amsterdam, 1969.

Gentzen, Gerhard. (1938): Neue Fassung des Widerspruchsfreiheitbeweiss fur die reine Zahlentheorie, Forschungen zur Log. u.z. Grund. der exacten Wiss., New Series No. 4, pp 19-44; English translation in M. E. Szabo (ed.), The Collected Works of Gerhard Gentzen, pp. 252-286, North Holland, Amsterdam, 1969.

Hitchcock, P. and D. Park (1973): Induction Rules and Proofs of Program Termination, in M. Nivat (ed.), Automata, Languages, and Programming, pp. 225-251, North-Holland, Amsterdam.

Hoare, C. A. R. (1969): An Axiomatic Basis of Computer Programming, Comm. ACM 12(10): 576-580 (October).

Manna, Z. (1974): Mathematical Theory of Computation, McGraw-Hill.

Manna, Z., S. Ness, and J. Vuillemin (1973): Inductive Methods for Proving Properties of Programs", Comm. ACM 16(8): 491-502 (August).

Manna, Z., and J. Vuillemin (1972): Fixpoint Approach to the Theory of Computation, Comm. ACM 15: 528-536.

Manna, Z., and R. Waldinger (1978): Is "Sometime" Sometimes Better Than "Always"?, Comm. ACM 21(2): 159-171.

McCarthy, J. (1963): A Basis for a Mathematical Theory of Computation, in P. Braffort and D. Hirschberg (eds.), Computer Programming and Formal Systems), pp. 33-70. North-Holland Publishing Company, Amsterdam.

McCarthy, J. (1978): Representation of Recursive Programs in First Order Logic, unpublished draft, Computer Science Department, Stanford University, Stanford, California.

Morris, J. H., and B. Wegbreit (1977): Program Verification by Subgoal Induction, Comm. ACM 20(4): 209-22 (April).

Park, D. (1969): Fixpoint Induction and Proofs of Program Properties, in Machine Intelligence 5, pp. 59-78, Edinbrugh University Press, Edinburgh.

Scott, D. (1970): Outline of a Mathematical Theory of Computation, Proceedings of Fourth Annural Princeton Conference on Information Science and Systems, Princeton, pp. 169-176.

Appendix: Equivalence of 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

$$\forall x{:}D\ (\underline{\text{last}}(f'(x)) \sqsubseteq f(x))$$

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

$$\forall x{:}D\ (t[\underline{\text{last}}\circ f'](x) \sqsubseteq \underline{\text{last}}\ f'(x))$$

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

$$\forall x{:}D\ (f(x) \sqsubseteq \underline{\text{last}}(f'(x)))$$

from the minimization schema and 3) above.

5. Combine the conclusions of 2) and 4) to finally prove

$$\forall x{:}D\ (f(x) = \underline{\text{last}}(f'(x))).$$

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

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

which reduces (by the equivalence axiom) to

$$(*)\quad \forall x{:}D\ (t[g](x) \sqsubseteq g(x)) \rightarrow \forall x{:}D\ (\underline{\text{last}}(f'(x)) \sqsubseteq g(x)).$$

But (*) follows immediately from the lemma

$$\forall x{:}D\ (t[g](x) \sqsubseteq g(x)) \rightarrow \forall x{:}D\ (\underline{\text{last}}(f'(x)) \sqsubseteq t[g](x))$$

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