Fixpoint Approach to the Theory of Computation

Zohar Manna and Jean Vuillemin
Stanford University

Following the fixpoint theory of Scott, the semantics of computer programs are defined in terms of the least fixpoints of recursive programs. This allows not only the justification of all existing verification techniques, but also their extension to the handling, in a uniform manner of various properties of computer programs, including correctness, termination, and equivalence.

Key Words and Phrases: verification techniques, semantics of programming languages, least fixpoints, recursive programs, computational induction

CR Categories: 5.23, 5.24

The research reported here was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract SD-183. Authors' address: Computer Science Department, Stanford University, Stanford, CA 94305.

Introduction

Substantial progress has recently been made in understanding the mathematical semantics of programming languages as a result of Scott's fixpoint theory. Our main purpose in this paper is to introduce the reader to some applications of this theory as a practical tool for proving properties of programs.

The paper consists of two parts. In Part 1 the notion of a recursive program and its associated (unique) least fixpoint function are introduced. We describe the computational induction method, a powerful tool for proving properties of the least fixpoint of a recursive program. We then illustrate how one could describe the semantics of an ALGOL-like program P by "translating" it into a recursive program P' such that the partial function computed by P is identical to the least fixpoint of P'. Works in this area include: [McCarthy, 1963a, 1963b; Landin, 1965; Strachey, 1966; Morris, 1968; Bekič, 1969; Park, 1969; deBakker and Scott, 1969; Scott, 1970; Scott and Strachey, 1971; Manna, Ness, and Vuillemin, 1972; Milner, 1972; Weyhrauch and Milner, 1972].

In Part 2 of the paper we illustrate some of the advantages of the fixpoint approach to program semantics. First, we justify the inductive assertion methods of [Floyd, 1967 and Hoare, 1969, 1971]. Other verification methods such as recursion induction [McCarthy, 1963a, 1963b], structural induction [Burstall, 1969], fixpoint induction [Park, 1969; Cooper, 1971], and the predicate calculus approach [Manna, 1969; Manna and Pnueli, 1970] can be justified in much the same way. Secondly, we emphasize that the fixpoint approach suggests a natural method for proving properties of programs: given a program P, we can translate it into the corresponding recursive program P', and then prove the desired properties for the least fixpoint of P' by com-

putational induction. In contrast to other existing methods, this approach gives a *uniform* way of expressing and proving different properties, including correctness, termination, and equivalence. This makes it very appealing for machine implementation [Milner, 1972].

Warning. The reader should be aware that some of the results presented in this paper hold only under certain restrictions which are ignored in this informal presentation.

Part 1. The Fixpoint Approach to Program Semantics

1.1 Recursive Programs

A *recursive program* is a LISP-like definition of the form

$$F(\mathbf{x}) \Leftarrow \tau[F](\mathbf{x}),$$

where $\tau[F](\mathbf{x})$ is a composition of base functions and the function variable F, applied to the individual variables $\mathbf{x} = (x, y, z, \ldots)$. The following, for example, is a recursive program over the integers

$$P_0: \quad F(x, y) \Leftarrow \textit{if } x = y \textit{ then } y + 1 \textit{ else } F(x, F(x - 1, y + 1)).$$

We allow our base functions to be partial, i.e. they may be undefined for some arguments. This is quite natural, since they represent the result of some computation which may, in general, give results for some inputs and run indefinitely for others. We include as limiting cases the partial functions defined for all arguments, called total functions, as well as the partial function undefined for all arguments.

Let us consider now the following functions:

$f_1(x, y)$: *if* $x = y$ *then* $y + 1$ *else* $x + 1$,
$f_2(x, y)$: *if* $x \geq y$ *then* $x + 1$ *else* $y - 1$, and
$f_3(x, y)$: *if* $(x \geq y) \wedge (x - y$ even$)$ *then* $x + 1$ *else undefined.*

These functions have an interesting common property. For each i $(1 \leq i \leq 3)$, if we replace all occurrences of F in the program P_0 by f_i, the left-hand side and the right-hand side of the symbol $\Leftarrow$ yield identical partial functions, i.e.

$$f_i(x, y) \equiv \textit{if } x = y \textit{ then } y + 1 \textit{ else } f_i(x, f_i(x - 1, y + 1)).$$

We say that the functions f_1, f_2, and f_3 are *fixpoints* of the recursive program P_0. There are two different ways to extend the regular equality relation. The natural extension, denoted by $=$, is *undefined* whenever at least one of its arguments is *undefined.* The other one, denoted by $\equiv$, is *true* if both arguments are *undefined*, and *false* if exactly one of them is *undefined.* Consequently, the function $f_4(x, y): x + 1$, is not a fixpoint of P_0 when y is *undefined.*

Among the three functions, f_3 has one important special property: for any (x, y) such that $f_3(x, y)$ is defined, i.e. $(x \geq y) \wedge (x - y$ even$)$, both $f_1(x, y)$ and $f_2(x, y)$ are also defined and have the same value as $f_3(x, y)$. We say that f_3 is "less defined than or equal to" f_1 and f_2, and denote this by $f_3 \subseteq f_1$ and $f_3 \subseteq f_2$. It can be shown that f_3 has this property not only with respect to f_1 and f_2 but with respect to all fixpoints of the recursive program P_0. Moreover, $f_3(x, y)$ is the *only* function having this property; f_3 is therefore said to be the *least (defined) fixpoint of* P_0.

One of the most important results related to this topic is due to Kleene, who showed that *every recursive program P has a unique least fixpoint (denoted by* f_P*).* (See [Kleene, 1952].)

In discussing our recursive programs, the key problem is: *what is the partial function f defined by a recursive program P?* There are two viewpoints: (a) *Fixpoint approach*: Let it be the unique least fixpoint f_P. (b) *Computational approach*: Let it be the computed function C_P for some given computation rule C (such as "call by name" or "call by value").

We now come to an interesting point: all the theory for proving properties of recursive programs is based on the assumption that the function defined by a recursive program is exactly the least fixpoint f_P, that is, the fixpoint approach is adopted. Unfortunately, many programming languages use implementations of recursion (such as "call by value"!) which do not necessarily lead to the least fixpoint [Morris, 1968].[1]

Let us consider, for example, the following recursive program over the integers

$$P_1: F(x, y) \Leftarrow \textit{if } x = 0 \textit{ then } 1 \textit{ else } F(x - 1, F(x, y)).$$

The least fixpoint f_{P_1} can be shown to be

$$f_{P_1}(x, y): \textit{if } x \geq 0 \textit{ then } 1 \textit{ else undefined.}$$

However, the computed function C_{P_1}, where C is "call by value," turns out to be

$$C_{P_1}(x, y): \textit{if } x = 0 \textit{ then } 1 \textit{ else undefined.}$$

Thus C_{P_1} is properly less defined than f_{P_1}—e.g. $C_{P_1}(1, 0)$ is *undefined* while $f_{P_1}(1, 0) = 1$.

There are two alternative ways to view this problem. (1) Existing computer languages should be modified, and language designers and implementors should seek computation rules which always lead to the least fixpoint. "Call by name" is one such computation rule, but unfortunately it often leads to very inefficient computations. An efficient computation rule which always leads to the least fixpoint can be obtained by modifying "call by value" so that the evaluation of the arguments of a procedure is delayed as long as possible [Vuillemin, 1972]. (2) Theoreticians are wasting their time by

[1] It can be shown in general that for every recursive program P and any computation rule C, C_P must be less defined than or equal to f_P, i.e. $C_P \subseteq f_P$ [Cadiou, 1972].

developing fixpoint methods for proving properties of programs which do not compute fixpoints. They should instead concentrate their efforts on developing direct methods for proving properties of programs as they are actually executed.

We shall indicate in Part 2 of this paper how the apparent conflict between these views can be resolved by a suitable choice of the semantic definition of the programming language.

1.2 The Computational Induction Method

The main practical reason for suggesting the fixpoint approach is the existence of a very powerful tool, the computational induction method, for proving properties of the least fixpoint f_P of a given recursive program P. The idea of the method is essentially to use an induction on the level of recursion.

Let us consider, for example, the recursive program

$$P_2: F(x) \Leftarrow \textit{if } x = 0 \textit{ then } 1 \textit{ else } x \cdot F(x - 1)$$

over the natural numbers. The least fixpoint $f_{P_2}(x)$ of this recursive program is the factorial function $x!$.

Let us denote by $f^i(x)$ the partial function indicating the "information" we have after the ith level of recursion. That is,

$f^0(x)$ is *undefined* for all x;
$f^1(x)$ is $\textit{if } x = 0 \textit{ then } 1 \textit{ else } x \cdot f^0(x - 1)$,
i.e. $\textit{if } x = 0 \textit{ then } 1 \textit{ else undefined}$;
$f^2(x)$ is $\textit{if } x = 0 \textit{ then } 1 \textit{ else } x \cdot f^1(x - 1)$,
i.e. $\textit{if } x = 0 \textit{ then } 1 \textit{ else } x \cdot (\textit{if } x - 1 = 0 \textit{ then } 1 \textit{ else undefined})$,
or in short,
$\textit{if } x = 0 \vee x = 1 \textit{ then } 1 \textit{ else undefined}$;
etc.

In general, for every i, $i \geq 1$,

$$f^i(x) \text{ is } \textit{if } x = 0 \textit{ then } 1 \textit{ else } x \cdot f^{i-1}(x - 1),$$

which is

$$\textit{if } x < i \textit{ then } x! \textit{ else undefined.}$$

This sequence of functions has a limit which is exactly the least fixpoint of the recursive program, that is,

$$\lim_{i \to \infty} \{f^i(x)\} \equiv x!.$$

This will in fact be the case for any recursive program P: if P is a recursive program of the form $F(\mathbf{x}) \Leftarrow \tau[F](\mathbf{x})$, and $f^i(\mathbf{x})$ is defined by

$f^0(\mathbf{x})$ is Ω (*undefined* for all $\mathbf{x}$), and
$f^i(\mathbf{x})$ is $\tau[f^{i-1}](\mathbf{x})$ for $i \geq 1$,[2]

then

$$\lim_{i \to \infty} \{f^i(\mathbf{x})\} \equiv f_P(\mathbf{x}).$$

This suggests an induction rule for proving properties of f_P: To show that some property φ holds for f_P, i.e. $\varphi(f_P)$, we show that $\varphi(f^i)$ holds for all $i \geq 0$, and that φ remains true in the limit; therefore we may conclude that $\varphi(\lim_{i \to \infty} \{f^i\})$, i.e. $\varphi(f_P)$, holds.

Note that it is not true in general that φ remains true in the limit. For example, for the recursive program P_2 introduced above, $f^i(x)$ is the nontotal function $\textit{if } x < i \textit{ then } x! \textit{ else undefined}$, while $\lim_{i \to \infty} \{f^i\}$, i.e., f_{P_2}, is the total function $x!$. Thus for $\varphi(f)$ being "f is not total," we have that $\varphi(f^i)$ holds for all $i \geq 0$, while $\varphi(\lim_{i \to \infty} \{f^i\})$ does not hold. However, the limit property holds of a rather large class of φ (called "admissible predicates"—see [Manna, Ness, and Vuillemin, 1972]); in particular, all the predicates that we use later have this property.

There are two well-known ways to prove that $\varphi(f^i)$ holds for all $i \geq 0$, the rules for simple and complete induction on the level of recursion.

(a) *Simple induction*:
If $\varphi(f^0)$ *holds and* $\forall i[\varphi(f^i) \Rightarrow \varphi(f^{i+1})]$ *holds, then* $\varphi(f_P)$ *holds.*

(b) *Complete induction*:
If $\forall i\{[(\forall j \textit{ such that } j < i)\, \varphi(f^j)] \Rightarrow \varphi(f^i)\}$ *holds,*[3] *then* $\varphi(f_P)$ *holds.*

The simple induction rule is essentially the "μ-rule" suggested by [deBakker and Scott, 1969], while the complete induction rule is the "truncation induction rule" of [Morris, 1971]. Scott actually suggested the more elegant rule:

If $\varphi(\Omega)$ *holds and* $\forall f[\varphi(f) \Rightarrow \varphi(\tau[f])]$ *holds, then* $\varphi(f_P)$ *holds,*

which does not assume any knowledge of the integers in its formulation. These rules generalize easily to systems of mutually recursive definitions.

Example. Consider the recursive programs

$$P_3: F(x, y, z) \Leftarrow \textit{if } x = 0 \textit{ then } y \textit{ else } F(x - 1, y + z, z)$$

and

$$P_4: G(x, y) \Leftarrow \textit{if } x = 0 \textit{ then } y \textit{ else } G(x - 1, y + 2x - 1).$$

We would like to prove, using computational induction,

$f_{P_3}(x, 0, x) \equiv g_{P_4}(x, 0)$ for any natural number x.

(Both functions compute the square of x.)

[2] $\tau[f^{i-1}]$ is the result of replacing all occurrences of F in $\tau[F]$ by f^{i-1}.

[3] Note that this includes implicitly the need to prove $\varphi(f^0)$, since for $i = 0$ there is no j such that $j < i$.

For this purpose, we shall prove a stronger result than the desired one by simple computational induction. Proving a stronger result often simplifies proofs by induction, since it allows the use of a stronger induction hypothesis. So, using

$$\varphi(f, g) : \forall x \forall y[f(y, x(x - y), x) \equiv g(y, x^2 - y^2)],$$

we try to show that

$$\varphi(f_{P_3}, g_{P_4}) : \forall x \forall y[f_{P_3}(y, x(x - y), x) \equiv g_{P_4}(y, x^2 - y^2)]$$

holds. The desired result then follows by choosing $x = y$. The induction proceeds in two steps:

a. $\varphi(f^0, g^0)$,
i.e. $\forall x \forall y[f^0(y, x(x - y), x) \equiv g^0(y, x^2 - y^2)]$.
Trivial, since $\forall x \forall y[\textit{undefined} \equiv \textit{undefined}]$.

b. $\forall i[\varphi(f^i, g^i) \Rightarrow \varphi(f^{i+1}, g^{i+1})]$.
We assume $\forall x \forall y[f^i(y, x(x - y), x) \equiv g^i(y, x^2 - y^2)]$
and prove $\forall x \forall y[f^{i+1}(y, x(x - y), x) \equiv g^{i+1}(y, x^2 - y^2)]$.

$$\begin{aligned}
&f^{i+1}(y, x(x - y), x)\\
&\quad \equiv \textit{if } y = 0 \textit{ then } x(x - y)\\
&\qquad\qquad \textit{else } f^i(y - 1, x(x - y) + x, x)\\
&\quad \equiv \textit{if } y = 0 \textit{ then } x^2\\
&\qquad\qquad \textit{else } f^i(y - 1, x(x - (y - 1)), x)\\
&\quad \equiv \textit{if } y = 0 \textit{ then } x^2\\
&\qquad\qquad \textit{else } g^i(y - 1, x^2 - (y - 1)^2)\\
&\qquad \text{by the induction hypothesis}\\
&\quad \equiv \textit{if } y = 0 \textit{ then } x^2 - y^2\\
&\qquad\qquad \textit{else } g^i(y - 1, (x^2 - y^2) + 2y - 1)\\
&\quad \equiv g^{i+1}(y, x^2 - y^2).
\end{aligned}$$

1.3 Semantics of Algol-like Programs

Our purpose in this section is to illustrate how one can describe the semantics of an ALGOL-like program P by translating it into a recursive program P' such that the partial function computed by P is identical to the least fixpoint of P'. The features of ALGOL we consider are very simple indeed, but there is no theoretical difficulty in extending them.

The translation is defined blockwise: to each block B (or elementary statement) we associate a partial function f_B describing the effect of the block (or statement) on the values of the variables. For example,

begin $x := x + 1; y := y + 1$ *end*,

will be represented by the function

$$f(x, y) \equiv (x + 1, y + 1).$$

Functions are then combined to represent the whole program using the rule:

$$f_{B_1 ; B_2}(\mathbf{x}) \equiv f_{B_2}(f_{B_1}(\mathbf{x})).$$

This definition is unambiguous, since composition of partial functions is associative, i.e.

$$f_{B_3}(f_{B_1 ; B_2}(\mathbf{x})) \equiv f_{B_1 ; B_2 ; B_3}(\mathbf{x}) \equiv f_{B_2 ; B_3}(f_{B_1}(\mathbf{x})).$$

All that remains to be done is to describe the partial function associated with each elementary statement of the language. For simplicity, we shall first consider only a "flowchartable" subset of a language, with no goto statements or procedure calls. We shall also ignore the problem of declarations.

1. *Assignment statements*:

if B is $x_i := E(\mathbf{x})$ where E is an expression,
$f_B(\mathbf{x})$ is $(x_1, \cdots, x_{i-1}, E(\mathbf{x}), x_{i+1}, \cdots, x_n)$.

2. *Conditional statements*:

if B is **if** $p(\mathbf{x})$ **then** B_1,
$f_B(\mathbf{x})$ is *if* $p(\mathbf{x})$ *then* $f_{B_1}(\mathbf{x})$ *else* $\mathbf{x}$,

and

if B is **if** $p(\mathbf{x})$ **then** B_1 **else** B_2,
$f_B(\mathbf{x})$ is *if* $p(\mathbf{x})$ *then* $f_{B_1}(\mathbf{x})$ *else* $f_{B_2}(\mathbf{x})$.

3. *Iterative statements*:

if B is **while** $p(\mathbf{x})$ **do** B_1,
$f_B(\mathbf{x})$ is the least fixpoint of the recursive program
$F(\mathbf{x}) \Leftarrow \textit{if } p(\mathbf{x}) \textit{ then } F(f_{B_1}(\mathbf{x})) \textit{ else } x$.

Example. Let us consider the following program for computing the greatest natural number x smaller than or equal to $\sqrt{a}$, i.e. $x^2 \leq a < (x + 1)^2$, where a is any natural number. (The computation method is based on the fact that $1 + 3 + 5 + \cdots + (2n - 1) = n^2$ for every $n > 0$.)

```
P5: begin integer x, y, z;
      x := 0; y := z := 1;
      while y ≤ a do
        begin x := x + 1;
          z := z + 2;
          y := y + z;
        end;
    end.
```

The partial function computed by P_5 is identical to the least fixpoint of P_5', where

$$\begin{aligned}
P_5' :\ & F_0(a) \Leftarrow F(a, 0, 1, 1)\\
& F(a, x, y, z) \Leftarrow \textit{if } y \leq a\\
&\qquad\qquad \textit{then } F(a, x + 1, y + z + 2, z + 2)\\
&\qquad\qquad \textit{else } (a, x, y, z).
\end{aligned}$$

4. *Goto statements*: There has been much discussion (see, for example [Dijkstra, 1968; Knuth and Floyd 1971; Ashcroft and Manna, 1971]) about the usefulness of **goto** statements: they tend to make programs diffi-

cult to understand and debug, and one might prefer to use **while** or **for** statements instead. Without entering further into this controversy, we shall see that the semantics of **goto** statements is quite complex. In particular, it may lead to systems of mutually recursive definitions, and (not too surprisingly) it is indeed harder to prove properties of programs with **goto** statements. We consider two simple cases.

If we have a block of the form

begin $\cdots$; L: B_1; $\cdots$; B_{i-1}; **goto** L; B_{i+1}; $\cdots$; B_n **end**,

then we define

$f_{\textbf{goto } L;\ B_{i+1};\ \ldots;\ B_n}(\mathbf{x})$ to be the least fixpoint of the recursive program $F_L(\mathbf{x}) \Leftarrow f_{B_1;\ldots;B_n}(\mathbf{x})$.

If we have a block of the form

begin $\cdots$; **goto** L; B_1; $\cdots$; B_{i-1}; L: B_i; B_{i+1}; $\cdots$; B_n **end**,

then we define

$f_{\textbf{goto } L;B_1;\cdots;B_n}(\mathbf{x})$ to be the least fixpoint of the recursive program $F_L(\mathbf{x}) \Leftarrow f_{B_i;\ldots;B_n}(\mathbf{x})$.

Note that we have revised our rule of composition, since $f_{B;B'}(\mathbf{x}) \equiv f_{B'}(f_B(\mathbf{x}))$ is not valid when B is a **goto** statement. Similarly, if we wish to allow **goto**'s which jump out of iterative statements or branches of conditional statements, then we must change their semantic definition accordingly.

Example. Let us consider another version of P_5, using only the operations *successor* and *predecessor*.

```
P6: begin integer x, y, z;
      x := 0;  y := z := 1;
      L: if y ≤ a then
        begin integer t;
          x := x + 1;
          z := z + 1;
          t := z + 1;
          M: if t > 0 then
          begin y := y + 1;
            t := t − 1;
            goto M;
          end;
          z := z + 1; goto L;
        end;
    end.
```

The partial function computed by P_6 is identical to the least fixpoint of P_6' where

$$P_6'\colon F_0(a) \Leftarrow F_L(a, 0, 1, 1),$$
$$F_L(a, x, y, z) \Leftarrow \textit{if } y \leq a \textit{ then } F_M(a, x+1, y, z+1, z+2) \textit{ else } (a, x, y, z),$$
$$F_M(a, x, y, z, t) \Leftarrow \textit{if } t > 0 \textit{ then } F_M(a, x, y+1, z, t-1) \textit{ else } F_L(a, x, y, z+1).$$

Let us now define the semantics of simple procedures without parameters. We shall not discuss problems such as "side effects," parameter passing, or the procedure copy-rule for call by name.

5. *Procedures*:

a. For the nonrecursive procedure

procedure P; B

(where P is the procedure name and B is its body), we define

$f_{\textbf{call } P}(\mathbf{x})$ to be $f_B(\mathbf{x})$.

b. For the recursive procedure

procedure P; $B[P]$,

we define

$f_{\textbf{call } P}(\mathbf{x})$ to be the least fixpoint of the recursive program $F(\mathbf{x}) \Leftarrow f_{B[P]}(\mathbf{x})$

where occurrences of **call** P will be replaced by F in the semantic definition $f_{B[P]}$.

6. *An answer to the problem of "call by value"*: Our semantic definition of recursive procedures assumes that the implementation of recursion in the language always leads to the least fixpoint. If this is not the case, we must change our semantic definition: to every program P we associate a recursive program P' such that the least fixpoint of P' *will always be identical* to the partial function computed by P. Consider, for example, the program

integer procedure P (**integer** x, y);
$P :=$ **if** $x = 0$ **then** 1 **else** $P(x - 1, P(x, y))$;

If the implementation is "call by name," its semantics will be

$f_{\textbf{call } P}(x, y)$ is the least fixpoint of
$F(x, y) \Leftarrow \textit{if } x = 0 \textit{ then } 1 \textit{ else } F(x - 1, F(x, y))$.

However, if the implementation is "call by value," its semantics will be

$f_{\textbf{call } P}(x, y)$ is the least fixpoint of
$F(x, y) \Leftarrow \textit{if } (x = 0) \wedge \mathrm{def}(y) \textit{ then } 1 \textit{ else } F(x - 1, F(x, y))$,

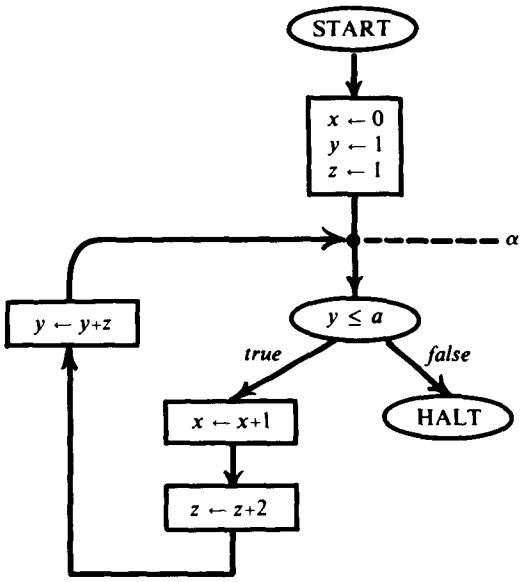
where the (computable) predicate $\mathrm{def}(y)$ is true whenever y is defined, and undefined otherwise.

Part 2. Application to the Verification Problem

Our purpose in the second part of the paper is to illustrate some of the advantages of the fixpoint approach to program semantics.

2.1 Justification of the Inductive Assertions Method

The most widely used method for proving properties of "flowchart programs" is presently the *inductive assertions method*, suggested in [Floyd, 1967 and Naur, 1966]. We shall illustrate the method on the simple program P_5 above. To clarify our discussion we shall describe the program as a flowchart:



We wish to show that this flowchart program, whenever it terminates, computes the greatest natural number smaller than or equal to $\sqrt{a}$, i.e. that $x^2 \leq a < (x+1)^2$ for any natural number a.

To do this we associate a predicate $Q(a, x, y, z)$, called an *inductive assertion*, with the point labelled α in the program, and show that Q must be true for the values of the variables (a, x, y, z) whenever execution of the program reaches point α. Thus, we must show: (a) that if we start execution with $a \geq 0$, then the assertion holds when point α is first reached, i.e. that $Q(a, 0, 1, 1)$ holds; and (b) that the assertion remains true when one goes around the loop from α to α, i.e. that $(y \leq a) \wedge Q(a, x, y, z)$ implies $Q(a, x+1, y+z+2, z+2)$. To prove the desired result we finally show (c) that $x^2 \leq a < (x+1)^2$ follows from the assertion $Q(a, x, y, z)$ when the program terminates, i.e. that $(y > a) \wedge Q(a, x, y, z)$ implies $x^2 \leq a < (x+1)^2$.

To verify the program, we take

$Q(a, x, y, z)$ to be
$$(x^2 \leq a) \wedge (y = (x+1)^2) \wedge (z = 2x+1).$$

We can then verify easily that conditions (a), (b), and (c) above, called the *verification conditions*, hold.

Hoare's inductive assertion method [Hoare, 1969, 1971] is actually a generalization of Floyd's method; he realized that if we wish to apply the method of inductive assertions to prove properties of a large program, we undoubtedly have to break the program into smaller parts, prove what we need about the parts, and then combine everything together. We will clearly break the program into pieces in the most convenient way for the proof, and, since composition of statements is associative, the way in which we group the statements of the program is irrelevant. For example, if the given program is of the form

$B_1 ; B_2 ; B_3 ; B_4$,

we can associate the statements in several different ways, e.g.

$((B_1 ; B_2) ; B_3) ; B_4$,
$(B_1 ; (B_2 ; B_3)) ; B_4$,
$(B_1 ; B_2) ; (B_3 ; B_4)$, or
$B_1 ; (B_2 ; (B_3 ; B_4))$.

Although the programs do not look the same, all of them yield the same least fixpoint, and therefore they are equivalent. If we express other verification techniques using this notation, we find that Floyd and Naur consider only the first possibility, i.e. grouping statements to the left, while McCarthy and Manna and Pnueli only consider the last possibility, i.e. grouping statements to the right. (See [McCarthy, 1963b; Manna and Pnueli, 1970].)

Following Hoare, we express this idea by writing[4] $\{R\}B\{T\}$ to mean that if $R(\mathbf{x})$ holds before executing the piece of program B and if B terminates, then $T(\mathbf{x})$ will hold after executing B.

We first apply *verification rules* to each statement of the program. Examples of such rules are:

a. *Assignment statement rule*:

$R \supset S_{x_i}^{E(\mathbf{x})}$ *implies* $\{R\}x_i := E(\mathbf{x})\{S\}$

where $S_{x_i}^{E(\mathbf{x})}$ stands for the result of replacing all occurrences of x_i in S by $E(\mathbf{x})$.

b. *Conditional statement rule*:

$\{R \wedge p\}B_1\{T\}$ *and* $\{R \wedge \sim P\}B_2\{T\}$ *implies* $\{R\}$ **if** p **then** B_1 **else** B_2 $\{T\}$.

c. *Iterative statement rule*:

$\{R \wedge p\}B\{R\}$ *implies* $\{R\}$ **while** p **do** $B\{R \wedge \sim p\}$.

We then compose pieces of the program until we get the entire program, using the following rules.

d. *Composition rule*:

$\{R\}B_1\{S\}$ *and* $\{S\}B_2\{T\}$ *implies* $\{R\}B_1 ; B_2\{T\}$.

e. *Consequence rules*:

$R \supset S$ *and* $\{S\}B\{T\}$ *implies* $\{R\}B\{T\}$, and
$\{R\}B\{S\}$ *and* $S \supset T$ *implies* $\{R\}B\{T\}$.

[4] We prefer this notation to Hoare's $R\{B\}T$.

Example. A proof of the correctness of the program P_5, given above, could be sketched as follows.

First, we establish, using the assignment statement rule, the following results:

Since $a \geq 0 \supset R(a, 0, 1, 1)$, where $R(a, x, y, z)$ is $(x^2 \leq a) \wedge (y = (x+1)^2) \wedge (z = 2x + 1)$, we get

$$\{a \geq 0\} x := 0; y := z := 1 \{R(a, x, y, z)\}. \quad (1)$$

Since $R(a, x, y, z) \wedge y \leq a \supset R(a, x+1, y+z+2, z+2)$, we get

$$\{R(a, x, y, z) \wedge y \leq a\} x := x + 1; z := z + 2; y := y + z \{R(a, x, y, z)\}. \quad (2)$$

By using the iterative statement rule, we get from (2)

$$\{R(a, x, y, z)\} \textbf{ while } y \leq a \textbf{ do begin } x := x + 1; z := z + 2; y := y + z \textbf{ end } \{R(a, x, y, z) \wedge y > a\}. \quad (3)$$

We now combine the results of (1) and (3) using the composition rule to obtain

$$\{a \geq 0\} P_5 \{R(a, x, y, z) \wedge y > a\}. \quad (4)$$

Since $[R(a, x, y, z) \wedge y > a] \supset x^2 \leq a < (x+1)^2$, we apply the consequence rule and finally get

$$\{a \geq 0\} P_5 \{x^2 \leq a < (x+1)^2\}. \quad (5)$$

The reader may wonder why those rules are valid. It is therefore important that Hoare's verification rules can in fact be *proved* from the semantics we gave, just by using computational induction. We shall illustrate this point by justifying two of the most powerful verification rules: the rule for **while** statements, and the rule for **call** of recursive procedures. For this purpose, we need to relate the notation $\{R\}B\{T\}$ to our $f_B(\mathbf{x})$, the partial function indicating the change of the values of the variables during the execution of B. $\{R\}B\{T\}$ simply means that whenever $R(\mathbf{x})$ is *true*, $T(f_B(\mathbf{x}))$ is either *true* (if B terminates) or *undefined*. We can express this by the relation

$$R(\mathbf{x}) \Rightarrow T(f_B(\mathbf{x})),$$

where "$\Rightarrow$" is the usual implication, with the additional convention that $a \Rightarrow b$ is *true* whenever a or b is *undefined*.

We are ready now to prove the following rules:

a. *Rule for while statements.* The verification rule for **while** statements indicates that if the execution of the body of the **while** statement leaves the assertion R invariant, R should hold upon termination of the **while** statement. More precisely,

$$\{R(\mathbf{x}) \wedge p(\mathbf{x})\} B \{R(\mathbf{x})\} \textit{ implies } \{R(\mathbf{x})\} \textbf{ while } p(\mathbf{x}) \textbf{ do } B \{R(\mathbf{x}) \wedge \sim p(\mathbf{x})\}.$$

We therefore have to prove the following:

THEOREM. $\forall \mathbf{x}[R(\mathbf{x}) \wedge p(\mathbf{x}) \Rightarrow R(f_B(\mathbf{x}))]$ *implies*

$\forall \mathbf{x}[R(\mathbf{x}) \Rightarrow [R(f_P(\mathbf{x})) \wedge \sim p(f_P(\mathbf{x}))]]$ *where* $P: F(\mathbf{x}) \Leftarrow$ *if* $p(\mathbf{x})$ *then* $F(f_B(\mathbf{x}))$ *else* $\mathbf{x}$.

PROOF. By computational induction.

1. $\forall \mathbf{x}[R(\mathbf{x}) \Rightarrow R(f^0(\mathbf{x})) \wedge \sim p(f^0(\mathbf{x}))]$ is clearly true according to our convention, of $\Rightarrow$, since $R(f^0(\mathbf{x}))$ and $\sim p(f^0(\mathbf{x}))$ are *undefined.*

2. We assume $\forall \mathbf{x}[R(\mathbf{x}) \Rightarrow R(f^i(\mathbf{x})) \wedge \sim p(f^i(\mathbf{x}))]$ and show $\forall \mathbf{x}[R(\mathbf{x}) \Rightarrow R(f^{i+1}(\mathbf{x})) \wedge \sim p(f^{i+1}(\mathbf{x}))]$. By definition of f^{i+1} we have

$R(f^{i+1}(\mathbf{x})) \equiv$ *if* $p(\mathbf{x})$ *then* $R(f^i(f_B(\mathbf{x})))$ *else* $R(\mathbf{x})$, and

$p(f^{i+1}(\mathbf{x})) \equiv$ *if* $p(\mathbf{x})$ *then* $p(f^i(f_B(\mathbf{x})))$ *else* $p(\mathbf{x})$.

We distinguish between two cases:[5]

Case 2 A. $p(\mathbf{x})$ is *false*. Then $R(f^{i+1}(\mathbf{x})) \equiv R(\mathbf{x})$ and $p(f^{i+1}(\mathbf{x})) \equiv p(\mathbf{x})$, so that $R(\mathbf{x}) \Rightarrow R(f^{i+1}(\mathbf{x})) \wedge \sim p(f^{i+1}(\mathbf{x}))$ is valid.

Case 2B. $p(\mathbf{x})$ is *true*. Then $R(f^{i+1}(\mathbf{x})) \equiv R(f^i(f_B(\mathbf{x})))$ and $p(f^{i+1}(\mathbf{x})) \equiv p(f^i(f_B(\mathbf{x})))$. By the assumption $R(\mathbf{x}) \wedge p(\mathbf{x}) \Rightarrow R(f_B(\mathbf{x}))$ holds, and since by the induction hypothesis $R(f_B(\mathbf{x})) \Rightarrow R(f^i(f_B(\mathbf{x}))) \wedge \sim p(f^i(f_B(\mathbf{x})))$, we get $R(\mathbf{x}) \Rightarrow R(f^i(f_B(\mathbf{x}))) \wedge \sim p(f^i(f_B(\mathbf{x})))$. Hence, $R(\mathbf{x}) \Rightarrow R(f^{i+1}(\mathbf{x})) \wedge \sim p(f^{i+1}(\mathbf{x}))$ as desired.

b. *Rule for recursive calls.* Let us consider a recursive procedure

procedure P; $B[P]$,

where P is the name of the procedure and $B[P]$ represents its body. The verification rule for proving properties of P is quite similar to computational induction, although its formulation might look rather paradoxical: in order to prove a property of the recursive procedure P, we are permitted to assume that the desired property holds for the body $B[P]$ of the procedure! This can be stated as follows:

$\forall g\, [\{R\}g\{T\}$ *implies* $\{R\}B[g]\{T\}]$
implies $\{R\}$ **call** $P\,\{T\}$.

In [Hoare, 1971, p. 109] it is stated, "this assumption of what we want to prove before embarking on the proof explains well the aura of magic which attends a programmers's first introduction to recursive programming."

The rule however is easy to justify. We have to prove the following:

THEOREM.

$\forall g[\forall \mathbf{x}[R(\mathbf{x}) \Rightarrow T(g(x))]$ *implies* $\forall \mathbf{x}[R(\mathbf{x}) \Rightarrow T(f_{B[g]}(\mathbf{x}))]]$
implies
$\forall \mathbf{x}[R(\mathbf{x}) \Rightarrow T(f(\mathbf{x}))]$ *where* $P{:}F(\mathbf{x}) \Leftarrow f_{B[F]}(\mathbf{x})$.

PROOF. Again by computational induction.

1. $\forall x[R(\mathbf{x}) \Rightarrow T(f^0(\mathbf{x}))]$ is *true*, since $T(f^0(\mathbf{x}))$ is *undefined.*

2. We assume $\forall \mathbf{x}[R(\mathbf{x}) \Rightarrow T(f^i(\mathbf{x}))]$ and show $\forall \mathbf{x}[R(\mathbf{x}) \Rightarrow T(f^{i+1}(\mathbf{x}))]$. By the induction hypothesis, $R(\mathbf{x}) \Rightarrow T(f^i(\mathbf{x}))$; therefore by the assumption of the theorem, $R(\mathbf{x}) \Rightarrow T(f_{B[f^i]}(\mathbf{x}))$. Thus from the definition of f^{i+1} we get $R(\mathbf{x}) \Rightarrow T(f^{i+1}(\mathbf{x}))$, as desired.

[5] A more rigorous treatment would also require checking the case in which $p(\mathbf{x})$ is *undefined.*

2.2 Translation to Recursive Programs

In the present state of the art of verifying programs, Hoare's method is presumably the most convenient for proving the correctness of programs. However, its main drawback is that it can handle only "partial correctness" of programs, i.e. we can only show that the final results of the programs, if any, satisfy some given input-output relation. The method does not provide us any means for proving termination, and seems rather ill fitted for proving equivalence between programs.

To show, for example, that the partial function defined by a given program P is monotonic increasing, we have to prove

$$\forall x, y[(x < y) \Rightarrow (f_{P'}(x) \leq f_{P'}(y))].$$

Note that it is rather awkward to express such a property as an input-output relation.

This is another case where our semantic definition of the programming language pays off: properties like termination and equivalence can be handled in exactly the same way as partial correctness. The idea is quite simple: To prove some property of a given program P, translate it to the corresponding recursive program P', and then prove the desired property for $f_{P'}$, by computational induction. In this method we may still associate the blocks of the program arbitrarily at our convenience.

Termination. To show that f_P is total, or in general that $g \subseteq f_P$ for some function g which is total on the desired domain, we cannot simply use computational induction choosing $\varphi(F)$ to be $g \subseteq F$, as then $\varphi(f^0)$ will always be *false*. However, we can overcome this difficulty by considering the domain over which our data range is defined by a recursive program.

For example, the natural numbers can be characterized[6] by the least fixpoint $\mathrm{num}(x)$ of the recursive program

$$N(x) \Leftarrow \textit{if } x = 0 \textit{ then true else } N(x - 1).$$

We can now translate any program P over the natural numbers into the corresponding recursive program P' and show that P' terminates by simply proving the relation

$$\forall x[\mathrm{num}(x) \subseteq \mathrm{num}(f_{P'}(x))].$$

In other words, $f_{P'}(x)$ is defined and its value is a natural number, whenever x is a natural number.

Equivalence. It should be quite clear at this point that equivalence of two recursive programs is no more difficult to prove than the other properties. Consider, for example, the two recursive programs over the natural numbers

$$P_7\colon F(x) \Leftarrow \textit{if } x = 0 \textit{ then } 1 \textit{ else } x \cdot F(x - 1),$$

and

$$P_8\colon G(x, y, z) \Leftarrow \textit{if } x = y \textit{ then } z \textit{ else } G(x, y + 1, (y + 1) \cdot z).$$

[6] Given that 0, 1, −, +, = have their usual meaning.

We want to show that

$$\forall x[f_{P_7}(x) \equiv g_{P_8}(x, 0, 1)].$$

Note that both $f_{P_7}(x)$ and $g_{P_8}(x, 0, 1)$ computes $x!$, but quite differently: $f_{P_7}(x)$ is "going down" from x to 0, while $g_{P_8}(x, 0, 1)$ is "going up" from 0 to x. This explains why a "direct" computational induction fails in this case.

However, if we consider the predicate $x \geq y$ over the natural numbers to be characterized by the least fixpoint $ge(x, y)$ of the recursive program

$$M(x, y) \Leftarrow \textit{if } x = y \textit{ then true else } M(x, y + 1),$$

we can show by computational induction that

$$\forall x, y[ge(x, y) \subseteq [f_{P_7}(x) = g_{P_8}(x, y, f_{P_7}(y))]].$$

Then, in particular, for $y = 0$ we get

$$\forall x[ge(x, 0) \subseteq [f_{P_7}(x) = g_{P_8}(x, 0, 1))]],$$

i.e. for every natural number x, both $f_{P7}(x)$ and $g_{P8}(x, 0, 1)$ must be defined and equal.

The proof by computational induction uses

$$\varphi(F)\colon \forall x, y[F(x, y) \subseteq [f_{P_7}(x) = g_{P_8}(x, y, f_{P_7}(y))]].$$

It is clear that $\varphi(f^0)$ holds. So, we assume that $\varphi(f^i)$ holds and show that $\varphi(f^{i+1})$ holds, i.e.

$$\forall x, y[f^{i+1}(x, y) \subseteq [f_{P_7}(x) = g_{P_8}(x, y, f_{P_7}(y))]],$$

or in other words,

$$\forall x, y[[\textit{if } x = y \textit{ then true else } f^i(x, y + 1)] \subseteq [f_{P_7}(x) = g_{P_8}(x, y, f_{P_7}(y))]].$$

The proof proceeds easily by distinguishing between the two cases where $x = y$ and $x \neq y$.

a. If $x = y$ we get $\forall x[\textit{true} \subseteq f_{P_7}(x) = f_{P_7}(x)]$, which is equivalent to showing that f_{P_7} is total.

b. If $x \neq y$ then

$$\forall x, y[f^i(x, y + 1) \subseteq [f_{P_7}(x) = g_{P_8}(x, y, f_{P_7}(y))]].$$

Using the definitions of f_{P_7} and g_{P_8} we get

$$\forall x, y[f^i(x, y + 1) \subseteq [f_{P_7}(x) = g_{P_8}(x, y + 1, f_{P_7}(y + 1))]],$$

which holds by the induction hypothesis.

References

Ashcroft, E., and Manna, Z. (1971). The translation of "goto" programs to "while" programs. Proc. IFIP Cong. 1971.

Bekić, H. (1969). Definable operations in general algebra and the theory of automata and flowcharts. Unpublished memo, IBM, Vienna. Dec. 1969.

Burstall, R.M. (1969). Proving properties of programs by structural induction. *Comput. J. 12*, 1 (Feb. 1969), 41–48.

Cadiou, J.M. (1972). Recursive definitions of partial functions and their computations. Ph.D. Th., Computer Sci. Dept., Stanford U., Stanford, Calif., June 1972.

Cooper, D.C. (1971). Programs for mechanical program verification. In *Machine Intelligence 6*, B. Meltzer and D. Michie, (Eds.), Edinburgh University Press, 1971, pp. 43–59.
deBakker, J.W., and Scott, D. (1969). A theory of programs. Unpublished memo, Aug. 1969.
Dijkstra, E. (1968). Goto statements considered harmful. *Comm. ACM 11*, 3 (Mar. 1968), 147–148.
Floyd, R.W. (1967). Assigning meanings to programs. In Proc. of a Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science, J.T. Schwartz (Ed.), Amer. Math. Soc., pp. 19–32.
Hoare, C.A.R. (1969). An axiomatic approach to computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–580, 583.
Hoare, C.A.R. (1971). Procedures and parameters: An axiomatic approach. In Symposium on Semantics of Algorithmic Languages, Lecture notes Mathematics, Vol. 188, E. Engelet (Ed.), Springer-Verlag, Berlin, pp. 102–116.
Kleene, S.C. (1952). *Introduction to Meta-mathematics*. Van Nostrand, Princeton, N.J., 1952.
Knuth, D.E., and Floyd, R.W. (1971). Notes on avoiding "goto" statements. *Information Processing Letters 1* (Jan. 1971), 23–31.
Landin, P.J. (1965). A correspondence between ALGOL 60 and Church's lamda-notation. *Comm. ACM 8*, 2 (Feb. 1965), 89–101; *Comm. ACM* 8, 3 (Mar. 1965), 158–165.
Manna, Z. (1969). The correctness of programs. *J. Computer & System Sciences 3*, 2 (May 1969), 119–127.
Manna, Z., Ness, S., and Vuillemin, J. (1972). Inductive methods for proving properties of programs. Proc. ACM Conf. on Proving Assertions about Programs, ACM, New York, 1972.
Manna, Z., and Pnueli, A. (1970). Formalization of properties of functional programs. *J. ACM 17*, 3 (July 1970), 555–569.
McCarthy, J. (1963a). A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (eds.), Humanities Press, New York, 1936, pp. 33–70.
McCarthy, J. (1963 b). Towards a mathematical science of computation. Proc. IFIP Cong. 1962, North Holland Pub. Co., Amsterdam, pp. 21–28.
Milner, R. (1972). Implementation and applications of Scott's logic for computable functions. Proc. ACM Conf. on Proving Assertions about Programs, ACM, New York, 1972.
Morris, J. H. (1968). Lambda-calculus models for programming languages, Ph.D. Th., Project MAC, MAC-TR-57, MIT, Cambridge, Mass., Dec. 1968.
Morris, J. H. (1971). Another recursion induction principle, *Comm. ACM 14*, 5 (May 1971), 351–354.
Naur, P. (1966). Proof of algorithms by general snapshots, *BIT 6* (1966), 310–316.
Park, D. (1969). Fixpoint induction and proofs of program properties. In *Machine Intelligence 5*, B. Meltzer and D. Michie (Eds.), Edinburgh University Press, pp. 59–78.
Scott, D. (1970). Outline of a mathematical theory of computation. Oxford U. Computing Lab., Programming Res. Group, Tech. Mono. PRG-2, Oxford, England, Nov. 1970.
Scott, D., and Strachey, C. (1971). Towards a mathematical semantics for computer languages. Tech. Mono. PRG-6, Oxford U., Oxford, England, Aug. 1971.
Strachey, C. (1966). Towards a formal semantics. Proc. IFIP Working Conf. 1964, North-Holland Pub. Co., Amsterdam, pp. 198–220.
Vuillemin, J. (1972). Proof techniques for recursive programs. Ph.D. Th., Computer Sci. Dept., Stanford U., Stanford, Calif. (to appear).
Weyhrauch, R., and Milner, R. (1972). Program semantics and correctness in a mechanized logic. The USA-Japan Computer Conf., Tokyo, Oct. 1972.