STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-164

STAN-CS-72-272

AD 742748

FIXPOINT APPROACH TO THE THEORY OF COMPUTATION

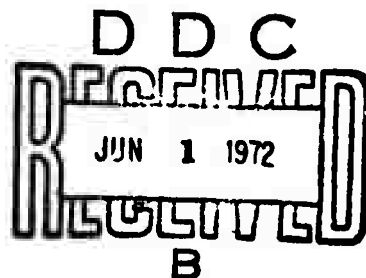
BY

ZOHAR MANNA

JEAN VUILLEMIN

SUPPORTED BY
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
AND
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

MARCH 1972



COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE

31

Stanford Artificial Intelligence Project
Memo AIM-164

April 1972

FIXPOINT APPROACH TO THE THEORY OF COMPUTATION

by

Zohar Manna and Jean Vuillemin
Computer Science Department
Stanford University

Abstract

Following the fixpoint theory of Scott, we propose to define the semantics of computer programs in terms of the least fixpoints of recursive programs. This allows one not only to justify all existing verification techniques, but also to extend them to handle various properties of computer programs, including correctness, termination and equivalence, in a uniform manner.

Keywords 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 in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract SD-183 and in part by NASA Contract 2FCZ 713.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Project Agency or the U.S. Government.

Reproduced in the USA. Available from the Clearinghouse for Federal Scientific and Technical Information. Springfield, Virginia 22151.

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 we first introduce the notion of a recursive program and its (unique) least fixpoint. 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 those of McCarthy [1963a, 1963b], Landin [1964], Strachey [1966], Morris [1968], Bekić [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 computational 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 convenient 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

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

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

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

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 partial functions:

$f_1(x,y)$: $x+1$

$f_2(x,y)$: $\underline{\text{if}}\; x \geq y \;\underline{\text{then}}\; x+1 \;\underline{\text{else}}\; y-1$, and

$f_3(x,y)$: $\underline{\text{if}}\; (x \geq y) \wedge (x-y \text{ even}) \;\underline{\text{then}}\; x+1 \;\underline{\text{else}}\; \underline{\text{undefined}}$.

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 lefthand side and the righthand side of the symbol $\Leftarrow$ yield identical partial functions, i.e.,*/

---

*/ $\equiv$ is an extension of the regular $=$ relation for handling undefined values. $a \equiv b$ is true if both a and b are undefined, but it is false if only one of them is undefined.

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

We say that the functions f_1 , f_2 and f_3 are <u>fixpoints</u> of the recursive program P_0 .

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., $(x \geq y) \wedge (x\text{-}y \text{ even})$, 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 $f_3 \sqsubseteq f_1$ and $f_3 \sqsubseteq f_2$. 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 <u>only</u> function having this property; f_3 is therefore said to be the <u>least (defined) fixpoint of P_0</u> .

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

In discussing our recursive programs, the key problem is:
<u>What is the partial function f defined by a recursive program P ?</u>
There are two viewpoints.

(a) <u>Fixpoint approach</u>: Let it be the unique least fixpoint f_P .

(b) <u>Computational approach</u>: 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]).*/

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

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

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

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

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

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

Thus, C_{P_1} is properly less defined than f_{P_1} -- e.g., $C_{P_1}(1,0)$ is <u>undefined</u> while $f_{P_1}(1,0) = 1$.

There are two alternative ways to view this problem: (a) 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

---

*/ 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., $C_P \sqsubseteq f_P$ (Cadiou [1972]).

"call by value" so that the evaluation of the arguments of a procedure is delayed as long as possible (Vuillemin [1972]). (b) Theoreticians are wasting their time by 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 recursive programs. The idea of the method is essentially to prove properties of the least fixpoint f_P of a given recursive program P by induction on the level of recursion.

Let us consider, for example, the recursive program

$$P_2: \quad F(x) \Leftarrow \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x \cdot F(x-1) \quad ,$$

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 i-th level of recursion. That is,

$f^0(x)$ is undefined (for all x);

$f^1(x)$ is if $x = 0$ then 1 else $x \cdot f^0(x-1)$,

i.e., if $x = 0$ then 1 else undefined ;

$f^2(x)$ is if $x = 0$ then 1 else $x \cdot f^1(x-1)$,

i.e., if $x = 0$ then 1 else $x \cdot$(if $x-1 = 0$ then 1 else undefined) ,

or in short, if $x = 0 \vee x = 1$ then 1 else undefined ;

etc.

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

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

which is

if $x < i$ then $x!$ else undefined .

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

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

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

$f^0(\bar{x})$ is Ω (undefined for all $\bar{x}$), and

$f^i(\bar{x})$ is $\tau[f^{i-1}](\bar{x})$ for $i \geq 1$,*/

then

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

---

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

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 $\varphi(\underline{\lim}_{i \to \infty}\{f^i\})$, 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 non-total function $\underline{\text{if}}\ x < i\ \underline{\text{then}}\ x!\ \underline{\text{else}}\ \underline{\text{undefined}}$, while $\underline{\lim}_{i \to \infty}\{f^i\}$, 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 shall 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) <u>Simple induction</u>:

<u>if $\varphi(f^0)$ holds and $\forall i[\varphi(f^i) \Rightarrow \varphi(f^{i+1})]$ holds ,</u>

<u>then $\varphi(f_P)$ holds .</u>

(b) <u>Complete induction</u>:

<u>if $\forall i\{[\forall j \text{ such that } j < i)\varphi(f^j)] \Rightarrow \varphi(f^i)\}$ holds ,</u>*

<u>then $\varphi(f_P)$ holds .</u>

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

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

$$\frac{\text{if } \varphi(\Omega) \text{ holds and } \forall f[\varphi(f) \Rightarrow \varphi(\tau[f])] \text{ holds ,}}{\text{then } \varphi(f_P) \text{ holds}} ,$$

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 : $F(x,y,z) \Leftarrow$ <u>if</u> $x = 0$ <u>then</u> y <u>else</u> $F(x-1,y+z,z)$

and

P_4 : $G(x,y) \Leftarrow$ <u>if</u> $x = 0$ <u>then</u> y <u>else</u> $G(x-1,y+2x-1)$.

We would like to prove, using computational induction, that

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

(Both functions compute the square of x .)

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) $\underline{\varphi(f^0, g^0)}$, i.e., $\forall x \forall y [f^0(y, x(x-y), x) \equiv g^0(y, x^2-y^2)]$.

Trivial, since $\forall x \forall y [\underline{\text{undefined}} \equiv \underline{\text{undefined}}]$.

(b) $\underline{\forall i [\varphi(f^i, g^i) \Rightarrow \varphi(f^{i+1}, g^{i+1})]}$.

We assume

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

and prove

$$\forall x \forall y [f^{i+1}(y, x(x-y), x) \equiv g^{i+1}(y, x^2-y^2)] .$$

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

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,

$$\underline{\text{begin}}\ x := x+1;\ y := y+1\ \underline{\text{end}}\ ,$$

will be represented by the function

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

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

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

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

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

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 $x_i := E(\bar{x})$ where E is an expression,

$f_B(\bar{x})$ is $(x_1,\ldots,x_{i-1},E(\bar{x}),x_{i+1},\ldots,x_n)$.

2) Conditional statements

if B is $\underline{\text{if}}\ p(\bar{x})\ \underline{\text{then}}\ B_1$,

$f_B(\bar{x})$ is $\underline{\text{if}}\ p(\bar{x})\ \underline{\text{then}}\ f_{B_1}(\bar{x})\ \underline{\text{else}}\ \bar{x}$,

and

if B is *if* $p(x)$ *then* B_1 *else* B_2 ,

$f_B(\bar{x})$ is if $p(\bar{x})$ then $f_{B_1}(\bar{x})$ else $f_{B_2}(\bar{x})$.

3) Iterative statements

if B is *while* $p(\bar{x})$ *do* B_1 ,

$f_B(\bar{x})$ is the least fixpoint of the recursive program

$$F(\bar{x}) \Leftarrow \text{if } p(\bar{x}) \text{ then } F(f_{B_1}(\bar{x})) \text{ else } \bar{x} \quad .$$

Example: Let us consider the following program for computing in x the greatest natural number 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 $1+3+5+\ldots+(2n-1) = n^2$ 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

$$P_5' : F_0(a) \Leftarrow F(a,0,1,1)$$

$$F(a,x,y,z) \Leftarrow \text{if } y \leq a \text{ then } F(a,x+1,y+z+2,z+2) \text{ else } (a,x,y,z) \quad .$$

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 difficult 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

$$\underline{\text{begin}} \ \ldots \ ; \ L\text{: } B_1 \ ; \ \ldots \ ; \ B_{i-1} \ ; \ \underline{\text{goto}} \ L \ ; \ B_{i+1} \ ; \ \ldots \ ; \ B_n \ \underline{\text{end}} \ ,$$

then we define

$f_{\underline{\text{goto}}\ L;B_{i+1};\ldots;B_n}(\bar{x})$ to be the least fixpoint of the recursive program

$$F_L(\bar{x}) \Leftarrow f_{B_1;\ldots;B_n}(\bar{x}) \quad .$$

If we have a block of the form

$$\underline{\text{begin}} \ \ldots \ ; \ \underline{\text{goto}} \ L \ ; \ B_1 \ ; \ \ldots \ ; \ B_{i-1} \ ; \ L\text{: } B_i \ ; \ B_{i+1} \ ; \ \ldots \ ; \ B_n \ \underline{\text{end}} \ ,$$

then we define

$f_{\underline{\text{goto}}\ L;B_1;\ldots;B_n}(\bar{x})$ to be the least fixpoint of the recursive program

$$F_L(\bar{x}) \Leftarrow f_{B_i;\ldots;B_n}(\bar{x}) \quad .$$

Note that we have revised our rule of composition, since $f_{B;B'}(\bar{x}) \equiv f_{B'}(f_B(\bar{x}))$ 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': $F_0(a) \Leftarrow F_L(a,0,1,1)$

$F_L(a,x,y,z) \Leftarrow \underline{\text{if}}\ y \le a\ \underline{\text{then}}\ F_M(a,x+1,y,z+1,z+2)$
$\qquad\qquad\qquad\qquad \underline{\text{else}}\ (a,x,y,z)$

$F_M(a,x,y,z,t) \Leftarrow \underline{\text{if}}\ t > 0\ \underline{\text{then}}\ F_M(a,x,y+1,z,t-1)$
$\qquad\qquad\qquad\qquad \underline{\text{else}}\ F_L(a,x,y,z+1)$.

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 non-recursive procedure

procedure P;B

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

$f_{\text{call } P}(\bar{x})$ to be $f_B(\bar{x})$.

(b) For the recursive procedure

procedure P; B[P] ,

we define

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

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

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

$$f_{\text{call } P}(x,y) \quad \text{is the least fixpoint of}$$

$$F(x,y) \Leftarrow \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ F(x-1,F(x,y)) \quad .$$

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

$$f_{\text{call } P}(x,y) \quad \text{is the least fixpoint of}$$

$$F(x,y) \Leftarrow \underline{\text{if}}\ (x = 0) \wedge \text{def}(y)\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ F(x-1,F(x,y)) \quad ,$$

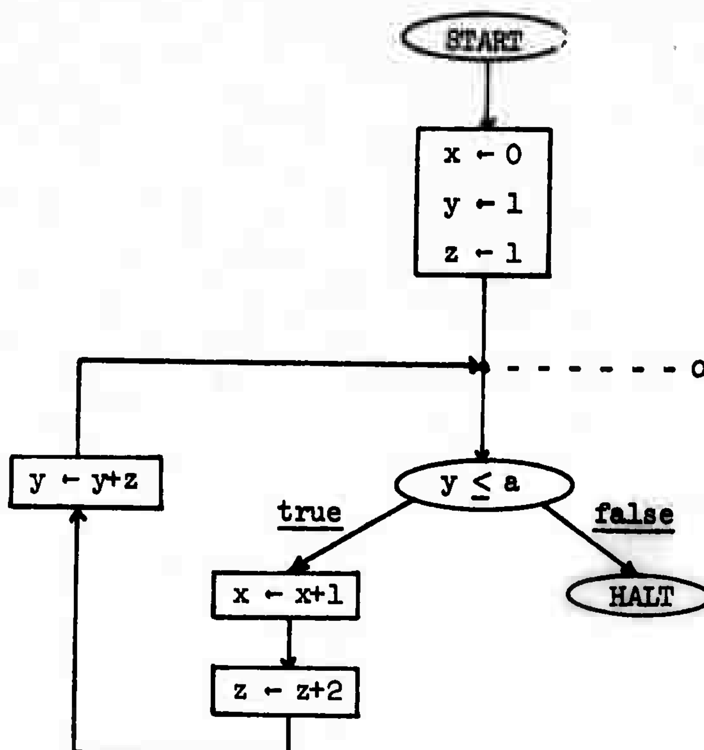
where the (computable) predicate def(y) is true whenever y is defined, and undefined otherwise.

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 by 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 <u>inductive assertion</u>, 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

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

One can then verify easily that conditions (a), (b) and (c) above, called the <u>verification conditions</u>, hold.

Hoare's inductive assertion method is actually a generalization of Floyd's method; Hoare [1969, 1971] realized that if we wish to apply the method of inductive assertions to prove properties of a large program, we shall 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

$$P: \quad B_1; B_2; B_3; B_4 \ ,$$

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

$((B_1; B_2); B_2); B_3$,

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

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

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 [1963b] and Manna and Pnueli [1970] only consider the last possibility, i.e., grouping statements to the right.

Following Hoare, we express this idea by writing $\{R\}B\{T\}$ */ to mean that if $R(\bar{x})$ holds before executing the piece of program B and if B terminates, then $T(\bar{x})$ 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:

$$\frac{R \supset S_{x_i}^{E(\bar{x})} \quad \text{implies} \quad \{R\}\ x_i := E(\bar{x})\ \{S\}}{}$$

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

---

*/ We prefer this notation to Hoare's $R\{B\}T$.

(b) conditional statement rule:

$$\frac{\{R_1\}B_1\{T\} \text{ and } \{R_2\}B_2\{T\} \quad \text{implies}}{\{\text{if } p(\bar{x}) \text{ then } R_1 \text{ else } R_2\}\ \mathbf{if}\ p(\bar{x})\ \mathbf{then}\ B_1\ \mathbf{else}\ B_2\ \{T\}}\ ,$$ */

(c) iterative statement rule:

$$\underline{\{R \wedge p(\bar{x})\}\ B\ \{R\} \quad \text{implies} \quad \{R\}\ \mathbf{while}\ p(\bar{x})\ \mathbf{do}\ B\ \{R \wedge \sim p(\bar{x})\}}\ .$$

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

(d) composition rule:

$$\underline{\{R\}B_1\{S\} \text{ and } \{S\}B_2\{T\} \quad \text{implies} \quad \{R\}B_1;B_2\{T\}}\ ,$$

(e) consequence rules:

$$\underline{R \supset S \text{ and } \{S\}B\{T\} \quad \text{implies} \quad \{R\}B\{T\}}\ , \text{ and}$$

$$\underline{\{R\}B\{S\} \text{ and } S \supset T \quad \text{implies} \quad \{R\}B\{T\}}\ .$$

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

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

---

*/ The reader should be aware of the difference between (if p then R_1 else R_2) in the mathematical language, which stands for $(p \supset R_1) \wedge (\sim p \supset R_2)$, and ($\mathbf{if}$ p $\mathbf{then}$ B_1 $\mathbf{else}$ B_2) in the programming language.

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

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

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

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

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

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

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

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

It is quite important that all of 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 $f_B(\bar{x})$, the partial function indicating the change of the values of the variables during the execution of B . $\{R\}$ B $\{T\}$ simply means that whenever $R(\bar{x})$ is true, $T(f_B(\bar{x}))$ is either true (if B terminates) or undefined. We can express this by the relation

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

adopting here the convention that $a \Rightarrow b$ is <u>true</u> whenever a or b is <u>undefined</u>.

We are ready now to prove the following rules:

(a) <u>rule for while statements</u>

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,

$$\{R(\bar{x}) \wedge p(\bar{x})\}\ B\ \{R(\bar{x})\} \quad \text{implies} \quad \{R(\bar{x})\}\ \textbf{while}\ p(x)\ \textbf{do}\ B\ \{R(\bar{x}) \wedge \sim p(\bar{x})\}\ .$$

We therefore have to prove the following theorem:

$$\forall \bar{x}[R(\bar{x}) \wedge p(\bar{x}) \Rightarrow R(f_B(\bar{x}))]$$

implies

$$\forall \bar{x}[R(\bar{x}) \Rightarrow R(f_P(\bar{x})) \wedge \sim p(f_P(\bar{x}))] \quad \text{where}$$

$$P: \quad F(\bar{x}) \Leftarrow \underline{\text{if}}\ p(\bar{x})\ \underline{\text{then}}\ F(f_B(\bar{x}))\ \underline{\text{else}}\ \bar{x}\ .$$

The proof is by computational induction.

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

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

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

$$p(f^{i+1}(\bar{x})) \equiv \underline{\text{if}}\ p(\bar{x})\ \underline{\text{then}}\ p(f^i(f_B(\bar{x})))\ \underline{\text{else}}\ p(\bar{x})\ .$$

We distinguish between two cases:[*]

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

Case 2B: $p(\bar{x})$ is true. Then $R(f^{i+1}(\bar{x})) \equiv R(f^{i}(f_B(\bar{x})))$ and $p(f^{i+1}(\bar{x})) \equiv p(f^{i}(f_B(\bar{x})))$. By the assumption $R(\bar{x}) \wedge p(\bar{x}) \Rightarrow R(f_B(\bar{x}))$ holds, and since by the induction hypothesis
$R(f_B(\bar{x})) \Rightarrow R(f^{i}(f_B(\bar{x}))) \wedge \sim p(f^{i}(f_B(\bar{x})))$, we get
$R(\bar{x}) \Rightarrow R(f^{i}(f_B(\bar{x}))) \wedge \sim p(f^{i}(f_B(\bar{x})))$. Hence,
$R(\bar{x}) \Rightarrow R(f^{i+1}(\bar{x})) \wedge \sim p(f^{i+1}(\bar{x}))$ 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 , one is permitted to assume that the desired property holds for the body B[P] of the procedure! This can be stated as follows:

$$\forall g\ [\ \{R\}\ g\ \{T\}\ \text{implies}\ \{R\}\ B[g]\ \{T\}\ ]\ \text{implies}\ \{R\}\ \textbf{call}\ P\ \{T\}\ .$$

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

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

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

$$\forall g[\forall \bar{x}[R(\bar{x}) \Rightarrow T(g(\bar{x}))] \text{ implies } \forall \bar{x}[R(\bar{x}) \Rightarrow T(f_{B[g]}(\bar{x}))]]$$

implies

$$\forall \bar{x}[R(\bar{x}) \Rightarrow T(f(\bar{x}))] \text{ where}$$

$$P: \quad F(\bar{x}) \Leftarrow f_{B[F]}(\bar{x}) \quad .$$

The proof is again by computational induction.

1. $\forall x[R(\bar{x}) \Rightarrow T(f^0(\bar{x}))]$ is <u>true</u>, since $T(f^0(\bar{x}))$ is <u>undefined</u>.

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

2.2 <u>Translation to Recursive Programs</u>

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.

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 actually still benefit from all the advantages of Hoare's approach since we may associate the blocks of the program arbitrarily at our convenience.

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

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

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

(A) Termination

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

For example, the natural numbers can be characterized[*] by the least fixpoint num(x) of the recursive program

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

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

[*] Given that 0 , 1 , - , + , = have their usual meaning.

$$\forall x[\underline{num}(x) \sqsubseteq \underline{num}(f_{P'}(x))] \quad .$$

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

(B) 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

$$P_7 : \; F(x) \Leftarrow \underline{if}\ x = 0\ \underline{then}\ 1\ \underline{else}\ x \cdot F(x-1) \quad ,$$

and

$$P_8 : \; G(x,y,z) \Leftarrow \underline{if}\ x = y\ \underline{then}\ z\ \underline{else}\ G(x,y+1,(y+1) \cdot z) \quad .$$

We want to show that

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

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 $\underline{ge}(x,y)$ of the recursive program

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

we can show by computational induction that

$$\forall x,y[\underline{ge}(x,y) \sqsubseteq [f_{P_7}(x) \equiv g_{P_8}(x,y,f_{P_7}(y))]] \quad .$$

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

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

i.e., for every natural number x, either both $f_{P_7}(x)$ and $g_{P_8}(x,0,1)$ are defined and equal, or both are undefined.

The proof is by computational induction with

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

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.,

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

or in other words

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

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

(a) If $x = y$ we get $\forall x[\underline{true} \sqsubseteq f_{P_7}(x) = f_{P_7}(x)]$, which clearly holds.

(b) If $x \neq y$ we get $\forall x,y[f^i(x,y+1) \sqsubseteq [f_{P_7}(x) = g_{P_8}(x,y,f_{P_7}(y))]]$.

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

$\forall x,y[f^i(x,y+1) \sqsubseteq [f_{P_8}(x) = g_{P_8}(x,y+1,f_{P_7}(y+1))]]$, which holds by the induction hypothesis.

References

ASHCROFT and MANNA [1971]. E. Ashcroft and Z. Manna, "The Translation of 'Goto' Programs to 'While' Programs", Proceedings of IFIP Congress 1971.

BEKIĆ [1969]. H. Bekić, "Definable Operations in General Algebra and the Theory of Automata and Flowcharts". Unpublished memo, IBM, Vienna (December 1969).

BURSTALL [1969]. R. M. Burstall, "Proving Properties of Programs by Structural Induction", Computer Journal, Vol. 12, No. 1 (February 1969), pp. 41-48.

CADIOU [1972]. J. M. Cadiou, "Recursive Definitions of Partial Functions and their Computations", Ph.D. Thesis, Computer Science Dept., Stanford University (to appear).

COOPER [1971]. D. C. Cooper, "Programs for Mechanical Program Verification", in Machine Intelligence 6 (B. Meltzer and D. Michie, Eds.), Edinburgh University Press, pp. 43-59.

DEBAKKER and SCOTT [1969]. J. W. deBakker and D. Scott, "A Theory of Programs", unpublished memo (August 1969).

DIJKSTRA [1968]. E. Dijkstra, "Goto Statements Considered Harmful", CACM, Vol. 11, No. 3 (March 1968), pp. 147-148.

FLOYD [1967]. R. W. Floyd, "Assigning Meanings to Programs", in Proceedings of a Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science (Ed. J. T. Schwartz), pp. 19-32.

HOARE [1969]. C. A. R. Hoare, "An Axiomatic Approach to Computer Programming", CACM, Vol. 12, No. 10 (October 1969), pp. 576-580, 583.

HOARE [1971]. C. A. R. Hoare, "Procedures and Parameters: an Axiomatic Approach", in Symposium on Semantics of Algorithmic Languages, Lecture notes Mathematics, Vol. 188 (E. Engeler, Ed.), Berlin, Springer-Verlag, pp. 102-116

KLEENE [1952]. S. C. Kleene, Introduction to Meta-mathematics, Van Nostrand, Princeton, New Jersey.

KNUTH and FLOYD [1971]. D. E. Knuth and R. W. Floyd, "Notes on Avoiding 'Goto' Statements", Information Processing Letters 1 (January 1971), pp. 23-31.

LANDIN [1964]. P. J. Landin, "The Mechanical Evaluation of Expressions", Computer Journal, Vol. 6, No. 4 (January 1964), pp. 308-320.

MANNA [1969]. Z. Manna, "The Correctness of Programs", JCSS, Vol. 3, No. 2 (May 1969), pp. 119-127.

MANNA, NESS, and VUILLEMIN [1972]. Z. Manna, S. Ness, and J. Vuillemin, "Inductive Methods for Proving Properties of Programs", in Proceedings of ACM Conference on Proving Assertions about Programs, ACM, New York (January 1972).

MANNA and PNUELI [1970]. Z. Manna and A. Pnueli, "Formalization of Properties of Functional Programs", JACM, Vol. 17, No. 3 (July 1970), pp. 555-569.

McCARTHY [1963a]. J. McCarthy, "A Basis for a Mathematical Theory of Computation". In Computer Programming and Formal Systems, (P. Braffort and D. Hirschberg, Eds.), pp. 33-70.

McCARTHY [1963b]. J. McCarthy, "Towards a Mathematical Science of Computation", in Information Processing: Proceedings of IFIP 62 (C. M. Popplewell, Ed.), Amsterdam, North Holland, pp. 21-28.

MILNER [1972]. R. Milner, "Implementation and Applications of Scott's Logic for Computable Functions", in Proceedings of ACM Conference on Proving Assertions about Programs, ACM, New York (January 1972).

MORRIS [1968]. J. H. Morris, "Lambda-Calculus Models of Programming Languages", Ph.D. Thesis, Project MAC, M.I.T., MAC-TR-57 (December 1968).

MORRIS [1971]. J. H. Morris, "Another Recursion Induction Principle", CACM, Vol. 14, No. 5 (May 1971), pp. 351-354.

NAUR [1966]. P. Naur, "Proof of Algorithms by General Snaptshots", BIT, Vol. 6 (1966), pp. 310-316.

PARK [1969]. D. Park, "Fixpoint Induction and Proofs of Program Properties", in Machine Intelligence 5 (B. Meltzer and D. Michie, Eds.), Edinburgh University Press, pp. 59-78.

SCOTT [1970]. D. Scott, "Outline of a Mathematical Theory of Computation", Oxford University Computing Lab., Programming Research Group, Technical Monograph PRG-2 (November 1970).

SCOTT and STRACHEY [1971]. D. Scott and C. Strachey, "Towards a Mathematical Semantics for Computer Languages", Technical Monograph PRC-6, Oxford University (August 1971).

STRACHEY [1966]. C. Strachey, "Towards a Formal Semantics", in Formal Languages Description Languages, (T. B. Steel, Ed.), Proc. IFIP Working Conf. 1964, Amsterdam, North-Holland, pp. 198-220.

VUILLEMIN [1972]. J. Vuillemin, "Proof Techniques for Recursive Programs", Ph.D. Thesis, Computer Science Dept., Stanford University (to appear).

WEYHRAUCH and MILNER [1972]. R. Weyhrauch and R. Milner, "Program Semantics and Correctness in a Mechanized Logic", The USA-Japan Computer Conference, Tokyo (October 1972).