AD-738 570

INDUCTIVE METHODS FOR PROVING PROPERTIES OF PROGRAMS

Z. Manna, et al

Stanford University
Stanford, California

November 1971

AD 738570

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-154

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS 243

INDUCTIVE METHODS FOR PROVING PROPERTIES OF PROGRAMS

BY

ZOHAR MANNA

STEPHEN NESS

JEAN VUILLEMIN

NOVEMBER 1971

COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY

LELAND STANFORD JUNIOR UNIVERSITY
ORGANIZED 1891

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-154

November 1971

COMPUTER SCIENCE DEPARTMENT REPORT
NO. CS 243

INDUCTIVE METHODS FOR PROVING PROPERTIES OF PROGRAMS*

by

Zohar Manna
Stephen Ness
Jean Vuillemin

ABSTRACT: We have two main purposes in this paper. First, we clarify and extend known results about computation of recursive programs, emphasizing the difference between the theoretical and practical approaches. Secondly, we present and examine various known methods for proving properties of recursive programs. We discuss in detail two powerful inductive methods, computational induction and structural induction, illustrating their applications by various examples. We also briefly discuss some other related methods.

Our aim in this work is to introduce inductive methods to as wide a class of readers as possible and to demonstrate their power as practical techniques. We ask the forgiveness of our more theoretical-minded colleagues for our occasional choice of clarity over precision.

*The research was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract SD-183.

INDUCTIVE METHODS FOR PROVING PROPERTIES OF PROGRAMS

ZOHAR MANNA, STEPHEN NESS, JEAN VUILLEMIN
Computer Science Department
Stanford University
Stanford, California

Abstract

We have two main purposes in this paper. First, we clarify and extend known results about computation of recursive programs, emphasizing the difference between the theoretical and practical approaches. Secondly, we present and examine various known methods for proving properties of recursive programs. We discuss in detail two powerful inductive methods, computational induction and structural induction, illustrating their applications by various examples. We also briefly discuss some other related methods.

Our aim in this work is to introduce inductive methods to as wide a class of readers as possible and to demonstrate their power as practical techniques. We ask the forgiveness of our more theoretical-minded colleagues for our occasional choice of clarity over precision.

Introduction

Many different inductive methods have been used to prove properties of programs. Well-known methods include for example recursion induction, structural induction, inductive assertions, computational induction, truncation induction, and fixedpoint induction. Our intention in this paper is to present and examine these methods, illustrating their application for proving properties of recursive programs.

In Section I we give the theoretical background necessary to understand the fixedpoint approach to recursive programs (essentially following Scott [1969, 1970]), as well as the practical computational approach. We emphasize that while existing inductive methods prove properties of the 'least fixedpoint function' of a recursive program, in practice the function computed by some common computation rules differs from it. We briefly suggest 'fixedpoint' computation rules which assure that the computed function is identical to the least fixedpoint.

In Section II we examine computational induction methods, i.e., methods in which the induction is based on the steps of the computation. We first present the extremely simple induction method introduced by Scott (deBakker and Scott [1969], Scott [1969]). Examples are presented which introduce various applications of the method. We also discuss another computational induction method, truncation induction (Morris [1971]). A related method, called fixedpoint induction, is described in Park [1969].

We describe the structural induction method and its application for proving properties of programs in Section III. This method was suggested explicitly by Burstall [1969], although it was often used previously, for example by McCarthy and Painter [1967] for proving the correctness of a compiler and by Floyd [1967] for proving termination of flowchart programs. Our intention in this section is to emphasize, by means of appropriately chosen examples, that the choice of a suitable partial ordering on the data structure and a suitable induction hypothesis leads to simple and clear inductive proofs. Although we show that computational induction and structural induction are essentially equivalent, there are practical reasons for keeping both of them in mind. Computational induction is best suited for proving the correctness and equivalence of programs, and because of its simplicity it is particularly convenient for machine implementation (Milner [1972a, 1972b]). On the other hand, termination of programs is usually easier to show by structural induction.

In Section IV, we introduce two additional methods: recursion induction (McCarthy [1963b]), which was actually the first method proposed for proving properties of recursive programs, and inductive assertions (introduced by Floyd [1967] and Naur [1966] for flowchart programs and generalized by Manna and Pnueli [1970] for recursive programs). We show that any proof by these methods can be effectively translated to a proof by computational induction.

I. RECURSIVE PROGRAMS

In this section, we introduce a theory of partial functions, and show its relation to recursive programs and their computations.

Partial Functions

We wish to consider partial functions from a domain D_1 into a range D_2, i.e., functions which may be undefined for some arguments. For example, the quotient function x/y, mapping $R \times R$ (pairs of real numbers) into R, is usually considered to have no value if $y = 0$. Partial functions arise naturally in connection with computation, as a computing process may give results for some arguments and run indefinitely for others.

In developing a theory for handling partial functions it is convenient to introduce the special element ω to represent the value undefined. We let D^+ denote $D \cup \{\omega\}$, assuming $\omega \notin D$ by convention; when D is the cartesian product $A_1 \times \dots \times A_n$, we let D^+ be $A_1^+ \times \dots \times A_n^+$.

The research reported here was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract SD-183.

Any partial function f mapping D_1 into D_2 may then be considered as a total function mapping D_1 into D_2^+ : if f is undefined for $d \epsilon D_1$, we let $f(d)$ be ω .

Since we shall consider compositions of partial functions, we may need to compute functions with some arguments being undefined. Thus we must extend every function mapping D_1 into D_2^+ to a function mapping D_1^+ into D_2^+ ; such extensions are discussed in the next section. Partial predicates are of course a special case, since a partial predicate is a partial function into $\{\underline{\text{true}}, \underline{\text{false}}\}$.

The Ordering $\subseteq$ on the Domain

To define appropriate extensions of partial functions from D_1 into D_2 to total functions from D_1^+ into D_2^+, we first introduce the partial ordering $\subseteq$ on every extended domain D^+ .[*] The partial ordering $\subseteq$ is intended to correspond to the notion 'is less defined than', and accordingly we define it by letting $\omega \subseteq d$ for all $d \epsilon D^+$. Note that distinct elements of D are unrelated by $\subseteq$: for distinct a and b in D , neither $a \subseteq b$ nor $b \subseteq a$ holds. If D^+ is the cartesian product $A_1^+ \times \dots \times A_n^+$, we define $\langle a_1,\dots,a_n\rangle \subseteq \langle b_1,\dots,b_n\rangle$ when $a_i \subseteq b_i$ for each i , $1 \le i \le n$.

Example. If $D = \{a,b\}$, then $D^+ = \{a,b,\omega\}$ and $(D \times D)^+ = \{\langle\omega,\omega\rangle, \langle\omega,a\rangle, \langle a,\omega\rangle, \dots, \langle a,b\rangle, \langle b,a\rangle, \langle b,b\rangle\}$. The partial orderings on D^+ and $(D \times D)^+$ are described in the diagrams below, where each connecting line indicates that the lower element is less defined than the upper element. (Lines implied by transitivity or reflexivity are not shown.) □

[*] A partial ordering is a binary relation which is reflexive $((\forall a)[a \subseteq a])$, antisymmetric $((\forall a,b)[a \subseteq b \wedge b \subseteq a \Rightarrow a \text{ is identical to } b])$, and transitive $((\forall a,b,c)[a \subseteq b \wedge b \subseteq c \Rightarrow a \subseteq c])$. As usual, we write $a \subset b$ if $a \subseteq b$ and a is not identical to b , $a \not\subseteq b$ if $a \subseteq b$ does not hold, etc.

$$\begin{tikzcd}
a \arrow[dr] & & b \arrow[dl] \\
& \omega &
\end{tikzcd}$$

D^+

$$\begin{tikzcd}
\langle a,a\rangle \arrow[d] \arrow[dr] & \langle a,b\rangle \arrow[dl] \arrow[dr] & \langle b,a\rangle \arrow[dl] \arrow[dr] & \langle b,b\rangle \arrow[dl] \arrow[d] \\
\langle a,\omega\rangle \arrow[drr] & \langle \omega,a\rangle \arrow[dr] & \langle \omega,b\rangle \arrow[d] & \langle b,\omega\rangle \arrow[dl] \\
& & \langle \omega,\omega\rangle &
\end{tikzcd}$$

$(D \times D)^+ = D^+ \times D^+$

Monotonic Functions

Any function f computed by a program has the property that whenever the input x is less defined than the input y , the output $f(x)$ is less defined than $f(y)$. We therefore require that the extended function f from D_1^+ into D_2^+ be monotonic, i.e.,

$$x \subseteq y \quad \text{implies} \quad f(x) \subseteq f(y) \quad .$$

We let $(D_1^+ \to D_2^+)$ denote the set of all monotonic functions from D_1^+ into D_2^+ .

If f has only one argument, monotonicity requires that $f(\omega)$ be ω, with one exception: the constant function $f(x) = c$ for all $x \epsilon D^+$. If f has many arguments, i.e., $D_1 = A_1 \times \dots \times A_n$, it may have many different monotonic extensions. A particularly important extension of any function is called the natural extension, defined by letting $f(d_1,\dots,d_n)$ be ω whenever at least one of the d_i is ω .[*] This corresponds intuitively to the functions computed by programs which must know all their inputs before beginning execution (e.g., Algol call by value).

Examples. (a) The identity function, mapping any x in D^+ into itself, is obviously monotonic.

(b) The quotient function, mapping $\langle x,y\rangle$ into x/y , extended to a total function by letting $x/0$ be ω for any x in R , becomes monotonic by the natural extension: let x/ω and ω/y be ω for any x and y in R^+ .

(c) The equality predicate mapping $D \times D$ into $\{\underline{\text{true}}, \underline{\text{false}}\}$ can be extended in the following particularly interesting ways:

(i) The natural extension (weak equality), denoted by $=$, yields the value ω whenever at least one of its arguments is ω . The weak equality predicate is of course monotonic.

[*] We assume all the functions of our examples to be naturally extended, unless otherwise noted.

(ii) Another extension (strong equality), denoted by $\equiv$, yields the value true when both arguments are ω and false when exactly one argument is ω; in other words, $x \equiv y$ if and only if $x \sqsubseteq y$ and $y \sqsubseteq x$. The strong equality predicate is not a monotonic mapping from $D^+ \times D^+$ into $\{true, false, \omega\}$, since $(\omega, d) \sqsubseteq (d,d)$ but $(\omega \equiv d) \not\sqsubseteq (d \equiv d)$ (i.e., false $\not\sqsubseteq$ true) for $d \in D$.

(d) The if-then-else function, mapping $\{true, false\} \times D \times D$ into D, is defined for any $a, b \in D$ by letting

the value of (if true then a else b) be a,

and

the value of (if false then a else b) be b.

It can be extended to a monotonic function mapping $\{true, false\}^+ \times D^+ \times D^+$ into D^+ by letting, for any $a, b \in D^+$,

the value of (if true then a else ω) be a,

the value of (if false then ω else b) be b,

and

the value of (if ω then a else b) be ω.

Note that this is not the natural extension of if-then-else. □

Composition of Functions

An important operation on functions is composition, which allows functions to be defined in terms of simpler functions. If f is a function from D_1^+ to D_2^+ and g a function from D_2^+ into D_3^+, then the composition of f and g is the function from D_1^+ into D_3^+ defined by $g(f(x))$ for every x in D_1^+. It is easy to show that, if f and g are monotonic functions, so is their composition.

Examples. (a) The function f, given by $f(x) \equiv$ (if $x = 0$ then 1 else x), is defined by composition of the weak equality predicate, the constant functions 0 and 1, the identity function, and the if-then-else function. Since all these functions are monotonic, f is monotonic.

(b) The function f, given by $f(x) \equiv$ (if $x \equiv \omega$ then 0 else 1), defined using the nonmonotonic predicate $\equiv$, is not monotonic, since $f(\omega) \equiv 0$ and $f(0) \equiv 1$ (i.e., $\omega \sqsubseteq 0$, but $f(\omega) \not\sqsubseteq f(0)$).

(c) The functions f_1 and f_2, given by

$$f_1(x) \equiv g(\text{if } p(x) \text{ then } h_1(x) \text{ else } h_2(x))$$

and

$$f_2(x) \equiv \text{if } p(x) \text{ then } g(h_1(x)) \text{ else } g(h_2(x)),$$

are defined by composition of if-then-else, p, g, h_1 and h_2. If p, g, h_1 and h_2 are monotonic, f_1 and f_2 are monotonic. There is an interesting relation between these two functions: (i) $f_2(x) \sqsubseteq f_1(x)$ for any x; (ii) if $g(\omega) \equiv \omega$, then $f_2(x) \equiv f_1(x)$ for any x. We shall use these results (and a similar result when g has several arguments) often in later proofs. □

The Ordering $\sqsubseteq$ on Functions

Let f and g be two monotonic functions mapping D_1^+ into D_2^+. We say that $f \sqsubseteq g$, read " f is less defined than g ", if $f(x) \sqsubseteq g(x)$ for any x in D_1^+; this relation is indeed a partial ordering on $(D_1^+ \rightarrow D_2^+)$. Two functions f and g are equal, $f \equiv g$, iff $f \sqsubseteq g$ and $g \sqsubseteq f$ (that is, $f \equiv g$ iff $f(x) \equiv g(x)$ for every $x \in D_1^+$). We denote by Ω the function which is always undefined: $\Omega(x)$ is ω for any $x \in D_1^+$. Note that $\Omega \sqsubseteq f$ for any function f.

Infinite increasing sequences $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \cdots$ of functions in $(D_1^+ \rightarrow D_2^+)$ are called chains. It can be shown that any chain has a unique limit function in $(D_1^+ \rightarrow D_2^+)$, denoted by $\lim_i \{f_i\}$, which has the characteristic properties that $f_i \sqsubseteq \lim_i \{f_i\}$ for every i, and for any function g such that $f_i \sqsubseteq g$ for every i, we have $\lim_i \{f_i\} \sqsubseteq g$.

Example. Consider the sequence of functions $f_0, f_1, \ldots$ over the natural numbers defined by $f_i(x) \equiv$ (if $x \leq i$ then $x!$ else ω). This sequence is a chain, as $f_i \sqsubseteq f_{i+1}$ for every i; $\lim_i (f_i)$ is the factorial function. □

Continuous Functionals

We now consider a function τ mapping the set of functions $(D_1^+ \rightarrow D_2^+)$ into itself, called a functional; that is, τ takes any monotonic function f as its argument and yields a monotonic function $\tau[f]$ as its value. As for functions, it is natural to restrict ourselves to monotonic functionals, i.e., τ such that $f \subseteq g$ implies $\tau[f] \subseteq \tau[g]$ for all f and g in $(D_1^+ \to D_2^+)$. For our purposes, however, we require that functionals satisfy a stronger property, called continuity: τ is continuous if for any chain of functions

$$f_0 \subseteq f_1 \subseteq f_2 \subseteq \cdots$$

we have

$$\tau[f_0] \subseteq \tau[f_1] \subseteq \tau[f_2] \subseteq \cdots$$

and

$$\tau[\lim_i(f_i)] \equiv \lim_i(\tau[f_i]) \quad .$$

We usually specify a functional τ by composition of known monotonic functions and the function variable F , denoted by $\tau[F](x)$; when F is replaced by any known monotonic function g , the composition rules determine $\tau[g](x)$. It can easily be shown that any functional defined by composition of monotonic functions and the function variable F is continuous.

Examples. (a) The identity functional τ_I defined by $\tau_I[F](x) \equiv F(x)$, mapping any f in $(D_1^+ \to D_2^+)$ into itself, is clearly continuous.

(b) The constant functional τ_g defined by $\tau[F](x) \equiv g(x)$, mapping any f in $(D_1^+ \to D_2^+)$ into the function g , is continuous.

(c) The functional τ defined by $\tau[F](x) \equiv$ (if $x = 0$ then 1 else $F(x+1)$) is constructed by composition of monotonic functions (if-then-else , addition, weak equality, and the constant functions 0 and 1) and the function variable F ; it is therefore continuous. Given any monotonic function g over the integers, $\tau[g]$ is another monotonic function over the integers:

if $g \equiv \Omega$, then $\tau[g](x) \equiv$ (if $x = 0$ then 1 else ω);

if $g(x) \equiv x-1$, then $\tau[g](x) \equiv$ (if $x = 0$ then 1 else x).

(d) The functional τ defined by $\tau[F](x) \equiv$ (if $\forall x[F(x) = x]$ then $F(x)$ else ω) is monotonic but not continuous; if we consider the chain $g_0 \subseteq g_1 \subseteq \cdots$ where $g_i(x) \equiv$ (if $x < i$ then x else ω) , $\tau[g_i] \equiv \Omega$ for any i so that $\lim_i(\tau[g_i]) \equiv \Omega$, whereas $\tau[\lim_i(g_i)]$ is the identity function. □

Fixedpoints

The fundamental property of a continuous functional τ mapping $(D_1^+ \to D_2^+)$ into itself is that it has a unique least-fixedpoint F_τ , having the two characteristic properties: $F_\tau \equiv \tau[F_\tau]$ and, for any g , $g \equiv \tau[g]$ implies $F_\tau \subseteq g$.

We can compute F_τ as the limit of the chain $\Omega \subseteq \tau[\Omega] \subseteq \tau^2[\Omega] \subseteq \cdots$ as follows from Kleene's first recursion theorem [1950].

Examples. All the functionals in the following examples are defined by composition of monotonic functions and the function variable F and are therefore continuous by construction and have unique least fixedpoints.

(a) The functional τ over $(N^+ \to N^+)$, given by

$\tau[F](x) \equiv$ (if $x = 0$ then 1 else $F(x+1)$) ,

has as fixedpoints the functions

$g_n(x) \equiv$ (if $x = 0$ then 1 else n)

for each $n \in N^+$.

The least fixedpoint is

$F_\tau(x) \equiv$ (if $x = 0$ then 1 else ω) .

(b) The only (and therefore least) fixedpoint of the functional τ over the integers given by

$\tau[F](x) \equiv$ if $x > 100$ then $x-10$ else $F(F(x+11))$,

is $F_\tau(x) \equiv$ if $x > 100$ then $x-10$ else 91 .

(c) The functional τ over the integers, defined by

$\tau[F](x_1,x_2) \equiv$

if $x_1 = x_2$ then x_2+1 else $F(x_1,F(x_1-1,x_2+1))$,

has as fixedpoints the functions

$f(x_1,x_2) \equiv x_1+1$,

$g(x_1,x_2) \equiv$ if $x_1 \geq x_2$ then x_1+1 else x_2-1 , and

$h(x_1,x_2) \equiv$ if $(x_1 \geq x_2) \wedge (x_1-x_2$ even$)$ then x_1+1 else ω ,

the latter being the least fixedpoint F_τ (Morris [1968]). □

#

We consider a functional τ over $(D_1^+ \to D_2^+)^n$ to be given by coordinate functionals

$\tau_1,\ldots,\tau_n$, so that $\tau[F_1,\ldots,F_n]$ is $\langle\tau_1[F_1,\ldots,F_n],\ldots,\tau_n[F_1,\ldots,F_n]\rangle$. It follows directly from the definition of the ordering on $(D_1^+ \to D_2^+)^n$ that τ is continuous iff each τ_i is continuous. A continuous functional τ over $(D_1^+ \to D_2^+)^n$ has a unique least fixedpoint $F_\tau = \langle F_{\tau_1},\ldots,F_{\tau_n}\rangle$; that is

(a) $F_{\tau_i} \equiv \tau_i[F_{\tau_1},\ldots,F_{\tau_n}]$ for all i , $1 \le i \le n$;

(b) For any fixedpoint $g \equiv \langle g_1,\ldots,g_n\rangle$ of τ , i.e., $g_i \equiv \tau_i[g_1,\ldots,g_n]$ for all i $(1 \le i \le n)$, $F_{\tau_i} \sqsubseteq g_i$ for all i $(1 \le i \le n)$.

Example. Consider the functional $\tau[F_1,F_2] \equiv \langle\tau_1[F_1,F_2],\tau_2[F_1,F_2]\rangle$ over $(N^+ \to N^+)^2$, where:

$$\tau_1[F_1,F_2](x) \equiv (\underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ F_1(x-1) + F_2(x-1))$$

and

$$\tau_2[F_1,F_2](x) \equiv (\underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ F_2(x-1)).$$

For any $n \in N^+$, the pair $\langle g_n,h_n\rangle$ defined by

$$g_n(x) \equiv (\underline{\text{if}}\ x = 0 \vee x = 1\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ (x-1)\cdot n+1)$$

and

$$h_n(x) \equiv (\underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ n)$$

is a fixedpoint of τ , since $g_n \equiv \tau_1[g_n,h_n]$ and $h_n \equiv \tau_2[g_n,h_n]$ (and therefore $\langle g_n,h_n\rangle \equiv \tau[g_n,h_n]$). The least fixedpoint is the pair:

$$\langle(\underline{\text{if}}\ x = 0 \vee x = 1\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ \omega)\ ,$$
$$(\underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ \omega)\rangle\ .$$ □

Recursive Programs

So far, we have been concerned only with functions considered abstractly, as purely mathematical objects. For example, we thought of the factorial function as a certain mapping between arguments and values, without considering how the mapping is specified. To continue our discussion we must introduce at this point a "programming language" for specifying functions. A function will be specified by a piece of code in the syntax of the language and then will be executed according to computation rules given by the semantics of the language.

In the rest of this paper we use for illustration a particularly simple language, chosen because of its similarities to familiar languages such as ALGOL or LISP.[*] A program in our language, called a <u>recursive definition</u> or <u>recursive program</u>, is of the form

$$F(x) \Leftarrow \tau[F](x)$$

where $\tau[F](x)$ is an expression representing composition of known monotonic functions and predicates and the function variable F , applied to the individual variable x .[**] For example, the following is a program for computing the factorial function:

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

This program resembles the ALGOL declaration

```
integer procedure f(x);
f := if x = 0 then 1 else x*f(x-1);
```

and the LISP definition

```
  DEFINE ((
(F (LAMBDA (X)(COND ((ZEROP X) 1)
                    (T (TIMES X (F (SUB1 X)))))))))).
```

Of course our programs are meaningless until we describe the semantics of our language, i.e., how to compute the function defined by a program. The next step is therefore to give <u>computation rules</u> for executing programs. Our aim is to characterize the rules such that for every program $F(x) \Leftarrow \tau[F](x)$ the computed function will be exactly the least fixedpoint F_τ .

---

[*] Although our programming language is very simple, it is powerful enough to express any "partial recursive" function, hence by Church's thesis any "computable" function (see, for example, Minsky [1967]).

[**] We shall purposely be vague in our definitions in this section to avoid the introduction of the notion of schemas and interpretations. For a formal approach, see Manna and Pnueli [1970] or Cadiou [1972].

Computation Sequence

Let $F(x) \Leftarrow \tau[F](x)$ be a program over some domain D^+. For a given input value $d \in D^+$ (for ω), the program is executed by constructing a sequence of terms $t_0, t_1, t_2, \ldots$, called a computation sequence for d, as follows:

(1) The first term t_0 is $F(d)$;

(2) For each i, $i \geq 0$, the term t_{i+1} is obtained from t_i in two steps: first

  (a) substitution: some occurrences of F (see below) in t_i are replaced by $\tau[F]$ simultaneously; and then

  (b) simplification: known functions and predicates are replaced by their values, whenever possible, until no further simplifications can be made;

(3) The sequence is finite and t_n is the final term in the sequence if and only if no further substitution or simplification can be applied to t_n (that is, when t_n is an element of D^+).

Computation Rules

A computation rule C tells us which occurrences of F should be replaced by $\tau[F]$ in each substitution step. For a given computation rule C, the program defines a partial function F_C mapping D^+ into D^+ as follows: If for input $d \in D^+$ the computation sequence for d is finite, we say that $F_C(d)$ is defined and $F_C(d) \equiv t_n$; if the computation sequence for d is infinite, we say that $F_C(d) \equiv \omega$.

The following are examples of typical computation rules:

(a) Kleene's computation rule: Replace all occurrences of F simultaneously. We denote the computed function by F_K.

(b) leftmost-innermost rule: Replace only the leftmost-innermost occurrence of F (that is, the leftmost occurrence of F with all arguments free of F's). We denote the computed function by F_{LI}. This is the rule which corresponds to the usual stack-implementation of recursion for languages like LISP or ALGOL. Any procedure evaluates all its arguments before execution.

(c) leftmost outermost rule: Replace only the leftmost-outermost occurrence of F. We denote the computed function by F_{LO}.

Example 1. We consider the recursive definition of the "91-function" over the integers:

$$F(x) \Leftarrow \underline{\text{if}}\ x > 100\ \underline{\text{then}}\ x-10\ \underline{\text{else}}\ F(F(x+11)).$$

We illustrate the computation sequences for $x = 99$ using the three rules.

(a) Using Kleene's rule:

```
t0  is  F(99)
  if 99 > 100 then 99-10
            else F(F(99+11))     [substitution]
t1  is  F(F(110))                [simplification]
  if [if 110 > 100 then 110-10
                 else F(F(110+11))] > 100
     then [if 110 > 100 then 110-10
                        else F(F(110+11))]-10
     else F(F([if 110 > 100
                then 110-10
                else F(F(110+11))]+11))
                                 [substitution]
t2  is  F(F(111))                [simplification]
  if [if 111 > 100 then 111-10
                 else F(F(111+11))] > 100
     then [if 111 > 100 then 111-10
                        else F(F(111+11))]-10
     else F(F([if 111 > 100
                then 111-10
                else F(F(111+11))]+11))
                                 [substitution]
t3  is  91 .                     [simplification]
```

In short, omitting simplifications and underlining the occurrences of F used for substitution:
$\underline{F}(99) \to \underline{F}(\underline{F}(110)) \to \underline{F}(\underline{F}(111)) \to 91$. Thus, $F_K(99) \equiv 91$.

(b) Using the leftmost-innermost rule:
$\underline{F}(99) \to F(\underline{F}(110)) \to \underline{F}(100) \to F(\underline{F}(111)) \to \underline{F}(101) \to 91$.
Thus, $F_{LI}(99) \equiv 91$.

(c) Using the leftmost-outermost rule:
$\underline{F}(99) \to \underline{F}(F(110))$
$\to \underline{\text{if}}\ \underline{F}(110) > 100\ \underline{\text{then}}\ F(110)-10\ \underline{\text{else}}\ F(F(F(110)+11))$
$\to \underline{F}(F(F(110)+11)) \to \ldots$
$\to \underline{\text{if}}\ \underline{F}(110)+11 > 100\ \underline{\text{then}}\ F(110)-9\ \underline{\text{else}}\ \ldots$

$\rightarrow \underline{F}(110)-9 \rightarrow 91$.
Thus, $F_{LO}(99) \equiv 91$. $\sqsupset$

An important property of F_C should be mentioned at this point (Cadiou [1972]):

For any computation rule C , *the computed function* F_C *is less defined than the least fixedpoint, i.e.*, $F_C \subseteq F_\tau$, but they are not necessarily equal.

A program may consist in general of a system of recursive definitions of the form

$$\begin{cases} F_1(\bar{x}) \Leftarrow \tau_1[F_1,\ldots,F_n](\bar{x}) \\ F_2(\bar{x}) \Leftarrow \tau_2[F_1,\ldots,F_n](\bar{x}) \\ \vdots \\ F_n(\bar{x}) \Leftarrow \tau_n[F_1,\ldots,F_n](\bar{x}) \end{cases} ,$$

where each τ_i is an expression representing a composition of known monotonic functions and predicates and the function variables $F_1,F_2,\ldots,F_n$ applied to the individual variables $\bar{x} = (x_1,\ldots,x_k)$. The generalization of the computation rules to systems of recursive definitions is straightforward; the computed function F_C of the system can be described as $(F_{C_1},F_{C_2},\ldots,F_{C_n})$, where each F_{C_i} is computed as described above. The results of this section still hold for systems of recursive definitions.

Fixpoint Computation

All the methods for proving properties of programs described in the rest of this paper are based on the assumption that the computed function is equal to the least fixedpoint. We are therefore interested only in the computation rules that yield the least fixedpoint. We call such computation rules *fixedpoint computation rules*.

Let $\alpha[F^1,\ldots,F^k](d)$ denote any term, where we use superscripts to distinguish the individual occurrences of F in α . Suppose that we choose for substitution the occurrences $F^1,\ldots,F^i$ (for some i , $1 \leq i \leq k$) of F in α . We say that this is a *safe substitution* if:

$$(\forall F^{i+1},\ldots,F^k)\alpha[\Omega,\ldots,\Omega,F^{i+1},\ldots,F^k](d) \equiv \omega .$$

Intuitively, the substitution is safe if the values of $F^{i+1},\ldots,F^k$ are not relevant: as long as $F^1,\ldots,F^i$ are not known, the value of $\alpha[F^1,\ldots,F^k](d)$ cannot be determined; hence, there is no need to compute $F^{i+1},\ldots,F^k$ at this point.

A *safe computation rule* is any computation rule which uses only safe substitutions. It can be shown that *any safe computation rule is a fixedpoint rule* (Vuillemin [1972]). For example, since Kleene's rule and the leftmost-outermost rule are safe, they are both fixedpoint rules.

The leftmost-innermost rule, however, is not safe. The following example illustrates a program for which $F_{LI} \not\equiv F_\tau$; that is, the *leftmost-innermost rule is not a fixedpoint rule*.

Example 2. Consider the program over the integers

$F(x,y) \Leftarrow$ *if* $x = 0$ *then* 1 *else* $F(x-1,F(x-y,y))$.

The least fixedpoint F_τ is

$F_\tau(x,y) \equiv$ (*if* $x \geq 0$ *then* 1 *else* ω) .

We compute $F(1,0)$ using the leftmost-innermost computation rule:

$\underline{F}(1,0) \rightarrow F(0,\underline{F}(1,0)) \rightarrow F(0,F(0,\underline{F}(1,0))) \rightarrow \ldots$

and so on. The sequence is infinite, and therefore $F_{LI}(1,0) \equiv \omega$. In fact

$F_{LI}(x,y) \equiv$ (*if* $x = 0 \vee (x > 0 \wedge y > 0 \wedge (y \text{ divides } x))$
then 1 *else* ω) ,

which is *strictly* less defined than F_τ .

This example is closely related to an example of Morris [1968]. $\sqsupset$

In practice, the fixedpoint computation rules described so far (Kleene's rule and the leftmost-outermost rule) lead to very inefficient computations. In the rest of this section we describe and illustrate a fixedpoint computation rule, called the *normal computation rule*, which leads to efficient computations. In fact, since our computation rules do not allow the reuse of previously computed values of the program, the normal computation rule can be shown to perform the *minimum possible number of substitutions* (Vuillemin [1972]).

By this rule, t_{i+1} is obtained from t_i by substituting $\tau[F]$ for one occurrence of F chosen as follows: we try first to replace the leftmost-outermost occurrence of F in t_i by $\tau[F]$, and start to evaluate the necessary tests in the new term, in order to eliminate the *if*-*then*-*else* connectives. If this is possible, we are done.

Otherwise, we choose a new occurrence of F in t_i which corresponds to the first F we had to test during the previous evaluation, and repeat the process.

In other words, the idea is to delay the evaluation of the arguments of the function variables F as much as possible. This rule is quite close to Algol 60 "call by name" for procedures. However, there are two important differences: (a) absolutely <u>no side effects</u> are allowed, and (b) each argument of the procedures is evaluated <u>at most once</u>, namely the first time (if ever) it is needed.

We denote the computed function by F_N. The normal rule is safe, and it is therefore a fixedpoint rule. The rule can be implemented in programming languages with almost no overhead, and provides an attractive alternative to call by value, which is not a fixedpoint rule, and call by name, which is not efficient.

<u>Example 3</u>. Consider the program over the natural numbers

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

We shall compute $F(2,1)$ using the normal computation rule. The occurrence of F chosen for substitution is underlined.

$$\begin{aligned}&\underline{F}(2,1) \to F(1,\underline{F}(2,0)) \to F(1,\underline{F}(1,1))\\ &\to F(1,\underline{F}(0,F(1,0))) \to F(1,\underline{F}(1,0)+1)\\ &\to F(1,\underline{F}(0,1)+1) \to \underline{F}(1,3) \to F(0,F(1,2))\\ &\to \underline{F}(1,2)+1 \to \underline{F}(0,F(1,1))+1 \to \underline{F}(1,1)+2\\ &\to \underline{F}(0,F(1,0))+2 \to \underline{F}(1,0)+3 \to \underline{F}(0,1)+3 \to 5\ .\end{aligned}$$

Note that in $F(1,\underline{F}(2,0))$, for example, the inner occurrence of F was chosen for substitution, since trying to substitute for the outer F would lead to

$$\begin{aligned}&\underline{\text{if}}\ 1=0\ \underline{\text{then}}\ \ldots\\ &\quad\underline{\text{else}}\ \underline{\text{if}}\ F(2,0)=0\ \underline{\text{then}}\ \ldots\\ &\quad\quad\underline{\text{else}}\ \ldots\ ,\end{aligned}$$

which requires testing for the value of $F(2,0)$.

We compare below the number of substitutions required for each computation rule on this example.

Normal rule: 14
Kleene's rule: 23
Leftmost-innermost: 14
Leftmost-outermost: 29

$F_N(x,y) \equiv F_\tau(x,y)$ is known as "Ackermann's function". This function is of special interest in recursive function theory because it grows faster than any primitive recursive function; for example, $F_\tau(0,0)=1$, $F_\tau(1,1)=3$, $F_\tau(2,2)=7$, $F_\tau(3,3)=61$, and

$$F_\tau(4,4) = 2^{2^{2^{2^{16}}}} - 3\ .$$

$\square$

<u>Example 4</u>. Consider the program over the integers:

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

We shall compute $F(2,1)$, using the normal computation rule:

$$\begin{aligned}&\underline{F}(2,1) \to \underline{F}(1,F(1,1))\\ &\quad\to \underline{F}(0,F(1-F(1,1),F(1,1))) \to 1\ .\end{aligned}$$

We again compare the substitutions required:

Normal rule: 3
Kleene's rule: 7
Leftmost-innermost: 7
Leftmost-outermost: 3

II. COMPUTATIONAL INDUCTION

The first method we shall describe is conceptually very simple: in order to prove some property of a program, we show that it is an invariant during the course of the computation.

For simplicity, we shall first explain the method for simple programs, consisting of a single recursive definition, then generalize to more complex programs.

Computational Induction for a Single Recursive Definition

To prove the property $P(F_\tau)$ of the function F_τ defined by $F \Leftarrow \tau[F]$, it is sufficient to:

(a) Check that P is true before starting the computation, i.e., $P(\Omega)$; and

(b) Show that, if P is true at one step of the computation, it remains true after the next step, i.e., $P(F)$ implies $P(\tau[F])$ for every F .

In short

from $P(\Omega)$ and $\forall F\{P(F) \Rightarrow P(\tau[F])\}$, infer $P(F_\tau)$.

Since this rule is not valid for any P,[*] we shall only consider admissible predicates[**] $P(F)$ which are simply conjunctions of inequalities: $\alpha[F] \sqsubseteq \beta[F]$, where α and β are two continuous functionals.

In this case, the justification of the principle is easy; if $\alpha[\Omega] \sqsubseteq \beta[\Omega]$ and $\forall F\{\alpha[F] \sqsubseteq \beta[F] \Rightarrow \alpha[\tau[F]] \sqsubseteq \beta[\tau[F]]\}$, then by a simple induction, $\alpha[\tau^i[\Omega]] \sqsubseteq \beta[\tau^i[\Omega]]$ for every $i \geq 0$. Therefore, since $\beta[\tau^i[\Omega]] \sqsubseteq \beta[F_\tau]$ for any i , $\alpha[\tau^i[\Omega]] \sqsubseteq \beta[F_\tau]$. By definition of the limit, this implies $\lim_i \alpha[\tau^i[\Omega]] \sqsubseteq \beta[F_\tau]$, and, α being continuous,

$$\alpha[F_\tau] \equiv \alpha[\lim_i[\tau^i[\Omega]]] \equiv \lim_i \alpha[\tau^i[\Omega]] \sqsubseteq \beta[F_\tau] .$$

Example 1. We wish to show that the program

$F(x) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ F(F(h(x)))$

defines an idempotent function, i.e., that $F_\tau F_\tau \equiv F_\tau$. By p and h , we understand respectively any naturally extended partial predicate and function. We prove $P(F_\tau)$, where $P(F)$ is $F_\tau F \equiv F$, i.e., $(F_\tau F \sqsubseteq F) \wedge (F \sqsubseteq F_\tau F)$.

(a) Show $P(\Omega)$ (i.e., $F_\tau\Omega \equiv \Omega$) .

$F_\tau(\Omega(x)) \equiv F_\tau(\omega)$	definition of Ω
$\equiv \underline{\text{if}}\ p(\omega)\ \underline{\text{then}}\ \omega\ \underline{\text{else}}\ F_\tau(F_\tau(h(\omega)))$	definition of F_τ
$\equiv \underline{\text{if}}\ \omega\ \underline{\text{then}}\ \omega\ \underline{\text{else}}\ F_\tau(F_\tau(h(\omega)))$	since $p(\omega) \equiv \omega$
$\equiv \omega$	definition of $\underline{\text{if}}\ \omega\ \underline{\text{then}}\ a\ \underline{\text{else}}\ b$
$\equiv \Omega(x)$.	definition of Ω

[*] Consider, for example, the recursive definition over the natural numbers $F(x) \Leftarrow \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x \cdot F(x-1)$, and the predicate $P(F)$: $\exists x[F(x) \equiv \omega \wedge x \not\equiv \omega]$. Then $P(\Omega)$ and $\forall F\{P(F) \Rightarrow P(\tau[F])\}$ hold; since $F_\tau(x)$ is a total function (the factorial function), $P(F_\tau)$ does not hold.

[**] A class AP of admissible predicates can be defined as:

$\langle AP\rangle \rightarrow \langle AP\rangle \wedge \langle AP\rangle \mid \forall\bar{x}\langle EP\rangle$

$\langle EP\rangle \rightarrow \langle EP\rangle \vee \langle EP\rangle \mid Q(\bar{x}) \mid \alpha[F](\bar{x}) \sqsubseteq \beta[F](\bar{x})$

where $Q(\bar{x})$ is any first order predicate, and α and β are two continuous functionals.

(b) <u>Show $\forall F\{P(F) \Rightarrow P(\tau[F])\}$</u> , i.e.,

$$\forall F\{F_\tau F \equiv F \Rightarrow F_\tau \tau[F] \equiv \tau[F]\} \;.$$

$$\begin{aligned}
F_\tau(\tau[F](x)) &\equiv F_\tau(\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x \\
&\qquad\qquad \underline{\text{else}}\ F(F(h(x)))) \\
&\qquad\qquad\qquad \text{definition of } \tau \\
&\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ F_\tau(x) \\
&\qquad \underline{\text{else}}\ F_\tau(F(F(h(x)))) \\
&\qquad\qquad \text{distributing } F_\tau \\
&\qquad\qquad \text{over conditional,} \\
&\qquad\qquad \text{since } F_\tau(\omega) \equiv \omega \\
&\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x \\
&\qquad \underline{\text{else}}\ F_\tau(F(F(h(x)))) \\
&\qquad\qquad \text{definition of } F_\tau \\
&\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x \\
&\qquad \underline{\text{else}}\ F(F(h(x))) \\
&\qquad\qquad \text{induction hypothesis} \\
&\equiv \tau[F](x) \;. \quad \text{definition of } \tau
\end{aligned}$$

□

The next example uses as domain the set Σ^* of finite strings over a given finite alphabet Σ, including the empty string Λ. There are three basic functions:

$h(x)$	gives the head (first letter) of the string x ;
$t(x)$	gives the tail of the string x (i.e., x with its first letter removed);
$a \cdot x$	concatenates the letter a to the string x .

For example, $h(BCD) = B$, $t(BCD) = CD$, $B \cdot CD = BCD$. These functions satisfy the following properties, for every $x \in \Sigma$ and $y \in \Sigma^*$:

$h(x \cdot y) = x$, $t(x \cdot y) = y$, $x \cdot y \neq \Lambda$,

and $y \neq \Lambda \Rightarrow h(y) \cdot t(y) = y$.[*]

<u>Example 2</u>. The program

$$F(x,y) \Leftarrow (\underline{\text{if}}\ x = \Lambda\ \underline{\text{then}}\ y\ \underline{\text{else}}\ h(x) \cdot F(t(x),y))$$

defines the <u>append</u> function $F_\tau(x,y)$, denoted $x*y$. We shall show that <u>append</u> is associative (i.e., that $x*(y*z) \equiv (x*y)*z$) by proving $P(F_\tau)$, where $P(F)$ is $F(x,y)*z \equiv F(x,y*z)$.

(a) <u>$P(\Omega)$</u> :

$$\begin{aligned}
\Omega(x,y)*z &\equiv \omega*z && \text{definition of } \Omega \\
&\equiv \text{if } \omega = \Lambda\ \underline{\text{then}}\ z \\
&\qquad \underline{\text{else}}\ h(\omega)\cdot(t(\omega)*z) \\
& && \text{definition of } \underline{\text{append}} \\
&\equiv \omega && \text{since } \omega = \Lambda \text{ is } \omega \\
&\equiv \Omega(x*y,z) \;. && \text{definition of } \Omega
\end{aligned}$$

(b) <u>$\forall F\{P(F) \Rightarrow P(\tau[F])\}$</u> :

$$\begin{aligned}
\tau[F](x,y*z) &\equiv \underline{\text{if}}\ x = \Lambda\ \underline{\text{then}}\ y*z \\
&\qquad \underline{\text{else}}\ h(x)\cdot F(t(x),y*z) \\
&\qquad\qquad \text{definition of } \tau \\
&\equiv \underline{\text{if}}\ x = \Lambda\ \underline{\text{then}}\ y*z \\
&\qquad \underline{\text{else}}\ h(x)\cdot(F(t(x),y)*z) \\
&\qquad\qquad \text{induction hypothesis} \\
&\equiv \underline{\text{if}}\ x = \Lambda\ \underline{\text{then}}\ y*z \\
&\qquad \underline{\text{else}}\ (h(x)\cdot F(t(x),y))*z \\
&\qquad\qquad \text{definition of } \underline{\text{append}} \\
&\equiv (\underline{\text{if}}\ x = \Lambda\ \underline{\text{then}}\ y \\
&\qquad \underline{\text{else}}\ h(x)\cdot F(t(x),y))*z \\
&\qquad\qquad \text{distributing } \underline{\text{append}} \\
&\qquad\qquad \text{over conditional,} \\
&\qquad\qquad \text{since } \omega*z \equiv \omega \\
&\equiv \tau[F](x,y)*z \;. \\
&\qquad\qquad \text{definition of } \tau
\end{aligned}$$

□

This proof was done formally with the LCF proof checker (Milner [1972a]).

<u>Parallel Induction</u>

We shall now present an application of computation induction to proving properties of two programs: $F \Leftarrow \tau[F]$ and $G \Leftarrow \sigma[G]$. To prove $P(F_\tau,G_\sigma)$, where $P(F,G)$ is an admissible predicate,[*] it is sufficient to:

[*] This system is sometimes called 'linear LISP'. There is no difficulty involved in generalizing our proofs to real LISP programs.

[*] That is, a conjunction of inequalities $\alpha[F,G] \subseteq \beta[F,G]$, where α and β are continuous functionals.

(a) Prove $P(\Omega,\Omega)$;

(b) Show that $P(F,G)$ implies $P(\tau[F],\sigma[G])$ for any F and G .

That is,

$$\frac{\underline{\text{from}}\ P(\Omega,\Omega)\ \text{and}\ (\forall F,G)\{P(F,G) \Rightarrow P(\tau[F],\sigma[G])\}}{\underline{\text{infer}}\ P(F_\tau,G_\sigma)}\ .$$

Example 3. Consider the programs

$$\begin{aligned}
F(y_1,y_2,y_3) &\Leftarrow \underline{\text{if}}\ y_1 = 0\ \underline{\text{then}}\ y_2\\
&\qquad\qquad \underline{\text{else}}\ F(y_1-1,y_2+y_3,y_3)\\
G(y_1,y_2,y_3) &\Leftarrow \underline{\text{if}}\ y_1 = 0\ \underline{\text{then}}\ y_2\\
&\qquad\qquad \underline{\text{else}}\ G(y_1-1,y_2+2y_1-1).
\end{aligned}$$

We want to show that both programs may be used to compute the squaring function; more precisely, that $F_\tau(x,0,x) \equiv G_\sigma(x,0)$ for any natural number x . Let

$$P(F,G):\ \forall x\forall y[F(y,x(x-y),x) \equiv G(y,x^2-y^2)]\ .$$

We shall prove $P(F_\tau,G_\sigma)$ which, for $y = x$, simplifies to $F_\tau(x,0,x) \equiv G_\sigma(x,0)$.

(a) $\underline{P(\Omega,\Omega)}$: $\Omega(y,x(x-y),x) \equiv \Omega(y,x^2-y^2)$ by definition of Ω .

(b) $\underline{(\forall F,G)\{P(F,G) \Rightarrow P(\tau[F],\sigma[G])\}}$:

$$\begin{aligned}
\tau[F](y,x(x-y),x) &\equiv \underline{\text{if}}\ y = 0\\
&\quad\ \ \underline{\text{then}}\ x(x-0)\\
&\quad\ \ \underline{\text{else}}\ F(y-1,x(x-y)+x,x)\\
&\equiv \underline{\text{if}}\ y = 0\\
&\quad\ \ \underline{\text{then}}\ x^2\\
&\quad\ \ \underline{\text{else}}\ F(y-1,x(x-(y-1)),x)\\
&\equiv \underline{\text{if}}\ y = 0\\
&\quad\ \ \underline{\text{then}}\ x^2\\
&\quad\ \ \underline{\text{else}}\ G(y-1,x^2-(y-1)^2)\\
&\qquad\qquad \text{induction hypothesis}\\
&\equiv \underline{\text{if}}\ y = 0\\
&\quad\ \ \underline{\text{then}}\ x^2-0^2\\
&\quad\ \ \underline{\text{else}}\ G(y-1,(x^2-y^2)+2y-1)\\
&\equiv \sigma[G](y,x^2-y^2)\ .
\end{aligned}$$

□

Example 4. Consider the two programs (Morris [1971])

$$\begin{aligned}
F(x,y) &\Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ h(F(k(x),y))\\
G(x,y) &\Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ G(k(x),h(y))\ ,
\end{aligned}$$

where p stands for any naturally extended partial predicate, and h and k for any naturally extended partial functions. In order to prove that $F_\tau(x,y) \equiv G_\sigma(x,y)$ for all x and y , we shall consider

$$\begin{aligned}
P(F,G):\ \forall x\forall y\{&[F(x,y) \equiv G(x,y)] \wedge\\
&[G(x,h(y)) \equiv h(G(x,y))]\}\ .
\end{aligned}$$

We prove $P(F_\tau,G_\sigma)$, which implies $F_\tau \equiv G_\sigma$ as follows:

(a) $\underline{P(\Omega,\Omega)}$:

$[\Omega(x,y) \equiv \Omega(x,y)] \wedge [\Omega(x,h(y)) \equiv h(\Omega(x,y))]$, since $h(\omega) \equiv \omega$.

(b) $\underline{(\forall F,G)\{P(F,G) \Rightarrow P(\tau[F],\sigma[G])\}}$:

(1)
$$\begin{aligned}
\tau[F](x,y) &\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\\
&\qquad\qquad\ \ \underline{\text{else}}\ h(F(k(x),y))\\
&\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\\
&\qquad\qquad\ \ \underline{\text{else}}\ h(G(k(x),y))\\
&\qquad\qquad\qquad \text{induction hypothesis}\\
&\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\\
&\qquad\qquad\ \ \underline{\text{else}}\ G(k(x),h(y))\\
&\qquad\qquad\qquad \text{induction hypothesis}\\
&\equiv \sigma[G](x,y)\ .
\end{aligned}$$

(2)
$$\begin{aligned}
\sigma[G](x,h(y)) &\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ h(y)\\
&\qquad\qquad\ \ \underline{\text{else}}\ G(k(x),h^2(y))\\
&\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ h(y)\\
&\qquad\qquad\ \ \underline{\text{else}}\ h(G(k(x),h(y)))\\
&\qquad\qquad\qquad \text{induction hypothesis}\\
&\equiv h(\sigma[G](x,y))\ .
\end{aligned}$$

□

Computational Induction for a Set of Recursive Definitions

We shall state the computational induction principle for a program consisting of two recursive definitions,

$$\begin{cases}
F_1 \Leftarrow \tau_1[F_1,F_2]\\
F_2 \Leftarrow \tau_2[F_1,F_2]\ \ ;
\end{cases}$$

the generalization to a system of n $(n > 2)$ recursive definitions is straightforward.

To prove $P(F_{\tau_1},F_{\tau_2})$, where $P(F_1,F_2)$ is an admissible predicate, it is sufficient to:

(a) Prove $P(\Omega,\Omega)$

(b) Show that, for all F_1 and F_2,

$P(F_1,F_2)$ implies $P(\tau_1[F_1,F_2],\tau_2[F_1,F_2])$.

That is,

$$\frac{\underline{\text{from}}\ P(\Omega,\Omega) \text{ and } (\forall F_1,F_2)\{P(F_1,F_2) \Rightarrow P(\tau_1[F_1,F_2],\tau_2[F_1,F_2])\}\ ,}{\underline{\text{infer}}\ P(F_{\tau_1},F_{\tau_2})\ .}$$

In the following examples, we omit variables and parentheses whenever possible.

Example 5. Consider the program

$$\begin{cases} F_1 \Leftarrow \underline{\text{if}}\ p\ \underline{\text{then}}\ F_1F_2h\ \underline{\text{else}}\ F_2g \\ F_2 \Leftarrow \underline{\text{if}}\ q\ \underline{\text{then}}\ fF_2F_1\ \underline{\text{else}}\ fh \\ F_3 \Leftarrow \underline{\text{if}}\ p\ \underline{\text{then}}\ F_3fF_4h\ \underline{\text{else}}\ fF_4g \\ F_4 \Leftarrow \underline{\text{if}}\ q\ \underline{\text{then}}\ fF_4F_3\ \underline{\text{else}}\ h \end{cases}$$

in which p and q stand for any naturally extended partial predicates, and f, g and h for any naturally extended partial functions. To prove that $F_{\tau_1} \equiv F_{\tau_3}$, let $P(F_1,F_2,F_3,F_4)$ be $(F_1 \equiv F_3) \wedge (F_2 \equiv fF_4)$; we show that $P(F_{\tau_1},F_{\tau_2},F_{\tau_3},F_{\tau_4})$ as follows:

(a) $\underline{P(\Omega,\dots,\Omega)}$:

$(\Omega \equiv \Omega) \wedge (\Omega \equiv f\Omega)$ is true since $f(\omega) \equiv \omega$.

(b) $\underline{(\forall F_1,\dots,F_4)\{P(F_1,\dots,F_4) \Rightarrow P(\tau_1[F_1,\dots,F_4],\dots,\tau_4[F_1,\dots,F_4])\}}$:

$$\begin{aligned} \tau_1[F_1,F_2,F_3,F_4] &\equiv \underline{\text{if}}\ p\ \underline{\text{then}}\ F_1F_2h\ \underline{\text{else}}\ F_2g \\ &\equiv \underline{\text{if}}\ p\ \underline{\text{then}}\ F_3fF_4h\ \underline{\text{else}}\ fF_4g \\ &\qquad\qquad \text{induction hypothesis} \\ &\equiv \tau_3[F_1,F_2,F_3,F_4] \\ \tau_2[F_1,F_2,F_3,F_4] &\equiv \underline{\text{if}}\ q\ \underline{\text{then}}\ fF_2F_1\ \underline{\text{else}}\ fh \\ &\equiv f(\underline{\text{if}}\ q\ \underline{\text{then}}\ F_2F_1\ \underline{\text{else}}\ h) \\ &\equiv f(\underline{\text{if}}\ q\ \underline{\text{then}}\ fF_4F_3\ \underline{\text{else}}\ h) \\ &\qquad\qquad \text{induction hypothesis} \\ &\equiv f(\tau_4[F_1,F_2,F_3,F_4])\ . \end{aligned}$$

□

Transformations which Leave F_τ Invariant

We can use computational induction to prove useful theorems about recursive definitions. For example, if we modify a recursive definition $F \Leftarrow \tau[F]$ by replacing some occurrences of F in $\tau[F]$ by either $\tau[F]$ or F_τ , the function computed by the new program is precisely the original F_τ .

To prove this, let us write $\tau[F] \equiv \tau'[F,F]$, where we use the second argument in $\tau'[F,F]$ to distinguish the occurrences of F which we wish to replace. We define $\tau_1[F] \equiv \tau'[F,\tau[F]]$ and $\tau_2[F] \equiv \tau'[F,F_\tau]$; our goal is to show that $F_\tau \equiv F_{\tau_1} \equiv F_{\tau_2}$. We show this in two steps:

(a) $\underline{(F_{\tau_1} \subseteq F_\tau) \text{ and } (F_{\tau_2} \subseteq F_\tau)}$. This part is easy since by definition of τ_1 and τ_2 , $F_\tau \equiv \tau_1[F_\tau] \equiv \tau_2[F_\tau]$. That is, F_τ is a fixedpoint of both τ_1 and τ_2 , therefore, it is more defined than both F_{τ_1} and F_{τ_2} .

(b) $\underline{(F_\tau \subseteq F_{\tau_1}) \text{ and } (F_\tau \subseteq F_{\tau_2})}$. This can be shown by computational induction. (Hint: prove $P(F_\tau,F_{\tau_1},F_{\tau_2})$ where $P(F_1,F_2,F_3)$ is $(F_1 \subseteq F_2) \wedge (F_1 \subseteq F_3) \wedge (F_1 \subseteq \tau[F_1]) \wedge (F_1 \subseteq F_\tau)$.)

Example 6. To prove that $F(x) \Leftarrow \underline{\text{if}}\ x > 10\ \underline{\text{then}}\ x-10\ \underline{\text{else}}\ F(F(x+13))$ and $G(x) \Leftarrow \underline{\text{if}}\ x > 10\ \underline{\text{then}}\ x-10\ \underline{\text{else}}\ G(x+3)$ define the same function over the natural numbers, one just has to replace $F(x+13)$ by the value of $F_\tau(x+13)$, which is $x+3$ since $x+13 > 10$.

□

Example 7. Consider the following program (Morris [1971])*/ over the positive natural numbers:

```
F_1(x) <= if x = 1 then 0
                   else (if even(x)
                            then F_1(x/2)
                            else F_1(3x+1))
F_2(x) <= if x = 1 then 0 else F_2(F_1(x))
```

*/ It is not known whether the function $F_{\tau_1}(x)$ is defined for all x or not; a computer program checked that it is defined for all positive integers up to 3*10^8 .

$$F_3(x) \Leftarrow \underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ (\underline{\text{if}}\ \text{even}(x)\ \underline{\text{then}}\ x/2\ \underline{\text{else}}\ F_3(F_3(\tfrac{3x+1}{2})))\ .$$

We shall prove that $F_{\tau_1} \equiv F_{\tau_2}$ by transforming the definition of F_1 and F_2 respectively, until we reach the same <u>recursive definition</u>.

First, $F_{\tau_1} \equiv F_{\tau_1'}$, where:

$$\tau_1'[F](x) \equiv (\underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ (\underline{\text{if}}\ \text{even}(x)\ \underline{\text{then}}\ F(x/2)\ \underline{\text{else}}\ \tau_1[F](3x+1)))\ .$$

Since $[\text{odd}(x)$ and $x > 1]$ imply $[\text{even}(3x+1)$ and $3x+1 > 1]$,

$$\tau[F](x) \equiv (\underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ (\underline{\text{if}}\ \text{even}(x)\ \underline{\text{then}}\ F(x/2)\ \underline{\text{else}}\ F(\tfrac{3x+1}{2})))\ .$$

Also $F_{\tau_2} \equiv F_{\tau_2'}$ where $\tau_2'[F_2,F_3] \equiv \tau_2[F_2,\tau_3[F_3]]$, i.e.,

$$\tau_2'[F_2,F_3](x) \equiv (\underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ (\underline{\text{if}}\ \text{even}(x)\ \underline{\text{then}}\ F_2(x/2)\ \underline{\text{else}}\ F_2(F_3(F_3(\tfrac{3x+1}{2})))))\ .$$

Again $F_{\tau_2'} \equiv F_{\tau_2''}$ where

$$\tau_2''[F_2,F_3](x) \equiv (\underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ (\underline{\text{if}}\ \text{even}(x)\ \underline{\text{then}}\ F_2(x/2)\ \underline{\text{else}}\ F_{\tau_2}(F_{\tau_3}(F_{\tau_3}(\tfrac{3x+1}{2})))))\ .$$

The result $F_{\tau_2}F_{\tau_3} \equiv F_{\tau_2}$ is easily established from the definition of F_2 by considering the three cases for $x = 1$: <u>true</u>, <u>false</u>, <u>undefined</u>(ω) . Thus

$$\tau_2''[F_2,F_3](x) \equiv (\underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ (\underline{\text{if}}\ \text{even}(x)\ \underline{\text{then}}\ F_2(x/2)\ \underline{\text{else}}\ F_{\tau_2}(\tfrac{3x+1}{2})))\ .$$

Finally, we consider

$$\sigma[F_2,F_3](x) \equiv (\underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ (\underline{\text{if}}\ \text{even}(x)\ \underline{\text{then}}\ F_2(x/2)\ \underline{\text{else}}\ F_2(\tfrac{3x+1}{2})))\ .$$

Clearly $\sigma[F_2,F_3] \equiv \tau[F_2]$; since $F_{\tau_1} \equiv F_\tau$ and $F_{\tau_2} \equiv F_\sigma$, we establish $F_{\tau_1} \equiv F_{\tau_2}$ as desired. □

<u>Example 8</u>. To prove $F_{\tau_1} \equiv F_{\tau_3}$ for the program:

$$\begin{cases} F_1 \Leftarrow \underline{\text{if}}\ p\ \underline{\text{then}}\ F_3F_2F_2f\ \underline{\text{else}}\ g \\ F_2 \Leftarrow \underline{\text{if}}\ q\ \underline{\text{then}}\ F_3h\ \underline{\text{else}}\ k \\ F_3 \Leftarrow \underline{\text{if}}\ p\ \underline{\text{then}}\ F_1F_4f\ \underline{\text{else}}\ g \\ F_4 \Leftarrow \underline{\text{if}}\ q\ \underline{\text{then}}\ F_2F_3h\ \underline{\text{else}}\ F_2k \end{cases}$$

we first change the definitions of F_1 and F_4 to

$$F_1 \Leftarrow \underline{\text{if}}\ p\ \underline{\text{then}}\ F_3F_{\tau_2}F_2f\ \underline{\text{else}}\ g$$

and

$$F_4 \Leftarrow \underline{\text{if}}\ q\ \underline{\text{then}}\ F_{\tau_2}F_3h\ \underline{\text{else}}\ F_{\tau_2}k\ ,$$

respectively, and then prove by computational induction that:

$$(F_1 \equiv F_3) \wedge (F_{\tau_2}F_2 \equiv F_4)\ .$$

The reader should be aware of the difficulties involved in proving that $F_{\tau_1} \equiv F_{\tau_3}$ without the above modifications. □

<u>Truncation Induction</u>

If for some continuous functional τ we define the sequence of functions f^i by letting $f^i \equiv \tau^i[\Omega]$, i.e.,

$$f^0 \equiv \Omega \text{ and } f^{i+1} \equiv \tau[f^i] \text{ for all } i \in N\ ,$$

then the same argument used to establish the validity of computational induction also shows the validity of the following very similar rule:

from $P(f^0)$ and $(\forall i \in N)[P(f^i) \supset P(f^{i+1})]$,
infer $P(F_\tau)$.

The resemblance of this rule to the usual mathematical induction on natural numbers suggests that we consider a similar rule using complete induction over natural numbers.[*] Morris [1971] called it truncation induction. More precisely,

In order to prove $P(F_\tau)$, $P(F)$ being an admissible predicate, we show that for any natural number i, the truth of $P(f^j)$ for all $j < i$ implies the truth of $P(f^i)$. That is,

from $(\forall i \in N)[[(\forall j \in N \text{ such that } j < i)P(f^j)] \supset P(f^i)]$,
infer $P(F_\tau)$.

The validity of this rule is established by first using induction on N to show that $P(f^n)$ holds for all $n \in N$; one can then use the proof given above for the validity of computational induction.

When the program consists of a system of recursive definitions such as

$$\begin{cases} F_1 \Leftarrow \tau_1[F_1,\ldots,F_k] \\ \quad \vdots \\ F_k \Leftarrow \tau_k[F_1,\ldots,F_k], \end{cases}$$

we let f^0 be $\langle\Omega,\ldots,\Omega\rangle$, f^{i+1} be $\langle\tau_1[f^i],\ldots,\tau_k[f^i]\rangle$, and F_τ be $\langle F_{\tau_1},\ldots,F_{\tau_n}\rangle$; the truncation induction rule is then precisely the same as above.

Example 9. (Morris [1971]). We consider again (see Example 4) the two programs:

$F(x,y) \Leftarrow (\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ h(F(k(x),y)))$

$G(x,y) \Leftarrow (\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ G(k(x),h(y)))$,

where p stands for any naturally extended partial predicate, and h and k for any naturally extended partial functions.

In order to prove that both programs define the same function, we check that $f^0 \equiv g^0$, $f^1 \equiv g^1$ and that $f^n \equiv g^n$ for all $n \geq 2$. (We treat the cases for $n = 0$ and $n = 1$ separately, since to prove $f^n \equiv g^n$ we have to use the induction hypothesis for both $n-1$ and $n-2$.)

(a) $f^0 \equiv g^0$: $\Omega \equiv \Omega$.

(b) $f^1 \equiv g^1$:

$f^1(x,y) \equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ h(f^0(k(x,y)))$
$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ \omega$
$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ g^0(k(x),h(y))$
$\equiv g^1(x,y)$.

(c) $f^n \equiv g^n$ for $n \geq 2$:

$f^n(x,y) \equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ h(f^{n-1}(kx,y))$
definition of f^n

$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ h(g^{n-1}(kx,y))$
induction hypothesis $(n-1)$

$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y$
$\quad \underline{\text{else}}\ h(\underline{\text{if}}\ p(k(x))$
$\quad\quad \underline{\text{then}}\ y$
$\quad\quad \underline{\text{else}}\ g^{n-2}(k^2(x),h(y)))$
definition of g^{n-1}

$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y$
$\quad \underline{\text{else}}\ (\underline{\text{if}}\ p(k(x))$
$\quad\quad \underline{\text{then}}\ h(y)$
$\quad\quad \underline{\text{else}}\ hg^{n-2}(k^2(x),h(y)))$

$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y$
$\quad \underline{\text{else}}\ (\underline{\text{if}}\ p(k(x))$
$\quad\quad \underline{\text{then}}\ h(y)$
$\quad\quad \underline{\text{else}}\ hf^{n-2}(k^2(x),h(y))$
induction hypothesis $(n-2)$

$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ f^{n-1}(k(x),h(y))$
definition of f^{n-1}

$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ y\ \underline{\text{else}}\ g^{n-1}(k(x),h(y))$
induction hypothesis $(n-1)$

$\equiv g^n(x,y)$. definition of g^n

□

It is often useful to define slightly different sequences of functions f^i and then apply a generalized form of truncation induction, as illustrated in the next example. Kleene's first recursion theorem can again be used to establish the

[*] When applied to natural numbers, these two inductions are equivalent; thus truncation induction and computational induction are equivalent from a theoretical point of view. Experience in using both methods shows that they are also equivalent in practice.

validity of such generalized truncation induction rules.

Example 10. We consider again the program

$$F \Leftarrow \underline{\text{if}}\ p\ \underline{\text{then}}\ \mathcal{I}\ \underline{\text{else}}\ FFh\ ,$$

where $\mathcal{I}$ is the identity function. We shall prove that $F_\tau F_\tau \equiv F_\tau$. For this purpose we define the sequence of functions f^i in the following way:

$f^0 \equiv \Omega$

$f^1 \equiv \tau[\Omega]$

$f^{n+2} \equiv \underline{\text{if}}\ p\ \underline{\text{then}}\ \mathcal{I}\ \underline{\text{else}}\ f^{n+1}f^n h$ for $n \geq 0$;

note that for $n \geq 2$, it is not necessarily the case that $f^n \equiv \tau^n[\Omega]$.

We shall prove the desired result by generalized truncation induction, letting the induction hypothesis be $f^{n+3}f^{n+2} \equiv f^{n+4}$.

We first check the cases $n = 0$ and $n = 1$ (the details are omitted); then, assuming that the induction hypothesis is true for all $i < n$, we get

$f^{n+3}f^{n+2} \equiv f^{n+3}(\underline{\text{if}}\ p\ \underline{\text{then}}\ \mathcal{I}\ \underline{\text{else}}\ f^{n+1}f^n h)$

definition of f^{n+2}

$\equiv \underline{\text{if}}\ p\ \underline{\text{then}}\ f^{n+3}\ \underline{\text{else}}\ f^{n+3}f^{n+1}f^n h$

$\equiv \underline{\text{if}}\ p\ \underline{\text{then}}\ \mathcal{I}\ \underline{\text{else}}\ f^{n+3}f^{n+1}f^n h$

definition of f^{n+3}

$\equiv \underline{\text{if}}\ p\ \underline{\text{then}}\ \mathcal{I}\ \underline{\text{else}}\ f^{n+3}f^{n+2}h$

induction hypothesis

$\equiv f^{n+4}$. definition of f^{n+4}

By complete induction, it follows that $f^{n+3}f^{n+2} \equiv f^{n+4}$ for all $n \in N$; then Kleene's theorem can be used to establish that $F_\tau F_\tau \equiv F_\tau$, as desired. ☐

III. STRUCTURAL INDUCTION

One familiar method of proving assertions on the domain N of natural numbers is that of complete induction: in order to prove that the statement $P(c)$ is true for every natural number c , we show that for any natural number a , the truth of $P(b)$ for all $b < a$ implies the truth of $P(a)$.

That is,

from $(\forall a \in D)\{[(\forall b \in D \text{ s.t. } b < a)P(b)] \Rightarrow P(a)]\}$,
infer $(\forall c \in D)P(c)$.

Since this induction rule is not valid for every ordered domain,[*] we shall first characterize the ordered domains which are 'good' for induction. We then present a general rule for proving statements over such domains, called structural induction; complete induction, as well as many other well-known induction rules, is a special case of structural induction. Finally, we give several examples using structural induction to prove properties of recursively defined functions.

Well-founded Sets

A partially ordered set $(S, <)$ consists of a set S and a partial ordering $<$ on S .[**] A partially ordered set $(S, <)$ which contains no infinite decreasing sequence $a_0 > a_1 > a_2 > \dots$ of elements of S is called a well-founded set.

Examples. (a) The set of all real numbers between 0 and 1 , with the usual ordering $\leq$, is partially ordered but not well-founded (consider the infinite decreasing sequence $\frac{1}{2} > \frac{1}{3} > \frac{1}{4} > \dots$).

(b) The set I of integers, with the usual ordering $\leq$, is partially ordered but not well-founded (consider $0 > -1 > -2 > \dots$).

(c) The set N of natural numbers, with the usual ordering $\leq$, is well-founded.

(d) If Σ is any alphabet, then the set Σ^* of all finite strings over Σ (i.e., sequences of letters of Σ), with the substring relation ($w_1 < w_2$ iff w_1 is a substring of w_2), is well-founded.

☐

Structural Induction

We may now state and prove the rule of structural induction on well-founded sets.[***] Suppose

[*] e.g., it is valid over the natural numbers with ordering $\leq$ but fails over the integers with ordering $\leq$ (consider P which is always false).

[**] Note that the ordering need not be total, i.e., it is possible that for some $a, b \in S$, neither $a < b$ nor $b < a$ holds.

[***] Structural induction is sometimes also called Noetherian induction. When the ordering $<$ is total, i.e., $a < b$ or $b < a$ holds for any $a, b \in S$, it is called transfinite induction.

P is a total predicate over the well-founded set $(S, \prec)$. If for any a in S, we can prove that the truth of $P(a)$ is implied by the truth of $P(b)$ for all $b \prec a$, then $P(c)$ is true for every c in S . That is:

<u>from</u> $(\forall a \in S)\{[(\forall b \in S \text{ s.t. } b \prec a)P(b)] \supset P(a)\}$,
<u>infer</u> $(\forall c \in S)P(c)$.

To prove the validity of this rule, we show that if the assumption is satisfied, there can be no element in S for which P does not hold. Consider the set A of elements $a \in S$ such that $P(a)$ is false. Let us assume that A is nonempty. Then, there is a least element a_0 such that $a \not\prec a_0$ for any $a \in A$: otherwise, there would be an infinite descending sequence in S . Then, for any element b such that $b \prec a_0$, $P(b)$ must hold; that is, $(\forall b \in S \text{ s.t. } b \prec a_0)P(b)$. But the assumption then implies that $P(a_0)$, in contradiction with the fact that $a_0 \in A$. Therefore A must be empty, i.e., $P(c)$ must hold for all elements $c \in S$.

<u>Applications</u>

We now give several examples using structural induction to prove properties of recursive programs. Such proofs require suitable choices of both the partial ordering $\prec$ and of the predicate P . Some of the examples show that the partial ordering to be chosen is not always the usual partial ordering on the domain. Other examples illustrate that it is often useful to prove a more general result than the desired theorem.

<u>Example 1</u>. (Cadiou) Factorial functions.

Consider the two programs

$F_1(x) \Leftarrow$ <u>if</u> $x = 0$ <u>then</u> 1 <u>else</u> $x \cdot F_1(x-1)$

and

$$\begin{cases} F_2(x) \Leftarrow F_3(x,0,1) \\ F_3(x,y,z) \Leftarrow \underline{\text{if}}\ x = y\ \underline{\text{then}}\ z\ \underline{\text{else}}\ F_3(x,y+1,(y+1)z) \end{cases}$$

F_{τ_1} and F_{τ_2} compute $x! = 1 \cdot 2 \cdot \ldots \cdot x$ for every $x \in N$ in two different ways: F_{τ_2} by 'going up' from 0 to x and F_{τ_1} by 'going down' from x to 0 . We wish to show that $F_{\tau_2} \equiv F_{\tau_1}$. The proof uses the predicate $P(x)$:

$$(\forall y \text{ s.t. } y \geq x)[F_{\tau_3}(y,y-x,F_{\tau_1}(y-x)) \equiv F_{\tau_1}(y)] ,$$

and the usual ordering on natural numbers.

(a) If $x = 0$, $P(0)$ is $\forall y[F_{\tau_3}(y,y,F_{\tau_1}(y)) \equiv F_{\tau_1}(y)]$, which is clearly true by definition of F_{τ_3} .

(b) If $x > 0$, we assume $P(x')$ for all $x' < x$ and show $P(x)$.

For any y s.t. $y \geq x$,

$$\begin{aligned} F_{\tau_3}(y,y-x,F_{\tau_1}(y-x)) &\equiv F_{\tau_3}(y,y-x+1,(y-x+1)\cdot F_{\tau_1}(y-x)) && \text{definition of } F_{\tau_3} \\ &&& (\text{since } x > 0) \\ &\equiv F_{\tau_3}(y,y-x+1,F_{\tau_1}(y-x+1)) && \text{definition of } F_{\tau_1} \\ &&& (\text{since } y-x+1 > 0) \\ &\equiv F_{\tau_3}(y,y-(x-1),F_{\tau_1}(y-(x-1))) \\ &\equiv F_{\tau_1}(y) \ . && \text{induction hypothesis} \\ &&& (\text{since } (x-1) < x \text{ and } (x-1) \leq y) \end{aligned}$$

By complete induction then, $P(x)$ holds for all $x \in N$. In particular, for $y = x$, $F_{\tau_3}(x,0,F_{\tau_1}(0)) \equiv F_{\tau_1}(x)$. Since $F_{\tau_1}(0) = 1$ and $F_{\tau_3}(x,0,1) \equiv F_{\tau_2}(x)$, we have $F_{\tau_2} \equiv F_{\tau_1}$ as desired. $\square$

Given a partially ordered set $(S, \prec)$, we define the <u>lexicographic ordering</u> $\underset{n}{\prec}$ on n-tuples of elements of S (i.e., on elements of S^n) by letting $(a_1,\ldots,a_n) \underset{n}{\prec} (b_1,\ldots,b_n)$ iff $a_1 = b_1 \wedge \ldots \wedge a_{i-1} = b_{i-1} \wedge a_i \prec b_i$ for some i, $1 \leq i \leq n$. It is easy to show that if $(S, \prec)$ is well-founded, so is $(S^n, \underset{n}{\prec})$. For example,

(a) For the natural numbers with the usual ordering $\leq$, the set $(N^2, \underset{2}{\leq})$ is well-founded: $(n_1,n_2) \underset{2}{\prec} (m_1,m_2)$ iff $n_1 < m_1$ or $(n_1 = m_1) \wedge (n_2 < m_2)$ (note that e.g. $(1,100) \underset{2}{\prec} (2,1)$).

(b) For the alphabet $\Sigma = \{A,B,\ldots,Z\}$ with the usual ordering $A < B < C < \ldots < Z$, the set

$(\Sigma^3, \underset{3}{\leq})$ of words of length three is well-founded. Note that this is the usual alphabetic order: e.g., ACK < BAT < CAD < CAT .

Example 2. Ackermann's function.

Consider the program

```
A(x,y) <= if x = 0 then y+1
              else if y = 0 then A(x-1,1)
                        else A(x-1,A(x,y-1)).
```

We wish to show that $A_\tau(x,y)$ is defined for any $x,y \in N$, i.e., that the computation of $A_\tau(x,y)$ always terminates. We shall use the structural induction rule applied on $(N^2, \underset{2}{\leq})$. Assuming that $A_\tau(x',y')$ is defined for any (x',y') such that $(x',y') \underset{2}{\leq} (x,y)$, we show that $A_\tau(x,y)$ must also be defined.

(a) If $x = 0$, termination is obvious.

(b) If $x \neq 0$ and $y = 0$, we note that $(x-1,1) \underset{2}{\leq} (x,y)$, so by the induction hypothesis $A_\tau(x-1,1)$ is defined. Thus $A_\tau(x,y)$ is also defined.

(c) Finally, if $x \neq 0$ and $y \neq 0$, $(x,y-1) \underset{2}{\leq} (x,y)$ and $A_\tau(x,y-1)$ is therefore defined by the induction hypothesis; then, regardless of the value of $A_\tau(x,y-1)$, $(x-1,A_\tau(x,y-1)) \underset{2}{\leq} (x,y)$ and the desired result follows by another application of the induction hypothesis. □

In each of the preceding examples we used the most natural ordering on the domain to perform the structural induction. In the next example it is natural to use a somewhat surprising ordering.

Example 3. (Burstall). The 91-function.

The 91-function F_τ is defined by the following program over the integers:

$F(x) \Leftarrow$ if $x > 100$ then $x-10$ else $F(F(x+11))$.

We wish to show that $F_\tau \equiv g$, where g is

$g(x) \equiv$ if $x > 100$ then $x-10$ else 91 .

The proof is by structural induction on the well-founded set $(I, \prec)$, where I is the integers and $\prec$ is defined as follows:

$$x \prec y \text{ iff } y < x \leq 101$$

(where $<$ is the usual ordering on the integers); thus $101 \prec 100 \prec 99 \prec \cdots$, but for example, $102 \nprec 101$. One can easily check that $(I, \prec)$ is well-founded.

Suppose $F_\tau(y) \equiv g(y)$ for all $y \in I$ such that $y \prec x$. We must show that $F_\tau(x) \equiv g(x)$.

(a) For $x > 100$, $F_\tau(x) \equiv g(x)$ directly.

(b) For $100 \geq x \geq 90$, $F_\tau(x) \equiv F_\tau(F_\tau(x+11)) \equiv F_\tau(x+1)$, and since $x+1 \prec x$ we have $F_\tau(x) \equiv F_\tau(x+1) \equiv g(x+1)$ by the induction assumption. But $g(x+1) \equiv 91 \equiv g(x)$, therefore $F_\tau(x) \equiv g(x)$.

(c) Finally, for $x < 90$, $F_\tau(x) \equiv F_\tau(F_\tau(x+11))$, and since $x+11 \prec x$ we have $F_\tau(x) \equiv F_\tau(F_\tau(x+11)) \equiv F_\tau(g(x+11))$ by induction. But $g(x+11) \equiv 91$, and we know by induction that $F_\tau(91) \equiv g(91) \equiv 91$, so $F_\tau(x) \equiv F_\tau(g(x+11)) \equiv F_\tau(91) \equiv 91 \equiv g(x)$, as desired.

We could alternatively have proven the above property by structural induction on the natural numbers with the usual ordering $<$, using the more complicated predicate $P(n)$: $(\forall x \in I)[x > 100-n \Rightarrow F_\tau(x) \equiv g(x)]$. The reader should note that the details of this proof and of the above proof are precisely the same. □

Since the set $(\Sigma^*, \prec)$ of finite strings Σ with the substring relation is well-founded, we may use it for structural induction. In the following example we use an induction rule that can easily be derived from structural induction, namely:

$$\frac{\text{from } P(\Lambda) \text{ and } (\forall x \in \Sigma^*)[x \neq \Lambda \wedge P(t(x)) \Rightarrow P(x)]}{\text{infer } (\forall x \in \Sigma^*) P(x)}.$$

Example 4. The reverse function.

The program reverse ,

$$\begin{cases} \underline{\text{reverse}}(x) \Leftarrow F(x,\Lambda) \\ F(x,y) \Leftarrow \text{if } x = \Lambda \text{ then } y \text{ else } F(t(x),h(x)\cdot y) \end{cases},$$

gives as value over Σ^* the string made up of the letters of x in reverse order. For example, if $\Sigma = \{A,B,C\}$ then reverse(ACBB) = BBCA .

We wish to prove that reverse(x) is defined and that reverse(reverse(x)) $\equiv x$ for all $x \in \Sigma^*$. Of course, proving that reverse has this property does not show that it actually reverses all words: many other functions, e.g. the identity function, also satisfy this property.

1. To prove that $\underline{reverse}(x)$ is always defined, we let

   $P(x)$ be $(\forall y \in \Sigma^*)[F_\tau(x,y)$ is defined$]$.

   (a) If $x = \Lambda$, then $F_\tau(x,y) \equiv y$, and thus $F_\tau(x,y)$ is defined.

   (b) If $x \neq \Lambda$, since $t(x) < x$ we may assume that $F_\tau(t(x),z)$ is defined for any z ; therefore $F_\tau(t(x),h(x)\cdot y)$ is defined for all $y \in \Sigma^*$. Thus $F_\tau(x,y)$ is defined for any y .

   It follows by structural induction that $F_\tau(x,y)$ is defined for all $x,y \in \Sigma^*$, and in particular, since $\underline{reverse}(x) \equiv F_\tau(x,\Lambda)$, $\underline{reverse}(x)$ is defined for all $x \in \Sigma^*$.

2. To prove $\underline{reverse}(\underline{reverse}(x)) \equiv x$ we let

   $P(x)$ be $(\forall y \in \Sigma^*)[\underline{reverse}(F_\tau(x,y)) \equiv F_\tau(y,x)]$.

   (a) If $x = \Lambda$, then for any y we have $\underline{reverse}(F_\tau(x,y)) \equiv \underline{reverse}(y) \equiv F_\tau(y,\Lambda) \equiv F_\tau(y,x)$.

   (b) If $x \neq \Lambda$, then for any y we have

   $\underline{reverse}(F_\tau(x,y))$

   $\equiv \underline{reverse}(F_\tau(t(x),h(x)\cdot y))$ — definition of F_τ (since $x \neq \Lambda$)

   $\equiv F_\tau(h(x)\cdot y,t(x))$ — induction hypothesis (since $x > t(x)$)

   $\equiv F_\tau(y,h(x)\cdot t(x)$ — definition of F_τ (since $h(x)\cdot y \neq \Lambda$)

   $\equiv F_\tau(y,x)$.

   Therefore $\underline{reverse}(F_\tau(x,y)) \equiv F_\tau(y,x)$ for all $x,y \in \Sigma^*$; in particular, for $y = \Lambda$, $\underline{reverse}(\underline{reverse}(x)) \equiv \underline{reverse}(F_\tau(x,\Lambda)) \equiv F_\tau(\Lambda,x) \equiv x$, as desired. ☐

Other properties of $\underline{reverse}$ may easily be proven by structural induction. In particular, the following example uses the properties that, for any $a,b \in \Sigma$ and $x \in \Sigma^*$:

(i) $\underline{reverse}(x*a) \equiv a\cdot\underline{reverse}(x)$,

(ii) $\underline{reverse}(a\cdot x) \equiv \underline{reverse}(x)*a$, and

(iii) $\underline{reverse}(a\cdot(x*b)) \equiv b\cdot(\underline{reverse}(x)*a)$.

Example 5. Another reverse function.

We wish to show that the program (due to Ashcroft)

```
R(x) <= if x = Λ
        then Λ
        else if t(x) = Λ
             then x
             else h(R(t(x)))·R(h(x)·R(t(R(t(x)))))
```

also defines a reversing function on Σ^* , i.e., that $R_\tau(x) \equiv \underline{reverse}(x)$ for all $x \in \Sigma^*$. Note that this program uses only one recursive definition.

In the proof we shall use the following lemma characterizing the elements of Σ^* : for any $x \in \Sigma^*$, either $x = \Lambda$, or $x \in \Sigma$ (i.e., $t(x) = \Lambda$), or $x = y\cdot(w*z)$ for some $y \in \Sigma$, $w \in \Sigma^*$, and $z \in \Sigma$. The lemma is easy to prove by a straightforward structural induction.

We now prove that $R_\tau \equiv \underline{reverse}$ by structural induction on $(\Sigma^*, <)$, where $<$ is the following partial ordering:

$x < y$ iff x is a substring of y or x is a substring of $\underline{reverse}(y)$. One can check that $(\Sigma^*, <)$ is well-founded.

Using the above lemma, the proof may be done in three parts.

(a) $x = \Lambda$: $R_\tau(x) \equiv \Lambda \equiv \underline{reverse}(x)$.

(b) $x \in \Sigma$: $R_\tau(x) \equiv x \equiv \underline{reverse}(x)$.

(c) $x = y\cdot(w*z)$ for some $y \in \Sigma$, $w \in \Sigma^*$, $z \in \Sigma$:

$R_\tau(x)$

$\equiv h(R_\tau(t(x)))\cdot R_\tau(h(x)\cdot R_\tau(t(R_\tau(t(x)))))$ — definition of R_τ

$\equiv h(R_\tau(w*z))\cdot R_\tau(y\cdot R_\tau(t(R_\tau(w*z))))$ — since $h(x) = y, t(x) = w*z$

$\equiv h(\underline{reverse}(w*z))\cdot R_\tau(y\cdot R_\tau(t(\underline{reverse}(w*z))))$ — induction hypothesis (since $w*z < x$)

$\equiv h(z\cdot\underline{\text{reverse}}(w))\cdot R_\tau(y\cdot R_\tau(t(z\cdot\underline{\text{reverse}}(w))))$

property (i) of <u>reverse</u>

$\equiv z\cdot R_\tau(y\cdot R_\tau(\underline{\text{reverse}}(w)))$

properties of h and t

$\equiv z\cdot R_\tau(y\cdot\underline{\text{reverse}}(\underline{\text{reverse}}(w)))$

induction hypothesis (since $\underline{\text{reverse}}(w) < x$)[*]

$\equiv z\cdot R_\tau(y\cdot w)$	property of <u>reverse</u> proven in previous example
$\equiv z\cdot\underline{\text{reverse}}(y\cdot w)$	induction hypothesis (since $y\cdot w < x$)
$\equiv z\cdot(\underline{\text{reverse}}(w)*y)$	property (ii) of <u>reverse</u>
$\equiv \underline{\text{reverse}}(x)$	property (iii) of <u>reverse</u>

We conclude that $R_\tau(x) \equiv \underline{\text{reverse}}(x)$ for all $x\epsilon\Sigma^*$, as desired. □

Comparison between Computational Induction and Structural Induction

Although computational induction and structural induction appear to be quite different methods, we shall show how any proof using one method can be translated to a proof using the other.

(a) Translation from Computational Induction to Structural Induction

Using the principle of structural induction and the hypothesis $P(\Omega)$ and $\forall F\{P(F) \Rightarrow P(\tau[F])\}$, where $P(F)$ is taken to be $\alpha[F] \subseteq \beta[F]$ for simplicity, we must prove $P(F_\tau)$. For that purpose, we shall consider a well-founded ordering $\prec$ over D^+ which resembles the computation of $\alpha[F_\tau]$ as follows: $x \prec y$ iff the Kleene computation sequence[**] of $\alpha[F_\tau](x)$ is shorter than that of $\alpha[F_\tau](y)$. Using the definition, it is straightforward to show that

$(\forall x\epsilon D^+)\{[(\forall y\epsilon D^+ \text{ such that } y \prec x)Q(y)] \Rightarrow Q(x)\}$,

where $Q(x)$ is $\alpha[F_\tau](x) \subseteq \beta[F_\tau](x)$; by structural induction, it follows that $(\forall x\epsilon D^+)Q(x)$, i.e., $P(F_\tau)$.

[*] Note that $\underline{\text{reverse}}(w) \prec x$, which is true because $\underline{\text{reverse}}(w)$ is a substring of $\underline{\text{reverse}}(x)$, as may be seen from property (c) of <u>reverse</u>.

[**] i.e., $\exists n$ s.t. $\tau^n[\Omega](x) \not\equiv \omega$ and $\tau^n[\Omega](y) \equiv \omega$.

(b) Translation from Structural Induction to Computational Induction

Unlike computational induction, structural induction may be used to prove general mathematical theorems, rather than just properties of programs. However, if we restrict ourselves to proving properties of programs,[*] Milner [1972b] has shown that structural induction can be nicely replaced by computational induction. The next two examples illustrate the use of this technique in cases where more direct computational induction fails.

<u>Example 6</u>. Factorial functions.

Consider again the two programs of Example 1:

$$F_1(x) \Leftarrow \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ x\cdot F_1(x-1) \quad ,$$

and

$$\begin{cases} F_2(x) \Leftarrow F_3(x,0,1)\ , \\ F_3(x,y,z) \Leftarrow \underline{\text{if}}\ x = y\ \underline{\text{then}}\ z \\ \qquad\qquad\qquad\qquad \underline{\text{else}}\ F_3(x,y+1,(y+1)\cdot z) \quad . \end{cases}$$

Since F_{τ_1} and F_{τ_2} are computed differently, the equivalence $F_{\tau_1} \equiv F_{\tau_2}$ cannot be proven directly by computational induction. However, we can consider the domain of natural numbers to be recursively defined by the program

$$\underline{\text{number}}(x) \Leftarrow \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{number}}(x-1),$$

and proceed to prove by computational induction on this program that:

$$(\forall x,y)\{[x > y \lor \underline{\text{number}}(x)] \subseteq [F_{\tau_3}(y,y-x,F_{\tau_1}(y-x)) \equiv F_{\tau_1}(y)]\}\ .$$

The proof can now be carried on, following the steps of the proof of Example 1. □

<u>Example 7</u>. Reverse function.

Consider again the program of Example 4 defining the <u>reverse</u> function:

$$\begin{cases} \underline{\text{reverse}}(x) \Leftarrow F(x,\Lambda) \\ F(x,y) \Leftarrow \underline{\text{if}}\ x = \Lambda\ \underline{\text{then}}\ y\ \underline{\text{else}}\ F(t(x),h(x)\cdot y)\ . \end{cases}$$

We shall show that $\underline{\text{reverse}}(x)$ (i.e., $F_\tau(x,\Lambda)$) is defined for any x in Σ^*. For this purpose, we characterize the elements of Σ^* by the program

[*] More precisely, when the well-founded ordering can be recursively defined.

$\underline{\mathrm{word}}(x) \Leftarrow \sigma[\underline{\mathrm{word}}](x)$, where

$$\sigma[G](x) \equiv (\underline{\mathrm{if}}\ x = \Lambda\ \underline{\mathrm{then}}\ \underline{\mathrm{true}}\ \underline{\mathrm{else}}\ G(t(x))).$$

We let $P(F,G)$ be $(\forall x,y \in \Sigma^*)\{[G(x) \wedge \underline{\mathrm{word}}(y)] \subseteq \underline{\mathrm{word}}(F(x,y))\}$.

(a) $\underline{P(\Omega,\Omega)}$: $[\Omega(x) \wedge \underline{\mathrm{word}}(y)] \subseteq \underline{\mathrm{word}}(\Omega(x,y))$ reduces to $\omega \subseteq \omega$.

(b) $\underline{\forall F,G\{P(F,G) \Rightarrow P(\tau[F],\sigma[G])\}}$:

$$
\begin{aligned}
&[\sigma[G](x) \wedge \underline{\mathrm{word}}(y)] \\
&\equiv \underline{\mathrm{if}}\ x = \Lambda\ \underline{\mathrm{then}}\ \underline{\mathrm{word}}(y) \\
&\qquad \underline{\mathrm{else}}\ G(t(x)) \wedge \underline{\mathrm{word}}(y) && \text{definition of } \sigma \\
&\equiv \underline{\mathrm{if}}\ x = \Lambda\ \underline{\mathrm{then}}\ \underline{\mathrm{word}}(y) \\
&\qquad \underline{\mathrm{else}}\ G(t(x)) \wedge \underline{\mathrm{word}}(h(x)\cdot y) && \text{definition of } \underline{\mathrm{word}} \\
&\subseteq \underline{\mathrm{if}}\ x = \Lambda\ \underline{\mathrm{then}}\ \underline{\mathrm{word}}(y) \\
&\qquad \underline{\mathrm{else}}\ \underline{\mathrm{word}}(F(t(x),h(x)\cdot y)) && \text{induction hypothesis} \\
&\equiv \underline{\mathrm{word}}(\tau[F](x,y)) \ . && \text{definition of } \tau
\end{aligned}
$$

Therefore, by computational induction, we have:

$$[\underline{\mathrm{word}}(x) \wedge \underline{\mathrm{word}}(y)] \subseteq \underline{\mathrm{word}}(F_\tau(x,y)) \qquad \text{for all } x,y \in \Sigma^* ,$$

which for $y = \Lambda$ gives $\underline{\mathrm{word}}(x) \subseteq \underline{\mathrm{word}}(\underline{\mathrm{reverse}}(x))$; i.e., $\underline{\mathrm{reverse}}(x)$ is defined and its value is a word whenever x is a word. □

IV. OTHER METHODS

In this section, we present two additional methods for proving properties of programs: recursion induction and inductive assertions. We show that any proof by either of these methods can be effectively translated into a proof by computational induction (and therefore also into a proof by structural induction).

Recursion Induction

To prove the equivalence of two functions f_1 and f_2 over some subdomain S of D , i.e., that $f_1(x) \equiv f_2(x)$ for all $x \in S$, it is sufficient to find a functional τ such that:

(a) f_1 is a fixedpoint of τ , i.e., $f_1 \equiv \tau[f_1]$,

(b) f_2 is a fixedpoint of τ , i.e., $f_2 \equiv \tau[f_2]$, and

(c) $F_\tau(x)$ is defined for any x in S .

The justification of this rule is easy: by definition of F_τ , we know that $F_\tau(x) \subseteq f_1(x)$ and $F_\tau(x) \subseteq f_2(x)$, for any x in D ; therefore, for any x in S , since $F_\tau(x)$ is defined, $F_\tau(x) \equiv f_1(x) \equiv f_2(x)$.

Example 1. (McCarthy [1963b]). We consider again the function $\underline{\mathrm{reverse}}$, defined in Example 4 (Section 3). We wish to prove by recursion induction that

$$\underline{\mathrm{reverse}}(x*y) \equiv \underline{\mathrm{reverse}}(y) * \underline{\mathrm{reverse}}(x) \qquad \text{for all } x,y \in \Sigma^* .$$

For this purpose we choose the functional τ defined by

$$
\begin{aligned}
\tau[F](x,y) \equiv\ & \underline{\mathrm{if}}\ x = \Lambda\ \underline{\mathrm{then}}\ \underline{\mathrm{reverse}}(y) \\
& \underline{\mathrm{else}}\ F(t(x),y)*h(x) \ .
\end{aligned}
$$

Then using known properties of $*$ and $\underline{\mathrm{reverse}}$, we get that

(a) $\underline{\mathrm{reverse}}(x*y)$ is a fixedpoint of τ , since

$$
\begin{aligned}
\underline{\mathrm{reverse}}(x*y) &\equiv \underline{\mathrm{if}}\ x = \Lambda \\
&\quad \underline{\mathrm{then}}\ \underline{\mathrm{reverse}}(y) \\
&\quad \underline{\mathrm{else}}\ \underline{\mathrm{reverse}}((h(x)\cdot t(x))*y) \\
&\equiv \underline{\mathrm{if}}\ x = \Lambda \\
&\quad \underline{\mathrm{then}}\ \underline{\mathrm{reverse}}(y) \\
&\quad \underline{\mathrm{else}}\ \underline{\mathrm{reverse}}(h(x)\cdot(t(x)*y)) \\
&\equiv \underline{\mathrm{if}}\ x = \Lambda \\
&\quad \underline{\mathrm{then}}\ \underline{\mathrm{reverse}}(y) \\
&\quad \underline{\mathrm{else}}\ (\underline{\mathrm{reverse}}(t(x)*y))*h(x) \ .
\end{aligned}
$$

(b) $\underline{\mathrm{reverse}}(y) * \underline{\mathrm{reverse}}(x)$ is a fixedpoint of τ , since

$$
\begin{aligned}
&\underline{\mathrm{reverse}}(y) * \underline{\mathrm{reverse}}(x) \\
&\quad \equiv \underline{\mathrm{if}}\ x = \Lambda \\
&\quad\quad \underline{\mathrm{then}}\ \underline{\mathrm{reverse}}(y)*\Lambda \\
&\quad\quad \underline{\mathrm{else}}\ \underline{\mathrm{reverse}}(y) * \underline{\mathrm{reverse}}(x) \\
&\quad \equiv \underline{\mathrm{if}}\ x = \Lambda \\
&\quad\quad \underline{\mathrm{then}}\ \underline{\mathrm{reverse}}(y) \\
&\quad\quad \underline{\mathrm{else}}\ \underline{\mathrm{reverse}}(y) * (\underline{\mathrm{reverse}}(t(x))*h(x))
\end{aligned}
$$

$\equiv$ **if** $x = \Lambda$
then reverse(y)
else (reverse(y) $\cdot$ reverse($t(x)$))$\cdot h(x)$.

(c) $F_\tau(x,y)$ is defined for every $x,y \in \Sigma^*$, as can be shown by a straightforward induction on x. □

Example 2. We consider a system over the natural numbers in which the primitives are the predicate zero(x) (true only when x is 0), the predecessor function pred(x) (where pred(0) is 0), and the successor function succ(x) . In this system the program

```
add(x,y) <= if zero(x)
            then y
            else add(pred(x),succ(y))
```

defines the addition function. We wish to prove that succ(add(x,y)) $\equiv$ add(x,succ(y)) for all $x,y \in N$ by recursion induction.

We consider the functional τ defined as

```
τ[F](x,y) ≡ (if zero(x)
             then succ(y)
             else F(pred(x),succ(y))) .
```

The reader can easily verify that both functions succ(add(x,y)) and add(x,succ(y)) are fixedpoints of τ . Furthermore, an easy induction on x shows that $F_\tau(x,y)$ is defined for any x and y in N, which completes the proof. □

It is interesting to compare the preceding example with Example 4 of Section 2, where we considered the program

$G(x,y) \Leftarrow$ **if** $p(x)$ **then** y **else** $G(k(x),h(y))$

and proved that

$hG_\sigma(x,y) \equiv G_\sigma(x,h(y))$ for all x and y .

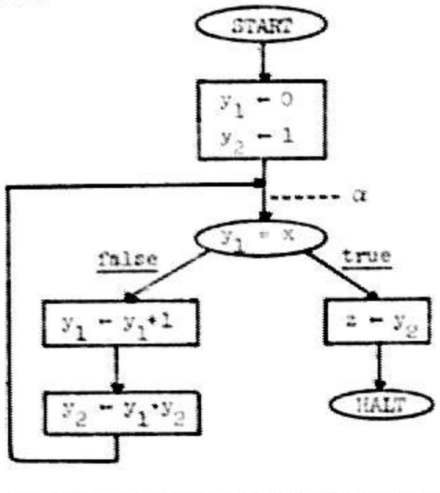
If we interpret $p(x)$ as zero(x) , $k(x)$ as pred(x) , and $h(y)$ as succ(y) , $G_\sigma(x,y)$ becomes add(x,y) , and the proof that $hG_\sigma(x,y) \equiv G_\sigma(x,h(y))$ is actually another proof that succ(add(x,y)) $\equiv$ add(x,succ(y)) . It is interesting to note that the algebraic manipulations used in both proofs are the same. However, the proof by recursion induction needed an argument of termination; if we consider the definition $G(x,y) \Leftarrow$ **if** $p(x)$ **then** y **else** $G(k(x),h(y))$ with no specific interpretation in mind for p , k and h , no argument for the termination of G_σ is possible (since, for instance, if $p(x)$ is always false, G_σ is never defined). A property like $hG_\sigma(x,y) \equiv G_\sigma(x,h(y))$ therefore cannot be proven by recursion induction (Morris [1971]).

An interesting special case of recursion induction, for which no proof of termination is needed, was described in Section 2. To prove the equivalence of two recursively defined functions, one can try to modify each definition until both definitions are the same, using transformations which leave the fixedpoint unchanged. This method was illustrated by Examples 6 and 7 of Section 2.

It is easy to show that every proof by recursion induction can be effectively translated to a proof by computational induction. Hint: assume $f_1 \equiv \tau[f_1]$ and $f_2 \equiv \tau[f_2]$ and prove $P(F_\tau)$, where $P(F)$ is $(F \sqsubseteq f_1) \wedge (F \sqsubseteq f_2)$.

Inductive Assertion Method

The most widely used method for proving properties of flowchart programs, called the inductive assertions method, was suggested by Floyd [1967]). We shall illustrate the method and its relation to computational induction on a simple flowchart program.



We wish to show that the above flowchart program over the natural numbers computes the factorial function, i.e., $z = x!$, whenever it terminates. To do this, we associate a predicate called an inductive assertion $Q(x,y_1,y_2)$ with the point labelled α in the program, and show that Q must be true for the values of the variables x,y_1,y_2 whenever execution of the program reaches point α . Thus, we must show (a) that the assertion holds when point α is first reached after starting execution (i.e., that $Q(x,0,1)$ holds) and (b) that it remains true when one goes around the loop from α to α (i.e., that $y_1 \neq x \wedge Q(x,y_1,y_2)$ implies $Q(x,y_1+1,(y_1+1)\cdot y_2)$. To prove the desired result we finally show (c) that $z = x!$ follows from the assertion $Q(x,y_1,y_2)$ when the program terminates (i.e., that $y_1 = x \wedge Q(x,y_1,y_2)$ implies $y_2 = x!$).

We take $Q_F(x,y_1,y_2)$ to be $y_2 = y_1!$. Then:

(a) $Q_F(x,0,1)$ is $1 = 0!$.

(b) We assume $y_1 \neq x$ and $Q_F(x,y_1,y_2)$, i.e., $y_2 = y_1!$. Then $Q_F(x,y_1+1,(y_1+1)\cdot y_2)$ is $(y_1+1)\cdot y_2 = (y_1+1)!$, i.e., $(y_1+1)\cdot y_1! = (y_1+1)!$.

(c) We assume $y_1 = x$ and $Q_F(x,y_1,y_2)$, i.e., $y_2 = y_1!$; then $y_2 = y_1! = x!$ as desired.

To show that the proof by inductive assertions corresponds to a computational induction proof in a natural way, we must first translate the flowchart program into a recursive program. Following the technique of McCarthy [1963a], we find that the output z of the above flowchart program is computed by the program

$$\begin{cases} \underline{\text{fact}}(x) \Leftarrow F(x,0,1) \\ F(x,y_1,y_2) \Leftarrow \underline{\text{if}}\ y_1 = x \\ \qquad\qquad \underline{\text{then}}\ y_2 \\ \qquad\qquad \underline{\text{else}}\ F(x,y_1+1,(y_1+1)\cdot y_2) \end{cases} .$$

We shall prove by computational induction that $F_\tau(x,0,1) \sqsubseteq x!$, i.e., that the value of $F_\tau(x,0,1)$ is $x!$ whenever $F_\tau(x,0,1)$ is defined. We take $P(F)$ to be the following admissible predicate:

$$(\forall x,y_1,y_2 \in N)\{\sim Q_F(x,y_1,y_2) \vee [F(x,y_1,y_2) \sqsubseteq x!]\}.$$

Obviously $P(\Omega)$ holds. To show that $P(F) \supset P(\tau[F])$ for every function F , we consider two cases: either $y_1 = x$, in which case the proof follows directly from (c) above, or $y_1 \neq x$, which follows directly from (b).

By computational induction we therefore have $P(F_\tau)$, i.e., $\sim Q_F(x,y_1,y_2) \vee [F_\tau(x,y_1,y_2) \sqsubseteq x!]$ for all $x,y_1,y_2 \in N$. But since $Q_F(x,0,1)$ is known from (a) to hold, we conclude that $F_\tau(x,0,1) \sqsubseteq x!$, as desired.

Manna and Pnueli [1970] generalized the inductive assertions method to apply to recursive programs. By their method it follows that the recursive program above computes the factorial function, when it terminates, if and only if there exists an inductive assertion $Q(x,y_1,y_2,z)$ such that

$$(\forall x,z \in N)[Q(x,0,1,z) \supset z = x!]$$
$$\wedge (\forall x,y_1,y_2,z \in N)\{\underline{\text{if}}\ y_1 = x\ \underline{\text{then}}\ Q(x,y_1,y_2,y_2)\ \underline{\text{else}}\ \forall t[Q(x,y_1+1,(y_1+1)\cdot y_2,t) \supset Q(x,y_1,y_2,t)]\} ,$$

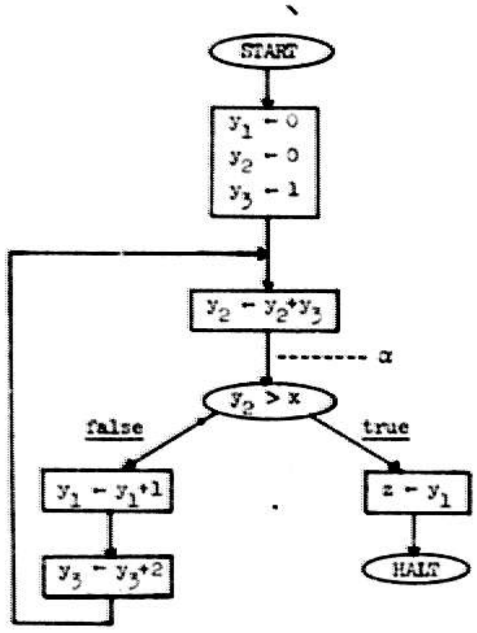
is true. (The formula implies that $Q(x,y_1,y_2,z)$ is true whenever $F_\tau(x,y_1,y_2) = z$.)

The reader may easily check that the above formula is satisfied by taking the predicate $Q_{MP}(x,y_1,y_2,z)$ to be $z\cdot y_1! = y_2\cdot x!$, which proves that $F_\tau(x,0,1) \sqsubseteq x!$ as desired. [*] The difference between the assertions $Q_F(x,y_1,y_2)$ and $Q_{MP}(x,y_1,y_2,z)$ should be noted: $Q_F(x,y_1,y_2)$ represents all possible values of the variables x,y_1,y_2 at arc α during execution, while $Q_{MP}(x,y_1,y_2,z)$ represents the relation between the final value of z and the initial value of x , y_1 and y_2 when computation starts at arc α .

As in the case of flowchart programs, the proof by the Manna-Pnueli method can easily be translated into a proof by computational induction, taking the predicate $P(F)$ to be $(\forall x,y_1,y_2 \in N)[F(x,y_1,y_2)\cdot y_1! \sqsubseteq y_2\cdot x!]$.

To further illustrate the use of the inductive assertion method, we briefly present a less familiar example. We shall show that for any natural number x the flowchart program given below, whenever it terminates, computes the smallest natural number greater than $\sqrt{x}$, i.e., that $z^2 \leq x < (z+1)^2$.

[*] Given an inductive assertion $Q_F(x,y_1,y_2)$ for the flowchart program, one can take the predicate $Q_{MP}(x,y_1,y_2,z)$ given by $Q_F(x,y_1,y_2) \supset (z = x!)$ and check that it also satisfies the above formula. However, the predicate obtained in this way is less natural than the predicate Q_{MP} given above.



We must show that there is an inductive assertion $Q(x,y_1,y_2,y_3)$ such that:

$$(\forall x \in N)Q(x,0,1,1)$$
$$\wedge\ (\forall x,y_1,y_2,y_3 \in N)[Q(x,y_1,y_2,y_3)$$
$$\supset \underline{\text{if}}\ y_2 > x$$
$$\underline{\text{then}}\ y_1^2 \le x < (y_1+1)^2$$
$$\underline{\text{else}}\ Q(x,y_1+1,y_2+y_3+2,y_3+2)]$$

is true. We take Q_P to be

$$(y_1^2 \le x) \wedge (y_2 = (y_1+1)^2) \wedge (y_3 = 2y_1+1) .$$

The reader may check that this assertion shows correctness as desired.

The corresponding recursive program is

$$\begin{cases} \underline{\text{sqrt}}(x) \Leftarrow F(x,0,1,1) \\ F(x,y_1,y_2,y_3) \Leftarrow \underline{\text{if}}\ y_2 > x \\ \qquad\qquad \underline{\text{then}}\ y_1 \\ \qquad\qquad \underline{\text{else}}\ F(x,y_1+1,y_2+y_3+2,y_3+2) \end{cases} ,$$

and the Manna-Pnueli formula is

$$(\forall x,z \in N)[Q(x,0,1,1,z) \supset z^2 \le x < z]$$
$$\wedge\ (\forall x,y_1,y_2,y_3 \in N)\{\underline{\text{if}}\ y_2 > x$$
$$\underline{\text{then}}\ Q(x,y_1,y_2,y_3,y_1)$$
$$\underline{\text{else}}\ \forall t[Q(x,y_1+1,y_2+y_3+2,y_3+2,t)$$
$$\supset Q(x,y_1,y_2,y_3,t)]\} .$$

This is satisfied by taking $Q_{MP}(x,y_1,y_2,y_3,z)$ to be

$$\underline{\text{if}}\ y_2 > x\ \underline{\text{then}}\ y_1 = z$$
$$\underline{\text{else}}\ (z-y_1)^2+(z-y_1)(y_3-1)+(y_2-y_3) \le x$$
$$< (z-y_1)^2+(z-y_1)(y_3+1)+y_2 .$$

Acknowledgments. We are indebted to Robin Milner for many stimulating discussions.

REFERENCES

BURSTALL [1969]. R. M. Burstall, "Proving Properties of Programs by Structural Induction". Computer Journal, Vol. 12, pp. 41-48 (1969).

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

deBAKKER and SCOTT [1969]. J. W. deBakker and Dana Scott, "A Theory of Programs", Unpublished memo., August 1969.

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

KLEENE [1950]. S. C. Kleene, Introduction to Metamathematics, D. Van Nostrand, Princeton, New Jersey (1950).

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

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

McCARTHY [1963b]. John McCarthy, "A Basis for a Mathematical Theory of Computation". In Computer Programming and Formal Systems, pp. 33-70 (eds. P. Braffort and D. Hirschberg). Amsterdam, North [illegible]nd (1963). Also in Proceedings of the Western Joint Computer Conference, pp. 225-238. New York, Spartan Books (1961).

McCARTHY and PAINTER [1967]. John McCarthy and J. A. Painter, "Correctness of a Compiler for Arithmetic Expressions". In Proceedings of a Symposium in Applied Mathematics, Vol. 19. Mathematical Aspects of Computer Science, pp. 33-41. (ed. J. T. Schwartz). Providence, Rhode Island, American Mathematical Society (1967).

MILNER [1972a]. Robin Milner, "Logic for Computable Functions - Description of a Machine Implementation", Computer Science report, Stanford University (to appear).

MILNER [1972b]. Robin Milner, "Implementation and Applications of Scott's Logic for Computable Functions", presented at the conference Proving Assertions About Programs, Las Cruces, New Mexico (January 1972).

MINSKY [1967]. Marvin Minsky, Computation - Finite and Infinite Machines, Prentice-Hall (1967).

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

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

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

PARK [1969]. David Park, "Fixpoint Induction and Proofs of Program Properties". In Machine Intelligence 5 (eds. B. Meltzer and D. Michie), Edinburgh University Press (1969).

SCOTT [1969]. Dana Scott, "A Type Theoretical Alternative to ISWIM, CUCH, OWHY". Unpublished notes, Oxford University (1969).

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

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