Programming Languages
T.A. Standish Editor

Inductive Methods for Proving Properties of Programs

Zohar Manna, Stephen Ness, Jean Vuillemin
Stanford University

There are two main purposes in this paper: first, clarification and extension of known results about computation of recursive programs, with emphasis on the difference between the theoretical and practical approaches; second, presentation and examination of various known methods for proving properties of recursive programs. Discussed in detail are two powerful inductive methods, computational induction and structural induction, including examples of their applications.

Key Words and Phrases: recursive programs, least fixedpoint, computational induction, structural induction

CR Categories: 4.2, 5.23, 5.24

The research reported here was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract SD-183. This is a modified version of a paper originally presented at the ACM Conference on Proving Assertions about Programs, Las Cruces, New Mexico, January 1972. Authors' addresses: Zohar Manna, Applied Mathematics Department, The Weizmann Institute of Science, Rehovot, Israel; Stephen Ness, Computer Science Department, Stanford University, Stanford, CA 94305: Jean Vuillemin, I.R.I.A., Domaine de Volceau, 78-Roquencourt, France.

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 introduce these methods to as wide a class of readers as possible, illustrating their power as practical techniques 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, 1970 [16]), 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 this function may differ from the function computed by some common computation rules. We briefly suggest "fixedpoint" computation rules which assure that the computed function is identical to the least fixedpoint. A brief informal exposition of the fixedpoint theory was given by Manna and Vuillemin, 1972 [8].

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 [3]). Examples are presented which introduce various applications of the method. We also discuss another computational induction method, truncation induction (Morris, 1971 [14]). A related method, called fixedpoint induction, is described in Park, 1969 [15].

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 [1], although it was often used previously, for example by McCarthy and Painter, 1967 [9], for proving the correctness of a compiler and by Floyd, 1967 [4] 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 of a suitable induction hypothesis leads to simple and clear inductive proofs.

Although it can be shown 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, 1972 [10, 11]). On the other hand, termination of programs is usually more convenient to show by structural induction.

We concentrate on these two methods because they form a natural basis for future automatic program verifiers. In particular, all other known verification tech-

niques can be justified rather directly by application of these methods. Grief, 1972 [5], discussed briefly the power of the different methods.

I. Recursive Programs

In this section, we introduce the fixedpoint 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. Partial predicates are of course a special case, since a partial predicate is a partial function mapping a domain D_1 into {**true,false**}.

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 $A_1 \times \cdots \times A_n$, we let D^+ be ${A_1}^+ \times \cdots \times {A_n}^+$. Any partial function f mapping $D_1 = A_1 \times \cdots \times A_n$ into D_2 may then be considered as a total function mapping D_1 into ${D_2}^+$: if f is undefined for $\langle d_1, \ldots, d_n \rangle \in D_1$, we let $f(d_1, \ldots, d_n)$ 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.

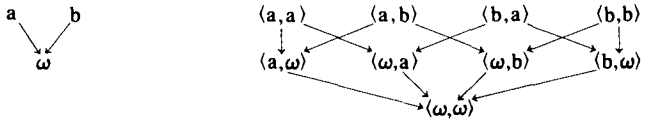
The Ordering $\subseteq$ on the Domain

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*[1] $\subseteq$ on every extended domain D^+. The partial ordering $\subseteq$ is intended to correspond to the notion "is less defined than or equal to," and accordingly we define it by letting $\omega \subseteq d$ and $d \subseteq d$ for all $d \in D^+$. 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 ${A_1}^+ \times \cdots \times {A_n}^+$, we define $\langle a_1, \ldots, a_n \rangle \subseteq \langle b_1, \ldots, b_n \rangle$ when $a_i \subseteq b_i$ for each i, $1 \leq i \leq n$.

Example 1. If $D = \{a, b\}$, then $D^+ = \{a, b, \omega\}$ and $(D \times D)^+ = \{\langle \omega, \omega \rangle, \langle \omega, a \rangle, \langle a, \omega \rangle, \ldots, \langle a, b \rangle, \langle b, a \rangle, \langle b, b \rangle\}$.

[1] A partial ordering is a binary relation which is reflexive ($(\forall a)[a \subseteq a]$), antisymmetric ($(\forall a,b)[a \subseteq b \wedge b \subseteq a \Rightarrow a$ 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.

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.) □



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

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.

$$x \subseteq y \text{ implies } f(x) \subseteq f(y) \quad \text{for all } x, y \in {D_1}^+.$$

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 $f(\omega)$ to be ω, with one exception: the constant function $f(x) = c$ for all $x \in D^+$ is always monotonic. In the following we denote such a constant function just by c. If f has many arguments, i.e. $D_1 = A_1 \times \cdots \times A_n$, it may have many different monotonic extensions. A particularly important extension of any function is called the *natural extension*, defined by letting $f(d_1, \ldots, d_n)$ 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 (i.e. ALGOL "call by value").

Example 2.

(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 {**true, 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.

(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 $x \subseteq y$ and $y \subseteq x$. The strong equality predicate is *not* a monotonic mapping from $D^+ \times D^+$ into {**true,false**}$^+$, since $\langle \omega, d \rangle \subseteq \langle d, d \rangle$ but $(\omega \equiv d) \not\subseteq (d \equiv d)$ (i.e. **false** $\not\subseteq$ **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

if true then a **else** $b = a$,
if false then a **else** $b = b$.

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

if true then a **else** $\omega = a$,
if false then ω **else** $b = b$,
if ω **then** a **else** $b = \omega$.

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

We shall assume all the functions of our examples to be naturally extended, except for the constant functions and the if-then-else function.

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.

Example 3.

(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. $\omega \subseteq 0$, but $f(\omega) \not\subseteq f(0)$). □

Finally, we discuss an important corollary which follows from properties of monotonic functions. Consider functions f_1 and f_2, given by

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

$f_2(x) \equiv$ **if** $p(x)$ **then** $g(h_1(x))$ **else** $g(h_2(x))$,

where p, g, h_1, and h_2 are monotonic. Then both f_1 and f_2 are monotonic, since each is defined by a composition of monotonic functions. There is an interesting relation between these two functions:

(i) $f_2(x) \subseteq f_1(x)$ *for any* x;
(ii) *if* $g(\omega) \equiv \omega$, *then* $f_2(x) \equiv f_1(x)$ *for any* x.

We shall use the second result often in later proofs. The above properties generalize to any n-ary ($n \geq 1$) monotonic function g. For example, if $g(\omega, y) \equiv \omega$ for all $\langle \omega, y \rangle \in D_1^+$, then

$g(\textbf{if } p(x) \textbf{ then } h_1(x) \textbf{ else } h_2(x),\ h_3(x)) \equiv$

if $p(x)$ **then** $g(h_1(x), h_3(x))$ **else** $g(h_2(x), h_3(x))$.

The Ordering $\subseteq$ on Functions

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

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

Example 4. Consider the sequence of monotonic functions $f_0, f_1, f_2, \ldots$ over the natural numbers defined by

$f_i(x) \equiv (\textbf{if } x \leq i \textbf{ then } x! \textbf{ else } \omega)$.

This sequence is a chain, as $f_i \subseteq f_{i+1}$ for every i; $\lim_i \{f_i\}$ is the factorial function. □

Continuous Functionals

We now consider a function τ mapping the set of functions $(D_1^+ \to D_2^+)$ 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 in the case of 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 consider only functionals satisfying a stronger property, called *continuity*. A functional τ is said to be continuous if for any chain of functions

$f_0 \subseteq f_1 \subseteq f_2 \subseteq \ldots$

we have

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

and

$\tau[\lim_i \{f_i\}] \equiv \lim_i \{\tau[f_i]\}$.

Every continuous functional is clearly monotonic.

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 f, the composition rules determine a monotonic function $\tau[f](x)$. It can easily be shown that *any functional defined by composition of monotonic functions and the function variable F is continuous.*

Example 5.

(a) The functional over the integers 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 f over the integers, $\tau[f]$ is another monotonic function over the integers:

if $f(x) \equiv \omega$, then $\tau[f](x) \equiv$ **if** $x = 0$ **then** 1 **else** ω;
if $f(x) \equiv x - 1$, then $\tau[f](x) \equiv$ **if** $x = 0$ **then** 1 **else** x.

(b) The functional over the natural numbers $(N^+ \to N^+)$ 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 $f_0 \subseteq f_1 \subseteq \cdots$ where $f_i(x) \equiv$ **if** $x < i$ **then** x **else** ω, $\tau[f_i] \equiv \Omega$ for any i so that $\lim_i \{\tau[f_i]\} \equiv \Omega$, whereas $\tau[\lim_i \{f_i\}]$ is the identity function. □

Fixedpoints

Let τ be a functional mapping $(D_1^+ \to D_2^+)$ into itself. We say that a function f is a *fixedpoint* of τ if $f \equiv \tau[f]$; i.e. τ maps the function f back into itself. We say that f is a *least fixedpoint* of τ if f is a fixedpoint of τ and $f \subseteq g$ for any other fixedpoint g of τ. An important fundamental result is that *any continuous* τ mapping $(D_1^+ \to D_2^+)$ into itself *has a unique least fixedpoint* f_τ in $(D_1^+ \to D_2^+)$. We can compute f_τ as the limit of the chain $\tau^0[\Omega] \subseteq \tau^1[\Omega] \subseteq \tau^2[\Omega] \subseteq \cdots$ (where $\tau^0[\Omega] \equiv \Omega$ and $\tau^{i+1}[\Omega] \equiv \tau[\tau^i[\Omega]]$), as follows from Kleene's first recursion theorem [6].

Example 6. 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 (for each $n \in N^+$)

$f_n(x) \equiv$ **if** $x = 0$ **then** 1 **else** 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

$$\tau[F](x_1, x_2) \equiv \textbf{if } x_1 = x_2 \textbf{ then } x_2 + 1 \textbf{ else } F(x_1, F(x_1 - 1, x_2 + 1))$$

has as fixedpoints the functions

$f(x_1, x_2) \equiv$ **if** $x_1 = x_2$ **then** $x_2 + 1$ **else** $x_1 + 1$,
$g(x_1, x_2) \equiv$ **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 [13]). Note that $f'(x_1, x_2) \equiv x_1 + 1$ is *not* a fixedpoint, since $\tau[f'](x_1, \omega) \equiv \omega$ while $f'(x_1, \omega) \equiv x_1 + 1$. □

We consider a functional τ over $(D_1^+ \to D_2^+)^n$, i.e. τ maps n-tuples of functions from $(D_1^+ \to D_2^+)$ into n-tuples of functions from $(D_1^+ \to D_2^+)$. Such a functional is given by coordinate functionals $\tau_1, \ldots, \tau_n$, so that $\tau[F_1, \ldots, F_n]$ is $\langle \tau_1[F_1, \ldots, F_n], \ldots, \tau_n[F_1, \ldots, F_n] \rangle$. 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 $f_\tau \equiv \langle f_{\tau_1}, \ldots, f_{\tau_n} \rangle$; that is

(a) $f_{\tau_i} \equiv \tau_i[f_{\tau_1}, \ldots, f_{\tau_n}]$ for all i, $1 \leq i \leq n$;
(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 $(1 \leq i \leq n)$, $f_{\tau_i} \subseteq g_i$ for all i $(1 \leq i \leq n)$.

Example 7. 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:

$\tau_1[F_1, F_2](x) \equiv$ **if** $x = 0$ **then** 1 **else** $F_1(x - 1) + F_2(x - 1)$
$\tau_2[F_1, F_2](x) \equiv$ **if** $x = 0$ **then** 0 **else** $F_2(x + 1)$.

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

$g_n(x) \equiv$ **if** $x = 0 \vee x = 1$ **then** 1 **else** $(x - 1) \cdot n + 1$
$h_n(x) \equiv$ **if** $x = 0$ **then** 0 **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:

$\langle$**if** $x = 0 \vee x = 1$ **then** 1 **else** ω, **if** $x = 0$ **then** 0 **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. 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 Minsky, 1967 [12]). A program in our language, called a *recursive definition* or *recursive program*, 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.[2] For example, the following is a program for

[2] We shall purposely be vague in our definitions in this section to avoid introducing the notions of schemata and interpretations. For a formal approach, see Manna and Pnueli, 1970 [7] or Cadiou, 1972 [2].

computing the factorial function:

$F(x) \Leftarrow$ **if** $x = 0$ **then** 1 **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 ((
  (FF (LAMBDA (X)(COND ((ZEROP X) 1)
                       (T (TIMES X (FF (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 *computation rules* 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.

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 x), 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 (see below) of $F(e)$ in t_i are replaced by $\tau[F](e)$ simultaneously, where e may be any subexpression; 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 $F(e)$ should be replaced by $\tau[F](e)$ 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 $f_C(d) \equiv t_n$; if the computation sequence for d is infinite, we say that $f_C(d) \equiv \omega$.

The following are examples of typical computation rules: (1) *full computation rule*: Replace all occurrences of F simultaneously. We denote the computed function by f_{FL}. (2) *leftmost-innermost ("call by value") 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 where a procedure evaluates *all* its arguments before execution. (3) *leftmost-outermost ("call by name") rule*: Replace only the leftmost-outermost occurrence of F. We denote the computed function by f_{LO}.

Example 8. We consider the recursive program for the "91-function" over the integers:

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

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

(a) Using the full rule:

t_0 is $F(99)$
  if $99 > 100$ **then** $99 - 10$
    else $F(F(99+11))$ [substitution]
t_1 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]
t_2 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]
t_3 is 91.

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

(b) Using the leftmost-innermost rule:

$\underline{F}(99) \rightarrow F(\underline{F}(110)) \rightarrow \underline{F}(100) \rightarrow F(\underline{F}(111)) \rightarrow \underline{F}(101) \rightarrow 91$.

Thus, $f_{\mathrm{LI}}(99) \equiv 91$.

(c) Using the leftmost-outermost rule:

$\underline{F}(99) \rightarrow \underline{F}(F(110))$
  $\rightarrow$ **if** $\underline{F}(110) > 100$ **then** $F(110) - 10$
    else $F(F(F(110)+11))$
  $\rightarrow \underline{F}(F(F(110)+11)) \rightarrow \cdots$
  $\rightarrow$ **if** $\underline{F}(110) + 11 > 100$ **then** $F(110) - 9$
    else $F(F(F(110)+22))-10$
  $\rightarrow \underline{F}(110) - 9 \rightarrow 91$.

Thus, $f_{\mathrm{LO}}(99) \equiv 91$. □

An important property of f_C should be mentioned at this point (Cadiou, 1972 [2]): 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

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

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 $\bar{x} = \langle x_1, \ldots, x_k \rangle$. 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 $\langle f_C^{(1)}, f_C^{(2)}, \ldots, f_C^{(n)} \rangle$, 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 of 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 $\alpha[F^1, \ldots, F^i, F^{i+1}, \ldots, F^k](d)$ denote any term α in the computation sequence for d under some computation rule C, 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, and 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, 1973 [17]). For example, since the full 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* (Morris, 1968 [13]).

Example 9. Consider the program over the integers

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

The least fixedpoint f_τ is

$$f_\tau(x,y) \equiv \textbf{if } x \geq 0 \textbf{ then } 1 \textbf{ else } \omega.$$

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

$$\underline{F}(1,0) \to F(0, \underline{F}(1,0)) \to F(0, F(0, \underline{F}(1,0))) \to \cdots$$

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

$$f_{LI}(x,y) \equiv \textbf{if } x = 0 \vee (x > 0 \wedge y > 0 \wedge (y \text{ divides } x)) \textbf{ then } 1 \textbf{ else } \omega,$$

which is *strictly less defined than* f_τ. □

In practice, the fixedpoint computation rules described so far (the full 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, the normal computation rule can be shown to perform the *minimum possible number of substitutions* of any rule within the class of computation rules which we described (Vuillemin, 1973 [17]).

The idea of the normal rule is to delay the evaluation of the arguments of procedures *as long as possible*, keeping arguments as formal expressions until they are needed. This rule is similar to ALGOL 60 "call by name," but with two important differences: (a) absolutely *no side effects are allowed, and* (b) any argument is evaluated *at most once*, namely the first time (if ever) it is needed.

Using the normal 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.

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.

Example 10. Consider the program over the natural numbers

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

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

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

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

$$\textbf{if } 1 = 0 \textbf{ then } \ldots \textbf{ else if } F(2, 0) = 0 \textbf{ then } \cdots \textbf{ else } \cdots,$$

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

Example 11. Consider the program over the integers

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

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

$$\underline{F}(2, 1) \to \underline{F}(1, F(1, 1)) \to \underline{F}(0, F(1 - F(1, 1), F(1, 1))) \to 1.$$

We again compare the substitutions required:

Normal rule: 3
Full 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 every P,[3] we shall only consider admissible predicates $P(F)$ which are simply conjunctions of inequalities $\alpha[F] \subseteq \beta[F]$, where α and β are two continuous functionals. In this case, the justification of the principle is easy; if $\alpha[\Omega] \subseteq \beta[\Omega]$ and $\forall F\{\alpha[F] \subseteq \beta[F] \Rightarrow \alpha[\tau[F]] \subseteq \beta[\tau[F]]\}$, then by a simple induction, $\alpha[\tau^i[\Omega]] \subseteq \beta[\tau^i[\Omega]]$ for every $i \geq 0$. Since by Kleene's first recursion theorem $\tau^i[\Omega] \subseteq f_\tau$ for all i, and β is monotonic, we have $\beta[\tau^i[\Omega]] \subseteq \beta[f_\tau]$, and therefore $\alpha[\tau^i[\Omega]] \subseteq \beta[f_\tau]$ for any i. By definition of the limit, this implies $\lim_i \{\alpha[\tau^i[\Omega]]\} \subseteq \beta[f_\tau]$, and since α is continuous, we have

$$\alpha[f_\tau] \equiv \alpha[\lim_i \{\tau^i[\Omega]\}] \equiv \lim_i \{\alpha[\tau^i[\Omega]]\} \subseteq \beta[f_\tau].$$

Thus,

$$\alpha[f_\tau] \subseteq \beta[f_\tau].$$

Example 12. We wish to show that the program

$F(x) \Leftarrow$ **if** $p(x)$ **then** x **else** $F(F(h(x)))$

defines an idempotent function, i.e. that $\forall x[f_\tau(f_\tau(x)) \equiv f_\tau(x)]$, or in short, $f_\tau f_\tau \equiv f_\tau$. 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. $(f_\tau F \subseteq F) \wedge (F \subseteq f_\tau F)$.

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

$f_\tau(\Omega(x))$
- $\equiv f_\tau(\omega)$ — definition of Ω
- $\equiv$ **if** $p(\omega)$ **then** ω **else** $f_\tau(f_\tau(h(\omega)))$ — definition of f_τ
- $\equiv$ **if** ω **then** ω **else** $f_\tau(f_\tau(h(\omega)))$ — since $p(\omega) \equiv \omega$
- $\equiv \omega$ — definition of **if** ω **then** a **else** b
- $\equiv \Omega(x)$ — definition of Ω.

(b) *Show* $\forall F\{P(F) \Rightarrow P(\tau[F])\}$, i.e.

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

$f_\tau(\tau[F](x))$
- $\equiv f_\tau$(**if** $p(x)$ **then** x **else** $F(F(h(x)))$) — definition of τ
- $\equiv$ **if** $p(x)$ **then** $f_\tau(x)$ **else** $f_\tau(F(F(h(x))))$ — distributing f_τ over conditional, since $f_\tau(\omega) \equiv \omega$
- $\equiv$ **if** $p(x)$ **then** x **else** $f_\tau(F(F(h(x))))$ — definition of f_τ
- $\equiv$ **if** $p(x)$ **then** x **else** $F(F(h(x)))$ — induction hypothesis
- $\equiv \tau[F](x)$ — definition of τ. □

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 $\sigma \in \Sigma$ and $w \in \Sigma^*$:

$h(\sigma \cdot w) = \sigma$, $t(\sigma \cdot w) = w$, $\sigma \cdot w \neq \Lambda$, and
$w \neq \Lambda \Rightarrow h(w) \cdot t(w) = w$.

This system is sometimes called "linear LISP." There is no difficulty involved in generalizing our proofs to real LISP programs.

Example 13. The program

$F(x, y) \Leftarrow$ **if** $x = \Lambda$ **then** y **else** $h(x) \cdot F(t(x), y)$

defines the **append** function $f_\tau(x, y)$, denoted $x*y$. We shall show that **append** is associative, i.e. that $x*(y*z) \equiv (x*y)*z$. For this purpose we prove $P(f_\tau)$, where $P(F)$ is $F(x, y)*z \equiv F(x, y*z)$.

(a) Show $P(\Omega)$; i.e. $\forall x, y, z[\Omega(x, y)*z \equiv \Omega(x, y*z)]$.

[3] Consider, for example, the recursive program over the natural numbers $F(x) \Leftarrow$ **if** $x = 0$ **then** 1 **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; but, since $f_\tau(x)$ is a total function (the factorial function), $P(f_\tau)$ does not hold.

$\Omega(x, y)*z$
$\equiv \omega*z$ definition of Ω
$\equiv$ **if** $\omega = \Lambda$ **then** z **else** $h(\omega)\cdot(t(\omega)*z)$ definition of **append**
$\equiv \omega$ since $\omega = \Lambda$ is ω
$\equiv \Omega(x, y*z)$ definition of Ω.

(b) Show $\forall F\{P(F) \Rightarrow P(\tau[F])\}$:
$\tau[F](x, y*z)$
$\equiv$ **if** $x = \Lambda$ **then** $y*z$ **else** $h(x)\cdot F(t(x), y*z)$ definition of τ
$\equiv$ **if** $x = \Lambda$ **then** $y*z$ **else** $h(x)\cdot(F(t(x), y)*z)$ induction hypothesis
$\equiv$ **if** $x = \Lambda$ **then** $y*z$ **else** $(h(x)\cdot F(t(x), y))*z$ definition of **append**
$\equiv$ (**if** $x = \Lambda$ **then** y **else** $h(x)\cdot F(t(x), y))*z$ distributing **append** over conditional, since $\omega*z \equiv \omega$
$\equiv \tau[F](x, y)*z$ definition of τ. □

Parallel Induction

We shall now present an application of computational induction to proving properties of two programs: $F \Leftarrow \tau[F]$ and $G \Leftarrow \sigma[G]$. To prove $P(f_\tau, g_\sigma)$ for an admissible predicate $P(F, G)$ (e.g. a conjunction of inequalities $\alpha[F, G] \subseteq \beta[F, G]$, where α and β are continuous functionals), use the following rule:

from $P(\Omega, \Omega)$ *and* $(\forall F, G)\{P(F, G) \Rightarrow P(\tau[F], \sigma[G])\}$, *infer* $P(f_\tau, g_\sigma)$.

Example 14. Consider the two programs (Morris, 1971 [14])

$F(x, y) \Leftarrow$ **if** $p(x)$ **then** y **else** $h(F(k(x), y))$
$G(x, y) \Leftarrow$ **if** $p(x)$ **then** y **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 $f_\tau(x, y) \equiv g_\sigma(x, y)$ for all x and y, we shall consider

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

We prove $P(f_\tau, g_\sigma)$, which implies $f_\tau \equiv g_\sigma$, as follows:
(a) Show $P(\Omega, \Omega)$.

$$\forall x,y\{[\Omega(x, y) \equiv \Omega(x, y)] \wedge [\Omega(x, h(y)) \equiv h(\Omega(x, y))]\},$$

clearly true, since $h(\omega) \equiv \omega$.
(b) Show $(\forall F, G)\{P(F, G) \Rightarrow P(\tau[F], \sigma[G])\}$.
(1)
$\tau[F](x, y)$
$\equiv$ **if** $p(x)$ **then** y **else** $h(F(k(x), y))$
$\equiv$ **if** $p(x)$ **then** y **else** $h(G(k(x), y))$ induction hypothesis
$\equiv$ **if** $p(x)$ **then** y **else** $G(k(x), h(y))$ induction hypothesis
$\equiv \sigma[G](x, y)$
(2)
$\sigma[G](x, h(y))$
$\equiv$ **if** $p(x)$ **then** $h(y)$ **else** $G(k(x), h(h(y)))$
$\equiv$ **if** $p(x)$ **then** $h(y)$ **else** $h(G(k(x), h(y)))$ induction hypothesis
$\equiv h($**if** $p(x)$ **then** y **else** $G(k(x), h(y)))$
$\equiv h(\sigma[G](x, y))$. □

Computational Induction for a System 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, use the following rule:

from $P(\Omega, \Omega)$ *and* $(\forall F_1, F_2)\{P(F_1, F_2) \Rightarrow P(\tau_1[F_1, F_2], \tau_2[F_1, F_2])\}$, *infer* $P(f_{\tau_1}, f_{\tau_2})$.

Example 15. Consider the program

$$\begin{cases} F_1(x) \Leftarrow \textbf{if } p(x) \textbf{ then } F_1(F_2(h(x))) \textbf{ else } F_2(g(x)) \\ F_2(x) \Leftarrow \textbf{if } q(x) \textbf{ then } f(F_2(F_1(x))) \textbf{ else } f(h(x)) \\ F_3(x) \Leftarrow \textbf{if } p(x) \textbf{ then } F_3(f(F_4(h(x)))) \textbf{ else } f(F_4(g(x))) \\ F_4(x) \Leftarrow \textbf{if } q(x) \textbf{ then } f(F_4(F_3(x))) \textbf{ else } h(x) \end{cases}$$

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 $(F_1 \equiv F_3) \wedge (F_2 \equiv fF_4)$. □

Transformations Which Leave f_τ Invariant

We can use computational induction to prove useful theorems about recursive programs. For example, if we modify a recursive program $F \Leftarrow \tau[F]$ 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) $(f_{\tau_1} \subseteq f_\tau)$ *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) $(f_\tau \subseteq f_{\tau_1})$ *and* $(f_\tau \subseteq f_{\tau_2})$. This can be shown by computational induction with $P(F, F_1, F_2)$ being the admissible predicate $(F \subseteq F_1) \wedge (F \subseteq F_2) \wedge (F \subseteq \tau[F]) \wedge (F \subseteq f_\tau)$.

Example 16. Consider the two recursive programs over the natural numbers

$F(x) \Leftarrow$ **if** $x > 10$ **then** $x - 10$ **else** $F(F(x + 13))$

and

$G(x) \Leftarrow$ **if** $x > 10$ **then** $x - 10$ **else** $G(x + 3)$.

We want to prove that $f_\tau \equiv g_\sigma$.

If we replace $F(x + 13)$ in τ by $\tau[F](x + 13)$, we get a new recursive program $F(x) \Leftarrow \tau'[F](x)$ where

$F(x) \Leftarrow$ **if** $x > 10$ **then** $x - 10$
else $F($**if** $x + 13 > 10$ **then** $x + 13 - 10$
else $F(F(x + 13 + 13)))$.

Since this is an f_τ invariant transformation $f_\tau \equiv f_{\tau'}$.

Since $x \geq 0$, we always have $x + 13 > 10$, therefore $f_{\tau'} \equiv g_\sigma$. The case $x \equiv w$ is immediate. Thus, $f_\tau \equiv g_\sigma$ as desired. □

Example 17. To prove $f_{\tau_1} \equiv f_{\tau_3}$ for the program

$$\begin{cases} F_1(x) \Leftarrow \textbf{if } p(x) \textbf{ then } F_3(F_2(F_2(f(x)))) \textbf{ else } g(x) \\ F_2(x) \Leftarrow \textbf{if } q(x) \textbf{ then } F_3(h(x)) \textbf{ else } k(x) \\ F_3(x) \Leftarrow \textbf{if } p(x) \textbf{ then } F_1(F_4(f(x))) \textbf{ else } g(x) \\ F_4(x) \Leftarrow \textbf{if } q(x) \textbf{ then } F_2(F_3(h(x))) \textbf{ else } F_2(k(x)), \end{cases}$$

we first change the definitions of F_1 and F_4 to

$$F_1(x) \Leftarrow \textbf{if } p(x) \textbf{ then } F_3(f_{\tau_2}(F_2(f(x)))) \textbf{ else } g(x)$$

and

$$F_4(x) \Leftarrow \textbf{if } q(x) \textbf{ then } f_{\tau_2}(F_3(h(x))) \textbf{ else } f_{\tau_2}(k(x)),$$

respectively, and then prove by computational induction that

$(f_{\tau_1} \equiv f_{\tau_3}) \wedge (f_{\tau_2} f_{\tau_2} \equiv f_{\tau_4})$, using
$P(F_1, F_2, F_3, F_4) : (F_1 \equiv F_3) \wedge (f_{\tau_2} F_2 \equiv F_4)$.

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

Truncation Induction

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

$f^\circ \equiv \Omega$ and $f^{i+1} \equiv \tau[f^i]$ for all $i \in N$ (natural numbers),

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

from $P(f^\circ)$ *and* $(\forall i \in N)[P(f^i) \Rightarrow P(f^{i+1})]$,
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, which Morris, 1971 [14] calls *truncation induction.*[4] 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 $(\forall i \in N)\{[(\forall j \in N \text{ such that } j < i) P(f^j)] \Rightarrow P(f^i)\}$,
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

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

we let f° 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 $\langle f_{\tau_1}, \ldots, f_{\tau_k} \rangle$; the truncation induction rule is then precisely the same as above.

Example 18. (Morris, 1971 [14]). We consider again (see Example 14) the two programs:

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

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 $f^\circ \equiv g^\circ$, $f^1 \equiv g^1$ and that $f^i \equiv g^i$ for all $i \geq 2$. We treat the cases for $i = 0$ and $i = 1$ separately, since to prove $f^i \equiv g^i$ we have to use the induction hypothesis for both $i - 1$ and $i - 2$. □

Termination

The examples introduced so far demonstrated that computational induction is convenient and natural for proving many kinds of properties of recursive programs. However, certain difficulties are involved in proving termination. To show that $g \subseteq f_\tau$ for some fixed function g (which is not Ω), we cannot simply choose $P(F)$ to be $g \subseteq F$, as then $P(\Omega)$ will always be false. In the next example we demonstrate that if the domain is specified by a "recursive predicate," it is possible to overcome this difficulty.

Example 19 (Milner). The function **reverse**$(x) \equiv f_\tau(x, \Lambda)$ where $F(x, y) \Leftarrow$ **if** $x = \Lambda$ **then** y **else** $F(t(x), h(x) \cdot y)$ 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 shall show that **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 function **word**$(x) \equiv g_\sigma(x)$, where

$G(x) \Leftarrow$ **if** $x = \Lambda$ **then true else** $G(t(x))$.

We let $P(F, G)$ be the admissible predicate

$(\forall x, y \in \Sigma^*)\{[G(x) \wedge \textbf{word}(y)] \subseteq \textbf{word}(F(x, y))\}$.

(a) Show $P(\Omega, \Omega)$.

$[\Omega(x) \wedge \textbf{word}(y)] \subseteq \textbf{word}(\Omega(x, y))$

holds since it reduces to $\omega \subseteq \omega$.

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

$\sigma[G](x) \wedge \textbf{word}(y)$
$\equiv$ (**if** $x = \Lambda$ **then true else** $G(t(x))) \wedge \textbf{word}(y)$
definition of σ
$\equiv$ **if** $x = \Lambda$ **then word**(y) **else** $G(t(x)) \wedge \textbf{word}(y)$
distributing $\wedge$ over conditional
$\equiv$ **if** $x = \Lambda$ **then word**(y) **else** $G(t(x)) \wedge \textbf{word}(h(x) \cdot y)$
definition of **word**
$\subseteq$ **if** $x = \Lambda$ **then word**(y) **else word**$(F(t(x), h(x) \cdot y))$
induction hypothesis
$\equiv$ **word**(**if** $x = \Lambda$ **then** y **else** $F(t(x), h(x) \cdot y)$)
distributing **word** over conditional
$\equiv$ **word**$(\tau[F](x, y))$ definition of τ.

[4] When applied to natural numbers, the two induction schemata are equivalent, i.e. we can validate either rule using the other. Thus in any system which includes a formalization of natural numbers, 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.

Therefore, by computational induction, we have:

$$[\mathbf{word}(x) \wedge \mathbf{word}(y)] \subseteq \mathbf{word}(f_\tau(x, y)) \quad \text{for all } x, y \in \Sigma^*,$$

which for $y = \Lambda$ gives $\mathbf{word}(x) \subseteq \mathbf{word}(\mathbf{reverse}(x))$. Since by definition we have that $\mathbf{word}(x)$ is **true** for all $x \in \Sigma^*$ and $\mathbf{word}(\omega) \equiv \omega$, this implies that $\mathbf{reverse}(x) \not\equiv \omega$ for any $x \in \Sigma^*$. □

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 $(\forall a \in N)\{[(\forall b \in N \text{ such that } b < a)P(b)] \Rightarrow P(a)]\}$,
infer $(\forall c \in N)P(c)$.

Since this induction rule is not valid for every ordered domain (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), we shall first characterize the ordered domains which are "good" for such induction. We then present a general rule called *structural induction*, for proving assertions over these domains; 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 programs.

Well-founded Sets

A partially ordered set $(S, \preccurlyeq)$ consists of a set S and a partial ordering $\preccurlyeq$ on S. Note that the ordering need not be total; i.e. it is possible that for some $a, b \in S$, neither $a \preccurlyeq b$ nor $b \preccurlyeq a$ holds. A partially ordered set $(S, \preccurlyeq)$ which contains no infinite decreasing sequence $a_0 \succ a_1 \succ a_2 \succ \ldots$ of elements of S is called a *well-founded* set.

Example 20.

(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 $\frac{1}{2} > \frac{1}{3} > \frac{1}{4} > \cdots$).

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

(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, with the substring relation ($w_1 \preccurlyeq w_2$ 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.[5] Let $(S, \preccurlyeq)$ be a well-founded set, and let P be a total predicate over S. 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

from $(\forall a \in S)\{[(\forall b \in S \text{ such that } b \prec a)P(b)] \Rightarrow P(a)\}$,
infer $(\forall c \in S)P(c)$.

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 is false. Consider the set A of elements $a \in S$ such that $P(a)$ is false. Let us assume that A is non-empty. Then there is a least element a_0 such that $a \not\preccurlyeq a_0$ for any $a \in A$; otherwise there would be an infinite descending sequence in S. Then, for any element b such that $b \prec a_0$, $P(b)$ is true; that is, $(\forall b \in S$ such that $b \prec a_0)\ P(b)$ must hold. But the assumption then implies that $P(a_0)$ is true, in contradiction with the fact that $a_0 \in A$. Therefore A must be empty, i.e. $P(c)$ is true for all elements $c \in S$.

Note that if there is no b in S such that $b \prec a$, the statement $(\forall b \in S$ such that $b \prec a)P(b)$ holds vacuously. For such a's we must therefore show $P(a)$ unconditionally to establish the hypothesis needed for the structural induction.

Applications

We now give several examples using structural induction to prove properties of recursive programs. Such proofs require suitable choices of both the partial ordering $\preccurlyeq$ 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 property.

Example 21 (Cadiou). Factorial functions. Consider the programs over the natural numbers

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

and

$$G(x, y) \Leftarrow \mathbf{if}\ x = y\ \mathbf{then}\ 1\ \mathbf{else}\ G(x, y + 1) \cdot (y + 1).$$

$f_\tau(x)$ and $g_\sigma(x, 0)$ compute $x! = 1 \cdot 2 \cdot \cdots \cdot x$ for every $x \in N$ in two different ways: $g_\sigma(x, 0)$ by "going up" from 0 to x and $f_\tau(x)$ by "going down" from x to 0. We wish to show that $g_\sigma(x, 0) \equiv f_\tau(x)$ for any $x \in N$ by using the predicate

$$P(x) : (\forall y \in N)[g_\sigma(x + y, y) \cdot f_\tau(y) \equiv f_\tau(x + y)]$$

and the usual ordering on natural members.

(a) If $x = 0$, $P(0)$ is $\forall y[g_\sigma(y, y) \cdot f_\tau(y) \equiv f_\tau(y)]$, which is clearly true by definition of g_σ.

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

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

For any $y \in N$,

$g_\sigma(x + y, y) \cdot f_\tau(y)$
$\equiv g_\sigma(x + y, y + 1) \cdot (y + 1) \cdot f_\tau(y)$ definition of g_σ (since $x > 0$)
$\equiv g_\sigma(x + y, y + 1) \cdot f_\tau(y + 1)$ definition of f_τ (since $y + 1 > 0$)
$\equiv g_\sigma((x - 1) + (y + 1), y + 1) \cdot f_\tau(y + 1)$ since $x > 0$
$\equiv f_\tau(x - 1 + y + 1)$ induction hypothesis (since $x - 1 < x$)
$\equiv f_\tau(x + y)$.

By complete induction, then, $P(x)$ holds for all $x \in N$. In particular, for $y = 0$, $g_\sigma(x, 0) \cdot f_\tau(0) \equiv f_\tau(x)$. Since $f_\tau(0) \equiv 1$, we have $g_\sigma(x, 0) \equiv f_\tau(x)$ as desired. □

In the preceding example 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 22 (Burstall). "McCarthy's 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 $(I, \preccurlyeq)$, where I is the integers and $\preccurlyeq$ is defined as follows:

$x \prec y$ iff $y < x \leq 101$
(where $<$ is the usual ordering on the integers); thus $101 \prec 100 \prec 99 \prec \ldots$, but for example, $102 \not\prec 101$. One can easily check that $(I, \preccurlyeq)$ 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 $(\Sigma^*, \preccurlyeq)$ of finite strings over 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:

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

Example 23. Consider again the program of Example 19 defining the function **reverse**$(x) \equiv f_\tau(x, \Lambda)$, where

$F(x, y) \Leftarrow$ **if** $x = \Lambda$ **then** y **else** $F(t(x), h(x) \cdot y)$.

We wish to prove 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.

To prove **reverse**(**reverse**(x)) $\equiv x$, we let

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

(a) If $x = \Lambda$, then for any y we have

reverse$(f_\tau(\Lambda, y)) \equiv$ **reverse**$(y) \equiv f_\tau(y, \Lambda)$.

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

reverse$(f_\tau(x, y))$
$\equiv$ **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 $x \succ t(x)$)
$\equiv f_\tau(y, h(x) \cdot t(x))$ definition of f_τ (since $h(x) \cdot y \neq \Lambda$)
$\equiv f_\tau(y, x)$.

Therefore **reverse**$(f_\tau(x, y)) \equiv f_\tau(y, x)$ for all $x, y \in \Sigma^*$; in particular, for $y = \Lambda$, **reverse**(**reverse**(x)) $\equiv$ **reverse**$(f_\tau(x, \Lambda)) \equiv f_\tau(\Lambda, x) \equiv x$, as desired. □

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

(i) **reverse**$(w * a) \equiv a \cdot$ **reverse**(w),
(ii) **reverse**$(a \cdot w) \equiv$ **reverse**$(w) * a$, and
(iii) **reverse**$(a \cdot (w * b)) \equiv b \cdot$ (**reverse**$(w) * a$),

where $*$ is the **append** function defined in Example 13 (Section II).

Example 24. Another reverse function. We wish to show that the program (due to Ashcroft)

$F(x) \Leftarrow$ **if** $x = \Lambda$ **then** Λ
else if $t(x) = \Lambda$ **then** x
else $h(F(t(x))) \cdot F(h(x) \cdot F(t(F(t(x)))))$

also defines a reversing function on Σ^*, i.e. that $f_\tau(x) \equiv$ **reverse**(x) for all $x \in \Sigma^*$. Note that this definition does not use any auxiliary functions.

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 $x = a \cdot (w * b)$ for some $a \in \Sigma$, $w \in \Sigma^*$, and $b \in \Sigma$. The lemma is easy to prove by a straightforward structural induction.

We now prove that $f_\tau \equiv$ **reverse** by structural induction on $(\Sigma^*, \preccurlyeq)$, where $\preccurlyeq$ is the following partial ordering:

$x \preccurlyeq y$ iff x is a substring of y or x is a proper substring of **reverse**(y). One can check that $(\Sigma^*, \preccurlyeq)$ is well founded.

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

(1) $x = \Lambda : f_\tau(x) \equiv \Lambda \equiv$ **reverse**(x).
(2) $x \in \Sigma : f_\tau(x) \equiv x \equiv$ **reverse**(x).
(3) $x = a \cdot (w * b)$ for some $a \in \Sigma$, $w \in \Sigma^*$, $b \in \Sigma$:

$f_\tau(x)$

$\equiv h(f_\tau(t(x)))\cdot f_\tau(h(x)\cdot f_\tau(t(f_\tau(t(x)))))$

definition of f_τ

$\equiv h(f_\tau(w*b))\cdot f_\tau(a\cdot f_\tau(t(f_\tau(w*b))))$

since $h(x) = a$, $t(x) = w*b$

$\equiv h(\mathbf{reverse}(w*b))\cdot f_\tau(a\cdot f_\tau(t(\mathbf{reverse}(w*b))))$

induction hypothesis (since $w*b \prec x$)

$\equiv h(b\cdot\mathbf{reverse}(w))\cdot f_\tau(a\cdot f_\tau(t(b\cdot\mathbf{reverse}(w))))$

property (i) of **reverse**

$\equiv b\cdot f_\tau(a\cdot f_\tau(\mathbf{reverse}(w)))$ properties of h and t

$\equiv b\cdot f_\tau(a\cdot\mathbf{reverse}(\mathbf{reverse}(w)))$

induction hypothesis (since $\mathbf{reverse}(w) \prec x$)[6]

$\equiv b\cdot f_\tau(a\cdot w)$ property of **reverse** proven in previous example

$\equiv b\cdot\mathbf{reverse}(a\cdot w)$ induction hypothesis (since $a\cdot w \prec x$)

$\equiv b\cdot(\mathbf{reverse}(w)*a)$ property (ii) of **reverse**

$\equiv \mathbf{reverse}(x)$ property (iii) of **reverse**.

We conclude that $f_\tau(x) \equiv \mathbf{reverse}(x)$ for all $x \in \Sigma^*$, as desired. □

Given a partially ordered set $(S, \preccurlyeq)$, we define the *lexicographic ordering* $\preccurlyeq_n$ on n-tuples of elements of S (i.e., on elements of S^n) by letting $\langle a_1, \ldots, a_n\rangle \prec_n \langle b_1, \ldots, b_n\rangle$ iff $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}$ and $a_i \prec b_i$ for some i, $1 \leq i \leq n$. It is easy to show that if $(S, \preccurlyeq)$ is well founded, so is $(S^n, \preccurlyeq_n)$. In the following example, we use the well-founded set $(N^2, \leq_2)$, i.e. the lexicographic ordering on pairs of natural numbers. Note that under this ordering $\langle n_1, n_2\rangle <_2 \langle m_1, m_2\rangle$ iff $n_1 < m_1$ or $n_1 = m_1$ and $n_2 < m_2$; for example, $\langle 1,100\rangle <_2 \langle 2, 0\rangle$.

Example 25. Consider again the recursive program over the natural numbers of Example 10 for computing Ackermann's function

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

We wish to show that $f_\tau(x, y)$ is defined, i.e. $f_\tau(x, y) \not\equiv \omega$, for any $x, y \in N$. We shall use the structural induction rule applied to the well-founded set $(N^2, \leq_2)$. Assuming that $f_\tau(x', y')$ is defined for any $\langle x', y'\rangle$ such that $\langle x', y'\rangle <_2 \langle x, y\rangle$, we show that $f_\tau(x, y)$ must also be defined.

(a) if $x = 0$, obviously $f_\tau(x, y)$ is defined.

(b) if $x \neq 0$ and $y = 0$, we note that $\langle x - 1, 1\rangle <_2 \langle x, y\rangle$, so by the induction hypothesis $f_\tau(x - 1, 1)$ is defined. Thus $f_\tau(x, y)$ is also defined.

(c) Finally, if $x \neq 0$ and $y \neq 0$, $\langle x, y - 1\rangle <_2 \langle x, y\rangle$, and therefore $f_\tau(x, y - 1)$ is defined by the induction hypothesis. Now, regardless of the value of $f_\tau(x, y - 1)$, we have $\langle x - 1, f_\tau(x, y - 1)\rangle <_2 \langle x, y\rangle$ and the desired result follows by another application of the induction hypothesis. □

[6] Note that $\mathbf{reverse}(w) \prec x$ because $\mathbf{reverse}(w)$ is a proper substring of $\mathbf{reverse}(x)$, as may be seen from property (iii) of **reverse**.

Acknowledgments. We are indebted to Robin Milner for many stimulating discussions and James Morris for suggesting many improvements to this paper.

Received November 1971: revised April 1972

References

1. Burstall, R.M. Proving properties of programs by structural induction. *Computer J.* 12, 1 (Feb. 1969), 41–48.
2. Cadiou, J.M. Recursive definitions of partial functions and their computations. Ph.D. Th. Computer Science Dept., Stanford U., 1972.
3. deBakker, J.W., and Scott, D. A theory of programs (unpublished memo., Aug. 1969).
4. Floyd, R.W. Assigning meanings to programs. Proc. Sympo. in Appl. Math. Vol. 19. *Mathematical Aspects of Computer Science*, (J.T. Schwartz, Ed.) AMS, Providence, R.I., 1967, pp. 19–32.
5. Grief, I.G. Induction in proofs about programs. Master's Th., M.I.T., 1972.
6. Kleene, S.C. *Introduction to Metamathematics.* D. Van Nostrand, Princeton, N.J., 1950.
7. Manna, Zohar, and Pnueli, Amir. Formalization of properties of functional programs. *J. ACM, 17*, 3 (July 1970), 555–569.
8. Manna, Zohar, and Vuillemin, John. Fixpoint approach to the theory of computation. *Comm. ACM 15*, 7 (July 1972), pp. 528–536.
9. McCarthy, John, and Painter, J.A. Correctness of a compiler for arithmetic expressions. Proc. Sympo. in Appl. Math. Vol. 19. *Mathematical Aspects of Computer Science*, (J.T. Schwartz, Ed.) AMS, Providence, R.I., 1967, pp. 33–41.
10. Milner, Robin. Logic for computable functions—description of a machine implementation. Comput. Sci. Rept., Stanford U., 1972.
11. Milner, Robin. Implementation and applications of Scott's logic for computable functions. Presented at Proc. ACM Conf. on Proving Assertions About Programs, Las Cruces, N.M., Jan. 1972, pp. 1–6.
12. Minsky, Marvin. *Computation–Finite and Infinite Machines.* Prentice-Hall, Englewood-Cliffs, N.J., 1967.
13. Morris, James H. Lambda-calculus models of programming languages. Ph.D. Th., Proj. MAC, MIT, MAC-TR-57, Dec. 1968.
14. Morris, James H. Another recursion induction principle. *Comm. ACM 14*, 5 (May 1971), 351–354.
15. Park, David. Fixpoint induction and proofs of program properties. In *Machine Intelligence 5*, (B. Meltzer and D. Michie, Eds.) Edinburgh U. Press, Edinburgh, 1969, pp. 59–78.
16. Scott, Dana. Outline of a mathematical theory of computation. Proc. Fourth Ann. Princeton Conf. on Information Sciences and Systems, Princeton U., 1970, pp. 169–176.
17. Vuillemin, Jean. Proof techniques for recursive programs. Ph.D. Th., Comput. Sci. Dept., Stanford U., 1973 (to appear).