INDUCTION IN PROOFS ABOUT PROGRAMS

Irene Gloria Greif

February 1972

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

INDUCTION IN PROOFS ABOUT PROGRAMS*

ABSTRACT

Four methods for proving equivalence of programs by induction are described and compared. They are recursion induction, structural induction, μ-rule induction, and truncation induction. McCarthy's formalism for conditional expressions as function definitions is used and reinterpreted in view of Park's work on results in lattice theory as related to proofs about programs. The possible application of this work to automatic program verification is commented upon.

---

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degree of Master of Science, December 1971.

Table of Contents

	Page
Abstract	2
Acknowledgements	3
Table of Contents	4
Introduction	5
Chapter 1	6
1.1 Notation	6
1.2 Recursion Induction as Defined by McCarthy	10
1.3 Structural Induction as Defined by Burstall	12
1.4 Recursive Definitions as Mappings Between Lattices	15
1.5 The μ-rule	19
1.6 Truncation Induction as Defined by Morris	21
Chapter 2. The Comparisons	23
2.1 Objectives	23
2.2 Structural Induction and Truncation Induction	24
2.3 The μ-rule and Truncation Induction	32
2.4 Recursion Induction and Truncation Induction	40
2.5 Recursion Induction and Structural Induction	47
Chapter 3. Some Comments on Automatic Program Verification	50
Chapter 4. Conclusion	56
Appendix	58
References	60

Introduction

Work on compilers, code optimizers, language definitions, and debugging systems often requires some notion of correctness of a program or of the equivalence of two programs. To determine the correctness of a compiler it is necessary to prove that the compilation process preserves meaning of programs, so that executions of a source language program and of its corresponding object language program produce identical results for all input. In optimizing a program it must be true that the transformations made preserve the program meaning in the same sense. Since there can be no general procedure for testing equivalence of programs [9], we must try to find partial decision procedures for this problem. For example, it can be shown that two programs are equivalent if a corresponding statement of first order logic is a theorem [4]. Or, in examining programs which operate on structured data, equivalence might be proved by an inductive argument based on the degree of complexity of the structured data [6].

Rules of this last type, i.e. rules with some induction step in them, will be the subject of this thesis. Several such methods are now known. Two of them, recursion induction and structural induction have been defined rather informally. Two others, application of a rule of inference, the μ-rule, and truncation induction, are somewhat more formally defined. All of these methods can be understood in terms of certain results of lattice theory as described by Park [8]. By using this information and by studying proofs by the various techniques we will explore and clarify the relationships between these induction rules.

Chapter 1.

1.1. Notation

Our language for describing functions will be that of McCarthy [5]. The set of forms written in terms of the set of primitive functions and predicates, $\mathcal{F}$, will be denoted by $C(\mathcal{F})$. The forms in $C(\mathcal{F})$ are:

if F is in $\mathcal{F}$, then F is in $C(\mathcal{F})$

if f and g are in $C(\mathcal{F})$, then $f \circ g$ is in $C(\mathcal{F})$ [composition]

if p and q are in $C(\mathcal{F})$ then $p \wedge q$, $p \vee q$, and $\neg p$ are in $C(\mathcal{F})$

if p,f and g are in $C(\mathcal{F})$, (where p is a predicate), then the form (if p then f else g) is in $C(\mathcal{F})$

In general the function and predicate symbols will be uninterpreted, particularly in definitions, but there will be examples of proofs for both interpreted and uninterpreted forms. Also, it is generally assumed that the constant functions $T,F \in C(\mathcal{F})$ and that the primitive predicates and functions are total.

A function definition is a function name, say f, followed by $\equiv$ and then some form $\mathcal{C}$ in the class $C(\mathcal{F})$. The function name itself may appear on the right hand side, in which case the definition is recursive. $\mathcal{C}(f)$ represents a conditional form which contains the function name f. The function name, f, can be considered to be a variable for which a substitution of another function name can be made. In general, script letters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ... will be used for forms. Sometimes it is convenient to define several functions at the same time, with references to each other, and occasionally a function may be defined to be the restriction of another function to a particular domain. The following are examples of function definitions:

(1) $f(x) \equiv$ if Px then Fx else FFx

(2) $f(x) \equiv$ if Px then f(Fx) else FFx

(3) $f(x) \equiv$ if Px then g(Fx) else f(FFx)

$g(x) \equiv$ if Px then f(x) else x

(4) $f(x) \equiv$ if Px then if PFx then fFx else Fx
else f(FFx)

(5) $f(x) \equiv g(x,1,1)$

where $g(x,y,z) \equiv$ if $x = y$ then z else $f(x, y+1, z \cdot y)$

McCarthy [5] shows that the class of functions definable from forms in C(succ, eq), where succ is the successor function and eq is the predicate for equality, is the class of all computable functions on the natural numbers, proving that this is a sufficiently powerful language to be of interest to us.

There are axioms for manipulating conditional expressions which constitute a complete system for checking equality of any pair of uninterpreted conditional forms. Two forms are equal if under any interpretation of function letters (except where a restriction is indicated) normal evaluation of either form results in the same value, where a conditional form, if p then a else b, has the value of a if p is T and of b if p is F. These axioms (again from McCarthy) are:

(A1) if p then a else a
is equal to a (if p is total)

(A2) if T then a else b
is equal to a

(A3) if F then a else b
is equal to b

(A4) If p then T else F
is equal to p

(A5) if p then (if p then a else b) else c
is equal to
if p then a else c

(A6) if p then a else (if p then b else c)
is equal to
if p then a else c

(A7) if (if p then q else r) then a else b
is equal to
if p then (if q then a else b)
else (if r then a else b)

(A8) if p then (if q then a else b) else (if q then c else a)
is equal to
if q then (if p then a else c) else (if p then b else d)

It is thus possible to test any two conditional expressions for equality. However, when conditional expressions are used as right hand sides of function definitions, the test for equality of any two function definitions is complicated by the possibility of recursions. It is obvious that if $f \equiv \mathcal{C}(f)$ and $g \equiv \mathcal{D}(g)$ are defined so that the right hand sides of their definitions can be shown to be identical forms by the axioms i.e. that $\mathcal{C}(x) = \mathcal{D}(x)$ then f and g are identical functions. However, there will be functions where the right hand sides appear to differ, but the recursion causes the results of computations to always be identical. For example, consider the following two definitions (from Morris [7]).

(1) $f(x,y) \equiv$ if Px then y else Gf(Fx,y)

$g(x,y) \equiv$ if Px then y else g(Fx,Gy)

Comparison of these forms by the axioms does not tell us enough about the functions f and g. To see how to compare these two function definitions we must understand how to interpret these recursive definitions.

A recursive definition can be considered to be an algorithm for computing a function. So the definition

$f(x) \equiv$ if Px then fFx else x

means "to compute f(x) test Px, if Px false the answer is x, if Px true then compute fFx." To compute fFx, we follow the same algorithm. Obviously, for some legal function definitions, and some arguments, execution by the algorithm can continue forever. Thus the functions defined by these conditional expressions may well be partial functions. With this interpretation

of the recursive definition, the following interpretation of the equivalence of two such definitions emerges. For any argument, following either of the two algorithms must always lead to the same answer. For the example (1) above it is clear that for any argument x, following the algorithm for f or for g leads to the same result. In one case, G is applied to y at each step in the computation and in the other case the appropriate number of applications of G are recorded to be carried out at the end of execution. It becomes clear that some way of carrying out induction over the number of appeals to the definition is needed i.e. induction on the level of recursion. For this particular example, we would like to be able to show that if $\neg Px$, then $Gf(Fx,y) = f(Fx,Gy)$. In other words, at the next level of recursion, we can either apply G or save it for later. If this is true then

$$f(x,y) \equiv \text{if } Px \text{ then } y \text{ else } Gf(Fx,y)$$

$$= \text{if } Px \text{ then } y \text{ else } f(Fx,Gy)$$

and we see that a computation of f(x) can actually be done by following the algorithm for g. Two definitions specifying functions which can be computed by identical algorithms clearly specify the same functions.

We now begin a review of the existing techniques for proving equivalence of recursive functions.

1.2 Recursion Induction as defined by McCarthy

In McCarthy's formalism the right hand side of a definition is a form which contains certain variable function names. For example, if $f(x) \equiv$ if Px then fFx else x, then $f \equiv \mathcal{C}(f)$ where $\mathcal{C}$ represents the conditional form. Then $g \equiv \mathcal{C}(g)$ defines exactly the same function. This is true because for all x, g(x) is evaluated by following the same algorithm as is followed in evaluation of f(x). A proof of the equivalence of two functions then would go as follows. Given two function definitions

$$f(x) \equiv \mathcal{C}(f)(x) \qquad \text{defines } f$$

$$g(x) \equiv \mathcal{D}(g)(x) \qquad \text{defines } g$$

find a third function $h(x) \equiv \mathcal{H}(h)(x)$ and show that

$$f(x) = \mathcal{H}(f)(x)$$

$$g(x) = \mathcal{H}(g)(x)$$

Then we can say that f = g <u>on the domain of h</u>. If h(x) converges, then f(x) and g(x) are defined and equal to h(x).

There are a few difficulties in using this method. Very little is said about how to transform the defining conditionals into the required third form, or for that matter, about how to come up with the third form in the first place. We can use the axioms about conditionals, and results of any other proofs by recursion induction or any other valid proof techniques (proof by analysis of cases, etc.). Often several subproofs and the generation of many intermediary "auxiliary functions" are required. No insights are offered as to how proof by recursion induction could be carried out mechanically. The generation of the third form is more often related to knowledge of the algorithm and alternatives to it, then to the forms of the conditional expressions themselves.

Also, the fact that equivalence can be proved only over the domain of a third function leaves the problem of analyzing the domains of the functions involved, a question [illegible] in general as was the original question of equivalence.

Example of Proof by Recursion Induction

Arithmetic example from McCarthy's paper

The primitive functions are succ and pred (successor and predecessor)

1. $g(m,n) \equiv succ(f(m,n))$

2. $h(m,n) \equiv f(succ(m),n)$

where $f(m,n) \equiv$ if $n = 0$ then m else $f(succ(m),pred(n))$

[$f(m,n) = m + n$; $g(m,n)$ is the succ of $(m + n)$ and $h(m)$ is the (succ of m) + n]

We will show $g = h$ on the domain of i where

$$i(m,n) \equiv \text{if } n = 0 \text{ then } succ(m) \text{ else } i(succ(m), pred(n))$$
$$\equiv \mathcal{J}(i)(m,n).$$

1. $g(m,n) \equiv succ(f(m,n))$

$= succ(\text{if } n = 0 \text{ then } m \text{ else } f(succ(m), pred(n)))$

$= \text{if } n = 0 \text{ then } succ(m) \text{ else } succ(f(succ(m), pred(n)))$

$= \text{if } n = 0 \text{ then } succ(m) \text{ else } g(succ(m), pred(n))$

$= \mathcal{J}(g)(m,n)$

2. $h(m,n) \equiv f(succ(m),n)$

$= \text{if } n = 0 \text{ then } succ(m) \text{ else } f(succ(succ(m)),pred(n))$

$= \text{if } n = 0 \text{ then } succ(m) \text{ else } h(succ(m), pred(n))$

$= \mathcal{J}(h)(m,n)$

$\therefore h = g$ on the domain of i.

1.3 Structural Induction as Defined by Burstall

This method for proving facts about programs applies to recursive programs which operate on a recursively defined domain of structured data. The data structure must be defined in such a way that a natural ordering exists on the class as a whole. This ordering must be a partial ordering which satisfies the minimum condition, i.e. every non-empty subset has a minimal element, where x is a minimal element of S if there is no $s < x$ in S. (This means that there can be no infinite descending chains.) The function definitions must be made in such a way that each level of recursion involves operations on structures which are less complex, or lower in this ordering, than the structures of the next higher level of recursion. By the minimum condition we know then that any such computation beginning on data in the class, must terminate. Proofs then proceed by course of values induction on the complexity of the structure, which means that to prove that a structure of arbitrary complexity has property P one must prove that the minimally complex elements have P and that if all structures of complexity $i < j$ have P then a structure of complexity j has P.[1] For example, for a program, f, which operates on list structures, to prove property P (where P might be that f(x) is equal to g(x) for all x) it must be shown that f(x) has P assuming f(x') has P for all x' sublists of x and that the null list has P. As long as a partial ordering can be defined on the domain, structural induction can be used even if it is not otherwise natural to consider the data to be structured in the sense that lists are structured.

You will notice that it is only meaningful to talk about applying structural induction, as described, to interpreted schema. An interpretation is needed to define either the domain and its ordering, or the structure with primitive functions being interpreted as the appropriate

---

[1] The general schema of course-of-values induction on integers is:

$$\frac{P(0),\ \forall j(\forall i((i < j \ \&\ P(i)) \rightarrow P(j)))}{\forall k\ P(k)}$$

constructor, selector and predicate operations. Referring again to the example of lists as the domain of definition, the primitive functions used in function definitions must correspond to such list operators as head-of-list, rest-of-list, concatenate and the primitive predicate might be "is atom" or "equals null list."

Example of a Proof by Structural Induction

We can prove the equivalence of g and h defined in the example of proof by recursion induction by recognizing that the following ordering on the ordered pairs on which the functions operate satisfies the minimum condition:

$$(x_1,y_1) \textcircled{<} (x_2,y_2)$$

iff $y_1 < y_2$

where < is the usual relation "is strictly less than" for integers. Now we have to prove two things:

1. g(x,0) = h(x,0) for any x

   (the set {(x,0)|x a number} contains all the ordered pairs which are "atoms" or minimally complex structures)

2. if g(x,i) = h(x,i) for any i < j

   then g(x,j) = h(x,j)

1. g(x,0) ≡ succ(f(x,0))

   = succ (if 0 = 0 then x else f (succ(x), pred(0)))

   = succ(x)

   h(x,0) ≡ f(succ(x),0)

   = if 0 = 0 then succ(x) else f(succ(succ(x)),pred(0))

   = succ(x)

2. $g(x,j) \equiv succ(f(x,j))$

$= succ(\text{if } j = 0 \text{ then } x \text{ else } f(succ(x),pred(j)))$

$= \text{if } j = 0 \text{ then } succ(x) \text{ else } succ(f(succ(x),pred(j)))$

$= \text{if } j = 0 \text{ then } succ(x) \text{ else } g(succ(x),pred(j))$

$= \text{if } j = 0 \text{ then } succ(x) \text{ else } h(succ(x),pred(j))$

(by induction)

$h(x,j) \equiv f(succ(x), j)$

$= \text{if } j = 0 \text{ then } succ(x) \text{ else } f(succ(succ(x)),pred(j))$

$= \text{if } j = 0 \text{ then } succ(x) \text{ else } h(succ(x),pred(j))$

For structural induction to apply the functions must be total over a particular well-defined domain. One method often used to prove totality of a function over a domain is to show that at each level of recursion, the value of some variable associated with the computation is decreased and that this variable cannot decrease in value indefinitely (Floyd [3]). This proof is implicit in a proof by structural induction, since the technique could not be applied at all unless such a variable could be defined. In the example given the variable was the value of the second argument in successive calls. In a sense equivalence is proved only over the domain of a third function, this function being the defining or testing function for the structure. It appears, then, that structural induction is only a special case of recursion induction, recursion induction applied only to programs meant to operate on structured data or on partially ordered domains. Also, it may be expected that for many proofs by recursion induction, no proof by structural induction exists. This occurs in any case where no structure can be ascribed to the data for which the function is defined, and in particular for uninterpreted schemas.

1.4 Recursive Definitions as Mappings Between Lattices

Before discussing the next two induction principles it is necessary to review some results of lattice theory (Park [8]) from which alternative interpretations of the recursive definitions can be formulated. We will not go into lattice theory in any depth, but will simply cite several definitions and theorems and show how they relate to the function definitions.

A lattice is a set with a partial ordering such that every pair of elements in the set has a least upper bound and a greatest lower bound. A complete lattice is a lattice in which every subset of the lattice has a least upper bound and a greatest lower bound. To see the relation to our study we must realize that the algebra of subsets of a set is a complete lattice and that functions mapping n-1 tuples into single values can be viewed as n-ary relations. The set of all n-ary relations on a set D forms a complete lattice with $\subseteq$ (set containment) as the partial order and anything true for complete lattices will be true for this set. Recursive definitions will be interpreted as mappings from the set of relations into itself. The function defined by such a definition will then correspond to a fixpoint of this mapping.

Let us first consider an arbitrary lattice L. A mapping $c:L \to L$ is monotonic if for all $m_1 \subseteq m_2$, $c(m_1) \subseteq c(m_2)$. We define now

(1) $\quad conv(c) = \bigcap \{m \mid c(m) \subseteq m\}.$

and notice that

(2) $\quad c(m) \subseteq m \to conv(c) \subseteq m$

The fixpoint theorem of Knaster-Tarski states that for c monotone

(3) $\quad c(conv(c)) = conv(c)$

$\quad conv(c) = \bigcap \{m \mid c(m) = m\}$

or, in words, first, conv(c) is a fixpoint of the map c, and second, that conv(c) is the minimal fixpoint. (Proofs in appendix.)

One further definition leads to an even more illuminating representation of the fixpoint conv(c). A map c is continuous if whenever $m_0 \subseteq m_1 \subseteq \ldots$ then

$$c\left(\bigcup_{i=0}^{\infty} m_i\right) = \bigcup_{i=0}^{\infty} c(m_i).$$

When c is monotone and continuous

$$\text{(4)} \qquad \text{conv}(c) = \bigcup_{i=0}^{\infty} c^i(0).$$

Now let us see the operation of function definition as a mapping. We use here an example from arithmetic. Let $\mathcal{C}(f) = \lambda x.$ if x = 0 then 1 else x*f(x-1). Then for any known relation f, $\mathcal{C}(f)$ is another relation,

$$\mathcal{C}(f) = \{(0,1),\ (1,1*f(0)),\ (2,2*f(1)),\ (3,3*f(2)),\ \ldots\}$$

For example, if f is the identity function, then $\mathcal{C}(f) = \{(0,1),(1,0),(2,2),(3,6),\ldots\}$. If f is the factorial $\{(0,1),(1,1),(2,2),(3,6),\ldots\}$, then $\mathcal{C}(f) = \{(0,1),(1,1),(2,2),(3,6),\ldots\}$, i.e. $\mathcal{C}(f) = f$, the factorial function is a fixpoint of $\mathcal{C}$. Also note that, if for a moment the domain of interest is extended to the integers, then if $f = \{\ldots,(-3,0),(-2,0),(-1,0),(0,1),(1,1),(2,2),(3,6),\ldots\}$, then again $\mathcal{C}(f) = f$.

When we write $f \equiv \mathcal{C}(f)$ our intention is that f represent the minimal fixpoint of the definition (or mapping) $\mathcal{C}$. From lattice theory, it is known that all monotonic mappings have fixpoints. If this method of writing definitions, i.e. using composition of functions and forming conditional expressions, always leads to monotonic mappings then all such recursive definitions are guaranteed to be meaningful in the sense that each such definition specifies a unique function and that function is the minimal fixpoint of the mapping associated with the form. By considering the logical implications of conditional expressions it can be shown that all such definitions are indeed monotonic (see Park [8]). By examining a

few conditional expressions for monotonicity, the reader should be able to convince himself that monotonicity will hold for the class of definitions we have been discussing. A rephrasing of the meaning of monotonicity may help. Say we are given a conditional expression $\mathcal{C}$, and we want to compute $\mathcal{C}(f)$ for a particular function f with infinite domain. As we begin to specify f we can begin to compute $\mathcal{C}(f)$. That is, a certain amount of information about f yields certain information about $\mathcal{C}(f)$. Added knowledge about f can reveal further members of $\mathcal{C}(f)$, but monotonicity guarantees that no new information will ever force a change in the former characterization of $\mathcal{C}(f)$. If f and $\mathcal{C}(f)$ are n-ary relations, then when an n-tuple is found to be in $\mathcal{C}(f)$ based on a finite number of n-tuples known to be in f, then that n-tuple is in $\mathcal{C}(f)$ and no further information about f will ever show that $\mathcal{C}(f)$ does not contain that n-tuple. In computer programming where programs often represent potentially infinite computations, although only finite computations can be carried out, it is good to know that results obtained in a finite amount of time are reliable. That is the importance of the monotonicity of these definitions.

Now, once we know that the definitions are meaningful, we would like to understand how the minimal fixpoint of one of them can be found. Since conditional expressions will always represent continuous mappings, there is a technique. Continuity of a mapping, $\mathcal{C}$, means that for a series of functions $f_1 \subseteq f_2 \subseteq \ldots$ such that $\bigcup_{i=1}^{\infty} f_i = f$, the series of functions $\mathcal{C}(f_1) \subseteq \mathcal{C}(f_2) \subseteq \ldots$ has the property that $\bigcup_{i=1}^{\infty} \mathcal{C}(f_i) = \mathcal{C}(f)$. The reader will find that if he allows himself to use only the operations set out above he can generate only continuous mappings (see appendix for a definition which is not continuous). Then, as noted above (4), the fixpoint of $\mathcal{C}$ is $\bigcup_{i=0}^{\infty} \mathcal{C}^i(0)$. In the lattice of n-ary relations 0 is Ω, the undefined relation.

For example, we compute the fixpoint of the factorial function as follows:

$$\mathcal{C}^0(\Omega) = \Omega \qquad = f_0$$

$$\mathcal{C}^1(\Omega) = \{(0,1)\} = f_1$$

$$\mathcal{C}^2(\Omega) = \{(0,1),\ (1,1)\} = f_2$$

$$\mathcal{C}^3(\Omega) = \{(0,1),\ (1,1),\ (2,2)\} = f_3$$

$$\vdots$$

$$f = \bigcup_{i=0}^{\infty} f_i$$

Notice that in this case a pair added to the function at $\mathcal{C}^n(\Omega)$ represents a computation which requires n levels of recursion.

$$f(3) = 3*f(2) = 3*2*\ f(1) = 3*2*1*f(0) = 3*2*1*1.$$

In a proof, induction on level of recursion is induction on the indexes of these finite approximations to a function.

Before leaving this section we will remark on one more theorem, the one which resulted immediately from the definition (1) of conv(c). It was (2) $c(x) \subseteq x \rightarrow conv(c) \subseteq x$. This is interpreted to be the rule for recursion induction, since it prescribes a very similar proof technique. Given functions f and g defined by $f \equiv \mathcal{C}(f)$ and $g \equiv \mathcal{O}(g)$ (in other words $f = conv(\mathcal{C})$, $g = conv(\mathcal{O})$ we can attempt to prove equality directly by showing

1. $f = \mathcal{O}(f)$
2. $g = \mathcal{C}(g)$.

These results and the theorem (2) imply 1. $f \subseteq g$, and 2. $g \subseteq f$. Therefore $f = g$. When the third function $h \equiv \mathcal{H}(h)$ is used as described by McCarthy this theorem is involved twice to get the results $h \subseteq f$ and $h \subseteq g$. This says that f and g are equal over the domain of h.

1.5 The μ-rule

In an unpublished paper Scott and deBakker build an algebraic theory of programs. In it are the usual axioms for conditional expressions, and axioms which force a model for the theory to contain only monotonic, continuous functions. There is one rule of inference, the μ-rule. It is essentially a rule for simple induction on the level of recursion. The rule is

$$\text{Premises:}\quad \frac{\Psi(\Omega) \qquad \Psi(X) \rightarrow \Psi(\mathcal{C}(X))}{\text{Consequent:}\quad \Psi(\text{conv}(\mathcal{C}))}$$

where Ψ is the predicate (perhaps it is equivalence of two definitions). If it can be shown that Ψ is true for the empty function, Ω, and that Ψ true for a function X implies that Ψ is true for $\mathcal{C}(X)$, then we can assume $\Psi(\text{conv}(\mathcal{C}))$. This is more meaningful if we recall that $\text{conv}(\mathcal{C}) = \Omega \cup \mathcal{C}^1(\Omega) \cup \mathcal{C}^2(\Omega) \cup \ldots$. By simple induction on i, $\Psi(\mathcal{C}^i(\Omega))$ is established for all i. By definition of $\text{conv}(\mathcal{C})$, this establishes $\Psi(\text{conv}(\mathcal{C}))$.

Facts to be proved are usually equivalences of two functions or containment of one function by another, but as with the other methods, its use can be extended to consider other predicates if suitable extensions are made to the language. With the μ-rule some questions of domains of functions seem to be taken care of more easily. If the programs "operate" in similar ways, i.e. always testing the same predicates on the same arguments, the domain comparison is automatically done. However, we do not find ourselves with an answer to the question of convergence (as we know we could not). Often it can be proved that one function is contained in another, but to prove equivalence would be to prove the totality of one of the functions. While such a proof may be possible given facts about the particular program (properties of numbers if a program involves arithmetic) it will not fall out of a proof in this theory.

Example of Proof by μ-rule

Again, we use the same example.

We prove $\Psi(f) \equiv succ(f(m,n)) = f(succ(m),n)$

where f is the minimal fixed point of the definition

i.e. $f(m,n) \equiv \mathcal{C}(f)(m,n) \equiv$ if $n = 0$ then m else $f(succ(m),pred(n))$

We need to prove

1. $\Psi(\Omega)$
2. $\Psi(X) \rightarrow \Psi(\mathcal{C}(X))$

1. $succ(\Omega(m,n)) = \Omega(m,n)$

   $\Omega(succ(m,n)) = \Omega(m,n)$

2. $succ(\mathcal{C}(X)(m,n)) = succ($if $n = 0$ then m else $X(succ(m),pred(n)))$

   $=$ if $n = 0$ then $succ(m)$ else $succ(X(succ(m),pred(n)))$

   $=$ if $n = 0$ then $succ(m)$ else $X(succ(succ(m)),pred(n))$

   (by assumption that $\Psi(X)$ holds)

   $= \mathcal{C}(X)\ (succ(m),n)$

Finally, note that recursion induction, in the form attainable from lattice theory, i.e. $\mathcal{C}(x) \subseteq x \rightarrow conv(\mathcal{C}) \subseteq X$ is a theorem in this theory. Recursion induction, then is clearly no stronger than the μ-rule on continuous functions.

1.6 Truncation Induction as Defined by Morris

This method also depends heavily on the realization that we are dealing only with continuous mappings. For any recursive definition $f \equiv \mathcal{C}(f)$ the truncations of f are the functions $f_0, f_1, f_2, \ldots$ where f_n is f restricted to n levels of recursion

This means $f_0 = \Omega$

$$f_1 = \mathcal{C}(f_0)$$

$$f_2 = \mathcal{C}(f_1) = \mathcal{C}^2(f_0)$$

.

.

.

and as stated before

(1) $$f = \bigcup_{i=0}^{\infty} \mathcal{C}^i(\Omega) = \bigcup_{i=0}^{\infty} f_i$$

For a particular x, $f(x) = y$ if and only if there is an i such that $f_i(x) = y$, since the ordered pair (x,y) is in f, as defined in (1), if and only if (x,y) is in f_i for some i. Therefore, a proof that for each i, the ordered pairs in f_i satisfy some condition, is a proof that this condition must also hold for the function f. For example, if for each i, f_i contains only ordered pairs of the form (x,0) then the ordered pairs in f can only be of that form. In comparing two functions, f and g, if it can be shown that for all i, $f_i = g_i$, then clearly $f = g$.

To obtain results about f_i proofs proceed by course-of-values induction on the index of the truncations. From $P(f_0)$ and $\forall_{i<j}\ P(f_i) \rightarrow P(f_j)$ we infer P(f).

The difference between truncation induction and the μ-rule is that the latter method is restricted to the use of simple induction on these truncations. Both methods are equally powerful, but whether course-of-values induction or simple induction is more convenient in dealing with recursive function can still be questioned. Morris' main argument for

truncation induction is to compare it to McCarthy's recursion induction and show that some of the convergence problems can be avoided. However, with a looser interpretation of recursion induction, which we will see is justified, as much can be said about convergence. And of course using truncation induction still does not make it possible to directly prove strong equivalence of a recursively defined function and a constant one.

Example of Proof by Truncation Induction

We want to prove succ(f(m,n)) = f(succ(m),n)
where f(m,n) ≡ if n = 0 then m else f(succ(m), pred(n))
From this definition we define the truncations of f to be

$$f_i(m,n) \equiv \text{if } n = 0 \text{ then } m \text{ else } f_{i-1}(succ(m),pred(n))$$

We must prove 1. $succ(f_0(m,n)) = f_0(succ(m),n)$

2. if $\forall i < j$, $succ(f_i(m,n)) = f_i(succ(m),n)$
then $succ(f_j(m,n)) = f_j(succ(m),n)$

1. $succ(f_0(m,n)) = succ(\Omega(m,n)) = \Omega(m,n)$

$f_0(succ(m),n) = \Omega(succ(m),n) = \Omega(m,n)$

2. $succ(f_j(m,n)) \equiv succ(\text{if } n = 0 \text{ then } m \text{ else } f_{j-1}(succ(m),pred(n)))$

$= \text{if } n = 0 \text{ then } succ(m) \text{ else } succ(f_{j-1}(succ(m),pred(n)))$

$= \text{if } n = 0 \text{ then } succ(m) \text{ else } f_{j-1}(succ(succ(m)),pred(n))$
(by induction)

$= f_j(succ(m),n)$

Chapter 2. The Comparisons

2.1 Objectives

In the following sections we will both look for similarities among the four methods previously described, and investigate the nature of some of their differences. We will do this by comparing proofs of identities by different techniques, and translates of proofs from one method to another. For cases where the methods are incomparable we will show how with more explicit use of assumed facts about program definition, proofs of the same facts can be obtained by a combination of arguments.

2.2 Structural Induction and Truncation Induction

While structural induction and truncation induction depend on different properties of functions, the former being induction on the complexity of a data structure, and the latter on the complexity of a computation structure, the restrictions on the form of function definition as specified by Burstall make these differences insignificant. The following comments on this restricted class of definitions will show that complexity of data structure and level of recursion are so closely related in these definitions, that they are interchangeable as indexes for inductive proofs.

Structural Induction is a special case among the inductive methods. It is the only one which is defined for interpreted programs, and only for those which operate in a particular way on structured data types. (i.e. We cannot use an arbitrary well-defined ordering on the domain for induction. It must be an ordering such that each recursive call by the function is guaranteed to be made with an argument which is lower in the ordering.) On these programs, structural induction is quite simple to use and to justify independent of facts about recursive functions. The induction involved is induction on the complexity of an ordering on data, a familiar type of induction not requiring the introduction of any new knowledge or notations about functions. The recursion implicitly involved in the function definition is not the interesting phenomenon. The proof works because of properties of the data structure, rather than properties of the recursive nature of the function. The actual proof is an examination of the function in terms of calls on less structured data, rather than in terms of calls requiring fewer and fewer nested calls, or calls on less fully specified partial functions. No insight into fixpoints or their constructions is necessary.

Structural induction could as easily be justified by considering the structure of the recursive functions. In fact, if truncation induction is used in a proof about functions defined in the specified way it would be found that exactly the same inferences are made though supposedly for different reasons. Let us examine the general pattern of a proof by

structural induction of the equality of two functions, $f \equiv \mathcal{F}(f)$ and $g \equiv \mathcal{G}(g)$, on a domain of structured data. To prove $\forall x f(x) = g(x)$ we prove first that $f(x_i) = g(x_i)$ for all x_i which are atoms or minimally complex structures. This part of the proof will necessarily be very straightforward since evaluation of $P(x_i)$, (or $g(x_i)$), x_i atomic, cannot involve recursive calls on f (or on g) since any recursive call on f (or g) is made with an argument with less complex structure than that of the original argument. When this original argument is already minimal, the final result must be directly obtainable from known functions. This is clear from the definition of the function which in general takes the form

(1) $\quad f(x) \equiv$ if (x atomic) then $\mathcal{C}_1(x)$ else $\mathcal{C}_2(f)\ (x)$

Therefore, by analysis by case (should there be more than one minimal element) this first part of the proof should be trivially true.

Next we must prove that for any x which is not minimal $f(x) = g(x)$, given that $\forall x' < x$, $f(x') = g(x')$. Again, we refer to the restriction on the forms of f and g. Since any occurrence of f on the right hand side of the definition $f(x) \equiv \mathcal{F}(f)(x)$ represents an application of f to a less complex structured argument, the assumption $f(x') = g(x')$ may as well be restated as an assumption that $\forall x(f(x) = g(x))$ for all subsequent calls on f and g. To be sure that the induction hypothesis is used only when valid, subscripts might be used to differentiate between the original use of f and subsequent recursive calls.

The similarity to truncation induction is becoming more obvious now. The induction step is virtually identical for proofs on this class of programs. It is necessary to prove $f_i(x) = g_i(x)$ assuming either that $f_j(x) = g_j(x)$ for $j < i$ or that for all recursive subcalls on f and g in the evaluation, $f(x) = g(x)$. How does the basis for induction compare? In truncation induction we show $f_0(x) = g_0(x)$. Then from the induction step it follows that $f_1(x) = g_1(x)$. In terms of the definitions of f and g, this means $\mathcal{F}(\Omega)(x) = \mathcal{G}(\Omega)(x)$. Referring back to the general form (1) of functions for which it is meaningful to consider proof by structural induction, this says that f and g agree for the atomic

structures. Thus the analogy is completed between structural induction and truncation induction. When we prove $f(x) = g(x)$ for x atomic we are establishing $f_1(x) = g_1(x)$. This is then the basis of induction, whether we consider it to be induction on the complexity of the data or on the depth of recursion.[1]

The validity of starting proofs from this basis seems questionable in light of the construction of the minimal fixpoint. For most statements provable by structural induction, it is true that the basis generally used for truncation induction proof is true as well and in translating to a trunction induction proof we could, in fact, write a valid proof with the statement for $i = 0$ as the basis. Thus any two recursively defined functions known to be equivalent due to proof by structural induction, can generally be shown to be strongly equivalent by truncation induction and therefore equivalent on the domain of interest, confirming the results of the structural induction proof. However, a structural induction proof may be dependent on the implicit assumption that any function to be examined is total on the domain of interest (this assumption can, in fact, be proved as noted at the end of section 1.3). For an example, take a definition $f \equiv \mathcal{F}(f)$ satisfying the criteria for structural induction and defining the function f, where $f(x)$ is equal to the constant A for all x. A structural induction proof that $f(x) = A$ is easy to write. However, by truncation induction alone, only the weaker statement $f \subseteq \lambda x.A$ can be proved. This is true because the strongest statement provable as a basis for induction is $f_0 \subseteq \lambda x.A$. The statement $f_0(x) = A$ is not true. The

---

[1] The reader might note that structural induction as stated here seems a bit redundant, in that the proof of equality of $f(x) = g(x)$ for x atomic is included in the proof of $f(x) = g(x)$ for general x, since clearly there must be a correspondence between the first then-clauses of both definitions for the induction step to carry through, and this correspondence proves $f(x) = g(x)$ for x atomic. This redundancy occurs here because of the consistent use of McCarthy's notation writing the whole conditional in each part of the proof. Burstall, in his paper [1], makes a few notational changes and writes functions by case. Then using analysis of cases in proof, the cases x atomic and x not atomic are separate and the two parts of the proof do not overlap. In examples, where identity is clear, unnecessary parts will be omitted.

1. succ(f(m,n)) ≡ succ(if n = 0 then m else f(succ(m),pred(n)))

   = if n = 0 then succ(m) else succ(f(succ(m),pred(n)))

   = if n = 0 then succ(m) else f(succ(succ(m)),pred(n))

   (by induction)

2. f(succ(m),n) ≡ if n = 0 then succ(m) else f(succ(succ(m)),pred(n))

The truncation induction proof of $succ(f_i(m,n)) = f_i(succ(m),n)$ follows simply by combining the two parts of the structural induction proof with subscripts added.

$succ(f_i(m,n))$ ≡ succ(if n = 0 then m else f_i(succ(m),pred(n)))

= if n = 0 then succ(m) else succ(f_{i-1}(succ(m),pred(n)))

= if n = 0 then succ(m) else f_{i-1}(succ(succ(m)),pred(n))

(by induction)

= f_i(succ(m),n)

Example 2 (from Burstall [1])

lit(f,s,y) ≡ if empty(s) then y else f(head(s),lit(f,rest(s),y))

[head(s), rest(s), concat(s1,s2) are primitive functions corresponding to the obvious list operators]

We want to prove

lit(f,concat(s1,s2),y) = lit(f,s1,lit(f,s2,y))

The proof will be by structural induction on the list s1.

1. for s1 = nil

   lit(f, concat(s1,s2),y) = lit(f,s2,y)

   lit(f,s1,lit(f,s2,y)) = lit(f,s2,y)

2. for s1 ≠ nil

$$lit(f,concat(s1,s2),y) \equiv f(head(concat(s1,s2)),lit(f,rest(concat(s1,s2)),y))$$

$$= f(head(s1),lit(f,concat(rest(s1),s2),y))$$

$$= f(head(s1),lit(f,rest(s1),lit(f,s2,y)))$$

(by induction)

$$lit(f,s1,lit(f,s2,y)) \equiv f(head(s1),lit(f,rest(s1),lit(f,s2,y))).$$

In this proof it is not obvious how the assumption about the domain is used until we try to set up the induction hypothesis for a truncation induction proof.

We want to prove $lit_i(f,concat(s1,s2),y) = lit_i(f,s1,lit_i(f,s2,y))$

However, for i,s1,s2 such that $i < length(concat(s1,s2))$ but $i \geq length(s1)$ and $i \geq length(s2)$ the left hand side is undefined while the right hand side is defined.

We can therefore only prove

$$lit_i(f,concat(s1,s2),y) \subseteq lit_i(f,s1,lit_i(f,s2,y))$$

1. base

$$\Omega(f,concat(s1,s2),y) \equiv \Omega(f,s1,\Omega(f,s2,y))$$

2. $lit_i(f,concat(s1,s2),y)$ = if empty(concat(s1,s2)) then $lit_i(f,s2,y)$

else $f(head(concat(s1,s2)),lit_{i-1}(f,rest(concat(s1,s2)),y))$

= if empty(concat(s1,s2)) then $lit_i(f,s2,y)$

else $f(head(s1),lit_{i-1}(f,concat(rest(s1),s2),y))$

$\subseteq$ if empty(concat(s1,s2)) then $lit_i(f,s2,y)$ else $f(head(s1),lit_{i-1}(f,$

$rest(s1),lit_{i-1}(f,s2,y)))$

$= lit_i(f,s1,lit_i(f,s2,y))$

Once the correct subscripting is decided on, all steps are just simple transcriptions of the steps of the structural induction proof.

Example 3. Illustrating the case in which domain assumptions are involved in a proof comparing a recursively defined function to a known function on the natural numbers.

$$f(x) \equiv \text{if } x = 0 \text{ then } 1 \text{ else } x*f(x-1)$$

we want to prove $f(x) = x!$

By structural induction:

for $x = 0$ $\quad f(x) = 1$ and $x! = 1$

for $x \neq 0$ $\quad f(x) = x*f(x-1)$

$= x*(x-1)!$ (by induction)

$= x!$

$\therefore \quad f(x) = x!$

To do a truncation induction proof we would want as our induction hypothesis $f_i(n) = n!$, however, this is not true for any i. Instead we will prove $f_i(x) \subseteq x!$

1. $f_0(x) = \Omega(x) \subseteq x!$
2. $f_i(x) \equiv \text{if } x = 0 \text{ then } 1 \text{ else } x * f_{i-1}(x-1)$

$= \text{if } x = 0 \text{ then } 1 \text{ else } x * (x-1)!$

$= x!$

$\therefore \quad f(x) \subseteq x!.$

From the form of the computation (which performs successive reductions in the initial integer argument) it is clear that f(n) converges for all natural numbers. Therefore, $f(x) = n!$ for n a natural number.

Notice that this sort of argument does not just allow one to ignore the difference between containment and equality in cases outside the class of programs for which structural induction can be used. For example, with the following function:

$$f(x) \equiv \text{if } x = 1 \text{ then } 0 \text{ else if } x \text{ even then } f(x/2) \text{ else } f(3x+1)$$

at present, only $f \subseteq \lambda x.0$ can be proved since there is no known independent way of describing an ordering on the integers such that a convergence argument can be made showing f to be total over the natural numbers.

For completeness, note the fact that although truncation induction has more general application it is consistent with structural induction whenever applied to definitions in the class to which this method applies. Therefore any fact proved by truncation induction about such a program will clearly be provable by structural induction. Furthermore, a legitimate proof by structural induction can be written quite mechanically by simply copying the truncation induction proof without the subscripts. This gives us the induction step, the basis being easily extractable as discussed above due to both the form of the definition and notational conventions.

2.3 μ-rule and Truncation Induction

The difference between μ-rule and truncation induction is essentially the difference between simple and course-of-values induction. Both methods are based on the same construction of fixed points, namely $\mathrm{conv}(\mathcal{C}) = \bigcup_{i=0}^{\infty} \mathcal{C}^i(\Omega)$. In comparing some pairs of functions you will find that often the equality of the two functions for a particular argument is based on their equality for other arguments found after some finite number of recursive calls. Or, to put it another way, it may be that for each x, if f(x) and g(x) are defined at stage i of the constructions of f and of g, then they are equal if and only if f(x') = g(x') at some stage $j < i$, $j \neq i - 1$. To prove equivalence by truncation induction is simple, since the assumption is for all $j < i$. By μ-rule, a series of proofs must be performed, establishing identities at stages j, j+1, ..., i-1 so that each proof requires looking back only one step in the construction. That there is no real difference in proving power between these two methods becomes clear by establishing procedures for transforming one type of proof into the other.

That μ-rule induction is no more powerful than truncation induction is clear, since a μ-rule proof can be interpreted as a truncation induction proof directly, where all induction hypotheses happen only to be applied at one level. If, on the other hand, an equality is established by truncation induction, it will also be possible to prove it by μ-rule. Keeping in mind that the only additional restriction is the limit of looking back one step at a time in the construction, it can be seen that the μ-rule proof will have to be a series of proofs, one for each level at which the induction hypothesis is used. We will now go into more detail on these observations and work out a few examples.

Truncation induction allows us to assume f = g for all occurrences of f in the computation of $\mathcal{H}(f)(x)$, while the μ-rule allows this assumption only for the occurrences of f at the first level of recursion. This explains the statement of the μ-rule induction step as being a proof that

$P(X) \Rightarrow P(\mathcal{H}(X))$. It is a notational convenience to help in avoiding the temptation to expand $\mathcal{H}(f)(x)$ to $\mathcal{H}(\mathcal{F}(f),x)$ and to possibly reuse the induction hypothesis. It is true that $f = g$ in $\mathcal{H}(f)(x)$ but it is no longer true that, for this f, $f \equiv \mathcal{H}(f)$. This f is a dummy variable about which only P(f) is known. We will see though that this may not be such a restricting condition, by showing how we can always find a μ-rule proof if we already have a truncation induction proof.

If P is a conjuction $P_1 \wedge \ldots \wedge P_k$ and we know P(X), then we know $P_1(X) \wedge \ldots \wedge P_k(X)$. Then proving $P(\text{conv}(\mathcal{H}))$ for some function specified by the definition $f \equiv \mathcal{H}(f)$ proves $P_1(\text{conv}(\mathcal{H}))$ and ... and $P_k(\text{conv}(\mathcal{H}))$ simultaneously. In a truncation induction proof there can only be a finite number of inferences which follow from the definition $f \equiv \mathcal{H}(f)$. For each of these inferences which would be invalid in a proof by μ-rule P_i can be defined to be the part of P which will allow us to make the necessary inference. Let us do the conversion for one case before discussing the general case any further.

Example 1.

prove $f(x,y) = g(x,y)$ by truncation induction

$$f(x,y) \equiv \text{if } Px \text{ then } y \text{ else } Gf(Fx,y)$$

$$g(x,y) \equiv \text{if } Px \text{ then } y \text{ else } g(Fx,Gy)$$

1. $f_i(x,y) \equiv$
2. if Px the y else $Gf_{i-1}(Fx,y)$
3. (ind) = if Px then y else $Gg_{i-1}(Fx,y)$
4. (def) = if Px then y else G(if PFx then y else $g_{i-2}(FFx,Gy)$)
5. = if Px then y else if PFx then Gy else $Gg_{i-2}(FFx,Gy)$
6. (ind) = if Px then y else if PFx then Gy else $Gf_{i-2}(FFx,Gy)$
7. (def) = if Px then y else $f_{i-1}(Fx,Gy)$
8. (ind) = if Px then y else $g_{i-1}(Fx,Gy)$
9. (def) = $g_i(x,Gy)$

2. $\equiv$ if Px then y else GX(Fx,y)

3. $(ind)_1$ = if Px then y else GY(Fx,y)

7. (ind_2) = if Px then y else X(Fx,Gy)

8. (ind_1) = if Px then y else Y(Fx,Gy)

9. (def) = $\mathcal{H}(X)(x,y)$

Proof of $P_2(\mathcal{H}(X), \mathcal{A}(Y))$

3. $G(\mathcal{A}(Y)(Fx,y) \equiv$

4. G(if PFx then y else $Y(F^2x, Gy)$)

5. = if PFx then Gy else $GY(F^2x,Gy)$

6. (ind_1) = if PFx then Gy else $GX(F^2x,Gy)$

7. (def) = $\mathcal{H}(X)(Fx,Gy)$

The reader may find that a shorter proof by μ-rule is possible if P_2 is formulated slightly differently, but the predicate as stated here shows the exact correspondence between the two proofs and would be the best choice in terms of a mechanical translation from one proof to the other.

In the general case, a few more properties of the proof may have to be taken into account in formulating some P_k. In the second level proof in example 1 the only facts used were the induction hypothesis and the fact that G was distributive over the clauses of a conditional. However, the entire subproof took place under the additional hypothesis that $\neg$ Px was true, since all changes were in the else-clause of a conditional based on the test Px. Actually P_2 should have been $\neg Px \rightarrow Gg(Fx,y) = f(Fx,Gy)$. It is not difficult to find examples of proofs where facts about the arguments implied by the result of a conditional test are necessary conditions for further equivalences. It just happens that in this case it is true that for all x, Gg(Fx,y) = f(Fx,Gy). Also, it may occasionally be necessary in a proof by truncation induction to use as justification for one step the result of some other proof by truncation induction. This result too can be treated as a constituent of the predicate to be proved by the μ-rule,

and its proof will be found in the same manner.

A more general procedure for changing a proof by truncation induction to a proof by μ-rule follows. Given a proof $\mathcal{P}_t$ of $P_t(f_1, \ldots f_n)$ by truncation induction we will prove $P_\mu(f_1, \ldots, f_n)$ where $P_\mu = P_t \wedge P_{y_1} \ldots \wedge P_{y_n} \wedge P_{x_1\text{-}z_1} \wedge \ldots \wedge P_{x_m\text{-}z_m}$ where

(1) P_{y_i} is a predicate corresponding to the subproof yi

(2) $P_{x_i\text{-}z_i}$ is the predicate $C_{x_i} \rightarrow P_i$

where P_i is the predicate which would justify skipping the block of lines x_i to y_i if x_{i+1} follows from x_i by an expansion by definition of a function (and $x_i \neq 1$) and y_i is the first line after x_i in which all functions are again at level of recursion of line x_i.

C_i describes the conditions which hold for the clause of the conditional in which the fact that P_i is true of the functions involved is used. Once the proper predicate is formulated, proof is written by transcribing of the lines of $\mathcal{P}_t$ into $\mathcal{P}_\mu$ in the obvious manner.

Example 2.

We will prove $g(x,x) = h(x)$, by showing $g_i(x,x) = h_i(x)$.

$f(x) \equiv g(x,x)$

$g(x,y) \equiv$ if Px then hy else $g(Fx,y)$

$h(x) \equiv$ if Px then x else hFx

basis $g_0(x,x) = \Omega(x,x) = \Omega(x) = h_0(x)$

1. $g_i(x,x) \equiv$

2. if Px then hx else $g_{i-1}(Fx,x)$

3. = if Px then x else $g_{i-1}(Fx,x)$ $\qquad (Px \Rightarrow h(x) = x)$

4. (subpr) = if Px then x else $g_{i-1}(Fx,Fx)$

5. (ind) = if Px then x else $h_{i-1}(Fx)$

6. (def) = $h_i(x)$

Proof that for all i $(\forall x \forall y (\neg Px \rightarrow g_i(y,x) = g_i(y,Fx)))$

1. $g_i(y,x) \equiv$
2. if Py then hx else $g_{i-1}(Fy,x)$
3. = if Py then hFx else $g_{i-1}(Fy,x)$ $(\neg Px \Rightarrow hx = hFx)$
4. (ind) = if Py then hFx else $g_{i-1}(Fy,Fx)$
5. (def) = $g_i(y,Fx)$

From this proof by truncation induction we formulate the following proof by μ-rule.

$P_\mu(g,h) \equiv g(x,x) = h(x) \wedge \neg Px \Rightarrow g(Fx,x) = g(Fx,Fx)$

The induction part of the μ-rule proof will be:

1. $\mathcal{J}(X)(x,x)$
2. $\equiv$ if Px then hx else X(Fx,x)
3. = if Px then x else X(Fx,x)
4. (ind2) = if Px then x else X(Fx,Fx)
5. (ind1) = if Px then x else Y(Fx)
6. (defn) = $\mathcal{H}(Y)(x)$.

and

1. $\mathcal{J}(X)(y,x)$
2. $\equiv$ if Py then hx else X(Fy,x)
3. = if Py then hFx else X(Fy,x)
4. (ind2) = if Py then hFx else X(Fy,Fx)
5. (defn) = $\mathcal{J}(X)(y,Fx)$

At the beginning of this section we also suggested a way of finding a truncation induction proof to correspond to a given μ-rule proof. The simplest way is to note that a μ-rule proof is a truncation induction proof in which only a restricted form of the induction hypothesis is used. A substitution throughout the proof of f_i for $\mathcal{F}(X)$ and of f_{i-1} for X will directly yield a truncation induction proof. If the μ-rule proof is of a complex statement (i.e. a conjunction of statements) it is simply necessary to use a corresponding conjunction for the truncation induction proof.

Example 3

We want to prove $f(x,y) = g(x,y)$ and will do this by the μ-rule by proving

$$f(x,y) = g(x,y) \wedge Gg(Fx,y) = g(Fx,Gy)$$

1. $\Omega(x,y) = \Omega(x,y)$

   $G(\Omega(Fx,y)) = \Omega(Fx,Gy)$

2. a. $\mathcal{F}(X)\ (x,y) \equiv$ if Px then y else GX(Fx,y)

   = if Px then y else GY(Fx,y)

   (by induction 1)

   = if Px then y else Y(Fx,Gy)

   (by induction 2)

   = $\mathcal{G}(Y)(x,y)$

   b. $G\mathcal{F}(Y)(fx,y) \equiv G$(if Px then y else Y(FFx,Gy))

   = if Px then Gy else GY(FFx,Gy)

   = if Px then Gy else Y(FFx,GGy)

   (by induction 2)

   = $\mathcal{G}(Y)(Fx,Gy)$.

A corresponding truncation induction proof is then:
We want to prove

$$f(x,y) = g(x,y) \wedge Gg(Fx,y) = g(Fx,Gy)$$

and do so by proving by induction that

$$\forall i(f_i(x,y) = g_i(x,y) \wedge Gg_i(Fx,y) = g(Fx,Gy))$$

1. $f_0(x,y) = g_0(x,y) \wedge Gg_0(Fx,y) = g_0(Fx,Gy)$
2. assume $f_j(x,y) = g_j(x,y) \wedge Gg_j(Fx,y) = g_j(Fx,Gy)$
   for $j < i$

$f_i(x,y)$ = if Px then y else G $f_{i-1}(Fx,y)$

= if Px then y else G $g_{i-1}(Fx,y)$

(by induction 1)

= if Px then y else $g_{i-1}(Fx,Gy)$

(by induction 2)

= $g_i(Fx,Gy)$

and $Gg_i(Fx,y)$ = G(if Px then y else $g_{i-1}(FFx,Gy)$)

= if Px then Gy else G $g_{i-1}(FFx,Gy)$

= if Px then Gy else $g_{i-1}(FFx,GGy)$

(by induction 2)

= $g_i(Fx,Gy)$

2.4 Recursion Induction and Truncation Induction

The comparison between these two methods begins with showing that truncation induction can be used to prove any result proved by recursion induction. The general principle behind the transformation of a recursion induction proof into a truncation induction proof is quite simple. If by recursion induction it can be proved that f = g, then it must be the case that for some $h \equiv \mathcal{H}(h)$

$$f = \mathcal{H}(f)$$

and

$$g = \mathcal{H}(g).$$

The truncations $f_i = \mathcal{G}(f_{i-1})$ and $g_i \equiv \mathcal{J}(g_{i-1})$ of f and g can then generally be expressed as $f_i = \mathcal{H}(f_{i-1})$

$$g_i = \mathcal{H}(g_{i-1}).$$

A proof by truncation induction is then just the combination of the two parts of the recursion induction proof, as follows:

$f_i \equiv \mathcal{G}(f_{i-1})$

$= \mathcal{H}(f_{i-1})$ (by the result of rec.ind. proof)

$= \mathcal{H}(g_{i-1})$ (by induction)

$= \mathcal{J}(g_{i-1})$ (by rec. ind.)

$= g_i$

For simple recursion induction proofs, this transformation can be done quite mechanically, based on this outline.

Example 1

We will prove that succ(f(m,n)) = f(succ(m),n) where f is defined as in section 1.2.

Define h = λm.λn. if n = 0 then succ(m) else h(succ(m), pred(n))

1. succ(f(m,n)) $\equiv$ succ(if n = 0 then m else f(succ(m), pred(n)))

   = if n = 0 then succ(m) else succ(f(succ(m), pred(n)))

   = $\mathcal{H}$(λm,λn. succ(f(m,n)))(m,n)

2. f(succ(m,n)) $\equiv$ if n = 0 then succ(m) else f(succ(succ(m)), pred(n))

   = $\mathcal{H}$ (λm.λn.f(succ(m),n))(m,n)

Then the induction part of a truncation induction proof could be written:

succ(f_i(m,n)) = succ(if n = 0 then m else f_{i-1}(succ(m), pred(n)))

   = if n = 0 then succ(m) else succ(f_{i-1}(succ(m), pred(n)))

   = $\mathcal{H}$(λm.λn.(succ(f_{i-1}(m,n))) (m,n)

(ind) = $\mathcal{H}$ (λm.λn.f_{i-1}(succ(m),n)) (m,n)

   = if n = 0 then succ(m) else f_{i-1}(succ(succ(m)), pred(n))

   = f_i(succ(m),n).

There will be cases where this rule will not be directly applicable. Proofs in which the recursion induction principle is used less directly (for example in a subproof contained in one of the clauses of the defining conditional), or in which it is unclear how we are using the definitions or to what depth of recursion we have descended will require some examination to determine at just what point this "combination of proofs" will go through to produce a proof by truncation induction.

Some further examples should illustrate the types of considerations to be made before applying the rule.

Example 2

We will prove another identity involving the same f of example 1.

$$f(f(m,n),p) = f(f(m,p),n).$$

First we define $h \Leftarrow \lambda m.\lambda n.\lambda p.$ if $p = 0$ then $f(m,n)$ else $h(succ(m),n,pred(p))$

1. $f(f(m,n),p) \equiv$ if $p = 0$ then $f(m,n)$ else $f(succ(f(m,n)),pred(p))$

   = if $p = 0$ then $f(m,n)$ else $f(f(succ(m),n),pred(p))$

   (by proof ex. 1)

   $= \mathcal{H}(\lambda m.\ \lambda n.\ \lambda p.f(f(m,n,p)))(m,n,p)$

2. $f(f(m,p),n) \equiv f($if $p = 0$ then m else $f(succ(m),pred(p)),n)$

   = if $p = 0$ then $f(m,n)$ else $f(f(succ(m),pred(p)),n)$

   $= \mathcal{H}(\lambda m.\ \lambda n.\ \lambda p.f(f(m,p),n))(m,n,p)$

Notice that in part 1 the first occurrence of f is expanded by definition, while in part 2 we expanded the second occurrence. In neither part did we have to expand both occurrences of f. This fact is important in determining exactly which indexes will be involved in the induction of the truncation proof. While a proof by truncation induction of $f_i(f_i(m,n),p) = f_i(f_i(m,p),n)$ is possible it is not the proof most closely related to the recursion induction proof. The goal should be one for which the identities already proven by recursion induction can be used to generate mechanically a truncation induction proof. To that end, considering the manner in which recursion induction was used to achieve the desired results, a goal in terms of truncations can be found which will enable us to do this.

From this proof we get

$$f_i(f(m,n),p) = f(f_i(m,p),n)$$

and the following proof, with subscripting the only revision necessary (i.e. no new steps need be added)

$f_i(f(m,n),p) \equiv$ if $p = 0$ then $f(m,n)$ else $f_{i-1}(\text{succ}(f(m,n)),\text{pred}(p))$

$= $ if $p = 0$ then $f(m,n)$ else $f_{i-1}(f(\text{succ}(m),n),\text{pred}(p))$

$= \mathcal{H}(\lambda m.\lambda n.\lambda p.\ f_{i-1}(f(m,n),p))(m,n,p)$

$= \mathcal{H}(\lambda m.\lambda n.\lambda p.\ f(f_{i-1}(m,p),n))(m,n,p)$

$= $ if $p = 0$ then $f(m,n)$ else $f(f_{i-1}(\text{succ}(m),\text{pred}(p)),n)$

$= f($if $p = 0$ then m else $f_{i-1}(\text{succ}(m),\text{pred}(p)),n)$

$= f(f_i(m,p),n)$.

Example 3

$f(x) \equiv g(x,x)$

$g(x,y) \equiv$ if Px then hx else $g(Gx,y)$

$h(x) \equiv$ if Px then x else hFx

proof that $f(x) \subseteq h(x)$ (by showing $f = \mathcal{H}(f)$)

1. $f(x) \equiv g(x,x) \equiv$ if Px then hx else $g(Fx,x)$

$= $ if Px then hx else $g(Fx,Fx)$ See second part of proof

$= $ if Px then hx else fFx

$= \mathcal{H}(f)(x)$

2. w.t.p. $\neg Px \Rightarrow g(y,x) = g(y,Fx)$

Define $i \equiv \lambda y.\lambda x.$ if Py then hFx else $i(Fy,x)$

a. $g(y,x) \equiv$ if Py then hx else $g(Fy,x)$

$= $ if Py then hFx else $g(Fy,x)$

(since $\neg Px \Rightarrow hx = hFx$)

$= \mathcal{I}(\lambda y.\lambda x.\ g(y,x))(y,x)$

b. $g(y,Fx) \equiv$ if Py then hFx else $g(Fy,Fx)$

$= \mathcal{I}(\lambda y.\lambda x.\ g(y,Fx))(y,x)$

The truncation induction proof follows:

$f_i(x) \equiv g_i(x,x) \equiv$ if Px then hx else $g_{i-1}(Fx,x)$

= if Px then hx else $g_{i-1}(Fx,Fx)$ (see next proof)

= if Px then hx else $f_{i-1}(Fx)$ (definition)

= $\mathcal{H}(f_{i-1})(x)$

= $\mathcal{H}(h_{i-1})(x)$ (by induction)

= $h_i(x)$.

w.t.p. $\neg Px \Rightarrow g_i(y,x) = g_i(y,Fx)$

$g_i(y,x) \equiv$ if Py then hx else $g_{i-1}(Fy,x)$

= if Py then hFx else $g_{i-1}(Fy,x)$

= $\mathcal{G}(\lambda y.\lambda x.\ g_{i-1}(y,x))(y,x)$

= $\mathcal{G}(\lambda y.\lambda x.\ g_{i-1}(y,Fx))(y,x)$

= if Py then hFx else $g_{i-1}(Fy,Fx)$

= $g_i(y,Fx)$.

Conversely, given a truncation induction proof, some of the information in the proof can be used towards the writing of a recursion induction proof. If the truncation induction proofs in this section are examined, the reader will notice that the steps in which the common form is explicitly written out are superfluous, the exact same induction could have been performed directly on the expanded form. In an ordinary truncation induction proof, at the places where the induction hypothesis is used it should be possible to extract forms for the third function used in a recursion induction proof. As a matter of fact, informal performance of proof by truncation induction is probably the most often used technique for deciding what the third function should look like. One starts at both ends, expanding by the definitions the forms which are being compared until a common form is found.

There will be complications in applying this procedure to arbitrary truncation induction proofs, generally of the sorts investigated in the previous section on μ-rule and truncation induction. That is, in proofs with several successive steps by induction or definition at levels below i-1, the proof must be broken up into one step proofs, since recursion induction generally works on less depth of recursion than does truncation induction. Also in recursion induction, levels of recursion are not part of the function definition. This means certain of the arguments used in a μ-rule proof could not be used. For instance, for a proof of $P_1(f) \wedge P_2(f)$ in which $P_1(\mathcal{G}(X))$ is proved using only $P_2(X)$ and $P_2(\mathcal{G}(X))$ is proved using only $P_1(X)$, the corresponding reasoning would appear circular by recursion induction, where we would be saying that we can prove $P_1(f)$ if $P_2(f)$ is true while $P_2(f)$ is true if $P_1(f)$ is true. When a truncation induction proof is carried back several levels of induction this kind of reasoning results. If, in extracting subproofs as was done in trying to get a μ-rule proof, these sequences of lines are examined for common forms across one or more induction calls, means can be found for getting around this. For example, looking back to Example 1 of section 2.3, by calling lines 3 to line 8 the subproof we will find line 5 to describe the third form desired. In short, the change from truncation induction to recursion induction takes the same sort of recognition of useful third forms as does the original proof by recursion induction.

A proof of $f(x,y) = g(x,y)$ by recursion induction would actually take two parts $f \subseteq g$, $g \subseteq f$. One corresponds to showing $g = \mathcal{G}(g)$ the other of $f = \mathcal{F}(f)$. These in turn correspond to making a subproof about lines 3 - 8 or lines 2 - 7 of the truncation induction proof, respectively.
We will do $g(x,y) \subseteq f(x,y)$.
Therefore we want to prove $f = \mathcal{G}(f)$

Proof 1

$$f(x,y) \equiv \text{if } Px \text{ then } y \text{ else } Gf(Fx,y)$$

$$= \text{if } Px \text{ then } y \text{ else } f(Fx,Gy)$$

(by proof 2)

$$g(x,y) = \text{if } Px \text{ then } y \text{ else } g(Fx,Gy)$$

Proof 2

We want to prove $Gf(Fx,y) = f(Fx,Gy)$

define $h = \lambda x.\lambda y.$ if Px then Gy else $Gh(Fx,y)$

$$Gf(Fx,y) \equiv G(\text{if } Px \text{ then } y \text{ else } Gf(FFx,y))$$

$$= \text{if } Px \text{ then } Gy \text{ else } GGf(FFx,y)$$

$$= \mathcal{H}(\lambda x.\lambda y.\ Gf(Fx,y))(x,y)$$

$$f(Fx,Gy) = \text{if } PFx \text{ then } Gy \text{ else } Gf(FFX,Gy)$$

$$= \mathcal{H}(\lambda x.\lambda y.\ f(Fx,Gy))(x,y)$$

2.5 Recursion Induction and Structural Induction

Since, as seen in section 2.2, structural induction and truncation induction are so similar, the relation between recursion induction and structural induction should not be very different from that between recursion induction and truncation induction with respect to the appropriate class of programs. There are a few additional observations which can be made, however, about Burstall's treatment of these methods in his paper.

Burstall defines structural induction first as induction on complexity of data structures for which there are constructor functions, selectors and decidable predicates. He then explains that in its full generality structural induction actually applies to programs on any domain which satisfies the minimum condition and for which the computations proceed in accordance with the decreasing order of arguments. We have used these definitions in this general sense in many of our examples. Burstall does not, however, really explore structural induction in its fullest generality in his paper. Every one of his examples is of the type (corresponding to those of section 2.4) which translates simply into a recursion induction proof. The induction is always simple, taking advantage of only one stage in the fixpoint construction (or in the structural ordering). It seems that his conception of structural induction was really quite a bit narrower than his own definitions indicate, as he only makes use of one level of recursion in each proof. This would explain why, as Burstall mentions without details at the start of his paper, McCarthy was able to give a very simple and convincing argument that structural induction is just a special case of recursion induction. As Burstall uses it, that is exactly what it is, a straightforward rewriting of the proof being enough to change from proof by one method to the other. While the transformation is possible, in general it is not a mechanical process, but depends heavily on the extent to which the structural induction proof was restricted to forms closely compatible with recursion induction. By the parallels drawn between structural induction and truncation induction we know that a structural induction proof can depend on

application of the inductive hypothesis at lower levels of recursion and that some study may be required to draw out the computational structure and associated conditional forms relevant to a recursion induction proof.

To illustrate let's look at two proofs of $f(x,y) = g(x,y)$ by structural induction where f and g are the following interpreted versions of our familiar example.

$$f(x,y) \equiv \text{if } x = 0 \text{ then } y \text{ else } f(x-1,y) + 1$$

$$g(x,y) \equiv \text{if } x = 0 \text{ then } y \text{ else } g(x-1,y+1)$$

The ordering on the domain is $(x_1,y_1) \textcircled{<} (x_2,y_2)$ iff $x_1 < x_2$
The first proof is by structural induction with preference for style of recursion induction

$$f(x,y) \equiv \text{if } x = 0 \text{ then } y \text{ else } f(x-1,y) + 1$$

$$= \text{if } x = 0 \text{ then } y \text{ else } g(x-1,y) + 1$$

(by induction)

$$g(x,y) \equiv \text{if } x = 0 \text{ then } y \text{ else } g(x-1,y+1)$$

$$= \text{if } x = 0 \text{ then } y \text{ else } g(x-1,y) + 1$$

(by proof of $g(x,y) + 1 = g(x,y+1)$

$$g(x,y) + 1 = \text{if } x = 0 \text{ then } y+1 \text{ else } g(x-1,y+1)+1$$

$$= \text{if } x = 0 \text{ then } y+1 \text{ else } g(x-1,y+2)$$

(induction)

$$g(x,y+1) = \text{if } x = 0 \text{ then } y + 1 \text{ else } g(x-1,y+2)$$

This second proof of $f(x,y) = g(x,y)$ is carried out just as in the uninterpreted case by truncation induction

$$f(x,y) \equiv \text{if } x = 0 \text{ then } y \text{ else } f(x-1,y) + 1$$

$$= \text{if } x = 0 \text{ then } y \text{ else } g(x-1,y) + 1$$

$$= \text{if } x = 0 \text{ then } y \text{ else if } x - 1 = 0 \text{ then } y+1$$

$$\text{else } g(x-2,y+1) + 1$$

= if x = 0 then y else if x - 1 = 0 then y + 1 else

f(x-2,y+1) + 1

= if x = 0 then y else f(x-1,y+1)

= if x = 0 then y else g(x-1,y+1)

= g(x,y)

This proof uses the induction hypothesis f(x',y') = g(x',y') for all (x',y') $\textcircled{<}$ (x,y) more generally than the previous proof did. The first proof clearly lends itself more easily to reinterpretation as a recursion induction proof.

As we said before, no proof of Burstall's in illustration of his definitions uses the induction hypothesis more than once. It seems that if he had to describe his conception of recursion it would be more like a μ-rule theory explanation than like a truncation induction one. The reason why he still needs to define his rule with course-of-values type induction rather than simple induction is that his only requirement of the computation with respect to the structure is that successive calls have less complex arguments, but this decrease in complexity is not restricted to comply with some notion of one stage in structure composition. So although Burstall probably didn't envisage uses of induction at more than one computation level, he did see the need for course-of-values induction due to the kind of programs he was concerned with.

Chapter 3. Some Comments on Program Verification

An automatic program verifier is a system for proving correctness of programs. Assuming that the programmer has use of a language for indicating the intended meaning of his program, the proof of correctness will consist of a proof that the program does indeed realize this intention. If this statement of intention is viewed as a second program, then the proof is in fact a proof of the equivalence of this program and the original one. Thus it is very likely that the techniques developed for proving equivalence of functions will be useful in work on program verification.

A program verifier will have to draw on several sources for its power. It is a theorem prover, the theorems it proves being in the form of equalities between a function definition and certain input-output specifications. To prove these theorems, the verifier must have a store of facts -- axioms and induction rules, perhaps some previously proved theorems -- to work with. These facts will be roughly of two sorts, some about the subject matter of the program's computations, and others about the abstract structure of computations. For example, a program verifier for geometry programs will be capable of proving theorems in geometry, while a verifier for arithmetic programs will be capable of proving theorems about numbers. But regardless of interpretation of the programs, if a program involves recursion, or iteration, in its computations the verifier will have to encompass a theory of computation with some rule of inference for dealing with the recursive computation structure. It is the need to get a handle on this part of the verifying process which has led to the study of uninterpreted schemas and which justifies their abstraction from the more concrete interpreted programs. If you look again at any of the interpreted function definitions which we have used for examples, you will see the obvious distinction between the steps justified by a theory of recursive schemas and those justified by the theory of the field of interpretation. An automatic verifier would have to handle both types of inferences to complete a proof.

Only the theories of computational structure as implied by certain rules of inference are being examined here. Since one of the chief goals of the theory of computation is finding means for automatic program verification, along with means of analysis or debugging aids to correct programs which are found to be incorrect, it is of interest to evaluate the methods we have been studying in terms of their applicability to the achievement of this goal. Some of the qualities desirable in a proof technique can be formulated now, though it is still a matter of speculation as to which will be most significant in determining final choices. To maximize the information gained from verification, the structure of the proof should reflect the structure of the computation. Should the proof fail to go through, it would be helpful to be able to identify the particular part of the function definition which diverged from the intention. And, of course, it must be possible to carry out the proof mechanically, or with some limited amount of help from the programmer. It is interesting at this point to see how the language and techniques defined in previous chapters hold up in light of these criteria.

First some thought must be given to how to write programs and correctness conditions. Programming style will undoubtably be influenced by decisions about how a program verifier is to be constructed. For example, if a programmer has written a long program which is supposed to compute a known mathematical function he might give the verifier his program and the function as stated concisely in the notation of mathematics. However, the verifier may need some help in understanding how the various sections of the program actually contribute to the computation of the function. Rather than expecting to find grounds for comparison between a function definition as an indivisible unit and a concise formulation of the intention, it will be more reasonable to try to indicate conditions for sections of the program. These sections will be shown to satisfy their subconditions and then by considering the interaction of these sections, which are already known to be correct, the entire program will be proved correct. This type of commenting on the program will most likely coincide with the writer's own

understanding of the program. While in many cases the programmer could state some concise relation between input and output representing his intentions for the operation of the program, in a program of any length or complexity he will understand the program as an interaction of subprograms each satisfying a condition necessary for meeting his final intention. These expressions of his intuitions into the workings of a computation by the program will then serve not only as justifications of the entire program but of all its parts. It will also make it easier to prove correctness of a slightly modified version of the program, since now it is sufficient to combine proof of correctness of a small part with the known correctness of the original program.

It may be necessary, given information about solvability of equivlence problems for different classes of programs, for a programmer to learn to think differently about programs, molding his approach to solving problems to the form of the language in which he will eventually state the solution. For example, the use of goto statements is known to make proof of equivalence of programs more difficult. However there are no programs written using goto statements, which cannot be rewritten without them. The program writing language used throughout this thesis, namely conditional statements, is powerful enough to express any program which could be written with goto expressions (even more powerful [10]). Many people originally learn to program in Fortran, or some such language, using goto's, and consequently feel that the goto is too useful to sacrifice. Yet those who learn to use a language like Lisp, find that they very quickly adopt a new approach to solving problems as the most natural. It is likely that as the prospects for having automatic program verifiers become better, through greater facility with proof techniques, the cost of this sort of restriction of programming style will begin to seem small as compared to the possible advantages to be had if a program verifier could be implemented.

We can find in the examples worked in previous chapters some cases illustrating the degree to which a proof may or may not reflect the more

obvious intuitive arguments for equivalence of definition. Clearest is the comparison of the several proofs of equivalence of

$$f(x,y) \equiv \text{if } Px \text{ then } y \text{ else } Gf(Fx,y)$$

$$g(x,y) \equiv \text{if } Px \text{ then } y \text{ else } g(Fx,Gy)$$

As noted earlier, the equivalence of f and g is most simply explained by the fact that in one case Gy is computed at each step in the evaluation and in the other the number of such computations is counted and all of them are done at the end. This explanation forms the basis for a recursion induction proof (and seems to be the most obvious choice for proceeding by μ-rule), but is completely lost in the truncation induction proof given in Morris [7] (Example 1 of section 2.3). For the purposes of following the writer's reasoning as closely as possible, allowing his comments to aid in proof, and of breaking up the proof into more manageable subproofs (examination of the proofs will reveal that this is another resulting difference in form), the former methods seem preferable.

It is possible, however, that efforts to mechanize this process will show reason to favor truncation induction. It may not be reasonable to expect to reflect human understanding directly in a computer system. Then a system which works blindly to the same end may be the only choice. The fact that with truncation induction a proof of the equivalence of f and g can be obtained without any real understanding of the functions may turn out to be a very good reason for choosing truncation induction over other methods. A basis for mechanical proof might be some generalization of the following simple non-deterministic algorithm by which the reader should be able to show the equivalence of f and g as defined above and as proved in section 2.3.

0. Defn. 1. $f_i(x,y) \equiv \mathcal{F}(f_{i-1})(x,y)$

   Defn. 2. $g_i(x,y) = \mathcal{G}(g_{i-1})(x,y)$

1. To prove $f_i(x,y) = g_i(x,y)$

   assume $f_j(x,y) = g_j(x,y)$ for all $j < i$

   Start with $f_i(x,y) \Rightarrow X$

2. If $X = g_i(x,y)$ go to End else go to 3a, 3b, 3c, 3d, 3e, or 3f.

3a. If X contains $\mathcal{Y}(f_j)(x,y)$ for $j < i$, substitute $f_{j+1}(x,y)$.
Go to 2.

3b. If X contains $\mathcal{Y}(g_j(x,y))$ for $j < i$ substitute $g_{j+1}(x,y)$.
Go to 2.

3c. If X contains f_j substitute $\mathcal{Y}(f_{j-1})(x,y)$.
Go to 2.

3d. If X contains g_j substitute $\mathcal{Y}(g_{j-1})(x,y)$.
Go to 2.

3e. If X contains f_j $j < i$ substitute g_j.
Go to 2.

3f. If X contains g_j $j < i$ substitute f_j.
Go to 2.

Assuming that by pattern matching the test for containment of a certain form in the arbitrary form X can be performed (distribution of functions over conditions, and such operations would have to be included), this algorithm will go through all possible sequences of valid inferences until it finds the desired identity. The sequence

0, 1, 2, 3c, 2, 3e, 2, 3d, 2, 3f, 2, 3a, 2, 3e, 2, 3b, 2, End.

corresponds to the proof of section 2.3.

If our programs are meant to operate on structured data, then structural induction will be a very natural tool to use. Probably anyone writing a program with a structure in mind is thinking in terms of performing operations on successive substructures -- this is the kind of thinking which develops with a language like Lisp. Also, in practice, a programmer will not want to write a program which might run forever for some input. If he is not sure that he is performing a calculation guaranteed to halt, then he will impose some artificial constraint on the program forcing halting in all cases. Therefore, any real program will have an ordering on its input, forcing convergence on a known domain. We could then say that for all practical purposes it is only necessary to

handle total functions, and perhaps then structural induction will turn out to be most useful.

In some cases, it may be necessary for the theorem prover for the field of interpretation to check that this ordering is indeed one which satisfies the minimum condition, a consideration outside the range of uninterpreted schematology. In a practical debugging system, however, we would want to check for the possibility that the programmer had erred in writing in a way which might allow infinite computation for some arguments. For this reason it might be advisable to carefully separate proof of equivalence where defined and proof of convergence, making a proof technique which assumes nothing about convergence the more desirable.

We are not prepared here to judge which will be the overriding criteria for the choice of method of automatic program verification, but hope we have indicated some of the possible bases for judging these methods, perhaps some justification for the independent study of program schema, and areas for future work.

Chapter 4. Conclusion

Of the four methods, recursion induction alone imposes any real restrictions on the kinds of steps we can take in proofs. This is due to the fact that the view of function definition taken by the user of this method (whether he knows it or not) is that a definition is a monotonic mapping, but may or may not be continuous. Therefore, he ignores some of the information that could be used to compare these functions, particularly the alternative definition of fixpoint, available for monotonic, continuous functions. The intermediary goals to be set up for performing a proof will be in the forms both of desired subproofs and in the discovery of alternative conditional forms which represent the common computation structure of the functions being compared. After finding a valid proof some unsettled questions about domains of function might be left when they could have been avoided with other techniques.

As for the three other methods, we have seen that while all three make identical claims about functions, they do so from such varying points of view as to effectively become different techniques. For someone working with structured data, structural induction provides proof justifiable on the basis of facts about the data apparently requiring no insight into the computation of recursively defined functions. The user of the μ-rule is more likely to break down his proofs into subproofs which make explicit all the identities he is actually proving, while the user of truncation induction, who presumably understands what the fixed point of a recursive definition looks like, will probably carry on a chain of expansions by definition and calls on the induction hypothesis, losing sight of subgoals as he proves the primary hypothesis.

We find ourselves then, with four seemingly different techniques -- and they are different in the following sense: For any particular problem one of these views will probably seem more natural or illuminating than the others and will facilitate a proof by induction. This proof, however, could in principle be carried out by any one of these methods.

Once a thorough understanding of the available methods of proof is obtained, the programmer will find it to his advantage to plan ahead for times when he may need to prove facts about his programs. The way he originally writes a program may affect his choice of proof techniques later. The development of mechanical program verifiers will require restrictions on the style of programming to fit the method of verification. In the preceding chapters it was seen both how the choice of method can eventually influence a programmer's conception of recursion and how the interpretation of recursive definitions, or of the domains of recursive programs, may influence his choice of proof technique. Future work in program verification may further influence and be influenced by these choices.

Appendix

A. Theorem: If c monotone

then

1. $c(conv(c)) = conv(c)$
2. $conv(c) = \cap \{m \mid c(m) = m\}$.

Proof:

1. a. Suppose $c(m) \subseteq m$, then $conv(c) \subseteq m$ (section 1.4 (2)). $c(conv(c)) \subseteq c(m)$ since c monotone, but $c(m) \subseteq m$ by hypothesis, so $c(conv(c)) \subseteq m$. This holds for all $m \in \{m \mid c(m) \subseteq m\}$

   $\therefore\ c(conv(c)) \subseteq \cap \{m \mid c(m) \subseteq m\}$

   $= conv(c)$.

   b. From a, $c(conv(c)) \subseteq conv(c)$

   $\therefore\ c(c(conv(c))) \subseteq c(conv(c))$

   since c monotone

   so $conv(c) \subseteq c(conv(c))$ (by 1.4 (2)).

2. a. From defn $conv(c) = \cap \{m \mid c(m) \subseteq m\} \subseteq \cup \{m \mid c(m) = m\}$

   b. From 1 $conv(c) = c(conv(c))$

   $\therefore\ conv(c) \in \{m \mid c(m) = m\}$ so $conv(c) \supseteq \cap \{m \mid c(m) = m\}$

B. The following, suggested by Paterson, is a definition outside the class $C(\mathcal{G})$ as an example of a non-continuous function.

$\mathcal{C}(f) = \lambda x.$ if $x = 1$ then if f defined for all even numbers then 1

else undefined

else undefined

so $\mathcal{C}(f)$ is a function g, undefined everywhere except possibly at $x = 1$ and $\mathcal{C}(f)(1) = 1$ iff $f(x)$ defined for all even x.

Let us try to apply the test for continuity that for all

$$m_1 \subseteq m_2 \subseteq \ldots$$

$$\mathcal{C}\left(\bigcup_{i=1}^{\infty} m_i\right) = \bigcup_{i=1}^{\infty} \mathcal{C}(m_i)$$

Look at the series of functions

$$\emptyset \subseteq \{(0,1)\} \subseteq \{(0,1)(2,1)\} \subseteq \{(0,1),(2,1),(3,1)\} \subseteq \ldots$$

then $\mathcal{C}\left(\bigcap_{i=1}^{\infty} f_i\right) = \{(1,1)\}$

while $\bigcup_{i=1}^{\infty} \mathcal{C}(f_i) = \emptyset$.

References

[1] Burstall, R., Proving Properties of Programs by Structural Induction. Computer Journal 12, 1, pp 41-48.

[2] deBakker, J. W. and Scott, D., A Theory of Programs. IBM Seminar, Vienna, August 1969.

[3] Floyd, R. W., Assigning Meaning to Programs. Proc. Symp. in Applied Math., 19, Math. Aspects of Comp. Sci. (ed. Schwartz), Amer. Math. Soc., Providence, R. I., pp. 19-32.

[4] Manna, Z., and Pnueli, A., Formalization of Properties of Recursively Defined Functions. ACM Symposium on Theory of Computing, Marina Del Rey, May 5-7, 1969.

[5] McCarthy, J., A Basis for a Mathematical Theory of Computation. Computer Programming and Formal Systems, North-Holland, Amsterdam, pp 33-70.

[6] McCarthy, J., and Painter, J., Correctness of a Compiler for Arithmetic Expressions. Stanford Artificial Intelligence Project Memo 40.

[7] Morris, J., Another Recursion Induction Principle, CACM 14, 5.

[8] Park, D., Fixpoint Induction and Proofs of Program Properties. Machine Intelligence 5, pp 59-78.

[9] Paterson, M. S., Equivalence Problems in a Model of Computation. Dissertation at Cambridge.

[10] Paterson, M. S., and Hewitt, C., Comparative Schematology. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, June 2-5, 1970.

CS-TR Scanning Project
Document Control Form

Date : 1/23/96

Report # LCS-TR-93

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 60(65-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☒ Single-sided or
☐ Double-sided

Print type:
☒ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
(A) IMAGE MAP: (1-60) UNPAD TITLE PAGE, 2-60 (61-65) SCANCONTROL DOD, TRGT'S(3)	
(B) A NUMBER OF PAGES HAVE CUT&PASTE WORDS AND OR SENTENCES.	

Scanning Agent Signoff:
Date Received: 1/23/96 Date Scanned: 1/26/96 Date Returned: 1/26/96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Massachusetts Institute of Technology Project MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP NONE

3. REPORT TITLE

Induction in Proofs About Programs

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

S. M. Thesis, Department of Electrical Engineering, December 1971

5. AUTHOR(S) *(Last name, first name, initial)*

Greif, Irene G.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
February 1972	60	10

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
N00014-70-A-0362-0001 b. PROJECT NO. c. d.	MAC TR-93 (THESIS) 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D. C. 20301

13. ABSTRACT

Four methods for proving equivalence of programs by induction are described and compared. They are recursion induction, structural induction, μ-rule induction, and truncation induction. McCarthy's formalism for conditional expressions as function definitions is used and reinterpreted in view of Park's work on results in lattice theory as related to proofs about programs. The possible application of this work to automatic program verification is commented upon.

14. KEY WORDS

Recursion Induction
Proofs About Programs

DD FORM 1 NOV 68 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 1/25/1996

M.I.T. Libraries
Document Services