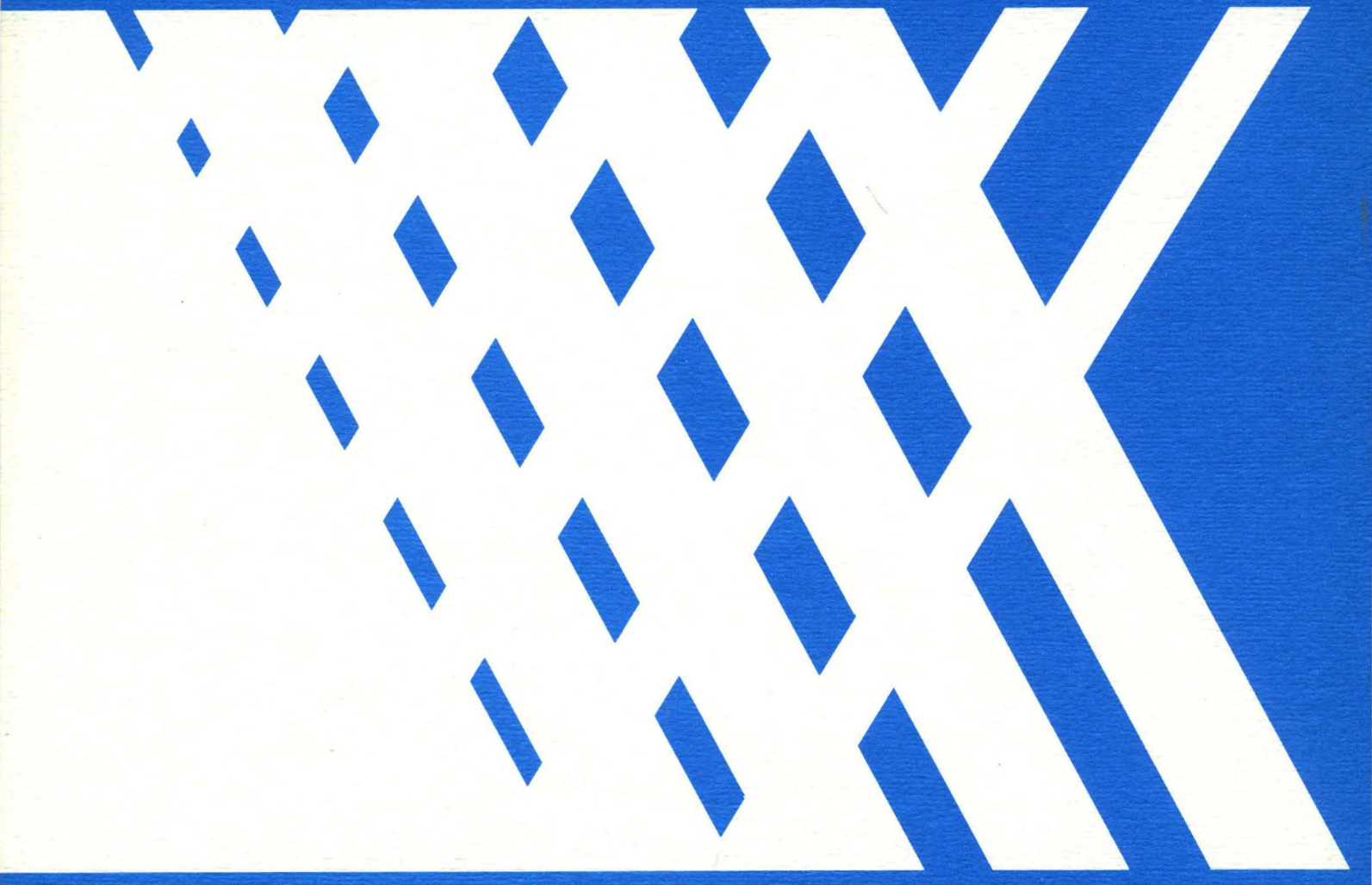
SUBGOAL INDUCTION

BY JAMES H. MORRIS JR. AND BEN WEGBREIT



XEROX
PALO ALTO RESEARCH CENTER

SUBGOAL INDUCTION

BY JAMES H. MORRIS JR. AND BEN WEGBREIT

CSL 75-6 JULY 1975

A new proof method, subgoal induction, is presented as an alternative or supplement to the commonly used inductive assertion method. Its major virtue is that it can often be used to prove a loop's correctness directly from its input-output specification without the use of an invariant. The relation between subgoal induction and other commonly used induction rules is explored and, in particular, it is shown that subgoal induction can be viewed as a specialized form of computation induction. Finally, a set of sufficient conditions are presented which guarantee that an input-output specification is strong enough for the induction step of a proof by subgoal induction to be valid.

Key Words and Phrases:

program verification, proving programs correct, induction rule, computation induction, inductive assertions, structural induction, proof rule, recursive programs, iterative programs

CR Categories:

4.19, 4.22, 5.21, 5.24

XEROX
PALO ALTO RESEARCH CENTER
3333 COYOTE HILL ROAD / PALO ALTO / CALIFORNIA 94304

INTRODUCTION

A variety of induction rules have been employed in program verification. Recursion induction [8] was the first, followed by structural induction [3,9], inductive assertions [4], and computation induction [11]. In this paper, we present a new induction rule, *subgoal induction*, which has several desirable properties: it is applicable to recursive as well as iterative programs; it is straightforward to use in mechanical program verification; and it leads to relatively simple proofs. We expand on these points below.

Similar proof methods have appeared in [1,6,10,13,15]. In this presentation, we have tended to emphasize programming methodology as well as theoretical issues. Thus our exposition includes both informal discussions and formal proofs.

THE RULE OF SUBGOAL INDUCTION

We begin by presenting and motivating subgoal induction in a hopefully intuitive fashion--by coupling the induction rule with program synthesis. That is, we consider the construction of a program and a proof of its correctness simultaneously. Let the program be specified by the requirement that given input x it is to produce output z such that $\Psi(x;z)$ for some given predicate[1] Ψ. We propose to construct a recursive program F, as follows. For certain x, an appropriate z can be computed using a previously defined function; let P(x) test for those x and H(x) be the previously defined function; this leads to the fragment:

if P(x) **then** H(x)

and the verification condition

(V1) $P(x) \rightarrow \Psi(x;H(x))$

---

[1] *For emphasis, we use the delimiter ";" to separate the input variable(s) from the output value. Thus* $\Psi(a,b,c;z)$ *relates input variables a, b, and c to the output z.*

For all other x, we propose to replace x by a somewhat simpler value, attempt to solve the problem for that simpler value, and then modify that solution to obtain a solution for x; let N be the function which maps x into the simpler value and L be the function which modifies the solution for N(x) in order to obtain a solution for x. This leads to the fragment

else L(x, F(N(x)))

The corresponding verification condition is

(V2) $\neg P(x) \wedge \Psi(N(x);z) \rightarrow \Psi(x;L(x,z))$

This may be read as: if ~P(x) and if z is an acceptable output for F with input N(x) then L(x,z) must be an acceptable output for F with input x. The program is then

(1.1) F(x) <= **if** P(x) **then** H(x) **else** L(x, F(N(x)))

The above reasoning is an informal demonstration that if the two verification conditions V1 and V2 are valid, then $\Psi(x;F(x))$ is valid. This is an example of the rule of *subgoal induction*.

The above argument tacitly assumes that F(N(x)) actually returns a value. If, to the contrary, F(N(x)) fails to terminate then F(x) fails to terminate. Proof of termination may be treated separately, e.g., based upon a well-ordering. In this paper, we will generally be concerned with *partial correctness*, i.e., consider only those cases where F(x) terminates. A more precise statement of subgoal induction reads: If V1 and V2 are valid, then for each x if F(x) terminates $\Psi(x;F(x))$ is true. We expand on this point and give a precise justification below.

Observe that V2 is a stronger requirement than actually needed to establish $\Psi(x; F(x))$. It would, in principle, suffice to prove the somewhat weaker implication

(V2*) $\neg P(x) \wedge \Psi(N(x);z) \wedge z = F(N(x)) \rightarrow \Psi(x;L(x,z))$

This may be read as: If ~P(x) and if z is the output of F(N(x)) and if the pair $\langle N(x),z\rangle$ is Ψ-acceptable then the pair $\langle x,L(x,z)\rangle$ must be Ψ-acceptable. V2* follows directly from expanding the definition of F.

It differs from V2 in that its hypothesis includes the additional conjunct $z = F(N(x))$. The essential idea of subgoal induction is the absence of this conjunct: *Provided that Ψ is a strong enough specification, z is adequately constrained by the requirement that $\Psi(N(x);z)$ be true.* In such cases, the conjunct $z = F(N(x))$ is unnecessary, and V2 is a valid theorem which, taken together with V1, establishes $\Psi(x;F(x))$. We will later discuss in detail and illustrate with examples the conditions under which Ψ is a strong enough specification for this to work. Here it suffices to observe a result we prove below: V1 and V2 are valid if and only if the verification conditions for computation induction [11] are valid.

Example 1 (Subgoal Induction)

Let "/" denote integer division (with truncation) and define

$E(x) \Leftarrow$ **if** $x = 0$ **then** 1 **else** $L(x, E(x/2))$, and

$L(x,y) \Leftarrow$ **if** $\text{even}(x)$ **then** y^2 **else** $K \cdot y^2$.

The property to be proved is $\Psi(x;E(x))$ where $\Psi(x;z) \equiv z = K^x$. The verification conditions are:

(V1) $x=0 \rightarrow 1 = K^x$ (trivial)

(V2) $x \neq 0 \wedge z = K^{x/2} \rightarrow L(x,z) = K^x$

Expanding L, this is $x \neq 0 \wedge z = K^{x/2} \rightarrow$ **if** $\text{even}(x)$ **then** $z^2 = K^x$ **else** $K \cdot z^2 = K^x$ which may be proved by cases on $\text{even}(x)$. ∎

In the case where L and H are the identity functions, the recursive function F defined in (1.1) is equivalent to the following **while** scheme

while $\sim P(x)$ **do** $x \leftarrow N(x)$

Let x_0 and x_f be the initial and final values of x in this scheme. To establish $\Psi(x_0;x_f)$ by subgoal induction, we must prove

(V1) $P(x) \rightarrow \Psi(x;x)$

(V2) $\sim P(x) \wedge \Psi(N(x);z) \rightarrow \Psi(x;z)$

Note that the idea of loop invariant does not appear. The output assertion Ψ need not be true within the loop and likely is not (otherwise we're wasting iterations!). In a sense, subgoal induction

allows the output assertion or intention of the loop to be used directly in its own proof.

Example 2 (Subgoal Induction on a **while** Loop)

Consider the well-known iteration for finding the greatest common divisor of two positive integers

while $x \neq y$ **do if** $x<y$ **then** $y \leftarrow y-x$ **else** $x \leftarrow x-y$

Suppose one wishes to prove just that the final value of x divides the original values of x and y. The output assertion is

$\Psi(x_0, y_0; x_f) \equiv x_f$ divides $x_0 \wedge x_f$ divides y_0

Let ξ be the state vector. To prove Ψ by subgoal induction , two verification conditions must be established. The first is .

(V1) $P(\xi) \rightarrow \Psi(\xi;\xi)$

where $\xi=\langle x,y\rangle$ and $P(\xi) \equiv x=y$. This becomes

$x=y \rightarrow x$ divides $x \wedge x$ divides y.

The second verification condition is

(V2) $\sim P(\xi) \wedge \Psi(N(\xi);z) \rightarrow \Psi(\xi;z)$

where $N(\xi)$ = **if** $x<y$ **then** $\langle x,y-x\rangle$ **else** $\langle x-y,y\rangle$.

This becomes

$x \neq y \wedge$

z divides (**if** $x<y$ **then** x **else** $x-y$) $\wedge$

z divides (**if** $x<y$ **then** $y-x$ **else** y) $\rightarrow$

z divides $x \wedge z$ divides y

(V1) is immediate and (V2) follows easily from the general observation that

z divides $a \wedge z$ divides $b \rightarrow z$ divides $a+b$

Notice that it was not necessary to invent an invariant assertion to prove Ψ; Ψ itself was a sufficient inductive hypothesis.

It happens that the following is a good invariant

$(\forall z)$ [z divides $x \wedge z$ divides $y \rightarrow z$ divides $x_0 \wedge z$ divides y_0].

However, it appears that a proof by inductive assertions will be more complicated than the foregoing proof. ∎

Because **while** loops are a commonly occurring syntactic form to which subgoal induction is directly applicable, it is appropriate to introduce a simple notation for their output assertions. We propose

while ⟨Boolean expression⟩ **do** ⟨statement⟩ **thus** ⟨output assertion⟩

The ⟨output assertion⟩ is to be true on exit from the **while** loop. It uses the following notation: If x is a free variable, then x_0 (read as "original x") is an initial value of the corresponding program variable x; other free appearances of x are understood to denote the final value of the corresponding program variable. Hence, the above program may be written

while $x \neq y$ **do if** $x<y$ **then** $y \leftarrow y-x$ **else** $x \leftarrow x-y$
thus x divides $x_0 \wedge$ x divides y_0

The extension of this notation to **for** loops and multiple-exit **while** loops should be clear.

Subgoal induction can be applied to more general program structures involving nested recursive calls and mutually recursive functions. For example, consider

$F(x) \Leftarrow$ **if** $P_0(x)$ **then** $L_0(G(N_0(x)))$
else if $P_1(x)$ **then** $L_1(G(N_2(F(N_1(x)))))$
else $L_2(F(N_3(x)), F(N_4(x)))$

$G(y) \Leftarrow$ **if** $P_5(y)$ **then** $H(y)$ **else** $F(y)$

with output assertions $\Psi_F(x;F(x))$ and $\Psi_G(y;G(y))$. To form the verification conditions for proof by subgoal induction, three additional concepts are required. We state and illustrate these in turn.

(1) *Mutual recursion is handled by using the output predicate for the called function in forming the verification condition for the calling function.* Thus, the verification condition for the first path through F is:

(FV1) $P_0(x) \wedge \Psi_G(N_0(x);z_G) \rightarrow \Psi_F(x;L_0(z_G))$

This is obtained as follows: let F be called with argument x and suppose that $P_0(x)$; to compute F's value, G is called with argument $N_0(x)$; let that result be z_G subject to the constraint $\Psi_G(N_0(x);z_G)$; the result of F is $L_0(z_G)$; to prove F correct, we must be able to show that $\Psi_F(x;L_0(z_G))$.

(2) *Multiple function calls on distinct functions are handled by introducing additional individual variables--one for each called function.* Thus, the verification condition for the second path through F is:

$$(FV2)\quad \sim P_0(x) \wedge P_1(x) \wedge \Psi_F(N_1(x);z_F) \wedge \Psi_G(N_2(z_F);z_G) \rightarrow \Psi_F(x;L_1(z_G))$$

This is obtained as follows: let F be called with argument x and suppose that $\sim P_0(x)$ and $P_1(x)$; to compute F's value, the first step is to call F recursively with argument $N_1(x)$; let that result be z_F subject to the constraint $\Psi_F(N_1(x);z_F)$; next, G is called with $N_2(z_F)$; let that result be z_G subject to the constraint that $\Psi_G(N_2(z_F);z_G)$; the result of the original call on F is $L_1(z_G)$; to prove F correct, we must be able to show that $\Psi_F(x;L_1(z_G))$.

(3) *Multiple calls on the same function are handled by introducing additional individual variables, subject to the constraint of the output assertion and the further constraint that when the arguments to the function are equal, then the outputs are equal.* Thus the verification condition for the third path through F is:

$$(FV3)\quad \sim P_0(x) \wedge \sim P_1(x) \wedge \Psi_F(N_3(x);z_3) \wedge \Psi_F(N_4(x);z_4) \wedge (N_3(x)=N_4(x) \rightarrow z_3=z_4) \rightarrow \Psi_F(x;L_2(z_3,z_4))$$

This is obtained as follows: let F be called with argument x and suppose that $\sim P_0(x)$ and $\sim P_1(x)$; to compute F's value, F is recursively called twice, with arguments $N_3(x)$ and $N_4(x)$; let the results be z_3 and z_4 respectively, subject to the constraints $\Psi_F(N_3(x);z_3)$, $\Psi_F(N_4(x);z_4)$ and the further constraint that if $N_3(x)$ equals $N_4(x)$ then z_3 equals

z_4; to prove F correct, we must be able to show that the final result, $L_2(z_3,z_4)$, satisfies $\Psi_F(x;L_2(z_3,z_4))$.

The two verification conditions for the two paths through G are obtained analogously:

(GV1) $P_5(y) \rightarrow \Psi_G(y;H(y))$

(GV2) $\sim P_5(y) \wedge \Psi_F(y;z_F) \rightarrow \Psi_G(y;z_F)$

We now turn to a more precise statement of the rule of subgoal induction. Let F be defined by

$F \Leftarrow \tau[F]$

where τ is a first-order program, i.e., all appearances of F are as the operator in a function call. Then to prove

$\Psi(x;F(x))$

it suffices to prove

$$(SGI) \quad \bigwedge_{i=1}^{n} \Psi(\gamma_i;z_i) \wedge \bigwedge_{1\le i,j\le n} (\gamma_i=\gamma_j \rightarrow z_i=z_j) \rightarrow \Psi(x;\beta)$$

where β is obtained from τ by replacing, *inside out*, each call on F, $F(\gamma_i)$, with a new individual variable z_i. Since τ is assumed to be first order, this eliminates all appearances of F from τ. Hence (SGI) is entirely free of F provided that the definition of Ψ, $\Psi(x;z)$, is free of F.

In practice, it is useful to split (SGI) into cases--one for each path through the program. Consider the schema

$F(x) \Leftarrow$ **if** $P_1(x)$ **then** $\alpha_1(x)$ **else if** $P_2(x)$ **then** $\alpha_2(x)$ ... **else** $\alpha_n(x)$

where the α's are arbitrary expressions possibly involving F. Subgoal induction may be used to directly compile one verification condition for each of the n cases in the definition. The consequent of the i^{th} term is $\Psi(x;\beta_i)$ where β_i is obtained from α_i by replacing, inside out, each call on F, i.e., $F(\gamma_i)$, with a new individual variable. For each

replacement thus made, a constraint term $\Psi(\gamma_i;z_i)$ is formed which forces z_i to be an acceptable output for F on input γ_i. The antecedent of the verification condition is the conjunction of the constraint terms and the path conditions, $\{P_i\}$. The i^{th} verification condition is

$$\bigwedge_{j=1}^{i-1} \sim P_j(x) \wedge P_i(x) \wedge \bigwedge_{j=1}^{k} \Psi(\gamma_j;z_j) \wedge \bigwedge_{1\le j,m\le k} (\gamma_j=\gamma_m \rightarrow z_j=z_m) \rightarrow \Psi(x;\beta_i)$$

An example may clarify how the β_i's are obtained.

Example 3 (Nested Recursive Function Calls)

The following function flattens an S-expression x, in the sense that it creates a one-level list whose elements are the atoms of x in print order, appended to y.

F(x,y) <= **if** atom(x) **then** cons(x,y) **else** F(car(x), F(cdr(x), y))

We wish to show that F(x,NIL) is identical to the result of a simpler procedure G, which flattens a list in a slower but more obvious way, as follows:

G(x) <= **if** atom(x) **then** cons(x,NIL) **else** append(G(car(x)), G(cdr(x))).

The auxiliary function append is defined

append(x,y) <= **if** null(x) **then** y **else** cons(car(x),append(cdr(x),y)).

The output assertion for F is $\Psi_F(x,y;z) \equiv \{z = \text{append}(G(x), y)\}$. The verification conditions are

(V1) atom(x) $\rightarrow$ cons(x,y) = append(G(x), y)

which is obviously valid, upon expansion of G and append.

(V2) $\sim$atom(x) $\wedge$ z_1 = append(G(cdr(x)), y) $\wedge$
z_2 = append(G(car(x)), z_1) $\rightarrow$
z_2 = append(G(x), y)

Substituting for equals and expanding G for a non-atomic argument, this simplifies to

$$\sim atom(x) \rightarrow append\{append(G(car(x)), G(cdr(x))), y\} = append\{G(car(x)), append(G(cdr(x), y)\}$$

which is an instance of a simple fact about append -- append(append(u,v), w) = append(u, append(v,w)), i.e., that append is associative. ∎

A general statement of subgoal induction for mutually recursive functions can now be presented. Let

$$F_1 <= \tau_1[F_1], \ \ldots, \ F_n <= \tau_n[F_n]$$

be a system of mutually recursive functions. Suppose we are given output specifications for the first $m \leq n$ of these

$$\Psi_1(x;F_1(x)), \ \ldots, \ \Psi_m(x;F_m(x))$$

Then to prove $\Psi_1(x, F_1(x))$, it suffices to prove

$$[\Psi_{i_1}(\gamma_1;z_1) \wedge \ldots \wedge \Psi_{i_k}(\gamma_k;z_k) \wedge \bigwedge_{1\leq p,q\leq k} (\gamma_p=\gamma_q \rightarrow z_p=z_q)] \rightarrow \Psi_1(x;\beta_1)$$

where β_1 is obtained from τ_1 by replacing, inside out, each function call

$$F_{i_j}(\gamma_j)$$

by z_j (where $1\leq i_j\leq m$). Provided that the Ψ's are free of F_j's ($1\leq j\leq m$), the resulting verification conditions will be free of such F_j's. Thus, *subgoal induction may be seen as a systematic way of using output specifications to eliminate function letters from theorems to be proved.*

RELATION TO OTHER PROOF METHODS

3.1 Inductive Assertions

Subgoal induction is, in a sense, a backward method of proving

something compared to the inductive assertion method. The two methods correspond to two different ways of expanding the flow chart in Figure 1a into an "infinite" flow chart. Figure 1b shows the conventional expansion into a flowchart with one starting point and an infinite number of stopping points. Figure 1c shows a backward expansion into a flow chart with one stopping point and an infinite number of starting points. This second expansion may seem unorthodox, but a careful examination will reveal that it describes the same set of finite (i.e., terminating) computations as the first. Each computation corresponds to the choice of a stopping node in Figure 1b or a starting node in Figure 1c.

Now suppose we wish to prove that the input-output relation $\Psi(x_0;x_f) \equiv \Phi(x_0) \rightarrow \Theta(x_0;x_f)$ holds for any finite computation which originates in state x_0 and finishes in state x_f. Obviously, to prove this for the infinite number of start-to-stop paths in either Figure 1b or Figure 1c we must use an inductive method based upon the method in which the n+1st path is generated from the first n.

The inductive assertion method uses the expansion in Figure 1b as follows. Given $\Phi(x_0)$ as an assumption, find an inductive assertion $\Gamma(x_0;x)$ and show

(1) $\Phi(x_0) \rightarrow \Gamma(x_0;x_0)$; i.e., it holds initially.

(2) $\sim P(x) \wedge \Gamma(x_0;x) \rightarrow \Gamma(x_0;N(x))$; i.e., if Γ holds between the states at points 0 and n then it holds between states 0 and n+1.

(3) $P(x) \wedge \Gamma(x_0;x) \rightarrow \Theta(x_0;x)$; i.e., $\Gamma(x_0;x)$ is sufficient to prove the desired input output relation, when the computation halts.

The subgoal induction method uses the expansion in Figure 1c as follows

(1) $P(x) \rightarrow \Psi(x,x)$, i.e., Ψ holds between points 1 and 0.

(2) $\sim P(x) \wedge \Psi(N(x);x_f) \rightarrow \Psi(x;x_f)$, i.e., if Ψ holds between points n and 0, then it holds between points n+1 and 0.

Thus the inductive assertion proof moves forward in the sense that it shows that some relation between the current state and the initial

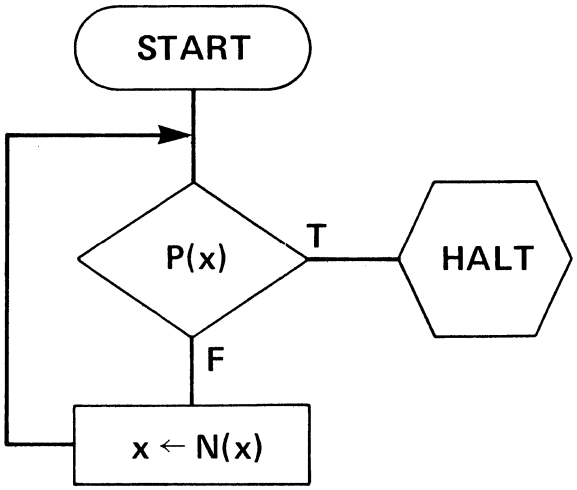


Figure 1a. A flow chart

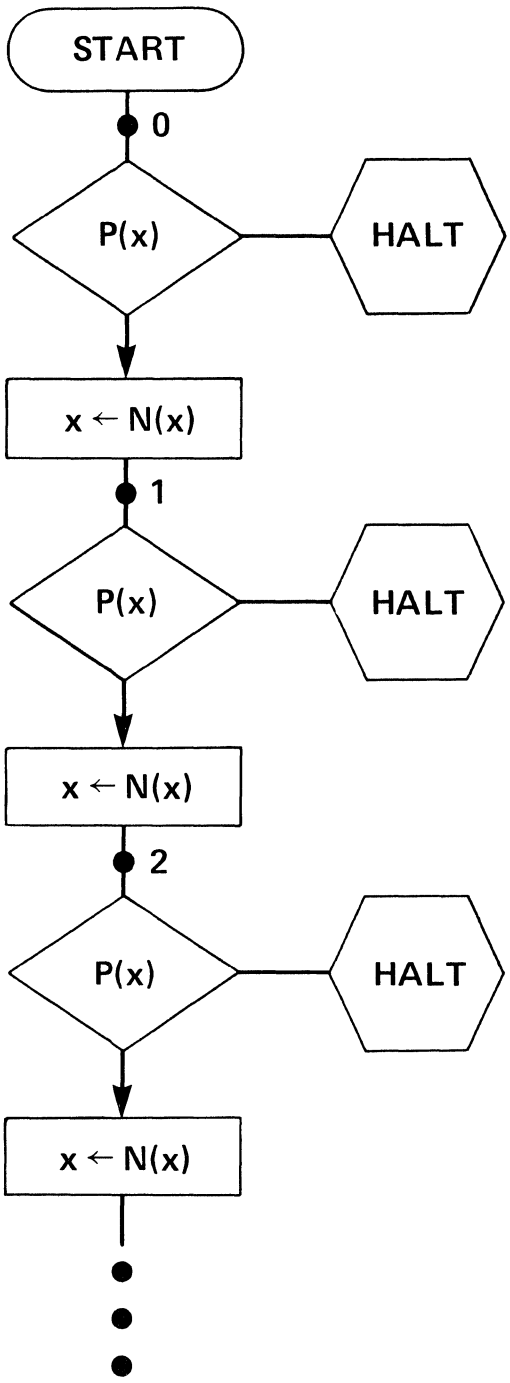


Figure 1b. A forward-expanded flow chart

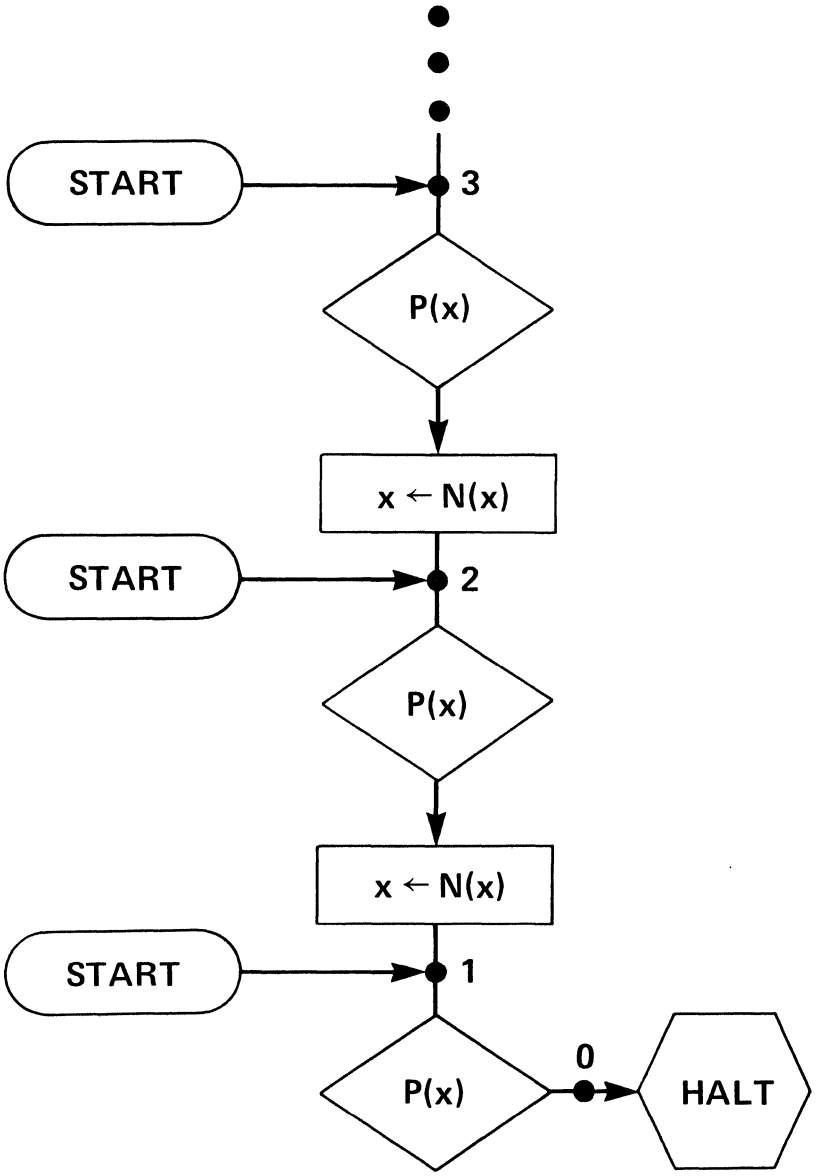


Figure 1c. A backward-expanded flow chart

state is maintained as the current state becomes the next state. The subgoal induction proof moves backward in the sense that it shows that a relation between the current state and the final state is maintained as the current state becomes a previous state.

The two methods are readily inter-translated. Performing the translation for the program in Figure 1 should serve to convince the reader.

Suppose an inductive assertion proof of $\Theta(x_0;x_f)$ at the end of the loop with input assertion $\Phi(x_0)$ has been achieved, i.e., we have proved

(F1) $\Phi(x_0) \rightarrow \Gamma(x_0;x_0)$

(F2) $P(x) \wedge \Gamma(x_0;x) \rightarrow \Theta(x_0;x)$

(F3) $\sim P(x) \wedge \Gamma(x_0;x) \rightarrow \Gamma(x_0;N(x))$

To establish the same result by subgoal induction, define

$$\Psi(x_0;x_f) \equiv (\forall y)\,[\Gamma(y;x_0) \rightarrow \Theta(y;x_f)]$$

Then prove Ψ by subgoal induction; i.e., prove

(V1') $P(x) \rightarrow (\forall y)\,[\Gamma(y;x) \rightarrow \Theta(y;x)]$

and

(V2') $\sim P(x) \wedge (\forall y)\,[\Gamma(y;N(x)) \rightarrow \Theta(y;x_f)] \rightarrow$
$(\forall y')\,[\Gamma(y';x) \rightarrow \Theta(y';x_f)]$

These follow from F2 and F3 respectively. To prove V2', for example, let y' be given; choose y to be y'; instantiate the free variable x_0 in F3 as y'; then V2' follows directly.

Finally, from F1 and $\Psi(x_0;x_f)$ (again choosing y to be x_0) we can deduce

$$\Phi(x_0) \rightarrow \Theta(x_0;x_f)$$

Using this method, any inductive assertion proof can be translated into a subgoal induction proof. ∎

The translation in the other direction is similar and was shown in [17] in a slightly different form. Suppose $\Psi(x_0;x_f) \equiv \Phi(x_0) \rightarrow \Theta(x_0;x_f)$ has been shown by subgoal induction, i.e.

(V1) $P(x) \rightarrow \Psi(x;x)$

(V2) $\sim P(x) \wedge \Psi(N(x);x_f) \rightarrow \Psi(x;x_f)$

To construct an inductive assertion proof define the inductive assertion $\Gamma(x_0;x)$ as follows

$$\Gamma(x_0;x) \equiv (\forall y)\,[\Psi(x;y) \rightarrow \Psi(x_0;y)]$$

Then show

$$(F1')\quad \Phi(x_0) \rightarrow (\forall y)\,[\Psi(x_0;y) \rightarrow \Psi(x_0;y)]$$

$$(F2')\quad P(x) \wedge (\forall y)\,[\Psi(x;y) \rightarrow \Psi(x_0;y)] \rightarrow \Psi(x_0;x)$$

$$(F3')\quad \sim P(x) \wedge (\forall y)\,[\Psi(x;y) \rightarrow \Psi(x_0;y)] \rightarrow (\forall y')\,[\Psi(N(x);y') \rightarrow \Psi(x_0;y')]$$

These are easily proved:

F1' is a tautology

F2' follows from V1, if we substitute x for y

F3' follows from V2, if we substitute x_f for y and y' ∎

3.2 Computation Induction

Subgoal induction may be viewed as a specialization of computation induction [11] to the familiar problem of proving partial correctness of first-order programs; i.e., the same problem that the inductive assertion method treats. The details of this connection are presented in the Appendix. Here it should suffice to illustrate the idea for the specific scheme

$$F(x) \Leftarrow \textbf{if } P(x) \textbf{ then } x \textbf{ else } L(F(N_1(x)),\ F(N_2(x)))$$

To prove $\Psi(x;F(x))$ the rule of computation induction [11] requires one to prove

$$(3.1)\quad (\forall g)\,[(\forall x')\,\Psi(x';g(x')) \rightarrow (\forall x)\,\Psi(x;\ \textbf{if } P(x) \textbf{ then } x \textbf{ else } L(g(N_1(x)),\ g(N_2(x))))]$$

The intuition behind this formula is that one may assume the property to be proved for the recursive calls of F while proving it for the body of F. The variable g is used instead of F to insure that no circular reasoning is possible.

A plausible strategy for proving (3.1) is to instantiate the premise

$$(3.2)\quad (\forall x')\,\Psi(x';g(x'))$$

twice, with $N_1(x)$ and $N_2(x)$ substituted for x' and use the resulting formulas to prove the conclusion. This is equivalent to proving

$$(3.3)\quad (\forall x,z_1,z_2)\ [\Psi(N_1(x);z_1) \wedge \Psi(N_2(x);z_2) \rightarrow \Psi(x;\ \textbf{if}\ P(x)\ \textbf{then}\ x\ \textbf{else}\ L(z_1,z_2))]$$

While it is straightforward to show that (3.3) implies (3.1) (i.e., that the strategy is sound), it is somewhat surprising that (3.1) "almost" implies (3.3). Specifically, if the clause

$$N_1(x)=N_2(x) \rightarrow z_1=z_2$$

is added to the premise of (3.3), yielding

$$(3.4)\quad (\forall x,z_1,z_2)\ [\Psi(N_1(x);z_1) \wedge \Psi(N_2(x);z_2) \wedge (N_1(x)=N_2(x) \rightarrow z_1=z_2) \rightarrow \Psi(x;\ \textbf{if}\ P(x)\ \textbf{then}\ x\ \textbf{else}\ L(z_1,z_2))]$$

then the resulting formula follows from (3.1). In fact, (3.4) is equivalent to (3.1). Showing (3.4) → (3.1) is straightforward. To show (3.1) → (3.4), suppose that (3.4) is false, i.e., for some x,z_1,z_2

$$(3.5)\quad \Psi(N_1(x);z_1) \wedge \Psi(N_2(x);z_2),$$

$$(3.6)\quad N_1(x)=N_2(x) \rightarrow z_1=z_2, \text{ and}$$

$$(3.7)\quad \sim\Psi(x;\ \textbf{if}\ P(x)\ \textbf{then}\ x\ \textbf{else}\ L(z_1,z_2)).$$

To show that (3.1) is false define the partial function h by

$$h(y) = \begin{cases} z_1 & \textbf{if } y=N_1(x) \\ z_2 & \textbf{if } y=N_2(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

By (3.6) h is well-defined. Then (3.1) is false for g=h since

$$(\forall x')\ \Psi(x';h(x'))$$

by (3.5), (recall that Ψ is a partial correctness relation) but

$$\sim\Psi(x;\ \textbf{if}\ P(x)\ \textbf{then}\ L(h(N_1(x)),h(N_2(x)))$$

by (3.7).

This informal demonstration is basically an outline of the following theorem proved in the Appendix.

Theorem 1. The statement to be proved in a subgoal induction proof,

i.e., (SGI) in section 2, is equivalent to the statement required by a computation induction proof.

COMBINING THE METHODS

The techniques of subgoal induction and inductive assertion provide alternative methodologies for stating and proving properties of programs--by "going backward" and "going forward" respectively. Because each of these directions is most natural for certain sorts of properties an obvious consideration is to combine them. A proof can then be partitioned and carried out partly with inductive assertions and partly with subgoal induction. Just how this can be done and under what circumstances it is a good strategy can best be understood by examining a special case of subgoal induction which leads very naturally to a partitioning.

It is commonly the case that an output specification takes a particular form which may be read as follows: If the input x satisfies certain constraints then the output is to have certain specified properties; if the input constraints are violated, the output is unspecified (usually because such inputs can never occur in program operation). That is, $\Psi(x;z)$ has the form

$$\Psi(x;z) \equiv \Phi(x) \rightarrow \Theta(x;z)$$

The constraint Φ is usually referred to as an *input specification*.

For simplicity, we discuss the schema F(x) <= **if** P(x) **then** H(x) **else** L(x, F(N(x))); the general case of multiple recursive functions is analogous. Writing the verification conditions for Ψ as defined above, and rearranging, we obtain

$$\text{(V1)} \quad P(x) \wedge \Phi(x) \rightarrow \Theta(x;H(x))$$

$$\text{(V2)} \quad \sim P(x) \wedge \Phi(x) \wedge [\Phi(N(x)) \rightarrow \Theta(N(x);z)] \rightarrow \Theta(x;L(x,z))$$

V1 is straightforward: it requires that if the input constraint is satisfied and the program terminates immediately then the output, H(x), be a Θ-acceptable output. V2, however, is more complex. Suppose the

left hand side of V2 is true, and consider proving the right hand side, $\Theta(x; L(x,z))$. Observe that $\Theta(x; L(x,z))$ involves the free variable z, but that z is restricted in the left hand side only by the conjunct $[\Phi(N(x)) \rightarrow \Theta(N(x);z)]$. As this conjunct could conceivably be true by virtue of $\Phi(N(x))$ being false, nothing necessarily is known about $\Theta(N(x);z)$ and hence nothing is known about z. A plausible proof strategy would be to establish that this cannot occur, i.e., to first prove

(V2a) $\sim P(x) \wedge \Phi(x) \rightarrow \Phi(N(x))$.

If true, (V2a) would guarantee that whenever the left hand side of (V2) is true, $\Phi(N(X))$ is true and hence $\Theta(N(x);z))$ is true. It would then suffice to prove that

(V2b) $\sim P(x) \wedge \Phi(x) \wedge \Phi(N(x)) \wedge \Theta(N(x);z) \rightarrow \Theta(x;L(x,z))$

V2a is of considerable interest. It requires that Φ be an invariant precondition for the function F: If Φ is true for some initial value of x given as input to F, then it must be true for all subsequent nested calls in F. Alternatively, it is useful to look at the syntactic form of V2a and observe that it has the form of a verification condition for the inductive assertion Φ around the loop

while $\sim P(x)$ **do** $x \leftarrow N(x)$.

Viewed in this way, Φ behaves as a normal invariant which, once established, may be used to assist the proof of V2b. This is somewhat surprising in that Φ and V2a were obtained from a subgoal induction proof using a "backwards going" induction. It illustrates that the two methods are really duals and that translation between them can be carried out on a very local level. As a first guideline as to how a proof should be partitioned, we observe that this sort of decomposition may be useful whenever it is possible to state a relatively simple invariant Φ describing which inputs are acceptable. Proof of this invariant is then decoupled from the remainder of the proof concerned with Θ.

Some additional syntax will help to crystallize this combined

method. We consider the case of **while** loops and extend our earlier notation to propose

maintain $\Phi(x)$ **while** $\sim P(x)$ **do** $x \leftarrow N(x)$ **thus** $\Theta(x_0;x)$

Proving V1, V2a, and V2b then establishes the input-output relation

$$\Psi'(x_0;x_f) \equiv \Phi(x_0) \rightarrow [\Phi(x_f) \wedge P(x_f) \wedge \Theta(x_0;x_f)]$$

Example 4 (Binary Search)

Consider the loop

maintain $\{b \leq c \wedge (\forall i | b \leq i \leq c)\ A[i] \leq A[i+1]\}$
while $b \neq c$
do begin $d \leftarrow (b+c)/2$; **if** $key > A[d]$ **then** $b \leftarrow d+1$ **else** $c \leftarrow d$ **end**
thus $\{key = A[b] \equiv (\exists i' | b_0 \leq i' \leq c_0)\ key = A[i']\}$

(Note that the output assertion does not require that the key be present in the table: it specifies that the key will be found if and only if it is present.)

Proving that the **maintain** clause is, in fact, an invariant is straightforward. Let $\Phi(b,c)$ denote this invariant, then the other verification conditions are

(V1) $b = c \wedge \Phi(b,c) \rightarrow$
$\{key = A[b] \equiv (\exists i' | b \leq i' \leq c)\ key = A[i']\}$

($V2b_1$) $\{b \neq c \wedge \Phi(b,c) \wedge d = (a+b)/2 \wedge key > A[d] \wedge$
$(key = A[z] \equiv (\exists i | d+1 \leq i \leq c)\ key = A[i])\} \rightarrow$
$\{key = A[z] \equiv (\exists i' | b \leq i' \leq c)\ key = A[i']\}$

($V2b_2$) $\{b \neq c \wedge \Phi(b,c) \wedge d = (a+b)/2 \wedge key \leq A[d] \wedge$
$(key = A[z] \equiv (\exists i | b \leq i \leq d)\ key = A[i])\} \rightarrow$
$\{key = A[z] \equiv (\exists i' | b \leq i' \leq c)\ key = A[i']\}$ ∎

Let us now consider a more technical aspect of this method: Is it "as powerful" as subgoal induction in the sense that V2 implies V2a and V2b? The answer, roughly speaking, is yes, except in cases which should never occur. More precisely, we reason as follows:

Definition. $\Psi(x;z) \equiv \Phi(x) \rightarrow \Theta(x;z)$ is said to be "well-behaved" with respect to F if

$$(\forall x)\ (\sim P(x) \wedge \Phi(x) \rightarrow (\exists z) \sim\Theta(x;L(x,z)))$$

That is, if Ψ is well-behaved, then whenever Φ is true and P is false of some x, there is some z such that L(x,z) is rejected by Θ. An output predicate Ψ which is *not* well-behaved has at least one x' which is acceptable input ($\Phi(x')$ = true) and for which the function recurses (P(x') = false), but for which any outcome whatever is acceptable according to Θ. This means that the function is needlessly continuing to recurse. We cannot think of any real examples in which such a situation occurs.

Theorem 2. If Ψ is well-behaved with respect to F and if V2 is valid, then V2a and V2b are valid.

Proof (by contrapositive). First note that V2 implies V2b immediately. Suppose that V2 is valid but that V2a is not valid; we will show that Ψ is not well-behaved. Since V2a is not valid, it is false for some x, say x',

$$(\sim V2a)\quad \sim P(x') \wedge \Phi(x') \wedge \sim\Phi(N(x'))$$

is true. Consider V2 for x'

$$\sim P(x') \wedge \Phi(x') \wedge [\Phi(N(x')) \rightarrow \Theta(N(x');z)] \rightarrow \Theta(x';L(x',z))$$

Using the truth of (~V2a), this simplifies to

$$\Theta(x';L(x',z))$$

Since (V2) is valid, this must be true for all z. Thus Ψ is not well-behaved. ∎

As a somewhat digressional point, we observe that this result may be employed in one other way. In mechanical program verification there is the possibility that a specification supplied by the programmer is incomplete and not strong enough to carry itself through the induction. V2 is then invalid and detecting this situation is necessary. Suppose that $\Psi(x;z)$ has the form $\Phi(x) \rightarrow \Theta(x;z)$ and that Θ is well-behaved. In many cases, it is possible to test for this syntactically, (e.g., if

$\Theta(x;z)$ has the form $z = g(x)$). Because V2a does not depend on Θ, it is less complex than V2. If the difference in complexity is significant, V2a may provide a useful filter for testing whether Ψ is complete. If V2a can be shown to be invalid, then the above theorem establishes that Ψ is incomplete, without further consideration of Θ.

The combined use of subgoal induction and inductive assertions may be applied, of course, to complete programs as well as simple loops. In general, a procedure has an *input assertion*, and an *output assertion*; intermediary points may be labeled with invariant *assertions*; **while** and **for** loops may be tagged with **maintain** invariants and **thus** subgoals. The inductive assertion method can be used to establish the correctness of the input assertion and the invariants by a "going forward" induction on program flow. Once established, a valid loop invariant can be used in the proof of a verification condition for a subgoal induction.

In particular, a **while** or **for** loop is treated as a recursive function in the sense that *its output condition is used in forming the verification condition for a path which passes through the loop.* For example, consider some **while** loop, W:

maintain $\Phi(x)$ **while** $\sim P(x)$ **do** $x \leftarrow N(x)$ **thus** $\Theta(x_0;x)$

and consider some case of a recursive function F which passes through W.

$F(x)$ <= **if** $P'(x)$ **then** $L(x, F(N_2(W(x))))$ **else** ...

The verification condition can be treated as being formed in two steps:

(1) Remove occurrences of F, by using Ψ_F:

$$P'(x) \wedge \Psi_F(N_2(W(x));z_F) \rightarrow \Psi_F(x;L(x,z_F))$$

(2) Remove occurrences of W, by using Ψ_W:

$$P'(x) \wedge \Psi_W(x;z_W) \wedge \Psi_F(N_2(z_W);z_F) \rightarrow \Psi_F(x;L(x,z_F))$$

where the loop specification Ψ_W is defined as

$$\Psi_W(x;z) \equiv \Phi(x) \rightarrow [\Phi(z) \wedge P(z) \wedge \Theta(x;z)].$$

(The treatment of **for** loops is analogous.) In practice, it is convenient to carry this out in a single step and regard Ψ_W as specifying the semantics of a loop.

Considering the other direction, subgoal induction can be used to establish the correctness of output assertions on recursive functions by a "going backward" induction. Once established, a valid output assertion describing the result returned by a called function can be asserted in a normal flowchart program. This allows a direct treatment of recursion mixed with normal program constructs such as loops, jumps, and exits. We illustrate this mixed case with an example.

Example 5 (Partition Sort)

So as to present the algorithm and its proof as simply as possible, we use a rather high-level notation--essentially Algol 68. Procedures may be passed and may return arrays; if A is an array, A[j:k] is the sub-array between A[j] and A[k] inclusive; length(A) returns the length of A; the infix operator "°" denotes concatenation of arrays.

```
real array procedure PSort(A), real array A; value A;
begin int n; n←length(A);
if n = 1 return A;
begin real array[1:n] S,B; int s,b; real x;
      s←b←0; x←A[n/2];
      for j from 1 to n do
            maintain (∀i|1≤i≤s) (S[i]<x) ∧ (∀i|1≤i≤b) (x≤B[i])
                  ∧ perm(A[1:j-1], S[1:s]°B[1:b]);
            if A[j]<x then  S[s←s+1]←A[j] else B[b←b+1] ← A[i];
      return PSort(S[1:s]) ° PSort(B[1:b])
end
end PSort output assertion ordered(PSort(A)) ∧ perm(A,PSort(A))
```

where ordered and perm are defined

$$ordered(A) \equiv (\forall i \mid 1 \le i < length(A))\ A[i] \le A[i+1]$$

$$\begin{aligned} perm(P,B) \equiv\ & length(A) = length(B) \wedge \exists R \\ & ((\forall i \mid 1 \le i < length(A))\ (1 \le R[i] \le length(A)) \\ & \wedge (\forall i,i' \mid 1 \le i < i' \le length(A))\ (R[i] \neq R[i']) \\ & \wedge (\forall i \mid 1 \le i \le length(A))\ (A[R[i]] = B[i])) \end{aligned}$$

Consider the proof by subgoal induction of the output assertion $\Psi_S(A;PSort(A))$ where

$$\Psi_S(A;z) \equiv ordered(z) \wedge perm(A,z)$$

Let ξ be the state vector. There are two cases

(V1) $P(\xi) \rightarrow \Psi_S(\xi;H(\xi))$

where $P(\xi) \equiv length(A)=1$ and $H(\xi) \equiv A$. This becomes

$$length(A) = 1 \rightarrow ordered(A) \wedge perm(A,A)$$

which is easily proved by expanding the definitions of ordered and perm. To prove the second case, assume that the invariant on the **for** loop has been validated by the inductive assertion technique. Further, observe that the invariant is initially true. Thus the verified input/output specification for the **for** loop is

$$\Psi_F(;A,S,B,s,b,x) \equiv (\forall i \mid 1 \le i \le s)\ (S[i] < x) \wedge (\forall i \mid 1 \le i \le b)\ (x \le B[i]) \wedge perm(A, S[1:s]\circ B[1:b])$$

(Note that in the absence of a **thus** clause which requires it, the input to the loop does not appear in this specification.) Let z_F be the state vector after the **for** loop terminates, let $N_1(z_F) = S[1:s]$, let $N_2(z_F) = B[1:b]$, and let $L(z_1,z_2) = z_1 \circ z_2$; then the second verification condition may be written

$$(V2)\quad \sim P(\xi) \wedge \Psi_F(\xi;z_F) \wedge \Psi_S(N_1(z_F);z_1) \wedge \Psi_S(N_2(z_F);z_2) \rightarrow \Psi_S(\xi;L(z_1,z_2))$$

That is,

$$length(A) \neq 1 \wedge \Psi_F(;A,S,B,s,b,x) \wedge ordered(z_1) \wedge perm(S[1:s],z_1) \wedge ordered(z_2) \wedge perm(B[1:b],z_1) \rightarrow ordered(z_1 \circ z_2) \wedge perm(A,\ z_1 \circ z_2)$$

Proof of this reduces to establishing two results

$$perm(A,U\circ V) \wedge perm(U,z_1) \wedge perm(V,z_2) \rightarrow perm(A,z_1\circ z_2)$$

$$ordered(z_1) \wedge ordered(z_2) \wedge perm(U,z_1) \wedge perm(V,z_2)$$
$$(\forall i \mid 1 \le i \le length(U))\ (U[i] < x)\ (\forall i \mid 1 \le i \le length(V))\ (x \le V[i]) \rightarrow$$

ordered($z_1 \circ z_2$)

both of which may be proved by using the definition of perm. ∎

A COMPLETENESS RESULT

We have previously touched upon a question which we now consider more fully: Under what circumstances is a specification strong enough to carry itself through an induction? Common experience has shown that input-output specifications are often too weak to be induction hypotheses, i.e., the resulting verification conditions are not valid. Section 4 presents a negative result: If $\Psi(x;z)$ has the form $\Phi(x) \rightarrow \Theta(x;z)$, if Ψ is well-behaved, and if Φ is not an invariant, then the induction formula is not valid. In this section we present a positive result, establishing a completeness result for subgoal induction. For simplicity, we consider the schema F(x) <= **if** P(x) **then** H(x) **else** L(x,F(N(x))); conditions for more general forms are analogous.

Definition. $\Psi(x;z) \equiv \Phi(x) \rightarrow \Theta(x;z)$ is said to be a "tight specification" if

$$\sim P(x) \wedge \Phi(x) \wedge \Theta(N(x);z_1) \wedge \Theta(N(x);z_2) \rightarrow L(x,z_1) = L(x,z_2)$$

Essentially, Ψ is a tight specification if two z's that are both accepted by Θ produce identical outputs from L.

Theorem 3. If $\Psi(x;z) \equiv \Phi(x) \rightarrow \Theta(x;z)$ is a tight specification, if Φ is an invariant, if F is total on the domain $\{x|\Phi(x)\}$, and if $\Psi(x;F(x))$ is valid, then

(V2) $\sim P(x) \wedge \Psi(N(x);z) \rightarrow \Psi(x;L(x,z))$

is valid.

Proof. Rewrite (V2) as

(5.1) $\sim P(x) \wedge \Phi(x) \wedge [\Phi(N(x)) \rightarrow \Theta(N(x);z)] \rightarrow \Theta(x;L(x,z))$

Consider some x', z' for which the left hand side is true

$$\sim P(x') \wedge \Phi(x') \wedge [\Phi(N(x')) \rightarrow \Theta(N(x');z')]$$

Since Φ is an invariant, $\Phi(x') \wedge \sim P(x') \rightarrow \Phi(N(x'))$; hence, it follows that

$\sim P(x') \wedge \Phi(x') \wedge \Phi(N(x')) \wedge \Theta(N(x');z')$

Since F is correct and $F(N(x'))$ is defined, $\Phi(N(x')) \rightarrow \Theta(N(x');F(N(x')))$; hence, it follows that

$\sim P(x') \wedge \Phi(x') \wedge \Phi(N(x')) \wedge \Theta(N(x');z') \wedge \Theta(N(x'); F(N(x')))$

From the definition of tight specification, this implies

(5.2) $L(x',z') = L(x',F(N(x')))$

Since $\Phi(x) \rightarrow \Theta(x;F(x))$ is valid, upon expanding the definition of F we obtain

(V2*) $\sim P(x') \wedge \Phi(x') \wedge [\Phi(N(x')) \rightarrow \Theta(N(x');F(N(x')))] \rightarrow \Theta(x';L(x',F(N(x'))))$

This, taken together with (5.2), implies

$\Theta(x';L(x',z'))$

which is the right hand side of (5.1). Thus (V2) is valid. ∎

Observe that if Θ characterizes z by a function, $\Theta(x;z) \equiv z=g(x)$, then Ψ is surely a tight specification. In particular, consider the case of proving two programs $F \lessdot= \tau[F]$ and $G \lessdot= \sigma[G]$ equivalent. Let the output assertion for F be $\Psi_F(x;z) \equiv z=G(x)$. This is a tight specification and it therefore follows that the verification conditions for subgoal induction are valid. Thus this theorem can be viewed as a generalization of results in [1] and [13].

Because of the intertranslatibility between subgoal induction, inductive assertions, and computation induction, as established in Section 3, it follows that results analogous to Theorem 3 apply to these other proof methods as well. Thus we have established a sufficient (but not of course) necessary criteria for judging when a specification is strong enough, so that the induction step is valid. Proof of this valid theorem depends, of course, on the decidability of the domain--which is a distinct issue.

CONCLUSION

Currently, there are three induction methods in common use for

mechanical program verification: structural induction [2,15], inductive assertions [5,7,16] and computation induction [14]. In proposing a fourth, *subgoal induction*, it is perhaps worthwhile to discuss just why it might be used in preference to one of the current methods.

At a formal level, all are equivalent when applicable: the results of Section 3 establish the formal equivalence of computation induction to subgoal induction and of inductive assertions to subgoal induction restricted to flowchart programs; further, [11] establishes the formal equivalence of structural and computational induction. The utility of subgoal induction lies not in formal power, but rather in its applicability, its directness, in the relative simplicity of the assertions it requires, and in the simplicity of the verification conditions it produces.

Subgoal induction may be useful in preference to structural induction in cases where the structure to be inducted on is complex. Structural induction requires an explicit determination of the structure so that the induction can be set-up. Such explicit determination may be difficult to mechanize where the well-ordering is complex, e.g., Binary Search, or Partition Sort. In such cases it may be easier to use subgoal induction which uses the computation sequence directly to establish the induction.

Subgoal induction may be preferable to computation induction since it has, in effect, "compiled" the computation induction rule into an equivalent but simpler form. In particular, subgoal induction generates first order formulas as verification conditions whereas computation induction generates second order formulas--due to the quantification over function letters. Thus subgoal induction avoids certain issues in the mechanization of higher-order logic which must be addressed when using computation induction.

With respect to inductive assertions, we regard subgoal induction as simply complementary. Subgoal induction can be used to generate the

verification conditions for function calls, thus allowing use of recursive functions in a flowchart program. Further, the rule of subgoal induction specialized to **while** loops can be used to verify such loops without explicit inductive assertions or with weaker-than-normal inductive assertions inside the loops. Finally, invariants verified by the inductive assertion method can be used as auxiliary information in proving subgoal induction verification conditions. Thus the two methods fit well together and each somewhat simplifies the work of the other.

ACKNOWLEDGMENT

The work reported here had its origins in trying to relate the methods of the Boyer-Moore theorem prover to other proof techniques. Numerous discussions with J Moore helped to clarify the relation and raised several of the questions answered here. D. Bobrow and L. P. Deutsch gave this paper sympathetic readings and suggested several improvements in its presentation.

REFERENCES

1. Basu, S. and Misra, J. Proving loop programs. *IEEE Transactions on Software Engineering, Vol. SE-1*, No. 1. (March 1975), 76-86.

2. Boyer, R. and Moore, J.S. Proving theorems about LISP functions. *JACM* 22, 1 (Jan. 1975), 129-144.

3. Burstall, R.M. Proving properties of programs by structural induction. *The Computer Journal* 12, 1 (Feb. 1969), 41-48.

4. Floyd, R.W. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, Schwartz, J.T. (Ed.), AMS, 1967, 19-32.

5. Good, D.I., London, R.L.and Bledsoe, W.W. An interactive program verification system. Int. Conf. on Reliable Software, Los Angeles, Ca., April 1975.

6. Hoare, C.A.R. Procedures and parameters: an axiomatic approach. *Lecture Notes in Mathematics 188*, Engeler, E. (Ed.), Springer-Verlag, 1971.

7. Igarashi, S., London, R.L. and Luckham, D.C. Automatic program verification I: Logical basis and its implementation. AIM-200, Stanford Artificial Intelligence Project, Stanford U., 1972.

8. McCarthy, J. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, Braffort and Hirschberg (Eds.), North-Holland, Amsterdam, 1963, 33-70.

9. McCarthy, J., and Painter, J.A. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, Schwartz, J.T. (Ed.), AMS, 1967, 33-41.

10. Manna, Z., and Pnueli, A. Formalization of properties of functional programs. *Journal of the ACM 17*, 3 (July 1970), 555-569.

11. Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. *CACM 16*, 8 (Aug. 1973), 491-502.

12. Manna, Z. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.

13. Mills, H. The new math of computer programming. *CACM 18*, 1 (Jan. 1975), 43-48.

14. Milner, R. Implementation and applications of Scott's logic for computable functions, *SIGPLAN Notices 7,1* and *SIGACT News 14* (Jan. 1972), 1-6.

15. Moore, J.S. Introducing prog into the pure lisp theorem prover. CSL-74-3, Xerox Palo Alto Research Center (Dec. 1974).

16. Waldinger, R.J. and Levitt, K.N. Reasoning about programs. *Artificial Intelligence, 5, 3* (Fall 1974), 235-316.

17. Wegbreit, B. Complexity of synthesizing inductive assertions. Computer Science Laboratory, Xerox Palo Alto Research Center, January 1975.

APPENDIX: RELATION TO COMPUTATION INDUCTION

We assume that the reader is familiar with the notation and concepts of [11]. The rule of computation induction as presented there is cast in a conceptual framework somewhat at variance from the current one. Before illustrating the close connection between computation induction and subgoal induction we shall attempt to reconcile this variance. First, [11] deals more explicitly with undefinedness: All functions are presumed to be total over a domain and range which include ω, the "undefined" value. Second, the meaning of a function definition is based upon call-by-name semantics, rather than the more familiar call-by-value semantics.

Let us illustrate the connection by translating a typical partial correctness problem into the framework of [11]. Given the function definition

$h(x) \Leftarrow$ **if** $P(x)$ **then** $H(x)$ **else** $h(N(x))$

prove $\Upsilon(x;h(x))$. First we define a new function f.

$f(x) \Leftarrow$ **if** $x = \omega$ **then** ω **else if** $P(x)$ **then** $H(x)$ **else** $f(N(x))$

Thus, the call-by-name meaning of f is precisely the call-by-value meaning of h. Second, define a new predicate

$$\Psi(x;z) \equiv z=\omega \vee \Upsilon(x;z)$$

This simply makes the partial correctness aspect of Ψ explicit. The correctness problem now becomes: Prove $\Psi(x;f(x))$.

With this translation of problem statement in mind let us now consider the rule of computation induction. Given a recursive definition of a function with the general form

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

the rule of computation induction says that to prove $\alpha(f)$ about f one may prove $\alpha(\Omega)$ and $(\forall g)\ [\alpha(g) \rightarrow \alpha(\tau[g])]$ where g ranges over the continuous functions. The predicate α must be *admissible* in the following sense: It must be the case that

$$(A.1)\quad (\forall i)\ \alpha(g_i) \rightarrow \alpha(\lim_i g_i)$$

In other words, a computation induction proof establishes only that α holds for g_0, g_1, etc. That α holds in the limit then follows from (A.1).

Subgoal induction is a specialization of computation induction in which the inductive statement $\alpha(f)$ is restricted to the form

$$(A.2)\quad \beta(f) \equiv \Psi(x;f(x)) \equiv (\forall x)\ [f(x)=\omega \vee \Upsilon(x;f(x))]$$

and the functional τ is restricted so that all occurrences of f in $\tau[f](x)$ are first-order in the sense that f is applied to some arguments rather than being an argument.

The first step in showing the correspondence between the methods is to show that the form (A.2) is restricted enough so that (A.1) holds. In particular, we shall show that the restrictions suggested in [12] are not necessary for predicates with the form (A.2) as long as the domain of values has no values definable only as limits of other values.

Lemma 1. Suppose for any chain of elements $x_1 \subseteq x_2 \subseteq \ldots$

$$x = \lim_i x_i \rightarrow (\exists j)\,[x = x_j]$$

(i.e., no element is definable only as a limit). Then

$$(A.3)\quad (\forall i)\,[\beta(g_i)] \rightarrow \beta(g)$$

where $g = \lim_i g_i$ and β is as in (A.2).

Proof. First note that, for any fixed x,

$$g(x) = (\lim_i g_i)(x) = \lim_i (g_i(x)) \text{ (see [11])}$$

Therefore, by the lemma's assumption, there is a j such that $g(x) = g_j(x)$. In general this means

$$(A.4)\quad (\forall x)\,(\exists j)\,[g(x) = g_j(x)]$$

Now, to prove (A.3) assume $(\forall i)\,[\beta(g_i)]$; i.e.,

$$(A.5)\quad (\forall i)\,(\forall x)\,[g_i(x)=\omega \vee \Upsilon(x;g_i(x))]$$

To show $\beta(g)$; i.e.,

$$(\forall x)\,[g(x) = \omega \vee \Psi(x;g(x))]$$

fix x and assume that $g(x)\neq\omega$. By (A.4) there is a j such that $g(x) = g_j(x)$. Since $g_j(x)\neq\omega$ by assumption we have $\Upsilon(x;g_j(x))$ by (A.5). Hence $\Upsilon(x;g(x))$, and (A.3) is established. ∎

The following lemma shows that if one restricts one's information about a function, g, to a partial correctness statement $\Psi(x;g(x))$, then the proof of any first-order statement about g can proceed in a rather straight-forward way.

Lemma 2. The statement

$$(A.6)\quad (\forall g)\,[(\forall x)\,[\Psi(x;g(x))] \rightarrow \Pi(g(\gamma_1),\ldots,g(\gamma_n))]$$

(where $\Psi(x;\omega)$ is true) is equivalent to

$$(A.7)\quad [\bigwedge_{i=1}^{n} \Psi(\gamma_i;z_i) \quad \wedge \quad \bigwedge_{1\leq i,j\leq n} (\gamma_i=\gamma_j \rightarrow z_i=z_j)] \rightarrow \Pi(z_1,\ldots,z_n)$$

where Π and the terms γ_i have no occurrences of g.

Proof. To show that (A.7) → (A.6) assume that g is given and

(A.8) $(\forall x)\ \Psi(x;g(x))$

Instantiate (A.8) n times to get

(A.9) $\bigwedge_{i=1}^{n} \Psi(\gamma_i;g(\gamma_i))$

Instantiate (A.7) to get

$$\bigwedge_{i=1}^{n} \Psi(\gamma_i;g(\gamma_i)) \wedge \bigwedge_{1\le i,j\le n} (\gamma_i=\gamma_j \rightarrow g(\gamma_i)=g(\gamma_j))$$

$$\rightarrow \Pi(g(\gamma_1),\ldots,g(\gamma_n))$$

which, with (A.9) implies

$\Pi(g(\gamma_1),\ldots,g(\gamma_n))$.

To show that (A.6) → (A.7) assume that (A.7) is false; i.e., there is some assignment to the free variables so that

(A.10) $\bigwedge_{i=1}^{n} \Psi(\gamma_i;z_i)$

and

(A.11) $\bigwedge_{1\le i,j\le n} \gamma_i=\gamma_j \rightarrow z_i=z_j$

and

(A.12) $\sim\Pi(z_1,\ldots,z_n)$.

Define the function h by

$$h(x) = \begin{cases} z_i \text{ if } x=\gamma_i \text{ for any } i,\ 1\le i\le n \\ \omega \text{ otherwise} \end{cases}$$

By (A.11) h is well-defined. Now (A.6) is false for g=h since

(a) $(\forall x)\ \Psi(x;h(x))$ by the definition of h

(b) $\sim\Pi(h(\gamma_1),\ldots,h(\gamma_n))$ for the same assignment of free variables that produced (A.12). ∎

The following more general form of the lemma can be proved in the same manner.

Lemma 2.1. The statement

$(\forall g)\ [(\forall x_1,\ldots,x_m)\ [\Psi(x_1,\ldots,x_m;\ g(x_1),\ldots,g(x_m))] \rightarrow \Pi(g(\gamma_1,\ldots,g(\gamma_n))]$

is equivalent to

$$\Big[\bigwedge_{\langle i_1,\ldots,i_m\rangle\in\{1,\ldots,n\}^m} \Psi(\gamma_{i_1},\ldots,\gamma_{i_m};z_{i_1},\ldots,z_{i_m}) \wedge \bigwedge_{1\le i,j\le n} (\gamma_i=\gamma_j \rightarrow z_i=z_j)\Big] \rightarrow \Pi(z_1,\ldots,z_n)$$

In other words, one can eliminate the function letter g from the discussion at the expense of introducing n^m instances of Ψ.

Finally, we prove the equivalence result between subgoal induction and computation induction.

Theorem 1. The statement to be proved in a subgoal induction proof, i.e., (SGI) in Section 2, is equivalent to the statement required by a computation induction proof.

Proof. Let the partial correctness statement be

$$\Psi(x;f(x))$$

where f is defined by

$$f \Leftarrow \tau[f]$$

and τ is a first-order functional of f. By Lemma 1, $\Psi(x;f(x))$ is an admissible predicate for a computation induction proof so $\Psi(x;f(x))$ follows from

(A.13) $\Psi(x;\Omega(x))$

and

(A.14) $(\forall g)\,[(\forall x)\,[\Psi(x;g(x))] \rightarrow (\forall y)\,[\Psi(y;\tau[g](y))]]$

These are equivalent, respectively, to

(A.13') $\Psi(x;\omega)$

and

(A.14') $(\forall g)\,[(\forall x)\,[\Psi(x;g(x))] \rightarrow \Psi(y;\tau[g](y))]$

assuming y is not free in $\Psi(x,g(x))$. Now (A.13') is true by definition so the proof of the theorem reduces to showing that proving (A.14') is equivalent to a subgoal induction proof. But (A.14') can be re-written in the form

(A.15) $(\forall g)\,[(\forall x)\,[\Psi(x;g(x))] \rightarrow \Pi(g(\gamma_1),\ldots,g(\gamma_n))]$

where Π has the form

(A.16) $(\forall z_1,\ldots,z_n)[\bigwedge_{i=1}^{n} z_i=g(\gamma_i) \rightarrow \Psi(y,\beta)]$

and β and the γ_i are derived as in Section 2; i.e., by replacing (inside out) each occurrence $g(\gamma_i)$ by a new variable z_i.

Now by Lemma 2, (A.15) is equivalent to

$$[\bigwedge_{i=1}^{n} \Psi(\gamma_i;z_i) \wedge \bigwedge_{1\le i,j\le n} (\gamma_i=\gamma_j \rightarrow z_i=z_j)] \rightarrow$$

$$\bigwedge_{i=1}^{n} z_i=z_i \wedge \Psi(y;\beta)$$

which is equivalent to the rule of subgoal induction. ∎