Programming Languages

B. Wegbreit*
Editor

Subgoal Induction

James H. Morris Jr. and Ben Wegbreit
Xerox Palo Alto Research Center

A 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. 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

1. Introduction

We describe a technique, subgoal induction, for proving programs correct. It may be used as an alternative or supplement to the commonly used inductive assertion method [4, 7]. Subgoal induction is a proof rule which matches the problem solving heuristic: "Transform a problem into a simpler one with the same general characteristics, solve the simpler problem, and use that solution to solve the original problem."

If we accept the premise that proof rules should guide the way one thinks about programs, then subgoal induction is an important addition to our understanding of programming methodology. The method of inductive assertions suggests that the programmer concentrate on invariants, i.e. the things his program does not change. The method of subgoal induction suggests that he concentrate on the dynamics of program segments, i.e. the relation between the current state and the result to be computed. Our experience with these two methods has shown both to be useful, and we later describe circumstances advantageous to each.

For example, when the task is to prove that a **while** loop satisfies a given input-output relation, subgoal induction can be applied directly without the intermediate step of framing an inductive assertion. This simplifies the problem of discovering an adequate induction hypothesis; the input-output relation must be made demanding enough to be an induction hypothesis.

Subgoal induction presents a way of "thinking recursively"; i.e. assuming one's ability to solve simpler problems and generating solutions to more complex problems. Its application to loop programs shows that one can program recursively without using function definitions, push down stacks or the other trappings of recursion.

We begin by presenting one particular case of subgoal induction to illustrate the basic idea. Consider the flowchart of Figure 1 which shows a simple loop. x is a vector containing all the program variables. Each execution of this loop will take place in one of two ways: If P is true the loop exits, as shown in Figure 2. If P is false then x is assigned $N(x)$ and the loop head is reached again, as shown in Figure 3.

Suppose the correctness condition for the loop is specified by the following requirement: given input x the loop is to produce output z such that $\Psi(x; z)$ for some given predicate[1] Ψ. We consider the two cases shown in Figures 2 and 3:

(1) If $P(x)$ is true then control leaves the loop with x unchanged. Thus the output is x. In order for the loop specification to be satisfied, it must be that $\Psi(x; x)$ in this case. That is,

$$P(x) \rightarrow \Psi(x; x) \tag{1.1}$$

(2) If $P(x)$ is false then x is assigned $N(x)$; let x' denote the new value of x, i.e. $x' = N(x)$. The loop head is now reached with x'. Suppose that eventually

Authors' address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

* Note. This paper was submitted prior to the time that Wegbreit became editor of the department, and editorial consideration was completed under the former editor, Thomas Standish.

[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.

the loop halts and that z is the output of the loop started with x'. Further, suppose that the input/output pair $\langle x', z\rangle$ satisfies the specification, i.e. $\Psi(x'; z)$. Observe that z is also the final output of the original loop starting with x. In order for the original loop specification to be satisfied, it must be that $\Psi(x; z)$ in this case. Therefore it suffices to prove:

$$\sim P(x) \wedge x' = N(x) \wedge \Psi(x'; z) \rightarrow \Psi(x; z) \qquad (1.2)$$

(1.1) and (1.2) are verification conditions. If they are both valid, then it is guaranteed that the loop meets its specification. *Subgoal induction is a way of constructing such verification conditions directly from a loop's specification.*

2. Rule of Subgoal Induction

In order to present subgoal induction in a more precise and rigorous fashion, we employ a functional view of programming in which programs are written as recursive functions. We first show how the induction rule can be coupled with the synthesis of a recursive function, considering program construction and proof of 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 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:

$$P(x) \rightarrow \Psi(x; H(x)) \qquad (2.1)$$

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 uses x and the solution for $N(x)$ to compute a solution for x. This leads to the fragment:

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

The corresponding verification condition is:

$$\sim P(x) \wedge \Psi(N(x); z) \rightarrow \Psi(x; L(x, z)) \qquad (2.2)$$

This may be read as: if $\sim P(x)$ 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:

$$F(x) \Leftarrow \textbf{if } P(x) \textbf{ then } H(x) \textbf{ else } L(x, F(N(x))) \qquad (2.3)$$

If the two verification conditions (2.1) and (2.2) are valid, then $\Psi(x; F(x))$ is valid. This is an example of the rule of *subgoal induction*, applied to a recursive function.

Fig. 1. A simple loop.

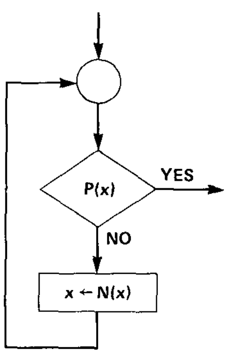


Fig. 2. Test taken in the positive direction.

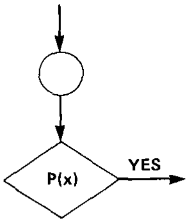
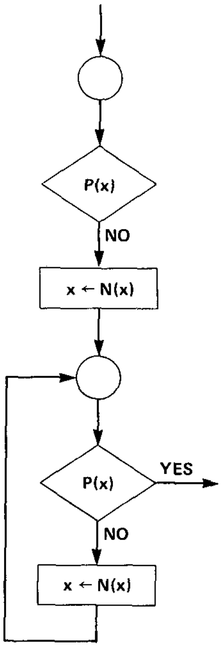


Fig. 3. Test taken once in the negative direction.



The above argument tacitly assumes that $F(N(x))$ actually returns a value. If, on 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 be concerned only with *partial correctness*, i.e. consider only those cases where $F(x)$ terminates. A more precise statement of subgoal induction reads: If (2.1) and (2.2) 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 (2.2) is a stronger requirement than actually needed to establish $\Psi(x; F(x))$. It would, in principle, suffice to prove the somewhat weaker implication:

$$\sim P(x) \wedge \Psi(N(x); z) \wedge z = F(N(x)) \rightarrow \Psi(x;L(x,z)) \qquad (2.4)$$

This may be read as: If $\sim P(x)$ and if z is the output of $F(N(x))$ and if the pair $\langle N(x), z\rangle$ satisfies $\Psi(N(x); z)$, then the pair $\langle x, L(x, z)\rangle$ must satisfy $\Psi(x; L(x, z))$. (2.4) follows directly from expanding the definition of F. It differs from (2.2) 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 (2.2) is a valid theorem which, taken together with (2.1), 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: (2.1) and (2.2) are valid if and only if the verification conditions for computation induction [12] are valid.

Example 1 (Subgoal induction). Let "/" denote integer division (with truncation) and define

$$E(x) \Leftarrow \mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ L(x, E(x/2)), \text{ and}$$
$$L(x, y) \Leftarrow \mathbf{if}\ even(x)\ \mathbf{then}\ y^2\ \mathbf{else}\ 2 \cdot y^2$$

The property to be proved is $\Psi(x; E(x))$ where $\Psi(x; z) \equiv z = 2^x$. The verification conditions are obtained from (2.1) and (2.2):

$$x = 0 \rightarrow 1 = 2^x \text{ (trivial)}$$
$$x \neq 0 \wedge z = 2^{x/2} \rightarrow L(x, z) = 2^x$$

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

In the case where H is the identity function and $L(x, z) = z$, the recursive function F defined in (2.3) is equivalent to the following **while** scheme:

$$\mathbf{while}\ {\sim}P(x)\ \mathbf{do}\ x \leftarrow N(x)$$

Let x_0 and x_f be the initial and final values of the state vector x in this scheme. To establish $\Psi(x_0; x_f)$ by subgoal induction, (2.1) and (2.2) require that we prove:

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

Observe that these verification conditions are identical to the verification conditions for the simple flowchart loop discussed in the introduction. Obtaining these as a special case of the general rule for recursive functions is an alternate derivation which may, perhaps, clarify the method. Note that the idea of loop invariant or inductive assertion does not appear. The output assertion Ψ need not be true within the loop and likely is not (otherwise we are wasting iterations!). 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 $x \leftarrow x - y$
```

The output assertion is:

$$\Psi(x_0, y_0; x_f) \equiv x_f = GCD(x_0, y_0)$$

where $GCD(x, y) = \max \{t \mid t \text{ divides } x \wedge t \text{ divides } y\}$. Let ξ be the state vector. To prove Ψ by subgoal induction, two verification conditions must be established. The first is:

$$P(\xi) \rightarrow \Psi(\xi; \xi) \tag{2.5}$$

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

$$x = y \rightarrow x = GCD(x, y)$$

The second verification condition is:

$${\sim}P(\xi) \wedge \Psi(N(\xi); z) \rightarrow \Psi(\xi; z) \tag{2.6}$$

where $N(\xi) = \mathbf{if}\ x < y\ \mathbf{then}\ \langle x, y - x\rangle\ \mathbf{else}\ \langle x - y, y\rangle$. This becomes:

$$\begin{aligned} &x \neq y \wedge \\ &z = (\mathbf{if}\ x < y\ \mathbf{then}\ GCD(x, y-x) \\ &\qquad \mathbf{else}\ GCD(x-y, y)) \rightarrow \\ &z = GCD(x, y) \end{aligned}$$

(2.5) is immediate and (2.6) follows easily from the general observation that

$$z \text{ divides } a \wedge z \text{ divides } b \rightarrow$$
$$z \text{ divides } a + b \wedge z \text{ divides } |a - b|$$

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

It happens that the following is a good invariant:

$$GCD(x_0, y_0) = GCD(x, y)$$

However, it appears that a proof by inductive assertions will be less direct 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 $x = GCD(x_0, y_0)$
```

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

Example 3 (Multiple-exit loop). Consider searching a table $A[1{:}n]$ for the first entry which is equal to key, returning the index of the entry if one exists or 0 otherwise. We wish to prove:

$$\{1 \leq j \leq n \wedge A[j] = key\} \vee$$
$$\{j = 0 \wedge (\forall i \mid 1 \leq i \leq n)\ A[i] \neq key\}$$

We write a multiple-exit loop as: **do** ⟨statement⟩ **end** with the understanding that the ⟨statement⟩ is to be repeated until an **exit** is executed. An **exit** and the ⟨output assertion⟩ associated with that **exit** is written:

exit thus ⟨output assertion⟩

In this notation, the search loop is written:

```
$j \leftarrow 1$;
do
if $j > n$ then
  begin $j \leftarrow 0$;
  exit thus $\{j = 0 \wedge (\forall i \mid j_0 \le i \le n)\ A[j] \ne \text{key}\}$
  end
else if $A[j] = \text{key}$ then
    exit thus $\{j_0 \le j \le n \wedge A[j] = \text{key}\}$
else $j \leftarrow j + 1$
end
```

Note that the two disjuncts of the specification are associated with distinct **exit** conditions. Note also that j_0 is used as the lower limit of the range conditions. The verification conditions for the case which exits on $j > n$ are:

$$j > n \rightarrow 0 = 0 \wedge (\forall i \mid j \le i \le n)\ A[i] \ne key$$

$$j_0 \le n \wedge A[j_0] \ne key \wedge j = 0 \wedge (\forall i \mid j_0 + 1 \le i \le n)\ A[i] \ne key \rightarrow j = 0 \wedge (\forall i \mid j \le i \le n) A[i] \ne key$$

both of which are valid. The verification conditions for the case which exits on $A[j] = \text{key}$ are:

$$j \le n \wedge A[j] = key \rightarrow j \le j \le n \wedge A[j] = key$$

$$j_0 \le n \wedge A[j_0] \ne key \wedge j_0 + 1 \le j \le n \wedge A[j] = key \rightarrow j_0 \le j \le n \wedge A[j] = key$$

which are also valid. Because the loop specification has been decomposed into two disjuncts, each associated with a distinct exit, the verification conditions are relatively simple.

As a final comment on this example, we stress the use of j_0 as the lower limit of the range conditions, e.g. on the first exit, the output specification $(\forall i \mid j_0 \le i \le n)$ $A[i] \ne key$. This expresses the final outcome, if control leaves via this exit, in terms of the input state j_0 on each iteration through the loop. □

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

$$\begin{aligned} F(x) \Leftarrow &\textbf{ if } P_0(x) \textbf{ then } L_0(G(N_0(x))) \\ &\textbf{ else if } P_1(x) \textbf{ then } L_1(G(N_2(F(N_1(x))))) \\ &\textbf{ else } L_2(F(N_3(x)), F(N_4(x))) \end{aligned}$$

$$G(y) \Leftarrow \textbf{if } P_5(y) \textbf{ then } H(y) \textbf{ else } F(y)$$

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:

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

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:

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

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:

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

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:

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

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

Now let us state the general rule for recursively defined functions. Consider a definition of the form:

$$F(x) \Leftarrow \textbf{if } P_1 \textbf{ then } E_1 \textbf{ else if } P_2 \textbf{ then} \ldots \textbf{ else } E_n \qquad (2.7)$$

where x is a vector of variables, the E_i are terms made up from constants, x, primitive functions, and F, and the P_i are predicates applied to terms made up from constants, x, primitive functions, but not F. (The generalization allowing the E_i to contain conditionals and the P_i to contain F is easy but unilluminating, as is the generalization to mutually recursive definitions.)

The rule of subgoal induction establishes the partial correctness relation $\Psi(x; F(x))$ by proving n statements, one for each E_i. Each statement has the form:

$$\bigwedge_{j=1}^{i-1} \sim P_j \wedge P_i \wedge \bigwedge_{k=1}^{m} \Psi(g_k; z_k) \wedge \bigwedge_{1 \le s < t \le m} (\gamma_s = \gamma_t \rightarrow z_s = z_t) \rightarrow \Psi(x; \gamma_0) \tag{2.8}$$

where the γ_k are derived from E_i by replacing all terms of the form $F(\gamma)$ with some new z_j in an inside-out manner. To be more precise, suppose E_i contains m occurrences of the function letter F. If $m = 0$, simply choose $\gamma_0 = E_i$, otherwise find a subterm of the form $F(\gamma)$ where γ does not contain F, choose $\gamma_m = \gamma$, replace the occurrence of $F(\gamma)$ in E_i by z_m, and repeat.

Consider the specific function definition:

$F(x, y) \Leftarrow$ **if** $P(x)$ **then** $H(x)$ **else** $F(M(x), F(N(x), y))$

then the two clauses to be proved are:

$$P(x) \rightarrow \Psi(x, y; H(x))$$

and

$$\sim P(x) \wedge \Psi(N(x), y; z_2) \wedge \Psi(M(x), z_2; z_1) \wedge (\langle N(x), y\rangle = \langle M(x), z_2\rangle \rightarrow z_2 = z_1) \rightarrow \Psi(x, y; z_1)$$

This rule is essentially the same as Manna and Pnueli's in [11] except for the equality "cross term" in the premise:

$$\bigwedge_{1 \le s < t \le m} (\gamma_s = \gamma_t \rightarrow z_s = z_t)$$

which requires that there be a functional relationship between the γ's and z's. This fact was implicit in the functional notation but was lost when the function letter F disappeared. The functionality can be important. Consider the definition:

$H(x) \Leftarrow$ **if** $x = 0$ **then** 1 **else** $H(x-1) - H(x-1)$

and suppose one wants to prove $H(x) \neq 2$. The rule of subgoal induction requires one to prove:

$$x = 0 \rightarrow 1 \neq 2$$

and

$$\sim(x = 0) \wedge z_1 \neq 2 \wedge z_2 \neq 2 \wedge ((x - 1) = (x - 1) \rightarrow z_1 = z_2) \rightarrow z_1 - z_2 \neq 2$$

It is clear that the equality cross term is essential in proving the second clause.

In the Appendix we show that this rule is sound and in fact equivalent to the rule of computation induction specialized to proofs of partial correctness.

Example 4 (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.

$F(x,y) \Leftarrow$ **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:

$G(x) \Leftarrow$ **if** $atom(x)$ **then** $cons(x, NIL)$
else $A)G(car(x)),G(cdr(x)))$.

The auxiliary function A (i.e. Append) is defined:

$A(x,y) \Leftarrow$ **if** $null(x)$ **then** y **else** $cons(car(x),A(cdr(x),y))$.

It suffices to prove the following output assertion for F: $\Psi_F(x, y; z) \equiv \{z = A(G(x), y)\}$. The verification conditions are:

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

which is obviously valid, upon expansion of G and A:

$$\begin{aligned} &\sim atom(x) \wedge z_1 = A(G(cdr(x)), y) \\ &\wedge z_2 = A(G(car(x)), z_1) \\ &\wedge [\langle cdr(x), y\rangle = \langle car(x), z_1\rangle \rightarrow z_1 = z_2] \rightarrow \\ &z_2 = A(G(x), y) \end{aligned}$$

Substituting for equals and expanding G for a nonatomic argument, this simplifies to

$$\begin{aligned} &\sim atom(x) \wedge [\langle cdr(x), y\rangle = \langle car(x), z_1\rangle \rightarrow z_1 = z_2] \rightarrow \\ &A\{A(G(car(x)), G(cdr(x))), y\} \\ &= A\ \{G(car(x)), A(G(cdr(x), y))\} \end{aligned}$$

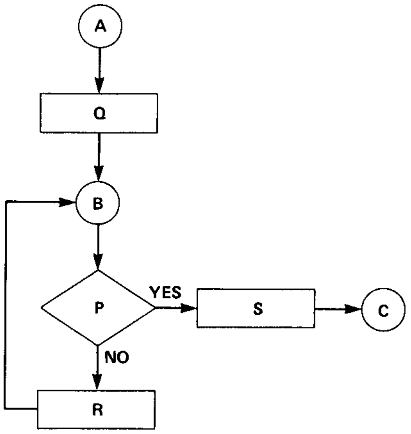
an instance of a simple fact about A—$A(A(u, v), w) = A(u, A(v, w))$, i.e. that A is associative. □

3. Comparison with Inductive Assertion Method

Subgoal induction and the inductive assertion method are symmetric alternatives for proving the following sort of statement: For all computations which start at A and end at B, relation Ψ holds between the beginning and final states of the data. The inductive assertion method is based upon an induction on the number of steps since the computation started at A. Subgoal induction is based upon the number of steps until it halts at B.

Consider the flow chart in Figure 4. Q, R, and S are input-output relations describing the effect of their

Fig. 4. A loop scheme.



respective boxes. P is a predicate. Suppose one wishes to show that a relation $\Delta(x_A; x_C)$ holds for any path from A to C. A conceivable approach might be to prove that Δ holds for A-to-C computations which visit point B once, twice, etc. This approach does not work out for the following reason: An A-to-C computation of length $n + 1$ does not contain any A-to-C computation of length n. An inductive method based on path length seems to work only when the longer paths contain the shorter ones. There are three families of paths one might consider: those from A to B, those from B to C, or those from B to B (i.e. ones which start and end at point B).

The inductive assertion method chooses the first family by inventing the subproblem of proving that a relation $\Gamma(x_A; x_B)$ holds for all paths from A to B, and then using Γ to prove that Δ holds between A and C. Thus, one must prove (for all x_A, x_B, x_C):

$$Q(x_A; x_B) \rightarrow \Gamma(x_A; x_B) \tag{3.1}$$
$$\Gamma(x_A; x_B) \wedge \sim P(x_B) \wedge R(x_B; x'_B) \rightarrow \Gamma(x_A; x'_B) \tag{3.2}$$
$$\Gamma(x_A; x_B) \wedge P(x_B) \wedge S(x_B; x_C) \rightarrow \Delta(x_A; x_C) \tag{3.3}$$

Figure 5 shows how Figure 4 can be expanded into an infinite flow chart containing all the paths under discussion. (3.1) and (3.2) establish that Γ holds between A and B_i for $i = 1, 2, \ldots$. (3.3) finishes the proof by showing that Δ holds between A and each C_i.

The subgoal induction method decomposes the problem in the reverse way: Invent a relation $\Psi(x_B; x_C)$ and prove that it holds for all B-to-C paths and then use Ψ to prove Δ holds between A and C, i.e. prove:

$$Q(x_A; x_B) \wedge \Psi(x_B; x_C) \rightarrow \Delta(x_A; x_C) \tag{3.4}$$
$$\sim P(x_B) \wedge R(x_B; x'_B) \wedge \Psi(x'_B; x_C) \rightarrow \Psi(x_B; x_C) \tag{3.5}$$
$$P(x_B) \wedge S(x_B; x_C) \rightarrow \Psi(x_B; x_C) \tag{3.6}$$

Figure 6 shows the flow chart of Figure 4 expanded in the reverse way so that point C occurs only once and there are an infinite number of starting points. Clauses (3.6) and (3.5) serve to show that Ψ holds between B_i and C for $i = 1, 2, \ldots$. Clause (3.4) shows that Δ holds between points A_i and C, for $i = 1, 2, \ldots$.

Formally speaking, a proof by either of these two methods can be used to produce a proof by the other.

Suppose one has a proof by inductive assertions; i.e. proofs of (3.1), (3.2), and (3.3). To prove Δ by subgoal induction, define:

$$\Psi(x_B; x_C) \equiv (\forall x_A)[\Gamma(x_A; x_B) \rightarrow \Delta(x_A; x_C)]$$

This formula is a sort of circumlocution of (3.2) and (3.3). It asks us to prove, for any path from B to C, and a state x_A such that $\Gamma(x_A; x_B)$ holds, that $\Delta(x_A; x_C)$ holds. Since (3.2) proves that Γ will be maintained as one passes around the loop, we need only consider the path directly from B to C; and (3.3) proves that Δ will hold if the $B - C$ path is followed. Finally, to complete the proof, one uses (3.1) to show that $\Gamma(x_A; x_B)$ does hold if point B is reached through Q. To summarize: Δ can be proved by subgoal induction be-

Fig. 5. A forward expansion of Fig. 4.

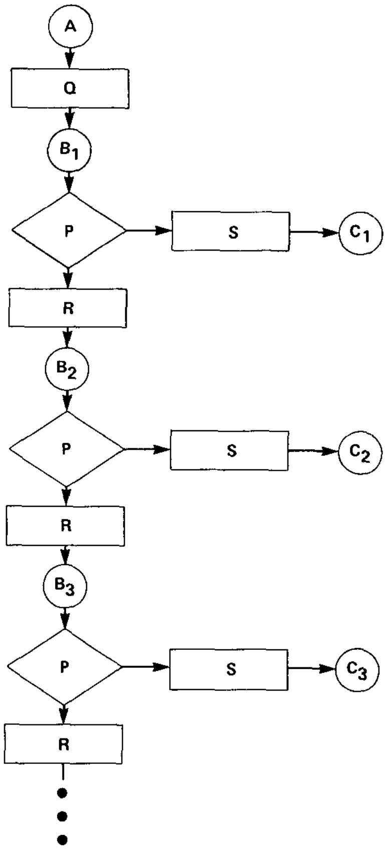
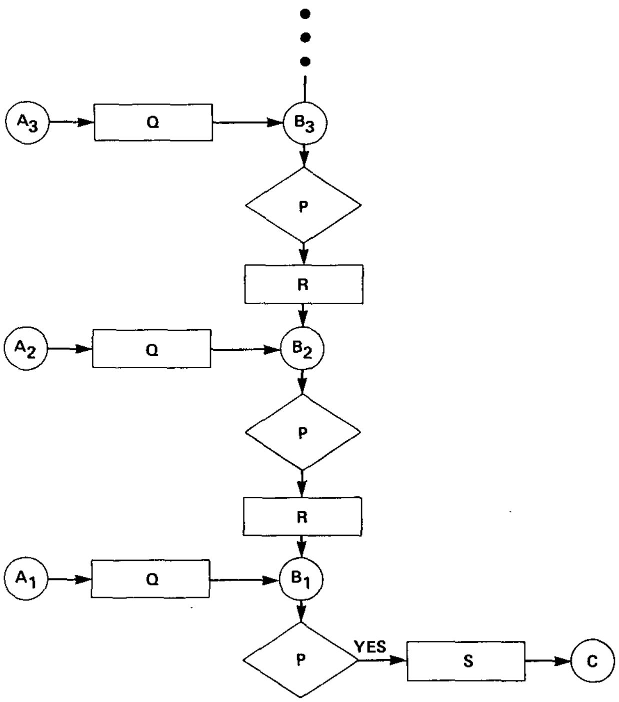


Fig. 6. A backward expansion of Fig. 4.



cause (3.4) follows from (3.1), (3.5) from (3.2), and (3.6) from (3.3).

Suppose one has a proof by subgoal induction; i.e. proofs of (3.4), (3.5), and (3.6). To prove Δ by inductive assertions define:

$$\Gamma(x_A; x_B) = (\forall x_C)[\Psi(x_B; x_C) \rightarrow \Delta(x_A; x_C)]$$

Then Δ can be proved by the inductive method because (3.1) follows from (3.4), (3.2) follows from (3.5), and (3.3) follows from (3.6). This translation technique is discussed in [19].

Despite this formal equivalence between the methods it appears that choosing the right one can sometimes make both the discovery of an induction hypothesis and the subsequent proof simpler.

First, if Q is a null operation then (3.4) becomes $\Psi(x_A; x_C) \rightarrow \Delta(x_A; x_C)$ so Δ itself is a good candidate for the subgoal hypothesis Ψ. Therefore subgoal induction seems like a better choice. On the other hand, when S is a null operation the corresponding advantage does not accrue to the inductive assertion method. (3.3) becomes:

$$\Gamma(x_A; x_B) \wedge P(x_B) \rightarrow \Delta(x_A; x_B)$$

and Δ is not a reasonable choice for Γ except in the unlikely case that it does not depend upon the test P. In other words, there is a special case where the subgoal induction method does not require any new assertions to be invented, and there is no such special case for the inductive assertion method.

A drawback of subgoal induction is that it cannot be used to prove invariants about nonterminating programs. For example, to prove that i is always positive in the program:

$i \leftarrow 1$;   **while** true **do** $i \leftarrow i + 1$

one just proves (3.1) and (3.2) for $\Gamma(i_0; i) \equiv i > 0$. The fact that subgoal induction can be used to prove invariants about programs which halt but never start is not much consolation.

Finally, let us consider two examples which illustrate the advantages of the methods. Consider the program:

```
$j \leftarrow 1$;
while $T[j] \neq 0$ do $j \leftarrow j + 1$;
$T[j] \leftarrow 2$
```

The relation to be proved is:

$$\Delta \equiv (\forall i)[T_f[i] = \textbf{if } i = m \textbf{ then } 2 \textbf{ else } T_0[i]]$$

where $m = \min \{i \mid i \geq 1 \wedge T_0[i] = 0\}$, i.e. T has had its first 0 changed to a 2. To prove this by the inductive assertion method, one might employ the loop assertion:

$$\Gamma(T_0; T, j) \equiv T_0 = T \wedge (\forall 1 \leq i < j)[T[i] \neq 0]$$

Having established it, one then must show that the final step achieves Δ. On the other hand, the induction hypothesis needed for a subgoal induction proof is the minor generalization one gets by replacing 1 in Δ by j, i.e.

$$\Psi(T, j; T_f) \equiv (\forall i)[T_f[i] = \textbf{if } i = m \textbf{ then } 2 \textbf{ else } T_0[i]]$$

where $m = \min \{i \mid i \geq j \wedge T[i] = 0\}$ Once this is proved, it is trivial to show that Δ itself holds by specializing to the case $j = 1$. The subgoal induction approach is doubly beneficial here: first, the effort to devise an induction hypothesis is less, and second, the final step of the proof is much simpler.

In other cases, the use of an inductive assertion proof seems more natural. Consider the program:

```
$i \leftarrow 1$
while $i < 10$ do $i = i + 1$
```

To prove that the final value of i is 10, one uses the inductive assertion $i \leq 10$. If one wanted to use the subgoal induction method he would use the relation:

$$\Psi(i_0; i_f) \equiv i_0 \leq 10 \rightarrow i_f = 10$$

This hypothesis contains the inductive assertion as a subpart and seems to be less intuitive than the former.

4. 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 $F(x) \Leftarrow$ **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 (2.1) and (2.2) for Ψ, and rearranging, we obtain:

$$P(x) \wedge \Phi(x) \rightarrow \Theta(x; H(x)) \tag{4.1}$$

$$\begin{aligned} &\sim P(x) \wedge \Phi(x) \wedge [\Phi(N(x)) \rightarrow \Theta(N(x); z)] \\ &\rightarrow \Theta(x; L(x, z)) \end{aligned} \tag{4.2}$$

(4.1) is straightforward: it requires that if the input constraint is satisfied and the program terminates immediately then the output, $H(x)$, be an acceptable output. (4.2), however, is more complex. Suppose the left-hand side of (4.2) 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

$$[\Phi(N(x)) \rightarrow \Theta(N(x); z)]$$

Since it is the case that 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:

$$\sim P(x) \wedge \Phi(x) \rightarrow \Phi(N(x)) \qquad (4.3)$$

If true, (4.3) would guarantee that whenever the left-hand side of (4.2) is true, $\Phi(N(X))$ is true and hence $\Theta(N(x); z))$ is true. It would then suffice to prove:

$$\begin{aligned} &\sim P(x) \wedge \Phi(x) \wedge \Theta(N(x); z) \\ &\rightarrow \Theta(x; L(x, z)) \qquad (4.4) \end{aligned}$$

Some insight may be gained by inspecting (4.3). 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 (4.3) 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 (4.4). This is somewhat surprising in that Φ and (4.3) 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 (4.1), (4.3), and (4.4) 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 5 (Binary search). Consider the loop:

```
maintain $\{x \leq y \wedge (\forall i \mid x \leq i < y)\ A[i] \leq A[i+1]\}$
while $x \neq y$
do begin $w \leftarrow (x+y)/2$;
        if $key > A[w]$ then $x \leftarrow w + 1$ else $y \leftarrow w$
   end
thus $\{key = A[x] \equiv (\exists i' \mid x_0 \leq i' \leq y_0)\ key = A[i']\}$
```

(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 $\Phi(x, y)$ denote this invariant, then the other verification conditions are:

$$\begin{aligned} &x = y \wedge \Phi(x, y) \\ &\rightarrow \{key = A[x] \equiv (\exists i' \mid x \leq i' \leq y)\ key = A[i']\} \end{aligned}$$

$$\begin{aligned} &\{x \neq y \wedge \Phi(x, y) \wedge w = (x + y)/2 \wedge key > A[w] \\ &\wedge (key = A[z] \equiv (\exists i \mid w + 1 \leq i \leq y)\ key = A[i])\} \\ &\rightarrow \{key = A[z] \equiv (\exists i' \mid x \leq i' \leq y)\ key = A[i']\} \end{aligned}$$

$$\begin{aligned} &\{x \neq y \wedge \Phi(x, y) \wedge w = (x + y)/2 \wedge key \leq A[w] \\ &\wedge (key = A[z] \equiv (\exists i \mid x \leq i \leq w)\ key = A[i])\} \\ &\rightarrow \{key = A[z] \equiv (\exists i' \mid x \leq i' \leq y)\ key = A[i']\} \qquad \square \end{aligned}$$

Let us now consider a more technical aspect of this combined method: Is it "as powerful" as subgoal induction in the sense that (4.2) implies (4.3) and (4.4)? 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 recurs ($P(x')$ = false), but for which any outcome whatever is acceptable according to Θ. This means that the function is needlessly continuing to recur. We cannot think of any real examples in which such a situation occurs.

THEOREM 1. *If* Ψ *is well-behaved with respect to* F *and if* (4.2) *is valid, then* (4.3) *and* (4.4) *are valid.*

PROOF (by contrapositive). First note that (4.2) implies (4.4) immediately. Suppose that 4.2 is valid but that (4.3) is not valid; we will show that Ψ is not well-behaved. Since (4.3) is not valid, it is false for some x, say x':

$$\sim P(x') \wedge \Phi(x') \wedge \sim\Phi(N(x')) \qquad (\sim 4.3)$$

is true. Consider (4.2) for x':

$$\begin{aligned} &\sim P(x') \wedge \Phi(x') \wedge [\Phi(N(x')) \rightarrow \Theta(N(x,); z)] \\ &\rightarrow \Theta(x'; L(x', z)) \end{aligned}$$

Using the truth of ($\sim$4.3), this simplifies to:

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

Since (4.2) is valid, this must be true for all z. Thus Ψ is not well-behaved.[2] □

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:

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

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

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

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.

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 6 (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 subarray 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 $n$;   $n \leftarrow length(A)$;
if $n = 1$ return $A$;
begin real array [1:n] $S, M, B$;   int $s, m, b$;   real $x$;
  $s \leftarrow m \leftarrow b \leftarrow 0$;   $x \leftarrow A[n/2]$;
  for $j$ from 1 to $n$ do
    maintain $(\forall i \mid 1 \le i \le s)\ (S[i] < x) \wedge (\forall i \mid 1 \le i \le m)\ (M[i] = x)$
             $\wedge (\forall i \mid 1 \le i \le b)\ (x < B[i])$
             $\wedge\ perm(A[1{:}j-1], S[1{:}s] \circ M[1{:}m] \circ B[1{:}b])$;
    if $A[j] < x$ then $S[s \leftarrow s+1] \leftarrow A[j]$
    else if $A[j] = x$ then $M[m \leftarrow m+1] \leftarrow A[j]$
    else $B[b \leftarrow b+1] \leftarrow A[j]$;
  return $PSort(S[1{:}s]) \circ M[1{:}m] \circ PSort(B[1{:}b])$
end
end $PSort$ output assertion ordered$(PSort(A)) \wedge perm(A, PSort(A))$
```

where ordered and perm are defined:

$$ordered(A) \equiv (\forall i \mid 1 \le i < length(A))\ A[i] \le A[i+1]$$
$$perm(A, B) \equiv length(A) = length(B)\ \wedge\ \exists R$$
$$((\forall i \mid 1 \le i < length(A))\ (1 \le R[i] \le length(A))$$
$$\wedge\ (\forall i, i' \mid 1 \le i < i' \le length(A))\ (R[i] \ne R[i'])$$
$$\wedge\ (\forall i \mid 1 \le i \le length(A))\ (A[R[i]] = B[i]))$$

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. The first is:

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

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:

$$\begin{aligned}\Psi_F(; A, S, B, s, b, x) \equiv\ & (\forall i \mid 1 \le i \le s)\ (S[i] < x)\\ &\wedge\ (\forall i \mid 1 \le i \le m)\ (M[i] = x)\\ &\wedge\ (\forall i \mid 1 \le i \le b)\ (x < B[i])\\ &\wedge\ perm(A, S[1{:}s] \circ M[1{:}m] \circ B[1{:}b])\end{aligned}$$

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 $L(c, d, e) = c \circ d \circ e$; then the second verification condition may be written:

[2] 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. (4.2) 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 (4.3) does not depend on Θ, it is less complex than (4.2). If the difference in complexity is significant, (4.3) may provide a useful filter for testing whether Ψ is complete. If (4.3) can be shown to be invalid, then the above theorem establishes that Ψ is incomplete, without further consideration of Θ.

$$\sim P(\xi) \wedge \Psi_F(\xi; z_F) \wedge \Psi_S(N_1(z_F); z_1)$$
$$\wedge\ \Psi_S(N_2(z_F); z_2) \rightarrow \Psi_S(\xi; L(z_1, M[1{:}m], z_2))$$

That is,

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

Proof of this reduces to establishing two results:

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

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

which may be proved using the definitions of perm and ordered. □

5. 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 $F(x) \Leftarrow$ **if** $P(x)$ **then** $H(x)$ **else** $L(x, F(N(x)))$; conditions for more general forms are analogous.

We begin by considering a particularly straightforward case. Suppose that the relation $\Psi(x; z)$ is a function: For every x there is at most one z which satisfies $\Psi(x; z)$. Further suppose that F is total and that $\Psi(x; F(x))$ is valid; then it follows that $F = \Psi$, i.e. $\Psi(x; z) \equiv z = F(x)$. Expanding the definition of F in the valid formula $\Psi(x; F(x))$, it follows that:

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

is valid. Since $\Psi(x; z) \equiv z = F(x)$, the third conjunct on the left-hand side is subsumed by the second conjunct. Simplifying, this yields the valid formula:

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

But this is exactly (2.2). We may therefore conclude that if $\Psi(x; z)$ is a total function then the induction formula is valid.

The following definition and theorem extend this argument to a larger class of specifications.

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

$$\Phi(x) \wedge \sim P(x) \rightarrow \Phi(N(x)) \tag{5.1}$$

$$\sim P(x) \wedge \Phi(x) \wedge \Theta(N(x); z_1) \wedge \Theta(N(x); z_2)$$
$$\rightarrow L(x, z_1) = L(x, z_2) \tag{5.2}$$

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

THEOREM 2. *If* $\Psi(x; z) \equiv \Phi(x) \rightarrow \Theta(x; z)$ *is a tight specification, if* F *is total on the domain* $\{x \mid \Phi(x)\}$, *and if* $\Psi(x; F(x))$ *is valid, then*

$$\sim P(x) \wedge \Psi(N(x); z) \rightarrow \Psi(x; L(x, z)) \tag{2.2}$$

is valid.

PROOF. Rewrite (2.2) as:

$$\sim P(x) \wedge \Phi(x) \wedge [\Phi(N(x)) \rightarrow \Theta(N(x); z)]$$
$$\rightarrow \Theta(x; L(x, z)) \tag{5.3}$$

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:

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

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

$$\sim P(x') \wedge \Phi(x') \wedge \Phi(N(x')) \wedge \Theta(N(x'); z')$$
$$\wedge\ \Theta(N(x'); F(N(x'))) \tag{5.4}$$

From the definition of tight specification, this implies:

$$L(x', z') = L(x', F(N(x'))) \tag{5.5}$$

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

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

This, taken together with (5.4) and (5.5), implies:

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

which is the right-hand side of (5.1). Thus (3.5) is valid. □

Observe that if Φ is everywhere true and 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 $F \Leftarrow \tau[F]$ and $G \Leftarrow \sigma[G]$ 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 [14].

Example 7 (Under-constrained specification). An example may serve to make these considerations more concrete. Consider

$$F(x) \Leftarrow \textbf{if } x \leq 1 \textbf{ then } \langle 0, x\rangle \textbf{ else } \langle F(x-1)[2], F(x-1)[1]$$
$$+ F(x-1)[2]\rangle$$

where angle brackets denote the forming of ordered pairs and subscripts denote the decomposition of ordered pairs. We wish to prove $\Psi(x; F(x))$ where

$$\Psi(x; z) \equiv x \geq 0 \rightarrow z[2] = Fib(x)$$

Here, *Fib* is the standard Fibonacci function defined as $Fib(n) \Leftarrow$ **if** $n = 0$ **then** 0 **else if** $n = 1$ **then** 1 **else** $Fib(n-1) + Fib(n-2)$. Referring to the schema of the theorem, $\Phi(x) \equiv x \geq 0$, $\Theta(x; z) \equiv z[2] = Fib(x)$, and $L(x, z) \equiv \langle z[2], z[1] + z[2]\rangle$. Referring to the definition, Φ is an invariant, F is total on the domain $\{x \mid x \geq 0\}$, and $\Psi(x; F(x))$ is valid. However, Ψ is not a tight specification: since Ψ constrains only the second component of z, it is possible to have two different z's which satisfy Θ; e.g. $x = 2$, $z_1 = \langle 1,1\rangle$, and $z_2 = \langle 13,1\rangle$ is an assignment of values for which the left-hand side of (5.2) is true but the right-hand side is false. As Ψ is not tight, Theorem 2 does not apply.

In fact, (2.2) is not valid; it reads:

$$x \geq 2 \wedge z[2] = Fib(x - 1) \rightarrow z[1] + z[2] = Fib(x)$$

Since the left-hand side in no way constrains $z[1]$, the right-hand side does not follow logically from the left.[3] Indeed, $x = 2$, and $z = \langle 13,1\rangle$ is a counterexample. Intuitively, the trouble is that Ψ is incomplete—it does not sufficiently constrain the value of z.

A tight specification is given by:

$$\Psi'(x; z) \equiv x \geq 0 \rightarrow z[2] = Fib(x) \wedge z[1] = Fib'(x - 1)$$

where $Fib'(x) \Leftarrow$ **if** $x < 0$ **then** 0 **else** $Fib(x)$. This satisfies the definition and, using it, (2.2) is valid. Of course, tight specifications are not unique and other extensions of Ψ are possible. □

Returning to the definition of tight specification, it is useful to examine the other requirement—that Φ be an invariant. Suppose the contrary, then $\Phi(x_0) = true$ but $\Phi(N^k(x_0)) = false$ for some k and x_0. Let $x = N^k(x_0)$. For such x, z is effectively unconstrained—any value whatever will do. In consequence, the validity of (2.2) is not guaranteed.

In summary, the definition of tight specification has been constructed so as to rule out two common defects of a specification:

(1) $\Psi(x; z)$ only specifies certain properties of z, so that additional conjuncts are needed to specify other properties.

(2) $\Psi(x; z)$ only specifies the outcome z for certain values of x, so that additional specification is needed for the remaining values.

Because of the intertranslatability of 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 separate issue.

6. Other Classes of Specifications

Thus far, we have considered only specifications of the form $(\forall x)\Psi(x; F(x))$ where F is a program and Ψ is a predicate. That is, our specifications have been concerned solely with establishing which input/output pairs $\langle x, F(x)\rangle$ are acceptable.

There are assertions one may wish to prove which do not have this form. For example, the requirement that F be monotonic may be expressed as:

$$(\forall x)\ (\forall y)\ [x < y \rightarrow F(x) < F(y)]$$

This has two occurrences of F and so does not fall neatly into the preceding paradigm. Similarly, the requirement that some binary operator be commutative or associative is not directly expressed as a set of acceptable input/output pairs. The purpose of this section is to discuss how cases such as these can be handled within the framework of subgoal induction.

The essential idea is to distinguish one occurrence of the function letter F. Subgoal induction is applied to the distinguished occurrence of F and used to construct verification conditions in the normal way. These conditions may contain the other occurrences of F. Thus, to prove these conditions are valid, it will then be necessary to reason about the properties of these other occurrences of F—by appealing to the definition. If this appeal is straightforward, the proof will go through without difficulty. An example will illustrate this.

Example 8 (Associativity of Append). Consider Append defined by:

$$A(u, v) \Leftarrow \textbf{if } null(u) \textbf{ then } v \textbf{ else } cons(car(u), A(cdr(u), v))$$

The assertion we wish to prove is:

$$A(A(u, v), w) = A(u, A(v, w))$$

We choose to distinguish the second appearance of A and, for the purpose of exposition, designate this as A^*. Thus, our goal is to prove that A^* satisfies:

$$A(A^*(u, v)\ w) = A(u, A(v, w))$$

Written as an input/output assertion, this is $\Psi_{A^*}(u, v; A^*(u, v))$ where

$$\Psi_{A^*}(u, v; z) \equiv A(z, w) = A(u, A(v, w))$$

[3] As a digressional point, we may observe that this can be simplified to $x \geq 2 \rightarrow z[1] = Fib(x-2)$ (by expanding the definition of $Fib(n)$). If the program is to be verified, this formula *must* be valid. Thus this formula can be added to the specification. In this way, it is sometimes possible to build up a complete specification by extracting information from proofs which fail. Details of this idea and its implementation in a program verifier are discussed in [5]. It should be noted that such techniques do not always yield a complete specification within a reasonable number of steps [19]. They are heuristics with certain circumscribed utility.

Applying the subgoal induction rule to this specification, we obtain two verification conditions. The first is:

$$null(u) \rightarrow A(v, w) = A(u, A(v, w))$$

This is valid since $A(u, \alpha) = \alpha$ when u is null. The second is:

$$\sim null(u) \wedge A(z, w) = A(cdr(u), A(v, w))$$
$$\rightarrow A(cons(car(u), z), w) = A(u, A(v, w))$$

Since $A(cons(\alpha, \beta), \gamma) = cons\ (\alpha, A(\beta, \gamma))$, the right-hand side of the implication is equal to:

$$cons(car(u), A(z, w)) = cons(car(u), A(cdr(u), A(v, w)))$$

which is an immediate consequence of the left-hand side of the implication. Thus the verification conditions are valid, establishing $\Psi_A(u, v; A(u, v))$, i.e. that A is associative. □

7. Related Work

Several papers have suggested related proof methods. Manna and Pnueli [11] showed how to transform a recursive function definition and specification into a first order formula containing an unspecified predicate Q, so that the function is partially correct with respect to the specification if a Q can be found which makes the formula true. By choosing Q based on the specification, this method can be viewed as a slightly weaker variant of subgoal induction. Manna in [20] applied the method of [11] to a flowchart program and noted that the resulting verification condition differed from that produced by the inductive assertion technique. The work of Basu, Misra [1], and Mills [14] has a similar rule, except that their specification is always a function rather than a general relation. Other authors [3, 7, 17] have presented similar ideas. Finally, subgoal induction can be viewed as a specialization of the rule of computation induction which was developed in the context of pure recursive functions [12].

8. Conclusion

Currently, there are three induction methods in common use for mechanical program verification: structural induction [2, 16], inductive assertions [6, 8, 18], and computation induction [15]. 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 Appendix A and Section 3 establish the formal equivalence of computation induction to subgoal induction and of inductive assertions to subgoal induction restricted to flowchart programs; further, [12] 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 setup. 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.

Appendix

In this section[4] we shall prove that the rule of subgoal induction is equivalent to the rule of computation induction specialized to proofs of partial correctness. Thus we show not only that it is sound but also that there is no need to use the apparently more general rule.

In order to treat termination questions carefully we shall adopt the view that a partial function is a special case of a relation. Instead of saying "$F(x)$ is defined and $F(x) = z$" we say "$F(x; z)$" which means "$\langle x, z\rangle \in F$". The use of the semicolon serves to distinguish this from an application of F to two arguments. A major convenience of this approach is that the partial correctness of F with respect to Ψ can be

[4] This section was written in collaboration with Howard Sturgis of Xerox, Palo Alto Research Center.

stated quite succinctly by "$F \subseteq \Psi$" which means "$(\forall x, z)[F(x; z) \rightarrow \Psi(x; z)]$".

First, let us specify precisely the function F defined by (2.7) from Section 2. We assume that all the primitive functions and predicates involved are total. Based upon the right-hand side of (2.7) we define a functional τ, mapping partial functions into partial functions, as follows:

$$\tau[F] = \lambda x \textbf{ if } P_1 \textbf{ then } E_1 \textbf{ else if } P_2 \textbf{ then } E_2 \ldots \textbf{ else } E_n$$

where the E_i may contain F and the P_i do not. In relational notation this is:

$$\tau[F] \equiv \tau_1[F] \cup \ldots \cup \tau_n[F]$$

where each τ_i is defined by a statement of the form:

$$\tau_i[F](x; z) \equiv \bigwedge_{j=1}^{i-1} \sim P_j$$
$$\wedge \ P_i \ \wedge \ (\exists z_1 \ldots z_m)\left[\bigwedge_{k=1}^{m} F(\gamma_k; z_k) \ \wedge \ z = \gamma_0\right]$$

where the $m + 1$ terms $\gamma_0, \ldots, \gamma_m$ are derived from E_i by replacing subterms of the form $F(\gamma)$ by z's in an inside-out manner until no occurrences of F remain. To be more precise, suppose E_i contains m occurrences of the function letter F. If $m = 0$, simply choose $\gamma_0 = E_i$, otherwise find a subterm of the form $F(\gamma)$ where γ does not contain F, choose $\gamma_m = \gamma$, replace the occurrence of $F(\gamma)$ in E_i by z_m, and repeat. This is the same as the algorithm for constructing the γ_i's given in Section 2.

For example, let us consider the specific function definition:

$$K(x, y) \Leftarrow \textbf{if } P(x) \textbf{ then } H(x) \textbf{ else } K(M(x), K(N(x), y))$$

then

$$\tau_1[F](x, y; z) \equiv P(x) \ \wedge \ z = H(x)$$
$$\tau_2[F](x, y; z) \equiv \sim P(x) \ \wedge \ (\exists z_1, z_2)[F[N[x], y; z_2)$$
$$\wedge \ F(M(x), z_2; z_1) \ \wedge \ z = z_1]$$

This definition of τ properly captures the intuitive notion of a functional mapping of a partial function F into another partial function, assuming that all the other functions involved are total.

Now define the family of functions $\{F_i\}$ by:

$F_0 = \varnothing$, i.e. the everywhere undefined function
$F_{i+1} = \tau[F_i]$

Then

$$F = \bigcup_{i=0}^{\infty} F_i.$$

This way of defining F, and the fact that it corresponds properly to our notion of how to compute F is essentially Kleene's first recursion theorem. Intuitively, F_i is that defective version of F which one can compute without ever using more than i stack frames; i.e. F_i is programmed to go into an infinite loop if it ever attempts to use the $(i + 1)$th frame. Then $\bigcup_{i=0}^{\infty} F_i$ is just the function one gets by letting the available stack space get arbitrarily large.

For example,

$K_0 = \varnothing$
$K_1(x, y; z) \equiv P(x) \ \wedge \ z = H(x)$
$K_2(x, y; z) \equiv P(x) \ \wedge \ z = H(x) \ \vee \ \sim P(x) \ \wedge \ P(N(x))$
$\wedge \ P(M(x)) \ \wedge \ z = H(M(x))$

Note that this definition uses "call-by-value" semantics. For example, if N is the identity function and $P(x)$ is false then $K(x, y)$ is undefined even if $P(M(x))$ is true. If "call-by-name" semantics were used, $K(x, y)$ would be $H(M(x))$.

Associated with this way of assigning meaning to a recursive definition is an induction rule for proving things about the defined function, the rule of computation induction [12]: To prove $\alpha[F]$, prove $\alpha[\varnothing]$ and $(\forall G)[\alpha[G] \rightarrow \alpha[\tau[G]]]$ where G ranges over all functions. There are various restrictions on α which need not concern us because we are interested only in the specialization of this rule to the case where $\alpha[G]$ is "$G \subseteq \Psi$". The base case for this specialization, $\varnothing \subseteq \Psi$, is always true; and $(\forall G)[G \subseteq \Psi \rightarrow \tau[G] \subseteq \Psi]$ is equivalent to

$$(\forall G)[G \subseteq \Psi \rightarrow (\tau_1[G] \cup \ldots \cup \tau_n[G]) \subseteq \Psi]$$

which in turn is equivalent to the conjunction of the n statements

$$(\forall G)[G \subseteq \Psi \rightarrow \tau_i[G] \subseteq \Psi] \qquad \text{(A1)}$$

for $i = 1, \ldots, n$. Thus a computation induction proof of $F \subseteq \Psi$ amounts to proving the n statements with the form (A1). From now on let us concentrate on proving one such statement.

At this point it is tempting to note that τ_i can be applied to any arbitrary relation and that $\tau_i[\Psi] \subseteq \Psi$ implies (A1) since $G \subseteq \Psi \rightarrow \tau_i[G] \subseteq \tau_i[\Psi]$ by the monotonicity of τ_i. The statement $\tau_i[\Psi] \subseteq \Psi$ turns out to be the ith clause of a subgoal induction proof with its equality cross-terms missing. Thus it would be easy to show that a slightly less powerful rule is sound because it implies a proof by computation induction. We are after bigger game, however.

We shall show that (A1) is equivalent to the ith clause of a subgoal induction proof, i.e.:

$$\bigwedge_{i=1}^{i=1} \sim P_j \ \wedge \ P_i \ \wedge \ \bigwedge_{k=1}^{m} \Psi(g_k; z_k)$$
$$\wedge \ \bigwedge_{1 \leq s < t \leq m} (\gamma_s = \gamma_t \rightarrow z_s = z_t) \rightarrow \Psi(x; \gamma_0) \qquad (2.8)$$

from Section 2. To reduce the notational complexity we shall use the abbreviation:

$$R(\Gamma; Z) \text{ means } \bigwedge_{k=1}^{m} R(\gamma_k; z_k)$$

for any relation R.

First, we prove a lemma which shows how the quantified function letter, G, can be dispensed with.

Lemma A1. Let $\gamma_0, \ldots, \gamma_m$ be terms in variables $x, z_1, \ldots, z_m$. Let Ψ be a relation. Then:

$$\Psi(\Gamma; Z) \wedge \bigwedge_{1 \leq s < t \leq m} (\gamma_s = \gamma_t \rightarrow z_s = z_t) \tag{A3}$$

is equivalent to

$$(\exists G)[G(\Gamma; Z) \wedge G \subseteq \Psi] \tag{A4}$$

where G ranges over all partial functions.

Proof. To show (A3) → (A4) define G to be the finite relation $\{\langle\gamma_1, z_1\rangle, \ldots, \langle\gamma_m, z_m\rangle\}$. First by

$$\bigwedge_{1 \leq s < t \leq m} (\gamma_s = \gamma_t \rightarrow z_s = z_t),$$

G is a function. Then by definition $G(\Gamma; Z)$. Finally, by $\Psi(\Gamma; Z)$, $G \subseteq \Psi$.

To show (A4) → (A3): by $G(\Gamma; Z)$ and $G \subseteq \Psi$ we have $\Psi(\Gamma; Z)$. The fact that G is a function and $G(\Gamma; Z)$ imply:

$$\bigwedge_{1 \leq s < t \leq m} (\gamma_s = \gamma_t \rightarrow z_s = z_t). \qquad \square$$

Theorem A1. The two statements (A1) and (2.8) are equivalent.

Proof. The proof proceeds by a sequence of equivalence-preserving transformation to (A1) which lead to (2.8). First, (A1) is equivalent to:

$(\forall G)[G \subseteq \Psi \rightarrow (\forall x, z)[\tau_i[G](x; z) \rightarrow \Psi(x; z)]]$, by the definition of $\subseteq$

$$(\forall G)\left[G \subseteq \Psi \rightarrow (\forall x, z)\left[\bigwedge_{j=1}^{i=1} \sim P_j \wedge P_i \wedge (\exists z_1 \ldots z_m)[G(\Gamma; Z) \wedge z = \gamma_0] \rightarrow \Psi(x; z)\right]\right]$$

by expanding τ_i'. Henceforth, let us abbreviate

$$\bigwedge_{j=1}^{i=1} \sim P_j \wedge P_i$$

by P^*.

$(\forall G)[G \subseteq \Psi \rightarrow [P^* \wedge G(\Gamma; Z) \wedge z = \gamma_0 \rightarrow \Psi(x; z)]]$,

by moving all quantifiers to the outside, and dropping universals (except G).

$(\forall G)[G \subseteq \Psi \rightarrow [P^* \wedge G(\Gamma; Z) \rightarrow \Psi(x; \gamma_0)]]$

by substitution for the lone occurrence of z.

$P^* \wedge (\exists G)[G \subseteq \Psi \wedge G(\Gamma; Z)] \rightarrow \Psi(x; \gamma_0)$,

by moving the quantifier inward. G does not occur in P^* or $\Psi(x; \gamma_0)$.

$$P^* \wedge \Psi(\Gamma; Z) \wedge \bigwedge_{1 \leq s < t \leq m} (\gamma_s = \gamma_t \rightarrow z_s = z_t) \rightarrow \Psi(x; \gamma_0),$$ by Lemma A1.

This statement, except for the abbreviations, is identical to statement (2.8). $\square$

Acknowledgments. 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.

Received June 1975; revised October 1975

References

1. Basu, S., and Misra, J. Proving loop programs. *IEEE Trans. Software Eng. SE-1*, 1 (March 1975), 76–86.
2. Boyer, R. and Moore, J.S. Proving theorems about LISP functions. *J. ACM 22*, 1 (Jan. 1975), 129–144.
3. Burstall, R.M. Proving properties of programs by structural induction. *Computer J. 12*, 1 (Feb. 1969), 41–48.
4. Floyd, R.W. Assigning meanings to programs. *Math. Aspects of Computer Science*, J.T. Schwartz, Ed., Amer. Math. Soc., Providence, R.I., 1967, pp. 19–32.
5. German, S.M., and Wegbreit, B. A synthesizer of inductive assertions. *IEEE Trans. Software Eng., SE-1*, 1 (March 1975), 68–75.
6. Good, D.I., London, R.L., and Bledsoe, W.W. An interactive program verification system. Proc. Int. Conf. on Reliable Software, Los Angeles, Calif., April 1975.
7. Hoare, C.A.R. Procedures and parameters: an axiomatic approach. *Lecture Notes in Mathematics 188*, E. Engeler, Ed., Springer-Verlag, 1971.
8. Igarashi, S., London, R.L., and Luckham, D.C. Automatic program verification I: Logical basis and its implementation. AIM-200, Stanford Artificial Intelligence Proj., Stanford U., Stanford, Calif., 1972.
9. McCarthy, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems,* P. Braffort and D.S. Hirschberg, Eds., North-Holland, Amsterdam, 1963, pp. 33–70.
10. McCarthy, J. and Painter, J.A. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, J.T. Schwartz, Ed., Amer. Math. Soc., Providence, R.I., 1967, pp. 33–41.
11. Manna, Z., and Pnueli, A. Formalization of properties of functional programs. *J. ACM 17*, 3 (July 1970), 555–569.
12. Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. *Comm. ACM 16*, 8 (Aug. 1973), 491–502.
13. Manna, Z. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.
14. Mills, H. The new math of computer programming. *Comm. ACM 18*, 1 (Jan. 1975), 43–48.
15. Milner, R. Implementation and applications of Scott's logic for computable functions. SIGPLAN Notices (ACM) *7*, 1 and SIGACT News (ACM) *14* (Jan. 1972), 1–6.
16. Moore, J.S. Introducing prog into the pure lisp theorem prover. CSL-74-3, Xerox Palo Alto Res. Center, Palo Alto, Calif. (Dec. 1974).
17. Topor, R.W. Interactive program verification using virtual programs. Ph.D. Th., University of Edinburgh, Edinburgh, 1975.
18. Waldinger, R.J., and Levitt, K.N. Reasoning about programs. *Artificial Intelligence 5*, 3 (Fall 1974), 235–316.
19. Wegbreit, B. Complexity of synthesizing inductive assertions. Comptr. Sci. Lab., Xerox Palo Alto Res. Center, Palo Alto, Calif., Jan. 1975.
20. Manna, Z. Mathematical theory of partial correctness. *J. Computer System Sci. 5*, 3 (June 1971), 239–253.