STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-127
COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS 174

TOWARDS AUTOMATIC PROGRAM SYNTHESIS

BY

ZOHAR MANNA
AND
RICHARD J. WALDINGER

JULY 1970

COMPUTER SCIENCE DEPARTMENT

STANFORD UNIVERSITY

TOWARDS AUTOMATIC PROGRAM SYNTHESIS

by

Zohar Manna
Computer Science Department
Stanford University
Stanford, California

Richard J. Waldinger
Artificial Intelligence Group
Stanford Research Institute
Menlo Park, California

Abstract: An elementary outline of the theorem-proving approach to automatic program synthesis is given, without dwelling on technical details. The method is illustrated by the automatic construction of both recursive and iterative programs operating on natural numbers, lists, and trees.

In order to construct a program satisfying certain specifications, a theorem induced by those specifications is proved, and the desired program is extracted from the proof. The same technique is applied to transform recursively defined functions into iterative programs, frequently with a major gain in efficiency.

It is emphasized that in order to construct a program with loops or with recursion, the principle of mathematical induction must be applied. The relation between the version of the induction rule used and the form of the program constructed is explored in some detail.

The research reported herein was sponsored in part by the Air Force Systems Command, USAF, Department of Defense through the Air Force Cambridge Research Laboratories, Office of Aerospace Research, under Contract No. F19628-70-C-0246, and by the Advanced Research Projects Agency of the Department of Defense and the National Aeronautics and Space Administration under Contract No. NAS 12-2221 (at Stanford Research Institute); and by the Advanced Research Projects Agency of the Office of the Secretary of Defense, ARPA Order No. SD-183 (at Stanford University).

1. INTRODUCTION

It is often easier to describe what a computation does than it is to define it explicitly. That is, we may be able to write down the relation between the input and the output variables easily, even when it is difficult to construct a program to satisfy that relation. A program synthesizer is a system that takes such a relational description and tries to produce a program that is guaranteed to satisfy the relationship, and therefore does not require debugging or verification.

On a more limited scale we can envision an automatic debugging system that corrects programs written by humans instead of merely verifying them. We can further imagine clever compilers and optimizers that understand the operation of the programs they manipulate and that can transform them intelligently.

Some program synthesizers have already been written, including the Heuristic Compiler (Simon [1963]), DEDUCOM (Slagle [1965]), QA3 (Green [1969a], [1969b]), and PROW (Waldinger and Lee [1969] and Waldinger [1969]). The last three of these systems use a theorem-proving approach: in order to construct a program satisfying a certain input-output relation, the system proves a theorem induced by this relation and extracts the program directly from the proof. All three used the resolution principle of Robinson [1965]. However, these systems have been fairly limited; for example, they either have been completely unable to produce programs with loops, or they introduced loops by underhanded methods.

When a theorem-proving approach is used in program synthesis, the introduction of loops into the extracted program is closely related to the use of the principle of mathematical induction in the corresponding proof. The induction principle presented special problems to the earlier program-

synthesis systems, problems which limited their ability to produce loop programs. These problems are discussed in this paper. We propose to use a variety of different versions of the induction rule, each of which applies to a particular data structure, and each of which induces a different form in the extracted program. The data structures treated are the natural numbers, lists, and trees.

We do not rely on any specific mechanical theorem-proving techniques here, both because we do not wish to restrict our class of readers to those familiar with, say, the resolution principle, and because we believe the approach to be more general and not dependent on one particular theorem-proving method. We give a large number of examples of programs, with the corresponding theorems and proofs used in their synthesis. The proofs we give are informal and in the style of a mathematics textbook. Some of them have been achieved by such systems as PROW and QA3; others we believe to be beyond the powers of existing automatic theorem provers, but none are unreasonably difficult, and we hope that the designers of theorem-proving systems will accept them as a challenge.

Section 2 gives the flavor of the approach illustrated by three examples. In that section we do not prove the induced theorems, and we present the constructed programs without describing the extraction process. In Section 3 we demonstrate the extraction process with complete examples of the synthesis of two programs without loops. We choose loop-free programs for these examples so as to postpone discussion of the principle of mathematical induction.

The heart of the paper is contained in Section 4, with the presentation of the induction principles and their corresponding iterative or recursive

program forms. One of the examples in this section gives details of the proof and program extraction process. Section 5 demonstrates a more general rule, the complete induction principle. Section 6 suggests applying program-synthesis techniques to translate recursive programs into iterative programs, and presents two examples, in which a striking gain in efficiency was achieved. Finally, in Section 7 we suggest further research in this field.

2. GENERAL DISCUSSION

We define the problem of automatic program synthesis as follows: given an input predicate $\varphi(\bar{x})$ and an output predicate $\psi(\bar{x},\bar{z})$, construct a program computing a partial function $\bar{z} = f(\bar{x})$ such that if $\bar{x}$ is an input vector satisfying $\varphi(\bar{x})$, then $f(\bar{x})$ is defined and $\psi(x,f(\bar{x}))$ is true. In short, the predicates $\varphi(\bar{x})$ and $\psi(\bar{x},\bar{z})$ provide the specifications for the program to be written.

In order to construct such a program, we prove the theorem

$$(\forall\bar{x})[\varphi(\bar{x}) \supset (\exists\bar{z})\psi(\bar{x},\bar{z})] .$$

The desired program is then implicit in the proof that the output vector $\bar{z}$ exists. The theorem prover must be restricted to show the existence of $\bar{z}$ constructively, so that the appropriate program can be extracted from the proof automatically.

Frequently, $\varphi(\bar{x})$ is identically true; i.e., we are interested in the performance of the program for every input $\bar{x}$. Then the theorem to be proved is simply

$$(\forall\bar{x})[T \supset (\exists\bar{z})\psi(\bar{x},\bar{z})] ,$$

or equivalently,

$$(\forall\bar{x})(\exists\bar{z})\psi(\bar{x},\bar{z}) .$$

In such cases we shall neglect to mention the input predicate.

Let us first illustrate the flavor of this idea with three examples:

(i) The construction of an iterative program to compute the quotient and the remainder of two natural numbers;

(ii) The translation of a LISP recursive program for reversing the top-level elements of a list into an equivalent LISP iterative program;

(iii) The construction of a recursive program for finding the maximum among the terminal nodes in a binary tree with integer terminals.

In each case we give the specifications for the program, the induced theorem, and the automatically synthesized program, without introducing the proofs of the theorems or the extraction of the programs from the proofs. Such details will be given in the examples of our later sections.

In our examples we express our input and output predicates in a modified predicate calculus language. However, this is not essential to the method; any language for describing relations may be used.

Example 1: Construction of an iterative division program.

We wish to construct an iterative program to compute the integer quotient and the remainder of two natural numbers x_1 and x_2, where $x_2 \neq 0$. The program should set the output variable z_1 to be the quotient of x_1 divided by x_2, and the output variable z_2 to be the corresponding remainder.

Thus, $\bar{x} = x_1, x_2$ and $z = z_1, z_2$. Since we are not interested in the program's performance for $x_2 = 0$, our input predicate is

$$\varphi(\bar{x}) : x_2 \neq 0 .$$

The output predicate is

$$\psi(\bar{x}, \bar{z}) \equiv (x_1 = z_1 \cdot x_2 + z_2) \cdot (z_2 < x_2) .$$

The theorem induced is then

$$(\forall x_1)(\forall x_2)\{x_2 \neq 0 \supset (\exists z_1)(\exists z_2)[(x_1 = z_1 \cdot x_2 + z_2) \wedge (z_2 < x_2)]\} .$$

The program synthesizer proves the theorem, and a program such as that

illustrated in Figure 1 is extracted from the proof.*/

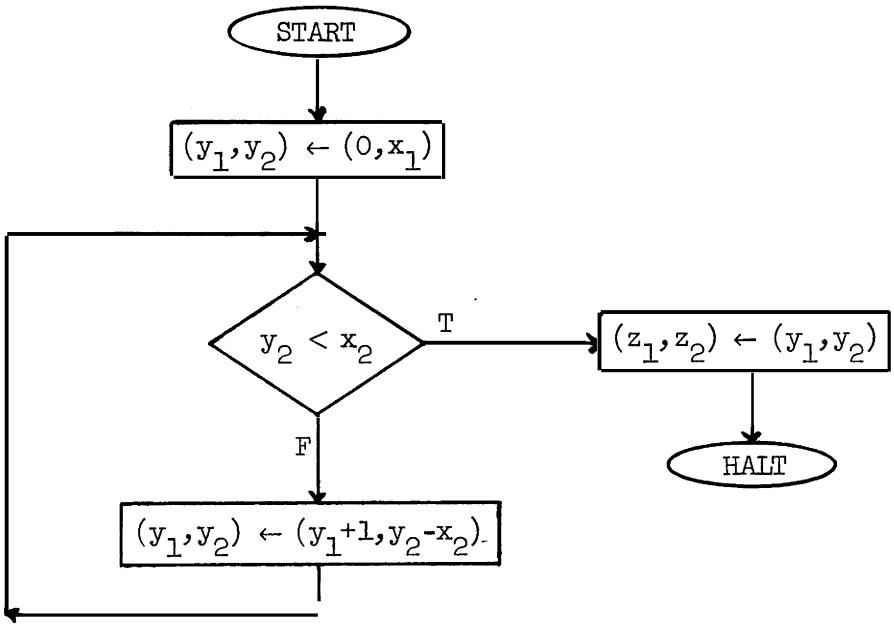


Figure 1: A division program

We have assumed that certain symbols, including the "minus" operator and the "less than" predicate, for instance, exist in our programming language; therefore, these operators are said to be primitive. However, if the use of the "minus" operator or the "less than" predicate is not permitted in the language (i.e., if they are non-primitive), the above program is illegal.

This suggests that the user must always specify a list of primitive operators, predicates, and constants that the derived program may use. If, for example, we allow our-system to use the constant "0", the "successor" and "predecessor" operators, and the "equality" predicate, but not the "minus" operator or the "less than" operator, the program illustrated in Figure 2 might be constructed.

Henceforth, we shall assume all commonly used symbols are primitive unless we make explicit mention to the contrary.

---

*/ Statements in which n-tuples of terms are assigned to n-tuples of variables represent simultaneous replacements. For example, $(y_1,y_2) \leftarrow (y_1+1,y_2-x_2)$ means that y_1 is replaced by y_1+1 and y_2 by y_2-x_2 simultaneously.

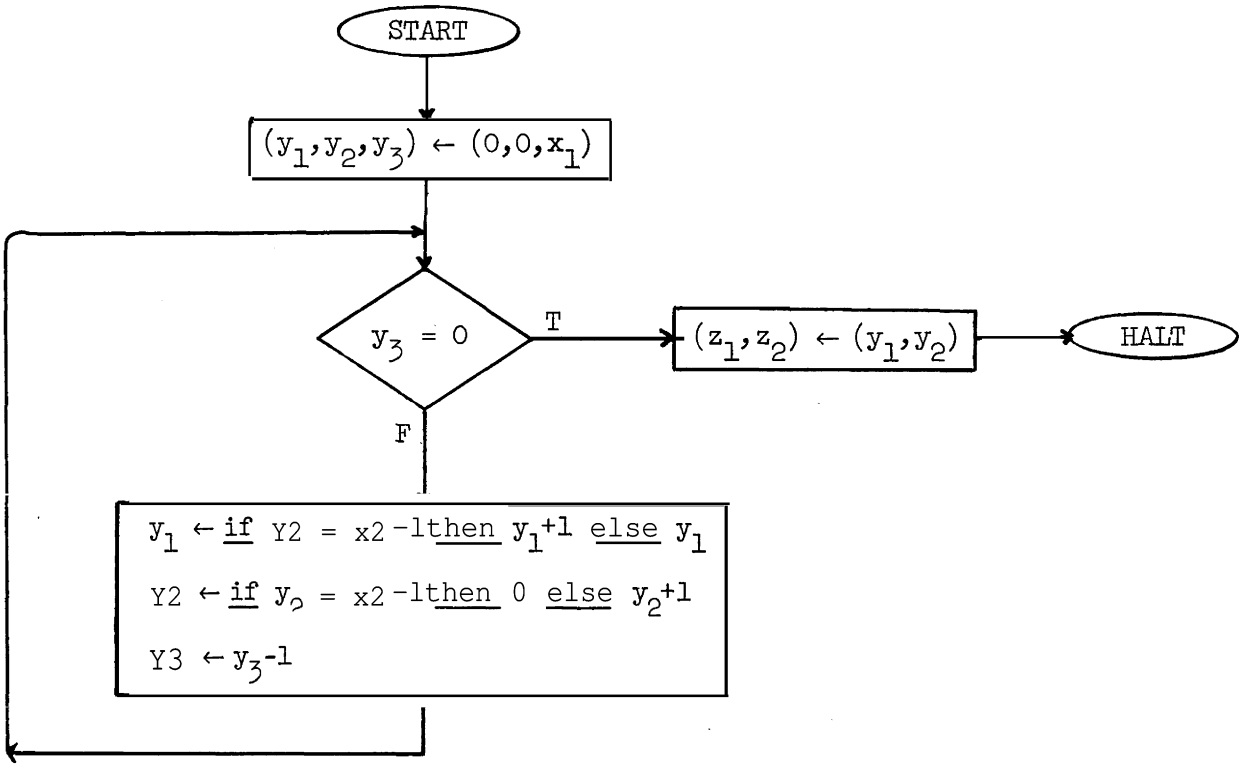


Figure 2: Another division program

Example 2: Translation of a recursive reverse program into an iterative program.

We wish to translate a LISP recursive program for reversing the top-level elements of a list into an equivalent LISP iterative program. For example, if x is the list (a b (c d) e) , then its reverse is (e (c d) b a) .

Here, $\bar{x} = x$, $\bar{z} = z$ and since we want the program to work on all lists, $\varphi(\bar{x})$ is T . The output predicate will be

$$\psi(\bar{x}, \bar{z}) : z = \underline{\text{reverse}}(x) ,$$

where reverse is defined by the recursive program (see McCarthy [1962]):

reverse(y) <= if Null(y) then NIL

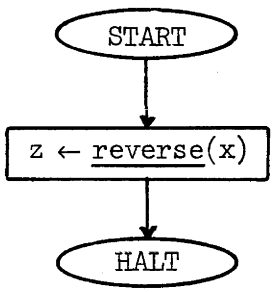
else append(reverse(cdr(y)),list(car(y))) .

The function append(y_1,y_2) concatenates the two lists y_1 and y_2. For example, if y_1 is the list (a b (c d)) and y_2 is the list (e), then append(y_1,y_2) is the list (a b (c d) e).

Thus, the theorem to be proved is

(vx) (3z) [z = reverse(x)] .

The above theorem has a trivial proof, taking z to be reverse(x) itself. Therefore our program synthesizer might construct the following unsatisfactory program



This introduces the problem of primitivity again. The reverse function should not be considered as a primitive in the programming language in this specific task because we clearly do not want reverse to occur in our iterative program. Henceforth, we shall assume that the name of the program to be constructed is never primitive.

If we allow our system to use the constant NIL, the operators car, cdr, and cons, and the predicate Null as primitives, the program illustrated in Figure 3 might be constructed.

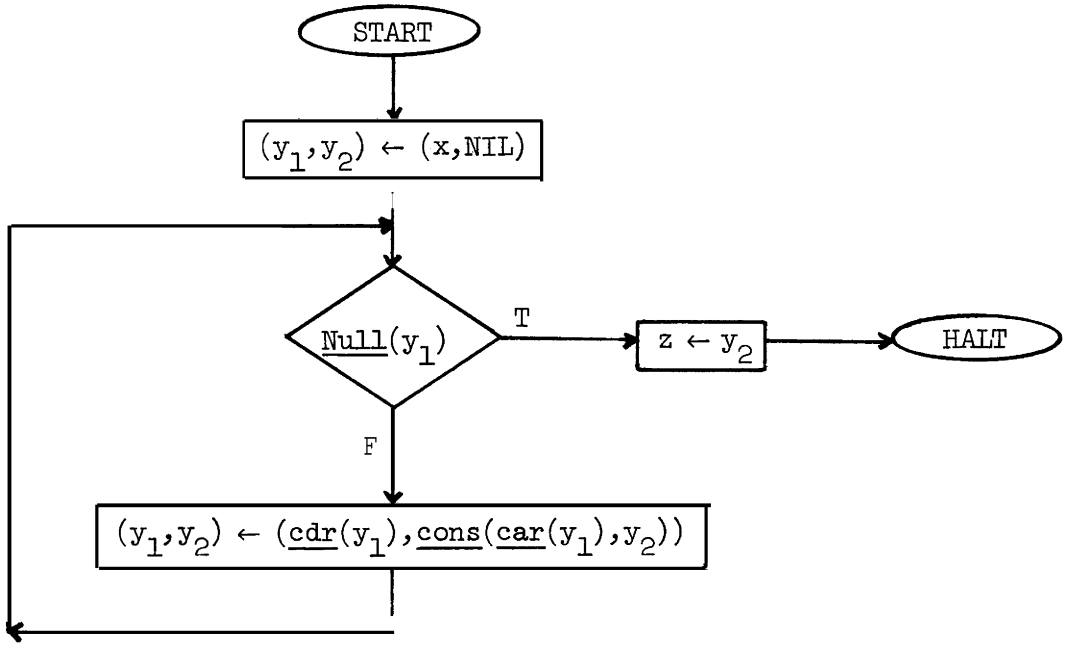


Figure 3: The reverse program

Note that the computation of the derived iterative program consumes less time and space than the computation of the given recursive program. This is not only because of the stacking mechanism necessary in general to implement recursive calls, but also because the repeated use of the append function during execution of the recursive program introduces redundancy in the computation.

Example 3: Construction of a recursive maxtree program.

We wish to construct a recursive program for finding the maximum among the terminal nodes in a binary tree with integer terminals. We shall introduce a special language called TREE for manipulating binary trees.

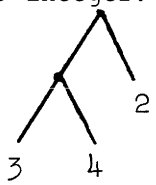
The primitives allowed in our TREE language are the operators

(a) left(y) : the left subtree of the tree y ,

(b) right(y) : the right subtree of the tree y ,

and the predicate

(c) Atom(y) : is the tree y a single integer?

For example, if y is the binary tree 

, then

left(y) is

, right(y) is 2 , Atom(y) is F , and

Atom(right(y)) is T .

Let $\bar{x} = x$, $\bar{z} = z$, $\varphi(\bar{x})$ be T , and the output predicate be

$$\psi(\bar{x},\bar{z}): \text{T erminal}(z,x) \wedge (\forall u)[\text{Terminal}(u,x) \supset u \leq z] ,$$

where $\text{Terminal}(y_1,y_2)$ means that the integer y_1 occurs as a terminal in the tree y_2 . The output predicate says that the integer z is a terminal of the tree x not less than any other terminal node of x .

Thus, the theorem to be proved is

$$(\forall x)(\exists z)\{\text{Terminal}(z,x) \text{ A } (\forall u)[\text{Terminal}(u,x) \supset u \leq z]\} .$$

If we allow the max operator over the integers,

$\max(y_1,y_2)$: the maximum of the integers y_1 and. Y2 ,

to be used as a primitive, the recursive program produced might be

```
z = maxtree(x) where
maxtree(y) <= if Atom(y) then y
                         else max(maxtree(left(y)),maxtree(right(y))) .
```

If we do not allow the max operator to be used as a primitive but allow the predicate "less than or equal to", the program produced might be

```
z = maxtree(x) where
maxtree(y) <= if Atom(y) then y
                         else if maxtree(left(y)) ≤maxtree(right(y))
                                  then maxtree(right(y))
                                  else maxtree(left(y)) .
```

Note that although the symbol maxtree, the name of the program, is not primitive, it may be used as a dummy function name in the recursive definitions. Any other function name could have been used instead.

We feel that at this point we should clarify the role of the input predicate. Compare the following program writing tasks: in the first, the input predicate is $\varphi(\bar{x})$ and the output predicate is $\psi(\bar{x},\bar{z})$, while in the second, the input predicate is $\varphi'(\bar{x})$: T and the output predicate is $\psi'(\bar{x},\bar{z}) : \varphi(\bar{x}) \supset \psi(\bar{x},\bar{z})$. In the first task we do not care how the synthesized program behaves if the input $\bar{x}$ does not satisfy $\varphi(\bar{x})$. In the second case we insist that the program terminates even if $\bar{x}$ does not satisfy $\varphi(\bar{x})$, but we still do not care what the value of the output is.

The theorems induced are:

$$(\forall\bar{x})[\varphi(\bar{x}) \supset (\exists\bar{z})\psi(\bar{x},\bar{z})]$$

and

$$(\forall\bar{x})(\exists\bar{z})[\varphi(\bar{x}) \supset \psi(\bar{x},\bar{z})] ,$$

respectively. Surprisingly enough these theorems are logically equivalent, even though they represent distinct tasks. This suggests that the program extractor must make use of the input predicate in the process of synthesizing the program.

Suppose, for instance, that in constructing our iterative division program (cf. Example 1) we had given the system the input predicate

$$\psi'(\bar{x}) : T$$

and the output predicate

$$\psi'(\bar{x},\bar{z}) : x_2 \neq 0 \supset (x_1 = z_1 \cdot x_2 + z_2) \wedge (z_2 < x_2) .$$

The theorem induced in this case would be

$$(\forall x_1)(\forall x_2)(\exists z_1)(\exists z_2)[x_2 \neq 0 \supset (x_1 = z_1 \cdot x_2 + z_2) \wedge (z_2 < x_2)] ,$$

which is logically equivalent to the theorem

$$(\forall x_1)(\forall x_2)\{x_2 \neq 0 \supset (\exists z_1)(\exists z_2)[(x_1 = z_1 \cdot x_2 + z_2) \wedge (z_2 < x_2)]\} .$$

However, the program extracted from the first theorem (Figure 4) halts for every natural number input, whereas the program extracted from the second theorem (Figure 1) does not halt when x_2 is 0 .

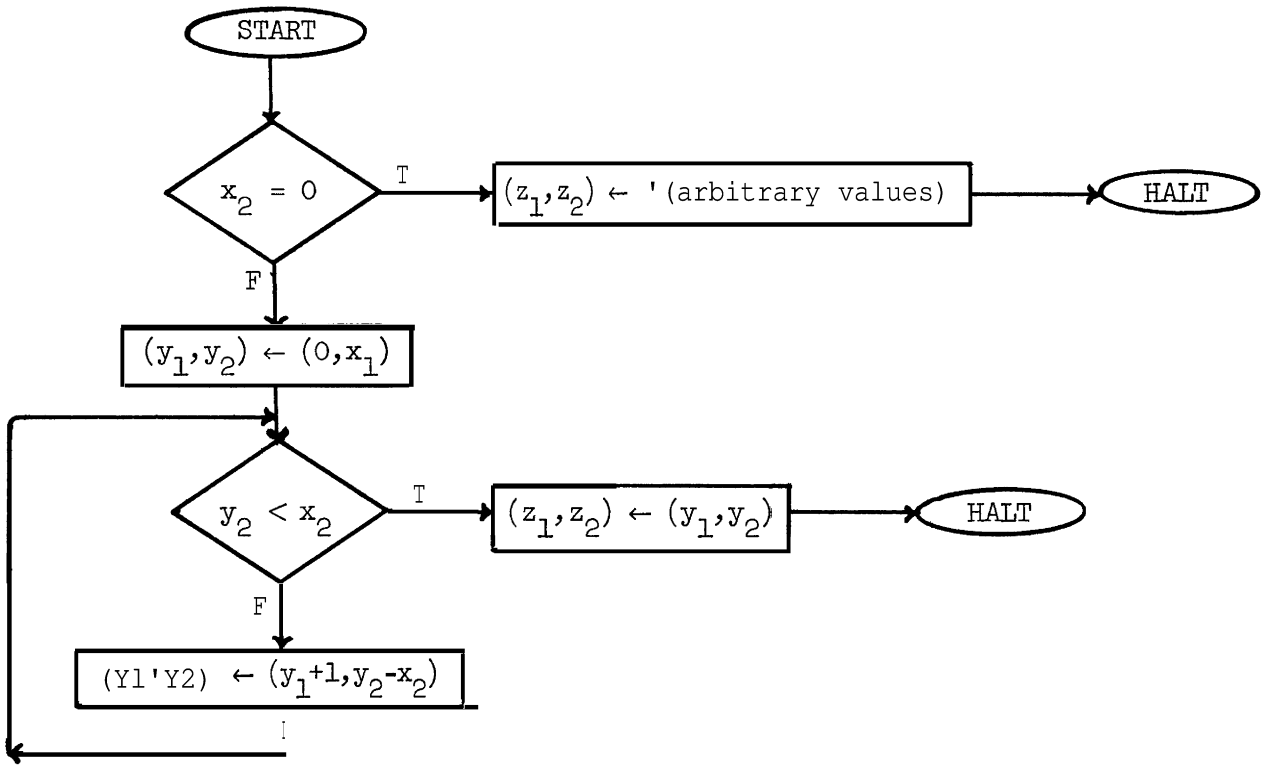


Figure 4: Another division program

3. CONSTRUCTION OF LOOP-FREE PROGRAMS

We would like first to illustrate with two examples the extraction of a program from a proof. The programs we will construct are especially simple since they have no loops. The program extraction process in this case may be roughly described as follows: substitutions into the output variables in the proof result in assignment statements in iterative programs and operator composition in recursive programs. Case analysis arguments in the proof result in conditional branching in both iterative and recursive programs.

Example 4: Reversing a two-element list.

We wish to construct a LISP program that takes as input a list of two elements, and produces as output the same list with the elements reversed.

Thus, the output predicate is

$$\psi(x,z) : (\forall u_1)(\forall u_2)[x = \underline{list}(u_1,u_2) \supset z = \underline{list}(u_2,u_1)],$$

and the theorem to be proved is

$$(\forall x)(\exists z)\{(\forall u_1)(\forall u_2)[x = \underline{list}(u_1,u_2) \supset z = \underline{list}(u_2,u_1)]\} .$$

We assume that any system used to prove this theorem has a large supply of facts about the data structure and the programatmning language to be used, stored in the form of axioms and rules of inference. We assume in particular that the rules of inference stored within the system can handle deductions of the first-order predicate calculus with equality such as those we use in the proof below.

During the process of proving the above theorem, the system will eventually choose the following axioms:

1. $\underline{\text{car}}(\underline{\text{list}}(u,v)) = u$,
2. $\underline{\text{cdr}}(\underline{\text{list}}(u,v)) = \underline{\text{list}}(v)$, and
3. $\underline{\text{append}}(\underline{\text{list}}(u),\underline{\text{list}}(v)) = \underline{\text{list}}(u,v)$.

Note that the operator list takes a variable number of arguments.

The proof will proceed as follows: Suppose $x = \underline{\text{list}}(u_1,u_2)$ for some arbitrary u_1 and u_2, Then by Axioms 1 and 2, respectively,

4. $\underline{\text{car}}(x) = u_1$, and
5. $\underline{\text{cdr}}(x) = \underline{\text{list}}(u_2)$.

From 4 we have

6. $\underline{\text{list}}(\underline{\text{car}}(x)) = \underline{\text{list}}(u_1)$.

Then combining 5 and 6 using Axiom 3, we obtain

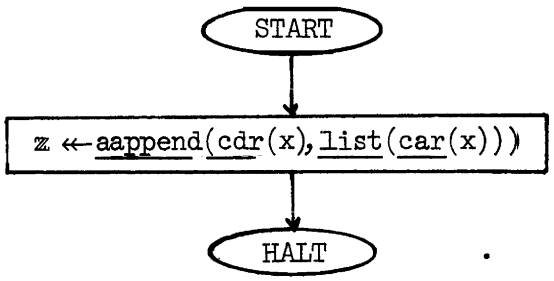
7. $\underline{\text{append}}(\underline{\text{cdr}}(x),\underline{\text{list}}(\underline{\text{car}}(x))) = \underline{\text{list}}(u_2,u_1)$.

Letting z be $\underline{\text{append}}(\underline{\text{cdr}}(x),\underline{\text{list}}(\underline{\text{car}}(x)))$, we obtain

8. $z = \underline{\text{list}}(u_2,u_1)$,

which is the desired conclusion.

Now, in order to extract the program, we keep track of the substitutions made for z during the proof. In the above proof we have replaced z by $\underline{\text{append}}(\underline{\text{cdr}}(x),\underline{\text{list}}(\underline{\text{car}}(x)))$; therefore, the desired program is simply



.

Example 5: The max of two numbers.

The program constructed in this example contains a conditional branch but no loops. We wish to find the maximum of two given integers. Thus, the output predicate is

$$\psi(x_1,x_2,z) : (z = x_1 \vee z = x_2) \wedge z \geq x_1 \wedge z \geq x_2 ,$$

and the corresponding theorem is

$$(\forall x_1)(\forall x_2)(\exists z)[(z = x_1 \vee z = x_2) \wedge z \geq x_1 \wedge z \geq x_2] .$$

The proof proceeds by case analysis; it may appear poorly motivated, but it is well within the capacity of existing theorem-proving programs. Translating the theorem into disjunctive normal form, we have

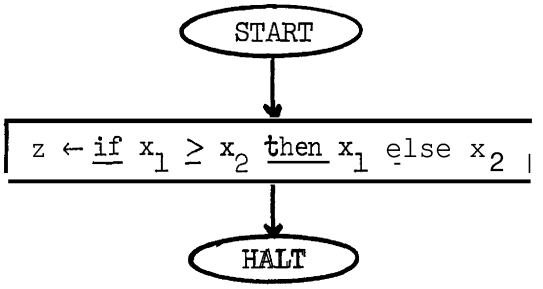
$$(\forall x_1)(\forall x_2)(\exists z)[(z = x_1 \wedge z \geq x_1 \wedge z \geq x_2) \vee (z = x_2 \wedge z \geq x_1 \wedge z \geq x_2)] .$$

If we assume $(u = v) \supset (u \geq v)$ as an axiom, we can simplify the above formula to

$$(\forall x_1)(\forall x_2)(\exists z)[(z = x_1 \wedge z \geq x_2) \vee (z = x_2 \wedge z \geq x_1)] .$$

Now suppose $x_1 \geq x_2$, then if we let z be x_1, the first disjunct is satisfied. On the other hand, suppose $x_1 < x_2$; then if we let z be x_2, the second disjunct is satisfied.

Since the substitution we made for z depends on whether or not x_1 was greater than or equal to x_2, the program extracted from the proof of the theorem is



The reader who is unsatisfied with our seat-of-the-pants description of the program extraction process may examine any of the more rigorous accounts in the literature (e.g. Green [1969a] [1969b], Waldinger and Lee [1969], Waldinger [1969], and Luckham and Nilsson [1970]).

The above programs are clearly of limited interest since neither contains a loop. In order to construct a program with loops, application of some version of the principle of mathematical induction is necessary. Therefore, in the next section we digress into a discussion of the induction principle.

4. THE INDUCTION PRINCIPLE

The induction principle is most commonly associated with proving theorems about the natural numbers, but analogues of it apply to other data structures, such as lists, trees, and strings. Furthermore, for each data structure there are many equivalent forms of the principle. Mathematicians use whichever version is most convenient. Similarly, the theorem prover chooses an appropriate induction principle from a given supply during the course of the proof. This choice directly determines the form of the program to be constructed, since each induction rule has an associated program form stored with it. Therefore, if we want to restrict the form of the extracted program, we must limit the set of available induction principles accordingly.

4.1 Natural numbers

We shall discuss four versions of the induction principle for the -natural numbers; two will be appropriate for writing recursive programs and two for writing iterative programs. In each class, one rule will be called a "going-up" principle and the other a "going-down" principle. We will illustrate each of these with a different version of the factorial program. The output predicate is

$$\psi(x,z) : z = \underline{\text{factorial}}(x) ,$$

where

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

This example will illustrate clearly the difference between the programs generated by using "going-up" induction and "going-down" induction: the

"going-up" programs compute x! in the order 1 , 1·2 , 1·2·3 , ... ,

while the "going-down'* programs compute x , x·(x-1) , x·(x-1)·(x-2) ,... .

The proofs required for the synthesis of the programs use two axioms induced by the above definition:

$$\underline{\text{factorial}}(0) = 1$$

and

$$u > 0 \supset [\underline{\text{factorial}}(u) = u \cdot \underline{\text{factorial}}(u-1)] .$$

We will not include those proofs, but merely will give the programs extracted, in order to illustrate the relationship between the form of the induction principle used in the proof and the form of the constructed program.

(a) Iterative going-up induction

The reader is probably familiar with the most common version of the induction principle over the natural numbers,

$$\text{do)}\quad \frac{(\forall y_1)[\mathcal{A}(y_1) \supset \mathcal{A}(y_1+1)\ 1}{(\forall x)\mathcal{A}(x)} \quad .$$

Intuitively, this means that if a property $\mathcal{A}$ holds for 0 and if whenever it holds for y_1 it holds for y_1+1 , then it holds for every natural number x . We call this version iterative going-up induction.

For our purpose we use a special form of the principle in which $\mathcal{A}(y_1)$ is $(\exists y_2)\mathcal{B}(y_1,y_2)$, where $\mathcal{B}$ still represents an unspecified property. The induction principle now becomes

$$(\exists y_2)\mathcal{B}(0,y_2)$$

$$\frac{(\forall y_1)[(\exists y_2)\mathcal{B}(y_1,y_2) \supset (\exists y_2)\mathcal{B}(y_1+1,y_2)\,1}{(\forall x)(\exists y_2)\mathcal{B}(x,y_2)} \quad .$$

The program form associated with this rule is illustrated in Figure 5. If the theorem to be proved happens to be of the form

$$(\forall x)(\exists z)\mathcal{B}(x,z) \ ,$$

and if going-up induction is applied, the program extractor then knows that the program must be of the form illustrated in Figure 5.

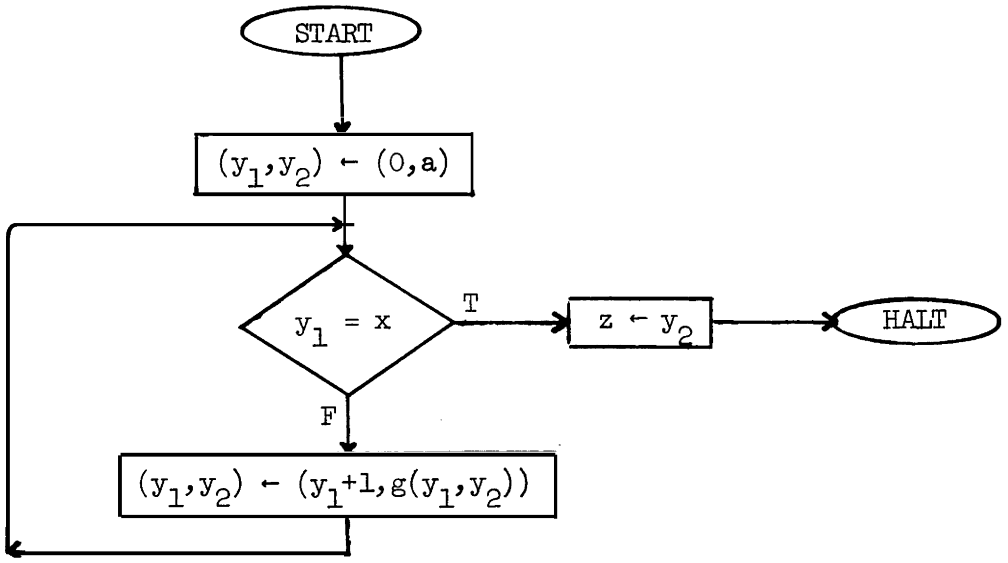


Figure 5: Iterative "going-up" program form

The constant a and the function $g(y_1,y_2)$ are unspecified in the above form. The task of the program constructor is now to write subroutines to compute a and g in such a way that the program of Figure 5 will satisfy the desired relation. This is done as follows.

The theorem to be proved is of the form $(\forall x)(\exists z)\mathcal{B}(x,z)$. This is precisely the form of the consequent of the induction principle. Therefore, if we can prove the two antecedents, then we are done. This suggests-that we attempt to prove the two lemmas:

(A) $\quad (\exists y_2)\mathcal{B}(0,y_2)$,

and

(B) $\quad (\forall y_1)[(\exists y_2)\mathcal{B}(y_1,y_2) \supset (\exists y_2)\mathcal{B}(y_1+1,y_2)\ 1$,

or equivalently, translating into prenex normal form,

(B') $\quad (\forall y_1)(\forall y_2)(\exists y_2^*)[\mathcal{B}(y_1,y_2) \supset \mathcal{B}(y_1+1,y_2^*)]$.

The proof of Lemma A generates a subroutine with no variables that yields a value for y_2 satisfying $\mathcal{B}(0,y_2)$. This is the desired definition of the constant a ; hence

(1) $\quad \mathcal{B}(0,a)$ is true.

The proof of Lemma B' generates another subroutine which yields a value of y_2^* in terms of y_1 and y_2 . This subroutine provides a definition of $g(y_1,y_2)$ satisfying

(2) $\quad \mathcal{B}(y_1,y_2) \supset \mathcal{B}(y_1+1,g(y_1,y_2))$ for all y_1 and y_2 .

The proof of the lemmas concludes the proof of the theorem $(\forall x)(\exists z)\mathcal{B}(x,z)$. We have now completely specified a program that computes a function $z = f(x)$ satisfying $\mathcal{B}(x,f(x))$ for all values of x .

For the suspicious reader we are ready to verify the above assertion. Consider the iterative "going-up" program form labeled as in Figure 6.

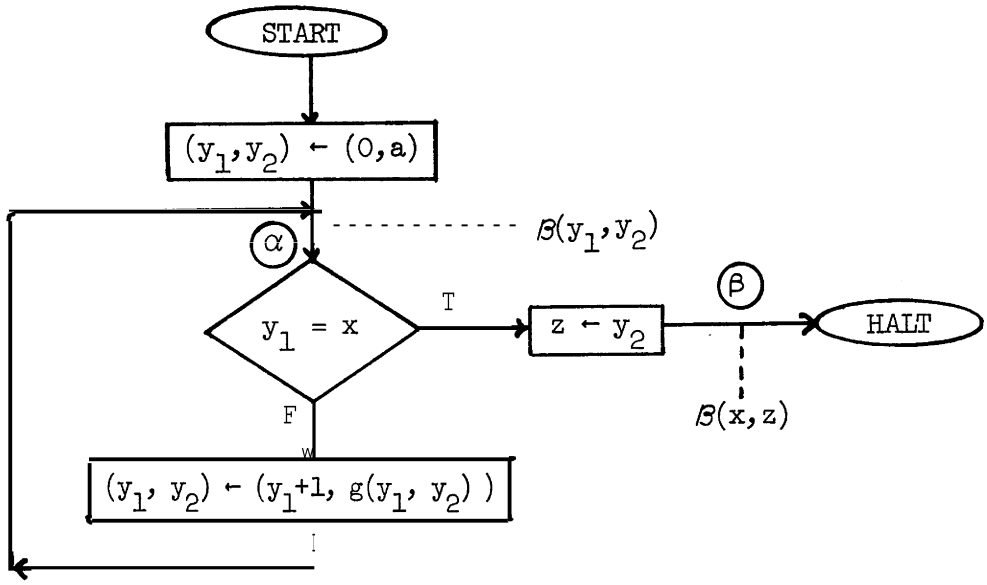


Figure 6: Labeled iterative 'going-up' program form

We will use Floyd's approach [1967] and show that whenever control passes through arc α , $\mathcal{B}(y_1, y_2)$ is true for the current values of y_1 and y_2 . Furthermore, whenever control passes through arc β , $\mathcal{B}(x, z)$ is true for the initial value of x and the final value of z .

Beginning at the START node, we set y_1 to 0 and y_2 to a , and so when we pass through arc α , $\mathcal{B}(y_1, y_2)$ (i.e., $\mathcal{B}(0, a)$) is true by (1).

Now suppose that at some point in the execution, control is passing through arc α and currently $\mathcal{B}(y_1, y_2)$ is true and $y_1 \neq x$. Then, by (2)

(3) $\mathcal{B}(y_1+1, g(y_1, y_2))$ is true.

Traveling around the loop we simultaneously set y_1 to y_1+1 and y_2 to $g(y_1, y_2)$ and reach arc α again. That $\mathcal{B}(y_1, y_2)$ is satisfied at this time follows directly from (3) and our assignments to y_1 and y_2 .

Clearly, we must at some time reach arc α with $y_1 = x$, since x is a natural number. Then we set z to y_2 and pass to arc β . Since $\mathcal{B}(y_1, y_2)$ was true at arc α and $y_1 = x$, $\mathcal{B}(x, z)$ is true at arc β . This concludes the proof that the program constructed has the desired properties.

Example 6: Iterative "going-up" factorial program.

We wish to construct an iterative "going-up" program for computing the factorial function. The theorem to be proved is

$$(\forall x)(\exists z)[z = \underline{\text{factorial}}(x)] \ .$$

Applying the iterative going-up induction principle (with $\mathcal{B}(y_1,y_2)$ being $y_2 = \underline{\text{factorial}}(y_1)$), we are presented with the two lemmas

(A) $$(\exists y_2)[y_2 = \underline{\text{factorial}}(0)]$$

and

(B') $$(\forall y_1)(\forall y_2)(\exists y_2^*)\{[y_2 = \underline{\text{factorial}}(y_1)] \supset [y_2^* = \underline{\text{factorial}}(y_1+1)]\} \ .$$

The lemmas are proven, and the values for y_2 and y_2^* (i.e., a and $g(y_1,y_2)$) found are 1 and $(y_1+1)\cdot y_2$, respectively. The program extracted is illustrated in Figure 7.

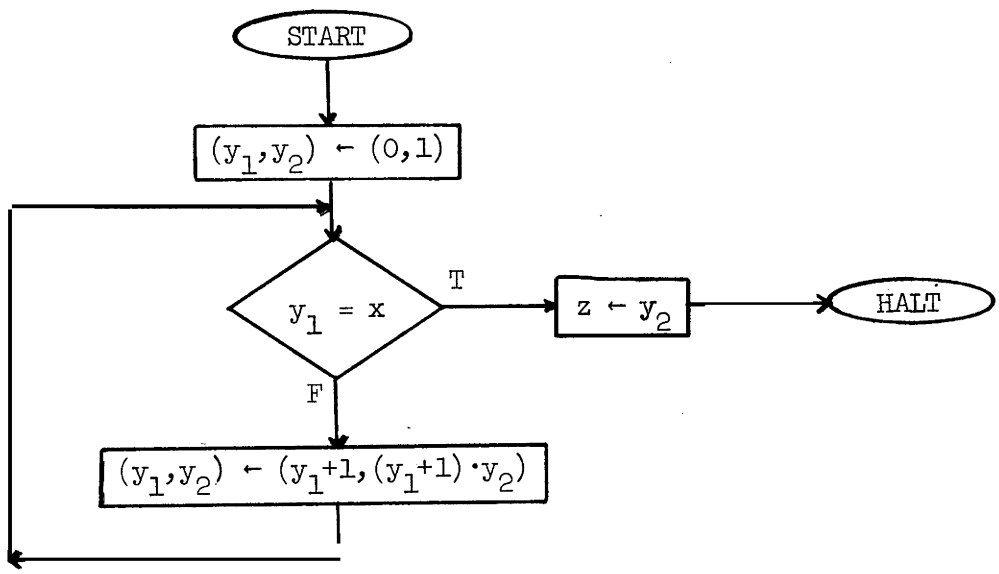


Figure 7: Iterative "going-up" factorial program

Note that for simplicity we have assumed in the above discussion that the program to be constructed has only one input variable x and one output variable z . This restriction may be waived by a straightforward generalization of the induction principle given above as illustrated in Examples 13 and 15.

(b) Recursive going-up induction

We present another going-up induction principle that leads to a different program form. The principle

$$\frac{\alpha(0) \qquad (\forall y_1)\,[y_1 \neq 0 \wedge \alpha(y_1\text{-}1) \supset \alpha(y_1)\,]}{(\forall x)\alpha(x)}$$

is logically equivalent to the first version but leads to the construction of a recursive program of the form

$$\begin{cases} z = f(x) \text{ where} \\ f(y) \Leftarrow \text{if } y = 0 \text{ then a else} g(y, f(y\text{-}1)) \ . \end{cases}$$

We call this version recursive going-up induction. Note that the f is a dummy function symbol that need not be declared primitive.

We have omitted the details concerning the derivation of this program but they are quite similar to those involved in Section (a) above.

(c) Iterative going-down induction

Another form of the induction principle is the iterative going-down form

$$\frac{(\exists y_1)\alpha(y_1) \qquad (\forall y_1)[y_1 \neq 0 \wedge \alpha(y_1) \supset \alpha(y_1\text{-}1)\,]}{\alpha(0)} \ .$$

The reader may verify that this rule is equivalent to the recursive going-up induction, replacing α by $\sim\alpha$ 'and twice using the fact that $p \wedge q \supset r$ is logically equivalent to $\sim r \wedge q \supset -p$.

In this case we use a special form of the principle in which $\alpha(y_1)$ is of the form $(\exists y_2)\beta(x, y_1, y_2)$, where x is a free variable.

The induction principle now becomes

$$\frac{(\exists y_1)(\exists y_2)\mathcal{B}(x,y_1,y_2) \qquad (\forall y_1)[y_1 \neq 0 \wedge (\exists y_2)\mathcal{B}(x,y_1,y_2) \supset (\exists y_2)\mathcal{B}(x,y_1\text{-}1,\, y_2)\,1}{(\exists y_2)\mathcal{B}(x,0,y_2)} \quad .$$

Suppose now that the theorem to be proved is of the form

$$(\forall x)(\exists z)\mathcal{B}(x,0,z) \quad .$$

The theorem may be deduced from the conclusion of the above induction principle. If the iterative going-down induction principle is used, the program to be extracted is automatically of the form illustrated in Figure 8.

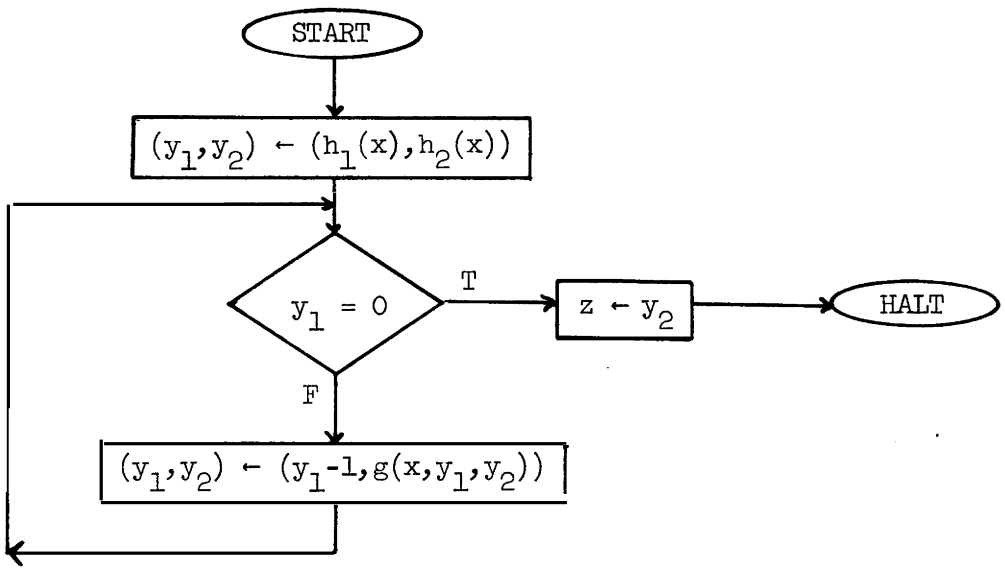


Figure 8: Iterative "going-down" program form

Thus, all that remains is to construct subroutines to compute the functions $h_1(x)$, $h_2(x)$, and $g(x,y_1,y_2)$.

The antecedents of the induction principle give us the two lemmas to be proved

(A) $(\exists y_1)\,(\exists y_2)\mathcal{B}(x,y_1,y_2)$

and

(B) $\quad (\forall y_1)[y_1 \neq 0 \wedge (\exists y_2)\mathcal{B}(x,y_1,y_2) \supset (\exists y_2)\mathcal{B}(x,y_1-1,y_2)]$,

or equivalently,

(B') $\quad (\forall y_1)(\forall y_2)(\exists y_2^*)[y_1 \neq 0 \wedge \mathcal{B}(x,y_1,y_2) \supset \mathcal{B}(x,y_1-1,y_2^*)]$.

The proof of Lemma A yields subroutines to canpute y_1 and y_2 in terms of x, which define the desired functions $h_1(x)$ and $h_2(x)$, respectively. The proof of Lemma B' yields a single subroutine to compute y_2^* in terms of x, y_1, and y_2 , thus defining the desired function $g(x, y_1, y_2)$. The program is then completely specified, and its correctness and termination can be demonstrated using Floyd's approach, as was done before.

Note that iterative going-down induction is of value only if the constant 0 occurs in the theorem to be proved. Otherwise, the theorem prover must manipulate the theorem to introduce 0 .

Example 7: Iterative "going-down" factorial program.

We wish to construct an iterative "going-down" program for computing the factorial function. The theorem to be proved is again

$$(\forall x)(\exists z)[z = \underline{\text{factorial}}(x)] \ .$$

The theorem contains no occurrence of the constant 0 . Thus, the theorem prover tries to introduce- 0 , using the first part of the definition of the factorial function (i.e., $\underline{\text{factorial}}(0) = 1$) and its supply of axioms ($u \cdot 1 = u$, in particular), deriving as a subgoal

$$(\exists z)[\underline{\text{factorial}}(0) \cdot z = \underline{\text{factorial}}(x)] \ ,$$

where x . is a free variable. This theorem is in the form of the consequent of the iterative going-down induction, i.e., $(\exists y_2)\mathcal{B}(x,0,y_2)$; hence, the theorem prover chooses the induction hypothesis $\mathcal{B}(x,y_1,y_2)$ to be $\underline{\text{factorial}}(y_1) \cdot y_2 = \underline{\text{factorial}}(x)$.

The lemmas proposed are

(A) $(\exists y_1)(\exists y_2)[\underline{\text{factorial}}(y_1)\cdot y_2 = \underline{\text{factorial}}(x)]$,

and

(B') $(\forall y_1)(\forall y_2)(\exists y_2^*)\{y_1 \neq 0 \text{ A } [\underline{\text{factorial}}(y_1)\cdot y_2 = \underline{\text{factorial}}(x)]$
$\supset [\underline{\text{factorial}}(y_1\text{-}1)\cdot y_2^* = \underline{\text{factorial}}(x)]\}$.

The values obtained for y_1 , y_2 , and y_2^* (i.e., $h_1(x)$, $h_2(x)$, and $g(x,y_1,y_2)$), respectively, are x , 1 , and $y_1\cdot y_2$. The program constructed is illustrated in Figure 9.

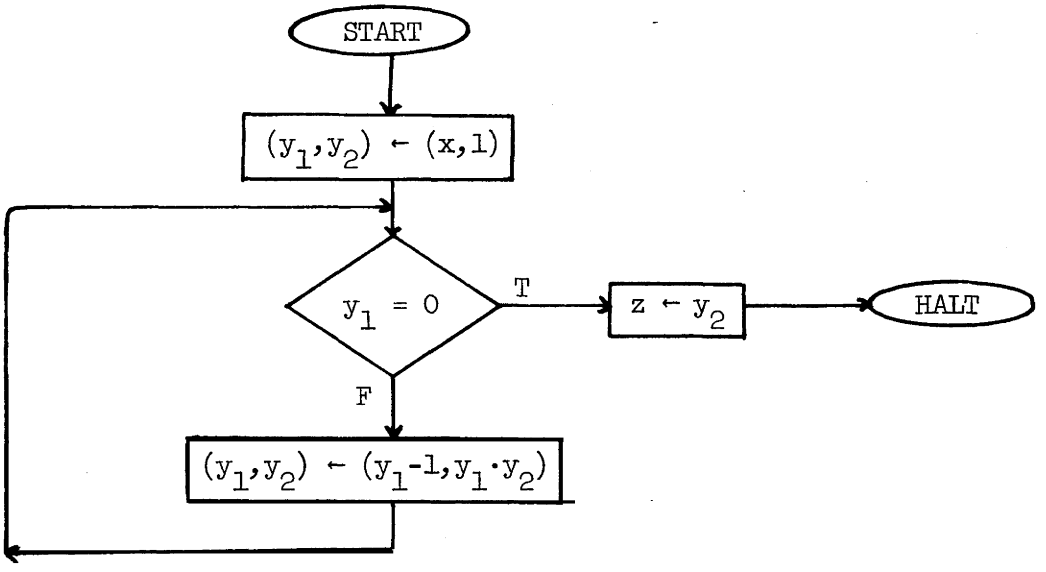


Figure 9: Iterative "going-down" factorial program

(d) Recursive -going-down induction

The recursive going-up induction was very similar to the iterative going-up induction. In the same way, the recursive going-down induction,

$$\frac{(\exists y_1)\mathcal{a}(y_1) \qquad (\forall y_1)\,[\mathcal{a}(y_1+1) \supset \mathcal{a}(y_1)\,]}{\mathcal{a}(0)} \ ,$$

is very similar to the iterative going-down induction. The form we are most interested in is

$$\frac{(\exists y_1)(\exists y_2)\mathcal{B}(x,y_1,y_2) \qquad (\forall y_1)[(\exists y_2)\mathcal{B}(x,y_1+1,y_2) \supset (\exists y_2)\mathcal{B}(x,y_1,y_2)]}{(\exists y_2)\mathcal{B}(x,0,y_2)} ,$$

where x is a free variable.

If the rule is used in generating a program, the two appropriate lemmas allow us to construct $h_1(x)$, $h_2(x)$ and $g(x,y_1,y_2)$ as before, and the program extracted is

$$\begin{cases} z = f(x) = f'(x,0) \text{ where} \\ f'(x,y) \Leftarrow \underline{\text{if}}\ y = h_1(x)\ \underline{\text{then}}\ h_2(x)\ \underline{\text{else}}\ g(x,y,f'(x,y+1)) \end{cases} .$$

Example 8: Recursive "going-down" factorial program.

The program we wish to construct this time is a recursive "going-down" program to compute the factorial function. Again the theorem

$$(\forall x)(\exists z)[z = \underline{\text{factorial}}(x)]$$

is transformed into

$$(\exists z)[\underline{\text{factorial}}(0)\cdot z = \underline{\text{factorial}}(x)] .$$

We continue as before and the program generated is

$$\begin{cases} z = f(x) = f'(x,0) \text{ where} \\ f'(x,y) \Leftarrow \underline{\text{if}}\ y = x\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ (y+1)\cdot f'(x,y+1) \end{cases} .$$

4.2 Lists

Our treatments of lists and natural numbers are in some ways analogous, since the constant NIL and the function cdr in LISP play the same role as the constant 0 and the "predecessor" function, respectively, in number theory. The induction principles of both data structures are closely related, but since there is no exact analogue in LISP to the "successor" function in number theory, there are no iterative going-up and recursive going-down list induction principles. Hence, we shall only deal with two induction rules in this section: recursive (going-up) and iterative (going-down) list inductions. In the discussion in this section we shall omit details since they are similar to the details in the previous section.

We shall illustrate the use of both induction rules by constructing two programs for sorting a given list of integers. The output predicate is

$$\psi(x,z) : z = \underline{sort}(x) \quad ,$$

where

$$(\forall y)(\forall z)\{[z = \underline{sort}(y)] \equiv \underline{if}\ \mathrm{Null}(y)\ \text{then}\ \mathrm{Null}(z)$$
$$\underline{else}\ (\forall u)[\underline{Member}(u,y) \supset z = \underline{merge}(u,\underline{sort}(\underline{delete}(u,y)))]\}.$$

Here,

$\underline{Member}(u,y)$ means that the integer u is a member of the list y ,

$\underline{delete}(u,y)$ is the list obtained by deleting the integer u from the list y , and

$\underline{merge}(u,v)$, where v is a sorted list that does not contain the integer u , is the list obtained by placing u in its place on the list v , so that the ordering is preserved.

The theorem to be proved is

$$(\forall x)(\exists z)[z = \underline{\text{sort}}(x)] \ .$$

(a) Recursive list induction

The recursive (going-up) list induction principle is

$$\frac{\mathcal{A}(\text{NIL}) \qquad (\forall y_1)[\sim \underline{\text{Null}}(y_1) \wedge \mathcal{A}(\underline{\text{cdr}}(y_1)) \supset \mathcal{A}(y_1)\ 1}{(\forall x)\mathcal{A}(x)} \ .$$

The program-synthesis form of the rule is

$$\frac{(\exists y_2)\mathcal{B}(\text{NIL},y_2) \qquad (\forall y_1)[\sim \underline{\text{Null}}(y_1) \wedge (\exists y_2)\mathcal{B}(\underline{\text{cdr}}(y_1),y_2) \supset (\exists y_2)\mathcal{B}(y_1,y_2)\ 1}{(\forall x)(\exists y_2)\mathcal{B}(x,y_2)} \ .$$

The corresponding program form generated is

$$\begin{cases} z = f(x) \ \underline{\text{where}} \\ f(y) \ \underline{<=}\ \underline{\text{if}}\ \underline{\text{Null}}(y)\ \underline{\text{then}}\ a\ \underline{\text{else}}\ g(y,f(\underline{\text{cdr}}(y))) \end{cases} \ .$$

Example 9: Recursive sort program.

The sort program obtained using the recursive list induction principle is

$$\begin{cases} z = \underline{\text{sort}}(x) \ \underline{\text{where}} \\ \underline{\text{sort}}(y) \ \underline{<=}\ \underline{\text{if}}\ \underline{\text{Null}}(y)\ \underline{\text{then}}\ \text{NIL}\ \underline{\text{else}}\ \underline{\text{merge}}(\underline{\text{car}}(y),\underline{\text{sort}}(\underline{\text{cdr}}(y))) \end{cases} \ .$$

(b) Iterative list induction

The reader can undoubtedly guess that the iterative (going-down) list induction principle is

$$\frac{(\exists y_1)\mathcal{A}(y_1) \qquad (\forall y_1)[\sim \underline{\text{Null}}(y_1) \wedge \mathcal{A}(y_1) \supset \mathcal{A}(\underline{\text{cdr}}(y_1))]}{\mathcal{A}(\text{NIL})} \ .$$

We are especially interested in the form

$$\frac{(\exists y_1)(\exists y_2)\mathcal{B}(x,y_1,y_2) \qquad (\forall y_1)[\sim \underline{\text{Null}}(y_1) \wedge (\exists y_2)\mathcal{B}(x,y_1,y_2) \supset (\exists y_2)\mathcal{B}(x,\underline{\text{cdr}}(y_1),y_2)]}{(\exists y_2)\mathcal{B}(x,\text{NIL},y_2)} \ ,$$

where x is a free variable.

The corresponding program form generated is illustrated in Figure 10; it employs the construction known among LISP programmers as the "cdr loop".

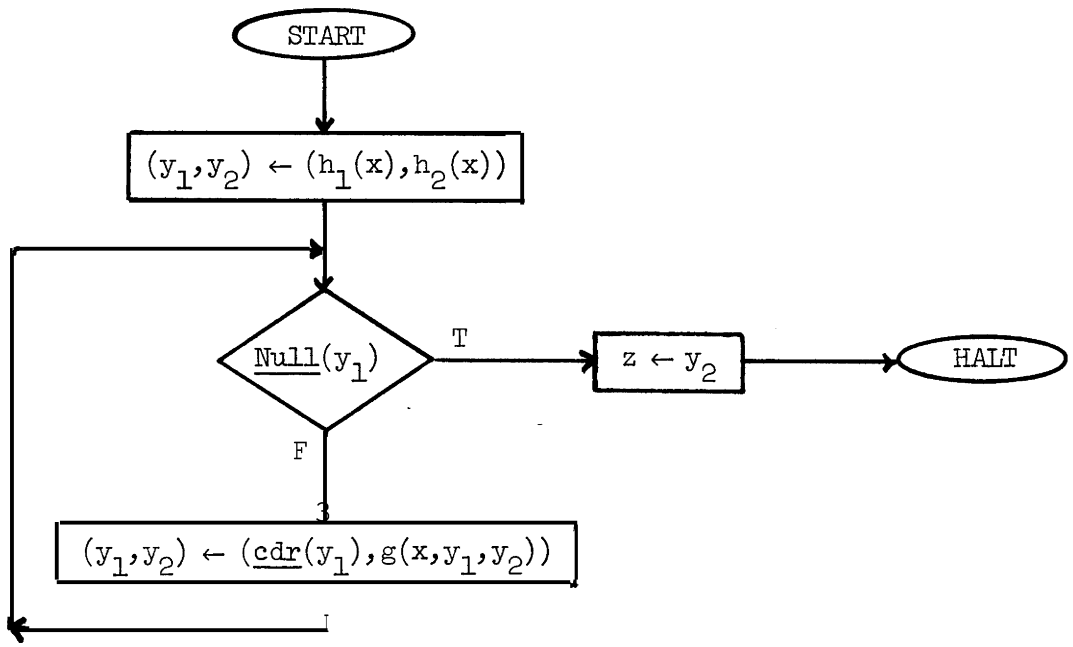


Figure 10: Iterative list program form

Example 10: Iterative sort program.

Using the iterative list induction, we can extract the sort program of Figure 11.

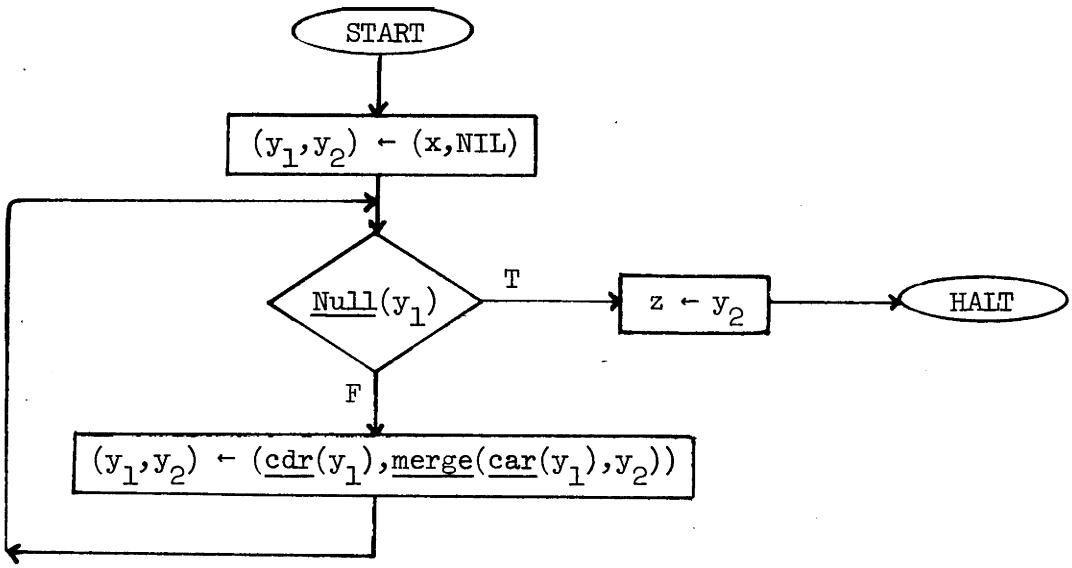


Figure 11: Iterative sort program

4.3 Trees

There is no simple induction rule for tree structures which gives rise to an iterative program form, because such a program would have to use a complex mechanism to keep track of its place in the tree. However, there is a simple recursive tree-induction rule:

$$\frac{(\forall y_1)\,[\underline{Atom}(y_1) \supset \mathcal{A}(y_1)\,] \qquad (\forall y_1)[\sim \underline{Atom}(y_1) \wedge \mathcal{A}(\underline{left}(y_1)) \wedge \mathcal{A}(\underline{right}(y_1)) \supset \mathcal{A}(y_1)]}{(\forall x)\mathcal{A}(x)}$$

In the automatic program synthesizer we are chiefly interested in the following form

$$(\forall y_1)[\underline{\text{Atom}}(y_1) \supset (\exists y_2)\mathcal{B}(y_1,y_2)\,]$$

$$\frac{(\forall y_1)[\sim \underline{\text{Atom}}(y_1) \wedge (\exists y_2)\mathcal{B}(\underline{\text{left}}(y_1),y_2) \wedge (\exists y_2)\mathcal{B}(\underline{\text{right}}(y_1),y_2) \supset (\exists y_2)\mathcal{B}(y_1,y_2)\,]}{(\forall x)(\exists y_2)\mathcal{B}(x,y_2)} \quad .$$

If we want to prove a theorem of form $(\forall x)(\exists z)\mathcal{B}(x,z)$ using tree induction, we must prove two lemmas

(A) $$(\forall y_1)\,[\underline{\text{Atom}}(y_1) \supset (\exists y_2)\mathcal{B}(y_1,y_2)\,] \ ,$$

or equivalently,

(A') $$(\forall y_1)\,(\exists y_2)\,[\underline{\text{Atom}}(y_1) \supset \mathcal{B}(y_1,y_2)\,] \ ,$$

and

(B) $$(\forall y_1)[\sim \underline{\text{Atom}}(y_1) \wedge (\exists y_2)\mathcal{B}(\underline{\text{left}}(y_1),y_2) \wedge (\exists y_2)\mathcal{B}(\underline{\text{right}}(y_1),y_2) \supset (\exists y_2)\mathcal{B}(y_1,y_2)\,] \ ,$$

or equivalently,

(B') $$(\forall y_1)(\forall y_2)(\forall y_2')(\exists y_2^*)[\sim \underline{\text{Atom}}(y_1) \wedge \mathcal{B}(\underline{\text{left}}(y_1),y_2) \wedge \mathcal{B}(\underline{\text{right}}(y_1),y_2') \supset \mathcal{B}(y_1,y_2^*)\,] \ .$$

From the proof of Lemma A' we define a subroutine $h(y_1)$ to compute y_2 in terms of y_1. The proof of Lemma B' yields a subroutine $g(y_1,y_2,y_2')$ to compute y_2^* in terms of y_1, y_2 and y_2'.

The corresponding program form is

$$\begin{cases} z = f(x) \ \underline{\text{where}} \\ f(y_1) \Leftarrow \text{if } \underline{\text{Atom}}(y_1) \ \underline{\text{then}} \ h(y_1) \ \underline{\text{else}} \ g(y_1,f(\underline{\text{left}}(y_1)),f(\underline{\text{right}}(y_1))) \end{cases} .$$

Note that this program form employs two recursive calls.

Example 11: Recursive maxtree program (see Example 3).

We wish to give the synthesis of a TREE recursive program for finding the maximum among the terminal nodes in a binary tree with integer terminals. This will be the first detailed example of the construction of a program containing loops.

The theorem to be proved is

$$(\forall x)(\exists z)[z = \underline{\text{maxtree}}(x)] \ ,$$

where

(1) $[z = \underline{\text{maxtree}}(x)] \equiv [\underline{\text{Terminal}}(z,x) \ A(\forall u)[\underline{\text{Terminal}}(u,x) \supset u \leq z]] \ ,$

and

(2) $\underline{\text{Terminal}}(u,v) \Leftarrow \underline{\text{if}}\ \underline{\text{Atom}}(v)\ \underline{\text{then}}\ u = v$

$\underline{\text{else}}\ \underline{\text{Terminal}}(u,\underline{\text{left}}(v)) \vee \underline{\text{Terminal}}(u,\underline{\text{right}}(v)) \ .$

We assume that **maxtree** itself is not primitive.

The theorem is of the form $(\forall x)(\exists z)\mathcal{B}(x,z)$, where $\mathcal{B}(x,z)$ is $z = \underline{\text{maxtree}}(x)$. Taking z to be y_2 , this is precisely the conclusion of the tree induction principle. Therefore, we apply the induction with

$$\mathcal{B}(y_1,y_2) \ : \ Y2 \ . \ \underline{\text{maxtree}}(y_1) \ .$$

Hence, it suffices to prove the following two lemmas:

(A') $(\forall y_1)(\exists y_2)[\underline{\text{Atom}}(y_1) \supset y_2 = \underline{\text{maxtree}}(y_1)] \ ,$

and

(B') $(\forall y_1)(\forall y_2)(\forall y_2')(\exists y_2^*)[\ \sim \underline{\text{Atom}}(y_1)$

$\wedge\ y_2 = \underline{\text{maxtree}}(\underline{\text{left}}(y_1))$

$A\ y_2' = \underline{\text{maxtree}}(\underline{\text{right}}(y_1))$

$\supset y_2^* = \underline{\text{maxtree}}(y_1)\ 1 \ .$

The proof of the lemmas will rely on the definition of **maxtree** (Axiom 1) and the following two axioms induced by the recursive definition (2) of the Terminal predicate:

(2a) $\underline{\text{Atom}}(v) \supset [\underline{\text{Terminal}}(u,v) \equiv (u = v)] \ ,$

and

(2b) $\sim \underline{\text{Atom}}(v) \supset [\underline{\text{Terminal}}(u,v) \equiv [\underline{\text{Terminal}}(u,\underline{\text{left}}(v)) \vee \underline{\text{Terminal}}(u,\underline{\text{right}}(v))]] .$

First we prove Lemma A'. By Axiom lit follows that we want to prove

$$(\forall y_1)(\exists y_2)\{\underline{\text{Atom}}(y_1) \supset [\underline{\text{Terminal}}(y_2,y_1) \wedge (\forall u)[\underline{\text{Terminal}}(u,y_1) \supset u \leq y_2]\ \text{I}]\ ,$$

or equivalently, using Axiom (2a) (with u being y_2 and v being y_1) ,

$$(\forall y_1)(\exists y_2)\{\underline{\text{Atom}}(y_1) \supset [y_2 = y_1 \wedge (\forall u)[u = y_1 \supset u \leq y_2]]\}\ .$$

It clearly suffices to take y_2 to be y_1 to complete the proof of the lemma. Therefore, the subroutine for $h(y_1)$ that we derive from the proof of this lemma is simply $h(y_1) = y_1$.

The proof of Lemma B' is a bit more complicated. Let us assume that y_1 is a tree such that $\sim \underline{\text{Atom}}(y_1)$ and let $y_2 = \underline{\text{maxtree}}(\underline{\text{left}}(y_1))$ and $y_2' = \underline{\text{maxtree}}(\underline{\text{right}}(y_1))$. We want to find a y_2^* (in terms of y_1, Y2 , and y_2') for which $y_2^* = \underline{\text{maxtree}}(y_1)$. This means, by Axiom 1, that we want y_2^* such that

$$\underline{\text{Terminal}}(y_2^*,y_1) \wedge (\forall u)[\underline{\text{Terminal}}(u,y_1) \supset u \leq y_2^*]\ .$$

This implies, by Axiom (2b) and our assumption $\sim \underline{\text{Atom}}(y_1)$, that we have to find a y_2^* satisfying the following three conditions:

(i) $\underline{\text{Terminal}}(y_2^*,\underline{\text{left}}(y_1)) \vee \underline{\text{Terminal}}(y_2^*,\underline{\text{right}}(y_1))$,

(ii) $(\forall u)[\underline{\text{Terminal}}(u,\underline{\text{left}}(y_1)) \supset u \leq y_2^*]$,

and

(iii) $(\forall u)[\underline{\text{Terminal}}(u,\underline{\text{right}}(y_1)) \supset u \leq y_2^*]$.

It was assumed that $y_2 = \underline{\text{maxtree}}(\underline{\text{left}}(y_1))$ and $y_2' = \underline{\text{maxtree}}(\underline{\text{right}}(y_1))$. Thus, using Axiom 1, condition (i) implies t& $y_2^* \leq y_2 \vee y_2^* \leq y_2'$, condition (ii) implies that $y_2 \leq y_2^*$, and condition(iii) implies that $y_2' \leq y_2^*$. This suggests that we take y_2^* to be $\underline{\text{max}}(y_2,y_2')$, which indeed satisfies the three conditions. Therefore,

the subroutine for $g(y_1,y_2,y_2')$ that we extract from the proof of this lemma is $g(y_1,y_2,y_2') = \underline{\max}(y_2,y_2')$.

The complete program derived from the proof is

$$\left\{\begin{array}{l} z = \underline{\text{maxtree}}(x) \ \underline{\text{where}} \\ \underline{\text{maxtree}}(y_1) \Leftarrow \text{if } \underline{\text{Atom}}(y_1) \ \underline{\text{then}} \ y_1 \\ \qquad\qquad\qquad\qquad \text{else } \underline{\max}(\underline{\text{maxtree}}(\underline{\text{left}}(y_1)),\underline{\text{maxtree}}(\underline{\text{right}}(y_1))) \ . \end{array}\right.$$

5. COMPLETE INDUCTION

The so-called complete induction principle is of the form

$$\frac{(\forall y_1)\{(\forall u)[u < y_1 \supset \alpha(u)] \supset \alpha(y_1)\}}{(\forall x)\alpha(x)} .$$

Intuitively, this means that if a property α holds for a natural number y_1 whenever it holds for every natural number u less than y_1, then it holds for every natural number x.

Although this rule is in fact logically equivalent to the earlier number-theoretic induction rules (see, for example, Mendelson [1964]), we shall see that it leads to a more general program form than the previous rules, and therefore it is more powerful for program-synthetic purposes. However, it puts more of a burden on the theorem prover because less of the program structure is fixed in advance and more is specified during the proof process.

We are most interested in the version of this rule in which $\alpha(y_1)$ has the form $(\exists y_2)\beta(y_1,y_2)$, i.e.,

$$\frac{(\forall y_1)\{(\forall u)[u < y_1 \supset (\exists y_2)\beta(u,y_2)] \supset (\exists y_2)\beta(y_1,y_2)\}}{(\forall x)(\exists y_2)\beta(x,y_2)} .$$

Thus, in order to prove a theorem of the form

$$(\forall x)(\exists z)\beta(x, z) ,$$

it suffices to prove a single lemma of the form

$$(A') \qquad (\forall y_1)(\exists u)(\forall y_2)(\exists y_2^*)\{[u < y_1 \supset \beta(u, y_2)] \supset \beta(y_1,y_2^*)\} .$$

From a proof of the lemma we extract one subroutine for computing the value of u in terms of y_1 (called $h(y_1)$), and another for computing the value of y_2^* in terms of y_1 and y_2 (called $g(y_1,y_2)$). These

functions satisfy the relation

(1) $[h(y_1) < y_1 \supset \mathcal{B}(h(y_1),y_2)] \supset \mathcal{B}(y_1,g(y_1,y_2))$ for every y_1 and y_2 .

The program form associated with the complete induction rule is then the recursive program form:

(2) $\begin{cases} z = f(x) \text{ where} \\ f(y) \Leftarrow g(y,f(h(y))) \end{cases}$.

This form requires some justification.

Assume that the function f satisfies the output predicate

(3) $\mathcal{B}(u, f(u))$ for all $u < x$.

We will try to show

$\mathcal{B}(x, f(x))$.

First, suppose $h(x) < x$. Then by the hypothesis (3), $\mathcal{B}(h(x),f(h(x)))$. Therefore, from (1) (taking y_1 to be x and y_2 to be $f(h(x))$) we obtain $\mathcal{B}(x,g(x,f(h(x))))$, i.e., by (2), $\mathcal{B}(x,f(x))$.

Now, suppose $h(x) \geq x$. Then taking y_1 to be x , the antecedent of (1) is true vacuously, and we conclude $\mathcal{B}(x,g(x,f(h(x))))$, i.e., by (2), $\mathcal{B}(x, f(x))$.

Example 12: The recursive quotient program (see Example 1).

We want to construct a recursive program to find (the integer part of) the quotient of two natural numbers x_1 and x_2 , given that $x_2 \neq 0$. Our output predicate is therefore

$\psi(x_1,x_2,z) : (\exists r)[x_1 = z \cdot x_2 + r \wedge r < x_2]$.

The theorem is then

$(\forall x_1)(\forall x_2)\{x_2 \neq 0 \supset (\exists z)(\exists r)[x_1 = z \cdot x_2 + r \wedge r < x_2]\}$.

Assume now that x_2 is a fixed positive integer. Then we wish to prove

$(\forall x_1)(\exists z)(\exists r)[x_1 = z \cdot x_2 + r \wedge r < x_2]$.

The theorem is now in the same form as the conclusion of the complete

induction rule, taking x to be x_1 , y_2 to be z , and

$$\mathcal{B}(y_1,y_2) : (\exists r)[y_1 = y_2 \cdot x_2 + r \wedge r < x_2]$$

Therefore, the single lemma to be proved is

$$(A') \quad (\forall y_1)(\exists u)(\forall y_2)(\exists y_2^*)\{[u < y_1 \supset (\exists r)[u = y_2 \cdot x_2 + r \wedge r < x_2]] \supset (\exists r)[y_1 = y_2^* \cdot x_2 + r \wedge r < x_2]\} ,$$

or equivalently,

$$(\forall y_1)(\exists u)(\forall y_2)(\exists y_2^*)(\forall r)(\exists r^*)\{[u < y_1 \supset [u = y_2 \cdot x_2 + r \text{ A } r < x_2]] \supset [y_1 = y_2^* \cdot x_2 + r^* \wedge r^* < x_2]\} .$$

If $y_1 < x_2$ we satisfy the conclusion of the lemma by taking y_2^* to be 0 (and r^* to be y_1) . If, on the other hand, $y_1 \geq x_2$, we take u to be $y_1 - x_2$, y_2^* to be y_2+1 (and r^* to be r) ; then the conclusion follows using an appropriate set of axioms for arithmetic. The program derived is then

$$\begin{cases} z = \underline{\text{div}}(x_1,x_2) \quad \underline{r \quad e} \\ \underline{\text{div}}(y_1,x_2) \Leftarrow \text{if } y_1 < x_2 \text{ } \underline{\text{then}} \text{ } 0 \text{ else } \underline{\text{div}}(y_1 - x_2,x_2)+1 . \end{cases}$$

Although the program we constructed has two input variables, we were able to use the single-variable induction principle in its synthesis by treating the second input x_2 as a free variable. Typically when constructing programs with more than one input variable, we shall have to use a suitably generalized induction rule.

The next example will use two input variables, and we will not be able to treat either of them as a free variable. Therefore we take this opportunity to demonstrate how to generalize the complete induction principle to construct programs with two inputs.

The form of complete induction was

$$\frac{(\forall y_1)\{(\forall u)[u < y_1 \supset \mathcal{A}(u)\ 1 \supset \mathcal{A}(y_1)\ 3}{(\forall x)\mathcal{A}(x)} \quad .$$

For the two-input-variable case, we take $\mathcal{A}(y_1)$ to be $(\forall y_2)\,(\exists y_3)\mathcal{B}(y_1,y_2,y_3)$, obtaining the version

$$\frac{(\forall y_1)\{(\forall u)[u < y_1 \supset (\forall y_2)(\exists y_3)\mathcal{B}(u,y_2,y_3)] \supset (\forall y_2)(\exists y_3)\mathcal{B}(y_1,y_2,y_3)\}}{(\forall x_1)(\forall x_2)(\exists y_3)\mathcal{B}(x_1,x_2,y_3)} \quad .$$

Suppose we want to prove a theorem of the form

$$(\forall x_1)(\forall x_2)(\exists z)\mathcal{B}(x_1,x_2,z) \quad .$$

This is the same as the consequent of the complete induction rule. Thus, it suffices to prove the antecedent as a lemma:

(A) $$(\forall y_1)\{(\forall u)[u < y_1 \supset (\forall y_2)(\exists y_3)\mathcal{B}(u,y_2,y_3)] \supset (\forall y_2)(\exists y_3)\mathcal{B}(y_1,y_2,y_3)\} \ ,$$

or equivalently,

(A') $$(\forall y_1)(\forall y_2)(\exists u)(\exists y_2^*)(\forall y_3)(\exists y_3^*)\{[u < y_1 \supset \mathcal{B}(u,y_2^*,y_3)\ 1 \supset \mathcal{B}(y_1,y_2,y_3^*)\} \quad .$$

From the proof of this lemma we extract three subroutines $h_1(y_1,y_2)$, $h_2(y_1,y_2)$, and $g(y_1,y_2,y_3)$ corresponding to u , y_2^* , and y_3^* , respectively. The program extracted from the proof of the theorem will be of the form

$$\begin{cases} z = f(x_1,x_2) \ \underline{\text{where}} \\ f(y_1,y_2) \Leftarrow g(y_1,y_2,f(h_1(y_1,y_2),h_2(y_1,y_2))) \quad . \end{cases}$$

Example 13: The greatest common divisor program.

The program to be constructed must find the greatest common divisor (gcd) of two positive integers x_1 and x_2. For simplicity, we ignore the possibility of x_1 or x_2 being 0, and the theorem to be proved is then

$$(\forall x_1)(\forall x_2)(\exists z)[z = \underline{gcd}(x_1, x_2)],$$

where

$$[z = \underline{gcd}(x_1, x_2)] \equiv \{z | x_1 \ A \ z | x_2 \ A \ \forall u[u | x_1 \ A \ u | x_2 \supset u \leq z]\}.$$

Here, $u|v$ means "u divides v evenly". Recall that the function gcd should not be considered to be primitive.

The theorem is in the same form as the conclusion of the complete induction rule (for two input variables), taking y_3 to be z, and

$$\mathcal{B}(y_1, y_2, y_3): Y3 = \underline{gcd}(y_1, y_2).$$

Therefore, we must prove the following lemma:

$$(A') \qquad (\forall y_1)(\forall y_2)(\exists u)(\exists y_2^*)(\forall y_3)(\exists y_3^*)\{[u < y_1 \supset Y3 = \underline{gcd}(u, y_2^*)] \supset y_3^* = \underline{gcd}(y_1, y_2)\}.$$

This is one of the proofs we consider to be a challenge for existing theorem-proving systems. We suggest taking

u to be $\underline{rem}(y_2, y_1)$

y_2^* to be y_1,

and y_3^* to be if $\underline{rem}(y_2, y_1) = 0$ then y_1 else y_3,

where $\underline{rem}(y_2, y_1)$ is the remainder when y_2 is divided by y_1. Therefore,

$h_1(y_1, y_2)$ is $\underline{rem}(y_2, y_1)$,

$h_2(y_1,y_2)$ is y_1 ,

and $g(y_1,y_2,y_3)$ is $\underline{\text{if}}\ \underline{\text{rem}}(y_2,y_1) = 0\ \underline{\text{then}}\ y_1\ \underline{\text{else}}\ y_3$.

The complete gcd program extracted is therefore

$$\left\{\begin{array}{l} z = \underline{\text{gcd}}(x_1,x_2)\ \underline{\text{where}} \\ \underline{\text{gcd}}(y_1,y_2) \Leftarrow \underline{\text{if}}\ \underline{\text{rem}}(y_2,y_1) = 0\ \text{then}\ y_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{else}\ \underline{\text{gcd}}(\underline{\text{rem}}(y_2,y_1),y_1) \end{array}\right. .$$

6. TRANSLATION FROM RECURSION TO ITERATION

Iterative programs and recursive programs compute the same class of functions (namely, the partial recursive functions). However, recursive programs are commonly far more inefficient in time and space than the corresponding iterative programs. Although it is straightforward to transform an iterative program into an equivalent recursive program, the reverse transformation presents difficulties. (See, for example, McCarthy [1963a], Strong [1970], and Paterson and Hewitt [1970].)

LISP and ALGOL compilers, for example, translate recursive programs into iterative programs that use stacks, without changing the essence of the computation. Using program-synthetic techniques, it is sometimes possible to perform the transformation in such a way that the resulting iterative program performs the computation in a fundamentally better way than the original recursive program. Although we have no mechanism to ensure this improvement in general, we shall see how this occurs in the two examples presented in this section, the first concerning the reverse function and the second, the Fibonacci sequence.

Example 14: The reverse function (see Example 2).

We are given a recursive reverse program:

```
z = reverse(x) where
reverse(y) <= if Null(y) then NIL
                        else append(reverse(cdr(y)),list(car(y))) .
```

As we mentioned earlier, this definition is quite inefficient since it involves repeated computation of the append function, which in itself requires a relatively complex computation.

The theorem to be proved is

$$(\forall x)(\exists z)[z = \underline{\text{reverse}}(x)] \ .$$

Recall that the reverse function is not considered to be primitive . For efficiency we also omit the append function from the list of primitives.

Since we want to write an iterative program, we must use the iterative list induction rule:

$$\frac{(\exists y_1)(\exists y_2)\mathcal{B}(x,y_1,y_2) \qquad (\forall y_1)[\sim \underline{\text{Null}}(y_1) \wedge (\exists y_2)\mathcal{B}(x,y_1,y_2) \supset (\exists y_2)\mathcal{B}(x,\underline{\text{cdr}}(y_1),y_2)]}{(\exists y_2)\mathcal{B}(x,\text{NIL},y_2)} \ .$$

Aside from this rule we have two axioms that result directly from the given definition of the reverse function:

(1a) $\underline{\text{reverse}}(\text{NIL}) = \text{NIL}$,

(1b) $\sim\underline{\text{Null}}(y) \supset [\underline{\text{reverse}}(y) = \underline{\text{append}}(\underline{\text{reverse}}(\underline{\text{cdr}}(y)),\underline{\text{list}}(\underline{\text{car}}(y)))]$.

Furthermore the system will use the following axioms chosen from its supply during the course of the proof;

(2a) $\underline{\text{append}}(\text{NIL},u) = u$,

(2b) $\underline{\text{append}}(u,\text{NIL}) = u$,

(3) $\underline{\text{append}}(u,\underline{\text{append}}(v,w)) = \underline{\text{append}}(\underline{\text{append}}(u,v),w)$, and

(4) $\underline{\text{append}}(\underline{\text{list}}(u),v) = \text{cons}(u,v)$.

The theorem to be proved, $(\forall x)(\exists z)[z = \underline{\text{reverse}}(x)]$, is not in the correct form to apply the iterative induction because NIL does not occur in it. However, by Axiom 2a and the definition of reverse

(Axiom 1a), our theorem prover will translate the theorem into the following satisfactory form:

$$(\forall x)(\exists z)[\underline{\text{append}}(\underline{\text{reverse}}(\text{NIL}),z) = \underline{\text{reverse}}(x)]$$

Therefore, we can apply the iterative list induction rule with

$$\mathcal{B}(x,y_1,y_2) : \underline{\text{append}}(\underline{\text{reverse}}(y_1),y_2) = \underline{\text{reverse}}(x) \quad ,$$

and the two lemmas to be proved are:

(A) $$(\exists y_1)(\exists y_2)[\underline{\text{append}}(\underline{\text{reverse}}(y_1),y_2) = \underline{\text{reverse}}(x)] \ ,$$

and

(B') $$(\forall y_1)(\forall y_2)(\exists y_2^*)[\sim \underline{\text{Null}}(y_1) \wedge \underline{\text{append}}(\underline{\text{reverse}}(y_1),y_2) = \underline{\text{reverse}}(x) \supset \underline{\text{append}}(\underline{\text{reverse}}(\underline{\text{cdr}}(y_1)),y_2^*) = \underline{\text{reverse}}(x)]$$

Using Axiom 2b, the system chooses y_1 to be x and y_2 to be NIL , concluding the proof of Lemma A.

To prove Lemma B', the system assumes $\sim \underline{\text{Null}}(y_1)$ and $\underline{\text{append}}(\underline{\text{reverse}}(y_1),y_2) = \underline{\text{reverse}}(x)$. Using the definition of reverse (Axiom 1b), and the assumption that $\sim \underline{\text{Null}}(y_1)$, it derives

$$\underline{\text{reverse}}(y_1) = \underline{\text{append}}(\underline{\text{reverse}}(\underline{\text{cdr}}(y_1)),\underline{\text{list}}(\underline{\text{car}}(y_1))) \ .$$

Substituting in the hypothesis, it deduces

$$\underline{\text{append}}(\underline{\text{append}}(\underline{\text{reverse}}(\underline{\text{cdr}}(y_1)),\underline{\text{list}}(\underline{\text{car}}(y_1))),y_2) = \underline{\text{reverse}}(x) \ .$$

Using the associative rule for append (Axiom 3), it obtains

$$\underline{\text{append}}(\underline{\text{reverse}}(\underline{\text{cdr}}(y_1)),\underline{\text{append}}(\underline{\text{list}}(\underline{\text{car}}(y_1)),y_2)) = \underline{\text{reverse}}(x)$$

Then, from Axiom 4, it derives

$$\underline{\text{append}}(\underline{\text{reverse}}(\underline{\text{cdr}}(y_1)),\underline{\text{cons}}(\underline{\text{car}}(y_1),y_2)) = \underline{\text{reverse}}(x) \ .$$

Comparing this formula with the desired conclusion, the system takes y_2^* to be $\underline{\text{cons}}(\underline{\text{car}}(y_1),y_2)$, concluding the proof.

Such a proof is well within the capabilities of existing theorem provers. In fact, the above proof of Lemma B' has actually been found (see Brice and Derksen [1970]) using the QA3 theorem-proving system (Green and Raphael [1968]) with Morris's E-resolution [1969].

Since in the proof of Lemma A, y_1 and y_2 were replaced by x and NIL, respectively, and in the proof of Lemma B', y_2^* was replaced by $\underline{cons}(\underline{car}(y_1),y_2)$, the iterative program illustrated in Figure 12 will be constructed. Note that this program is far more efficient than its recursive counterpart.

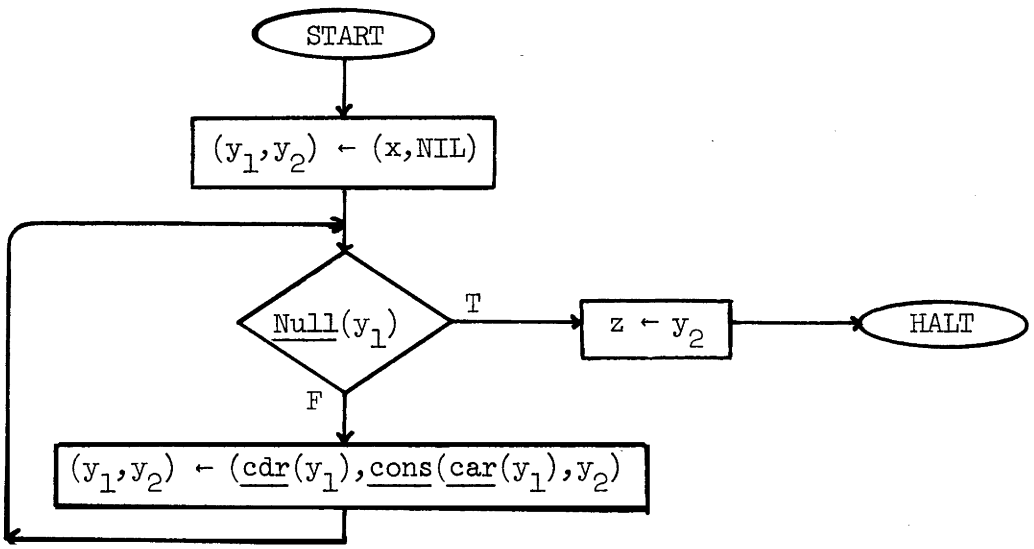


Figure 12: Iterative reverse program

Example 15: The Fibonacci sequence.

The advantage of iteration over recursion is particularly apparent in the computation of the Fibonacci series

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...,

each of whose terms (after the second) is the sum of the preceding two. Given a natural number x, the value of the x-th Fibonacci number

is most simply defined by the recursive program.

$$\left\{\begin{array}{l} z = \underline{\text{fibonacci}}(x) \ \underline{\text{where}} \\ \underline{\text{fibonacci}}(y) \Leftarrow \underline{\text{if}}\ (y = 0 \vee y = 1)\ \underline{\text{then}}\ 1 \\ \qquad\qquad\qquad\qquad \underline{\text{else fibonacci}}(y-1) + \underline{\text{fibonacci}}(y-2) \end{array}\right. .$$

In practice this program is grossly inefficient, involving many repetitions of the same computation. We would like to use our approach to translate this program into an efficient iterative program with no redundant computation.

The theorem to be proved is simply

$$(\forall x)(\exists z)[z = \underline{\text{fibonacci}}(x)] \ .$$

The recursive definition of the fibonacci function implies the axioms:

(1a) $\quad (u = 0 \vee u = 1) \supset \underline{\text{fibonacci}}(u) = 1 ,$

(1b) $\quad (u \geq 2) \supset \underline{\text{fibonacci}}(u) = \underline{\text{fibonacci}}(u-1) + \underline{\text{fibonacci}}(u-2) ,$

or equivalently,

$$\underline{\text{fibonacci}}(u'+2) = \underline{\text{fibonacci}}(u'+1) + \underline{\text{fibonacci}}(u') \ .$$

Axiom 1a suggests to the theorem prover that the case $(x = 0 \vee x = 1)$ be treated separately; in this event, we take z to be 1 , and the output relation is satisfied.

It remains to prove

$$(\forall x)\{x \geq 2 \supset (\exists z)[z = \underline{\text{fibonacci}}(x)]\} ,$$

or equivalently (using Axiom 1b)

$$(\forall x)\{x \geq 2 \supset (\exists z)[z = \underline{\text{fibonacci}}(x-1) + \underline{\text{fibonacci}}(x-2)]\} .$$

Thus, taking x' to be x-2, we have

$$(\forall x')(\exists z)[z = \underline{\text{fibonacci}}(x'+1) + \underline{\text{fibonacci}}(x')] .$$

Since the "plus" operator is primitive, taking z to be z_1+z_2, it suffices to prove

$$(\forall x')(\exists z_1)(\exists z_2)[z_1 = \underline{\text{fibonacci}}(x'+1) \wedge z_2 = \underline{\text{fibonacci}}(x')] .$$

Note that we now have two output variables z_1 and z_2 rather than one. However, the proof procedure is precisely analogous to the single-variable case; the iterative going-up induction principle used is

$$\frac{(\exists y_2)(\exists y_3)\beta(0,y_2,y_3) \qquad (\forall y_1)[(\exists y_2)(\exists y_3)\beta(y_1,y_2,y_3) \supset (\exists y_2)(\exists y_3)\beta(y_1+1,y_2,y_3)]}{(\forall x)(\exists y_2)(\exists y_3)\beta(x,y_2,y_3)} ,$$

with

$$\beta(y_1,y_2,y_3) : y_2 = \underline{\text{fibonacci}}(y_1+1) \wedge y_3 = \underline{\text{fibonacci}}(y_1) .$$

Taking z_1 to be y_2 and z_2 to be y_3 (i.e., z is y_2+y_3), the conclusion of this induction rule is identical to the modified theorem. Thus, the two lemmas to be proved are

(A) $$(\exists y_2)(\exists y_3)[y_2 = \underline{\text{fibonacci}}(1) \wedge y_3 = \underline{\text{fibonacci}}(0)]$$

and

(B') $$(\forall y_1)(\forall y_2)(\forall y_3)(\exists y_2^*)(\exists y_3^*)\{[y_2 = \underline{\text{fibonacci}}(y_1+1) \wedge y_3 = \underline{\text{fibonacci}}(y_1)] \supset [y_2^* = \underline{\text{fibonacci}}(y_1+2) \wedge y_3^* = \underline{\text{fibonacci}}(y_1+1)]\} .$$

Lemma A is proved using Axiom 1a taking y_2 and y_3 both to be 1.

To prove Lemma B' we assume $y_2 = \underline{\text{fibonacci}}(y_1+1)$ and $y_3 = \underline{\text{fibonacci}}(y_1)$. Then taking $y_2^* = y_2+y_3$, as suggested by Axiom 1b, and. $y_3^* = Y2$, as suggested by the hypothesis, we have completed the proof of Lemma B'.

The program extractor combines all the replacements and substitutions made in the proof to form the program of Figure 13, which exhibits none of the crude inefficiencies of the original recursive program. The reader may observe how closely the operations in the program mirror the steps of the proof.

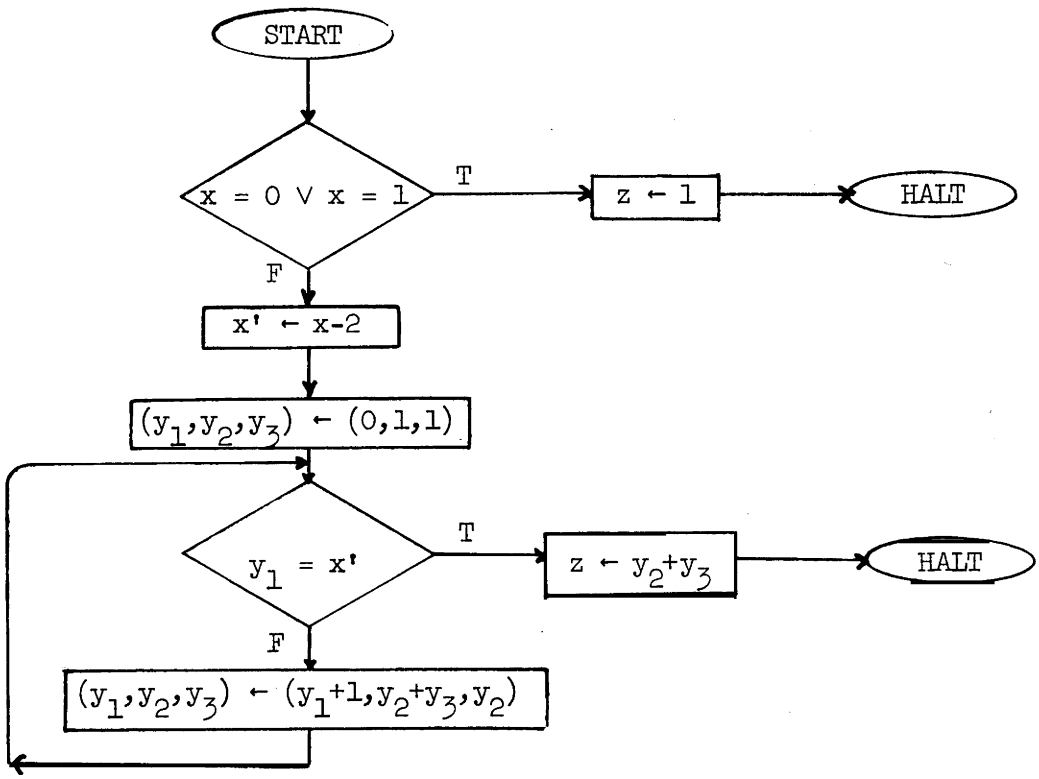


Figure 13: Iterative fibonacci program

7. FUTURE RESEARCH

Clearly the results reported in this note represent but a step in the direction of automatic program synthesis. Our chief goal was not to present a completed work, but rather to stimulate other people to examine these problems.

(a) Suggested theorem-proving research

The foundation of our approach, and its chief weakness, lie in the theorem prover. We have mentioned that many of our proofs are probably beyond the state of the art of mechanical theorem proving, although none of them are terribly difficult. We therefore can use our experience to pinpoint some weaknesses in the current methods and to suggest some directions for theorem-proving research.

Any theorem-proving system stores its knowledge either in the form of axioms (which are simply assertions) or rules of inference (which are methods for transforming assertions). A system that relies mainly on axioms is very general; new facts may be introduced without modifying the system because new axioms may be added long after the system is written. However, without restrictive strategies about how each axiom is to be used, such systems tend to thrash and flounder. On the other hand, systems such as King's[1969] (see also King and Floyd [1970]), which rely on rules of inference applying to a specific semantic domain, proceed with a great sense of direction but usually require reprogramming when new facts are introduced.

We therefore would like to see a system that combines the virtues of both approaches, using rules of inference when possible and axioms when

necessary. We further hope that the user would be able to introduce new rules of inference without being forced to reprogram the system. Thus we would be able to give the system special knowledge about the semantic domain (the integers or lists, for example) without affecting its generality.

We are dissatisfied with the large number of equivalent induction principles required by our system. One might prefer to have a single general induction rule with a more powerful program extraction mechanism (see, for example, Burstall [1969], Park [1970], and Scott [1969]). It is not yet clear what this mechanism would be, and we are not sure that the machine implementation of such a rule in a theorem-proving system would be feasible.

Finally, it occurred to us during the preparation of this paper that partial function logic (see McCarthy [1963b]) would be a more appropriate vehicle for program synthesis, because in this language we may discuss partial functions, whereas in the usual predicate calculus all operations and predicates are assumed to be total. We believe the techniques we have already outlined above apply to partial function logic as well. Some work has already been done by Hayes [1969] towards the machine implementation of this logic. Taking this remark in conjunction with a paper by Manna and McCarthy [1970] suggests that partial function logic may be the most natural language for program analysis and synthesis.

(b) -Language and representation

In our discussion we have used a modified predicate calculus in specifying the program to be constructed. This suggests that predicate calculus could be used as a higher-level programming language, where the

compiler would be a program synthesis system, extracting a program in a lower-level language.

On the other hand, we are not bound to the use of predicate calculus as our source language. Programmers might find such a language lacking in readability and conciseness. However, any language that might be developed for expressing input and output relations would be satisfactory so long as the system could translate it into the language its theorem prover understood.

Of course, there are cases in which it is as easy to write the program itself as to write input and output relations describing it. However, this is more likely to be the case with trivial examples than with complex realistic programs.

(c) Interactive program synthesis

We have not considered the possibility that the synthesizer might interact with the user in constructing its programs. However, an interactive approach might lead immediately to a more practical system. For example, if the theorem prover were interactive the power of the program synthesizer would be greatly increased. Alternatively, we might interact by allowing the user to suggest program segments to the synthesizer, allowing the system to incorporate them into the program.

(d) Program modification

We have not approached the problem of constructing efficient programs in any systematic way. We have contented ourselves with the construction of correct programs, and have seldom been very critical of the programming

quality exhibited. Although in Section 6 we illustrated that we can write more efficient programs by avoiding recursion and declaring inefficient subroutines non-primitive, more general work in this direction is clearly needed.

Once we have developed a method for controlling the efficiency of the extracted program, we not only can produce better programs with the purely synthetic approach, but also can use our techniques to write better compilers and program optimizers, which transform programs written by human beings. We take such a program (or a portion thereof) and transform it into its representation in predicate calculus (see Ashcroft[1970], Burstall [1970], Manna [1969], and Manna and Pnueli [1970]), which is then taken as the specification of a new, more efficient reconstruction.

Another way program-synthetic techniques may be used in the improvement of an already existing program is in the construction of an automatic debugging system. Current program verification methods give us a way to detect and locate errors in a program; we then can use the program-synthetic approach to replace the incorrect segment without affecting the remainder of the program.

Acknowledgments

We would like to thank Claude Brice and Jan Derksen for programming in connection with this work. We are indebted to Johns Rulifson for discussions leading to the suggestion of examples presented in the paper. We are also grateful to Edward Ashcroft, Terry Davis, Jan Derksen, Stephen Ness, and Nils Nilsson for critical reading of the manuscript and subsequent helpful suggestions. Nils Nilsson obtained simplifications in several of our examples.

REFERENCES

E. A. ASHCROFT [1970], "Mathematical Logic Applied to the Semantics of Computer Programs," Ph.D. Thesis, Imperial College, London.

C. BRICE and J. DERKSEN [1970], "The QA3 Implementation of E-Resolution", Stanford Research Institute, Artificial Intelligence Group (to appear).

R. M. BURSTALL [1969], "Proving Properties of Programs by Structural Induction," Comp. J. 12, 1, 41-48.

R. M. BURSTALL [1970], "Formal Description of Program Structure and Semantics in First-Order Logic," in Machine Intelligence 5 (Eds. Meltzer and Michie), Edinburgh University Press, 79-98.

R. W. FLOYD [1967], "Assigning Meaning to Programs," in Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Vol. 19, 19-32.

C. GREEN [1969a], "Application of Theorem Proving to Problem Solving," Proc. of the International Joint Conference on Artificial Intelligence, Washington, D.C.

C. GREEN [1969b], "The Application of Theorem Proving to Question-Answering Systems," Ph.D. Thesis, Stanford University, Stanford, California.

C. GREEN and B. RAPHAEL [1968], "The Use of Theorem-Proving Techniques in Question-Answering Systems," Proc. 23rd Nat. Conf. ACM, Thompson Book Company, Washington, D.C.

P. J. HAYES [1969], "A Machine-Oriented Formulation of the Extended Functional Calculus," Stanford Artificial Intelligence Project, Memo 62. Also appeared as Metamathematics Unit report, University of Edinburgh, Scotland.

J. KING [1969], "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pa.

J. KING and R. W. FLOYD [1970], "Interpretation Oriented Theorem Prover Over Integers," Second Annual ACM Symposium on Theory of Computing, Northampton, Mass.

D. LUCKHAM and N. J. NILSSON [1970], "Extracting Information from Resolution Proof Trees," Stanford Research Institute, Artificial Intelligence Group, Technical Note 32.

Z. MANNA [1969], "The Correctness of Programs," J. of Computer and System Sciences, Vol. 3, No. 2, 119-127.

Z. MANNA and J. McCARTHY [1970], "Properties of Programs and Partial Function Logic," in Machine Intelligence 5 (Eds. Meltzer and Michie), Edinburgh University Press, 79-98.

Z. MANNA and A. PNUELI [1970], "Formalization of Properties of Functional Programs," JACM, Vol. 17, No. 3.

J. McCARTHY [1962], "Towards a Mathematical Science of Computation," Proc. IFIP Congress 62, North-Holland, Amsterdam.

J. McCARTHY [1963a], "A Basis for a Mathematical Theory of Computation," in Computer Programming and Formal Systems (Eds. Braffort and Hirshberg), North Holland, Amsterdam.

J. McCARTHY [1963b], "Predicate Calculus with 'Undefined' as a Truth-Value," Stanford Artificial Intelligence Project, Memo 1, Stanford University.

E. MENDELSON [1964], Introduction to Mathematical Logic, Van Nostrand, Princeton, New Jersey.

J. B. MORRIS [1969], "E-Resolution: Extension of Resolution to Include the Equality Relation," Proc. of the International Joint Conference on Artificial Intelligence, Washington, D. C.

D. PARK [1970], "Fixpoint Induction and Proofs of Program Properties," in Machine Intelligence 5 (Eds. Meltzer and Michie), Edinburgh University Press, 59-78.

M. S. PATERSON and C. E. HEWITT [1970], "Comparative Schematology (Part 1)," Unpublished notes.

J. ROBINSON [1965], 'A Machine-Oriented Logic Based on the Resolution Principle," JACM, Vol. 12.

D. SCOTT [1969], "A Type-Theoretical Alternative to CUCH, ISWIM, OWHY," Unpublished memo.

H. SIMON [1963], "Experiments with a Heuristic Compiler," JACM, Vol. 10.

J. SLAGLE [1965], "Experiments with a Deductive Question-Answering Program," CACM, Vol. 8.

H. R. STRONG [1970], "Translating Recursion Equations into Flow Charts," Second Annual ACM Symposium on Theory of Computing, Northampton, Mass.

R. J. WALDINGER [1969], "Constructing Programs Automatically Using Theorem Proving," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pa.

R. J. WALDINGER and R. C. T. LEE [1969], "PROW: A Step Toward Automatic Program Writing," Proc. of the International Joint Conference on Artificial Intelligence, Washington, D.C.