A Deductive Approach to Program Synthesis

ZOHAR MANNA
Stanford University and Weizmann Institute
and
RICHARD WALDINGER
SRI International

Program synthesis is the systematic derivation of a program from a given specification. A deductive approach to program synthesis is presented for the construction of recursive programs. This approach regards program synthesis as a theorem-proving task and relies on a theorem-proving method that combines the features of transformation rules, unification, and mathematical induction within a single framework.

Key Words and Phrases: mathematical induction, program synthesis, program transformation, resolution, theorem proving
CR Categories: 3.64, 4.20, 5.21, 5.24

MOTIVATION

The early work in program synthesis relied strongly on mechanical theorem-proving techniques. The work of Green [5] and Waldinger and Lee [13], for example, depended on resolution-based theorem proving; however, the difficulty of representing the principle of mathematical induction in a resolution framework hampered these systems in the formation of programs with iterative or recursive loops. More recently, program synthesis and theorem proving have tended to go their separate ways. Newer theorem-proving systems are able to perform proofs by mathematical induction (e.g., Boyer and Moore [2]) but are useless for program synthesis because they have sacrificed the ability to prove theorems involving existential quantifiers. Recent work in program synthesis (e.g., Burstall and Darlington [3] and Manna and Waldinger [7]), on the other hand, has abandoned

This research was supported in part by the National Science Foundation under Grants MCS 76-83655 and MCS 78-02591, in part by the Office of Naval Research under Contracts N00014-76-C-0687 and N00014-75-C-0816, in part by the Defense Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, and in part by the United States–Israel Binational Science Foundation.
Authors' addresses: Z. Manna, Department of Computer Science, Stanford University, Stanford, CA 94305; R. Waldinger, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.

the theorem-proving approach and has relied instead on the direct application of transformation or rewriting rules to the program's specification; in choosing this path, these systems have renounced the use of such theorem-proving techniques as unification or induction.

In this paper we describe a framework for program synthesis that again relies on a theorem-proving approach. This approach combines techniques of unification, mathematical induction, and transformation rules within a single deductive system. We outline the logical structure of this system without considering the strategic aspects of how deductions are directed. Although no implementation exists, the approach is machine oriented and ultimately intended for implementation in automatic synthesis systems.

In the next section we give examples of specifications accepted by the system. In the succeeding sections we explain the relation between theorem proving and our approach to program synthesis.

SPECIFICATION

The specification of a program allows us to express the purpose of the desired program, without indicating an algorithm by which that purpose is to be achieved. Specifications may contain high-level constructs that are not computable, but are close to our way of thinking. Typically, specifications involve such constructs as the quantifiers *for all*... and *for some*..., the set constructor $\{x: \ldots\}$, and the descriptor *find z such that*....

For example, to specify a program to compute the integer square root of a nonnegative integer n, we would write

$$\begin{aligned} &\mathit{sqrt}(n) \Leftarrow \mathit{find}\ z\ \mathit{such\ that} \\ &\qquad \mathit{integer}(z)\ \mathit{and}\ z^2 \leq n < (z+1)^2 \\ &\quad \mathit{where}\ \mathit{integer}(n)\ \mathit{and}\ 0 \leq n. \end{aligned}$$

Here, the *input condition*

$$\mathit{integer}(n)\ \mathit{and}\ 0 \leq n$$

expresses the class of legal inputs to which the program is expected to apply. The *output condition*

$$\mathit{integer}(z)\ \mathit{and}\ z^2 \leq n < (z+1)^2$$

describes the relation the output z is intended to satisfy.

To describe a program to sort a list l, we might write

$$\begin{aligned} &\mathit{sort}(l) \Leftarrow \mathit{find}\ z\ \mathit{such\ that} \\ &\qquad \mathit{ordered}(z)\ \mathit{and}\ \mathit{perm}(l, z) \\ &\quad \mathit{where}\ \mathit{islist}(l). \end{aligned}$$

Here, *ordered*(z) expresses that the elements of the output list z should be in nondecreasing order; *perm*(l, z) expresses that z should be a permutation of the input l; and *islist*(l) expresses that l can be assumed to be a list.

To describe a program to find the last element of a nonempty list l, we might write

$$\begin{aligned} &\mathit{last}(l) \Leftarrow \mathit{find}\ z\ \mathit{such\ that} \\ &\qquad \mathit{for\ some}\ y,\ l = y <> [z] \\ &\quad \mathit{where}\ \mathit{islist}(l)\ \mathit{and}\ l \neq [\,]. \end{aligned}$$

Here, $u<>v$ denotes the result of appending the two lists u and v; $[u]$ denotes the list whose sole element is u; and [] denotes the empty list. (Thus, [A B C]<>[D] yields [A B C D]; therefore, by the above specification, *last*([A B C D]) = D.)

In general, we are considering the synthesis of programs whose specifications have the form

$$\begin{aligned} &f(a) \Leftarrow \textit{find } z \textit{ such that } R(a, z)\\ &\quad \textit{where } P(a). \end{aligned}$$

Here, a denotes the input of the desired program and z denotes its output; the input condition $P(a)$ and the output condition $R(a, z)$ may themselves contain quantifiers and set constructors (but not the *find* descriptor).

The above specification describes an applicative program, one which yields an output but produces no side effects. To derive a program from such a specification, we attempt to prove a theorem of the form

$$\begin{aligned} &\textit{for all } a,\\ &\textit{if } P(a)\\ &\textit{then for some } z,\ R(a, z). \end{aligned}$$

The proof of this theorem must be constructive, in the sense that it must tell us how to find an output z satisfying the desired output condition. From such a proof, a program to compute z can be extracted.

The above notation can be extended to describe several related programs at once. For example, to specify the programs $div(i, j)$ and $rem(i, j)$ for finding the integer quotient and remainder, respectively, of dividing a nonnegative integer i by a positive integer j, we write

$$\begin{aligned} (div(i, j), rem(i, j)) \Leftarrow\ &\textit{find } (y, z) \textit{ such that integer}(y) \textit{ and}\\ &\quad \textit{integer}(z) \textit{ and } i = y \cdot j + z \textit{ and } 0 \le z \textit{ and } z < j\\ &\textit{where integer}(i) \textit{ and integer}(j) \textit{ and } 0 \le i \textit{ and } 0 < j. \end{aligned}$$

BASIC STRUCTURE

The basic structure employed in our approach is the *sequent*, which consists of two lists of sentences, the *assertions* $A_1, A_2, \ldots, A_m$, and the *goals* $G_1, G_2, \ldots, G_n$. With each assertion or goal there may be associated an entry called the *output expression*. This output entry has no bearing on the proof itself, but records the program segment that has been constructed at each stage of the derivation (cf. the "answer literal" in Green [5]). We denote a sequent by a table with three columns: assertions, goals, and outputs. Each row in the sequent has the form

assertions	*goals*	*outputs*
$A_i(a, x)$		$t_i(a, x)$

or

	$G_j(a, x)$	$t_j(a, x)$

The meaning of a sequent is that if all instances of each of the assertions are true, then some instances of at least one of the goals is true; more precisely, the sequent has the same meaning as its *associated sentence*

if for all x, $A_1(a, x)$ *and*
for all x, $A_2(a, x)$ *and*
⋮
for all x, $A_m(a, x)$
then for some x, $G_1(a, x)$ *or*
for some x, $G_2(a, x)$ *or*
⋮
for some x, $G_n(a, x)$

where a denotes all the constants of the sequent and x denotes all the free variables. (In general, we denote constants or tuples of constants by $a, b, c, \ldots, n$ and variables or tuples of variables by $u, v, w, \ldots, z$.) If some instance of a goal is true (or some instance of an assertion is false), the corresponding instance of its output expression satisfies the given specification. In other words, if some instance $G_j(a, e)$ is true (or some instance $A_i(a, e)$ is false), then the corresponding instance $t_j(a, e)$(or $t_i(a, e)$) is an acceptable output.

Note that (1) an assertion or goal is not required to have an output entry; (2) an assertion and a goal never occupy the same row of the sequent; (3) the variables in each row are "dummies" that we can systematically rename without changing the meaning of the sequent.

The distinction between assertions and goals is artificial and does not increase the logical power of the deductive system. In fact, if we delete a goal from a sequent and add its negation as a new assertion, we obtain an equivalent sequent; similarly, we can delete an assertion from a sequent and add its negation as a new goal without changing the meaning of the sequent. This property is known as *duality*. Nevertheless, the distinction between assertions and goals makes our deductions easier to understand.

If initially we are given the specification

$$f(a) \Leftarrow \textit{find } z \textit{ such that } R(a, z) \quad \textit{where } P(a),$$

we construct the initial sequent

assertions	*goals*	*outputs* $f(a)$
$P(a)$		
	$R(a, z)$	z

In other words, we assume that the input condition $P(a)$ is true, and we want to prove that for some z, the goal $R(a, z)$ is true; if so, z represents the desired output of the program $f(a)$. The output z is a variable, for which we can make substitutions; the input a is a constant. If we prefer, we may remove quantifiers in $P(a)$ and $R(a, z)$ by the usual skolemization procedure (see, e.g., Nilsson [11]).

The input condition $P(a)$ is not the only assertion in the sequent; typically, simple, basic axioms, such as $u = u$, are represented as assertions that are tacitly present in all sequents. Many properties of the subject domain, however, are represented by other means, as we shall see.

The deductive system we describe operates by causing new assertions and goals, and corresponding new output expressions, to be added to the sequent without changing its meaning. The process terminates if the goal *true* (or the assertion *false*) is produced, whose corresponding output expression consists entirely of primitives from the target programming language; this expression is the desired program. In other words, if we develop a row of form

	true	t

or

false		t

where t is a primitive expression, the desired program is of form

$$f(a) \Leftarrow t.$$

Note that this deductive procedure never requires us to establish new sequents or (except for strategic purposes) to delete an existing assertion or goal. In this sense, the approach more resembles resolution than "natural deduction."

Suppose we are required to construct two related programs $f(a)$ and $g(a)$; i.e., we are given the specification

$$\begin{aligned}&(f(a), g(a)) \Leftarrow \textit{find } (y, z) \textit{ such that } R(a, y, z)\\ &\quad \textit{where } P(a).\end{aligned}$$

Then we construct an initial sequent with two output columns

assertions	*goals*	*outputs* $f(a)$	$g(a)$
$P(a)$			
	$R(a, y, z)$	y	z

If we subsequently succeed in developing a terminal row, say of form

	true	s	t

where both s and t are primitive expressions, then the desired programs are

$$f(a) \Leftarrow s$$

and

$$g(a) \Leftarrow t.$$

In the remainder of this paper we outline the deductive rules of our system and their application to program synthesis.

SPLITTING RULES

The splitting rules allow us to decompose an assertion or goal into its logical components. For example, if our sequent contains an assertion of form F *and* G, we can introduce the two assertions F and G into the sequent without changing its meaning. We will call this the *andsplit rule* and express it in the following notation:

assertions	*goals*	*outputs*
F and G		*t*
F		*t*
G		*t*

This means that if rows matching those above the double line are present in the sequent, then the corresponding rows below the double line may be added.

Similarly, we have the *orsplit rule*

assertions	*goals*	*outputs*
	F or G	*t*
	F	*t*
	G	*t*

and the *ifsplit rule*

assertions	*goals*	*outputs*
	if F then G	*t*
F		*t*
	G	*t*

There is no *orsplit rule* or *ifsplit rule* for assertions and no *andsplit rule* for goals. Note that the output entries for the consequents of the splitting rules are exactly the same as the entries for their antecedents.

Although initially only the goal has an output entry, the *ifsplit rule* can introduce an assertion with an output entry. Such assertions are rare in practice, but can arise by the action of such rules.

TRANSFORMATION RULES

Transformation rules allow one assertion or goal to be derived from another. Typically, transformations are expressed as conditional rewriting rules

$$r \Rightarrow s \quad \textit{if } P$$

meaning that in any assertion, goal, or output expression, a subexpression of form r can be replaced by the corresponding expression of form s, provided that the condition P holds. We never write such a rule unless r and s are equal terms or equivalent sentences, whenever condition P holds. For example, the transformation rule

$$u \in v \Rightarrow u = \textit{head}(v) \textit{ or } u \in \textit{tail}(v) \quad \textit{if islist}(v) \textit{ and } v \neq []$$

expresses that an element belongs to a nonempty list if it equals the head of the list or belongs to its tail. (Here, *head*(v) denotes the first element of the list v, and *tail*(v) denotes the list of all but the first element.) The rule

$$u \mid 0 \Rightarrow \textit{true} \quad \textit{if integer}(u) \textit{ and } u \neq 0$$

expresses that every nonzero integer divides zero.

If a rule has the vacuous condition *true*, we write it with no condition; for example, the logical rule

$$Q \textit{ and true} \Rightarrow Q$$

may be applied to any subexpression that matches its left-hand side.

A transformation rule

$$r \Rightarrow s \quad \textit{if } P$$

is not permitted to replace an expression of form s by the corresponding expression of form r when the condition P holds, even though these two expressions have the same values. For that purpose, we would require a second rule

$$s \Rightarrow r \quad \textit{if } P.$$

For example, we might include the rule

$$x + 0 \Rightarrow x \quad \textit{if number}(x)$$

but not the rule

$$x \Rightarrow x + 0 \quad \textit{if number}(x).$$

Assertions and goals are affected differently by transformation rules. Suppose

$$r \Rightarrow s \quad \textit{if } P$$

is a transformation rule and F is an assertion containing a subexpression r' which is not within the scope of any quantifier. Suppose also that there exists a *unifier* for r and r', i.e., a substitution θ such that $r\theta$ and $r'\theta$ are identical. Here, $r\theta$ denotes the result of applying the substitution θ to the expression r. We can assume that θ is a "most general" unifier (in the sense of Robinson [12]) of r and r'. We rename the variables of F, if necessary, to ensure that it has no variables in common with the transformation rule. By the rule, we can conclude that if $P\theta$ holds, then $r\theta$ and $s\theta$ are equal terms or equivalent sentences. Therefore, we can add the assertion

$$\textit{if } P\theta \textit{ then } F\theta[r\theta \leftarrow s\theta]$$

to our sequent. Here, the notation $F\theta[r\theta \leftarrow s\theta]$ indicates that every occurrence of $r\theta$ in $F\theta$ is to be replaced by $s\theta$.

For example, suppose we have the assertion

$$a \in l \textit{ and } a \neq 0$$

and we apply the transformation rule

$$u \in v \Rightarrow u = \textit{head}(v) \textit{ or } u \in \textit{tail}(v) \quad \textit{if islist}(v) \textit{ and } v \neq [\,],$$

taking r' to be $a \in l$ and θ to be the substitution $[u \leftarrow a;\ v \leftarrow l]$; then we obtain the new assertion

$$\begin{array}{l} \textit{if islist}(l) \textit{ and } l \neq [\,] \\ \textit{then } (a = \textit{head}(l) \textit{ or } a \in \textit{tail}(l)) \textit{ and } a \neq 0. \end{array}$$

Note that a and l are constants, while u and v are variables, and indeed, the substitution was made for the variables of the rule but not for the constants of the assertion.

In general, if the given assertion F has an associated output entry t, the new output entry is formed by applying the substitution θ to t. For, suppose some instance of the new assertion "*if* $P\theta$ *then* $F\theta[r\theta \leftarrow s\theta]$" is false; then the corresponding instance of $P\theta$ is true, and the corresponding instance of $F\theta[r\theta \leftarrow s\theta]$ is false. Then, by the transformation rule, the instances of $r\theta$ and $s\theta$ are equal; hence the corresponding instance of $F\theta$ is false. We know that if any instance of F is false, the corresponding instance of t satisfies the given specification. Hence, because some instance of $F\theta$ is false, the corresponding instance of $t\theta$ is the desired output.

In our deduction rule notation, we write

assertions	*goals*	*outputs*
F		t
if $P\theta$ *then* $F\theta[r\theta \leftarrow s\theta]$		$t\theta$

The corresponding dual deduction rule for goals is

assertions	*goals*	*outputs*
	F	t
	$P\theta$ *and* $F\theta[r\theta \leftarrow s\theta]$	$t\theta$

For example, suppose we have the goal

	$a \mid z$ *and* $b \mid z$	$z + 1$

and we apply the transformation rule

$$u \mid 0 \Rightarrow \textit{true} \quad \textit{if integer}(u) \textit{ and } u \neq 0,$$

taking r' to be $a \mid z$ and θ to be the substitution $[z \leftarrow 0;\ u \leftarrow a]$. Then we obtain the goal

	($\textit{integer}(a)$ *and* $a \neq 0$) *and* (*true and* $b \mid 0$)	$0 + 1$

which can be further transformed to

	$\textit{integer}(a)$ *and* $a \neq 0$ *and* $b \mid 0$	1

Note that applying the transformation rule caused a substitution to be made for the occurrences of the variable z in the goal and the output entry.

Transformation rules can also be applied to output entries in an analogous manner.

Transformation rules need not be simple rewriting rules; they may represent arbitrary procedures. For example, r could be an equation $f(x) = a$, s could be its solution $x = e$, and P could be the condition under which that solution applies. Another example: the skolemization procedure for removing quantifiers can be represented as a transformation rule. In fact, decision methods for particular subtheories may also be represented as transformation rules (see, e.g., Bledsoe [1] or Nelson and Oppen [9]).

Transformation rules play the role of the "antecedent theorems" and "consequent theorems" of PLANNER (Hewitt [6]). For example, a consequent theorem that we might write as

$$\begin{array}{l} \textit{to prove } f(u) = f(v) \\ \quad \textit{prove } u = v \end{array}$$

can be represented by the transformation rule

$$f(u) = f(v) \Rightarrow \textit{true} \quad \textit{if } u = v.$$

This rule will have the desired effect of reducing the goal $f(a) = f(b)$ to the simpler subgoal $a = b$, and (like the consequent theorem) will not have the pernicious side effect of deriving from the simple assertion $a = b$ the more complex assertion $f(a) = f(b)$. The axiomatic representation of the same fact would have both results. (Incidentally, the transformation rule has the beneficial effect, not shared by the consequent theorem, of deriving from the complex assertion $\textit{not}(f(a) = f(b))$ the simpler assertion $\textit{not}(a = b)$.)

RESOLUTION

The original resolution principle (Robinson [12]) required that sentences be put into conjunctive normal form. As a result, the set of clauses sometimes exploded to an unmanageable size and the proofs lost their intuitive content. The version of resolution we employ does not require the sentences to be in conjunctive normal form.

Assume our sequent contains two assertions F and G, containing subsentences P_1 and P_2, respectively, that are not within the scope of any quantifier. For the time being, let us ignore the output expressions corresponding to these assertions. Suppose there exists a unifier for P_1 and P_2, i.e., a substitution θ such that $P_1\theta$ and $P_2\theta$ are identical. We can take θ to be the most general unifier. The *AA-resolution rule* allows us to deduce the new assertion

$$F\theta[P_1\theta \leftarrow \textit{true}] \textit{ or } G\theta[P_2\theta \leftarrow \textit{false}]$$

and add it to the sequent. Recall that the notation $F\theta[P_1\theta \leftarrow \textit{true}]$ indicates that every instance of the subsentence $P_1\theta$ in $F\theta$ is to be replaced by *true*. (Of course, we may need to do the usual renaming to ensure that F and G have no variables in common.) We will call θ the *unifying substitution* and $P_1\theta(=P_2\theta)$ the *eliminated subexpression*; the deduced assertion is called the *resolvent*. Note that the rule is symmetric, so the roles of F and G may be reversed.

For example, suppose our sequent contains the assertions

$$\textit{if } (P(x) \textit{ and } Q(b)) \textit{ then } R(x)$$

and

$$P(a) \textit{ and } Q(y).$$

The two subsentences "$P(x)$ *and* $Q(b)$" and "$P(a)$ *and* $Q(y)$" can be unified by the substitution

$$\theta = [x \leftarrow a; y \leftarrow b].$$

Therefore, the AA-resolution rule allows us to eliminate the subexpression "*P(a) and Q(b)*" and derive the conclusion

$$(\textit{if true then } R(a)) \textit{ or false},$$

which reduces to

$$R(a)$$

by application of the appropriate transformation rules.

The conventional resolution rule may be regarded as a special case of the above AA-resolution rule. The conventional rule allows us to derive from the two assertions

$$(\textit{not } P_1) \textit{ or } Q$$

and

$$P_2 \textit{ or } R$$

the new assertion

$$Q\theta \textit{ or } R\theta,$$

where θ is a most general unifier of P_1 and P_2. From the same two assertions we can use our AA-resolution rule to derive

$$(((\textit{not } P_1 \textit{ or } Q)\theta)[P_1\theta \leftarrow \textit{true}] \textit{ or } (((P_2 \textit{ or } R)\theta)[P_2\theta \leftarrow \textit{false}]$$

i.e.,

$$((\textit{not true}) \textit{ or } Q\theta) \textit{ or } (\textit{false or } R\theta),$$

which reduces to the same conclusion

$$Q\theta \textit{ or } R\theta$$

as the original resolution rule.

The justification for the AA-resolution rule is straightforward: Because F holds, if $P_1\theta$ is true, then $F\theta[P_1\theta \leftarrow \textit{true}]$ holds; on the other hand, because G holds, if $P_1\theta(=P_2\theta)$ is false, $G\theta[P_2\theta \leftarrow \textit{false}]$ holds. In either case, the disjunction

$$F\theta[P_1\theta \leftarrow \textit{true}] \textit{ or } G\theta[P_2\theta \leftarrow \textit{false}]$$

holds.

A "nonclausal" resolution rule similar to ours has been developed by Murray [8]. Other such rules have been proposed by Wilkins [14] and Nilsson [10].

THE RESOLUTION RULES

We have defined the AA-resolution rule to derive conclusions from assertions.

The *AA-resolution rule*

assertions	*goals*
F G	
$F\theta[P_1\theta \leftarrow \textit{true}]$ *or* $G\theta[P_2\theta \leftarrow \textit{false}]$	

where $P_1\theta = P_2\theta$, and θ is most general.

By duality, we can regard goals as negated assertions; consequently, the following three rules are corollaries of the AA-resolution rule.

The *GG-resolution rule*

assertions	*goals*
	F G
	$F\theta[P_1\theta \leftarrow true]$ *and* $G\theta[P_2\theta \leftarrow false]$

The *GA-resolution rule*

assertions	*goals*
G	F
	$F\theta[P_1\theta \leftarrow true]$ *and* *not* $(G\theta[P_2\theta \leftarrow false])$

The *AG-resolution rule*

assertions	*goals*
F	G
	not$(F\theta[P_1\theta \leftarrow true])$ *and* $G\theta[P_2\theta \leftarrow false]$

where P_1, P_2, and θ satisfy the same condition as for the AA-resolution rule.

Up to now, we have ignored the output expressions of the assertions and goals. However, if at least one of the sentences to which a resolution rule is applied has a corresponding output expression, the resolvent will also have an output expression. If only one of the sentences has an output expression, say t, then the resolvent will have the output expression $t\theta$. On the other hand, if the two sentences F and G have output expressions t_1 and t_2, respectively, the resolvent will have the output expression

$$\textit{if } P_1\theta \textit{ then } t_1\theta \textit{ else } t_2\theta.$$

(Of course, if $t_1\theta$ and $t_2\theta$ are identical, no conditional expression need be formed; the output expression is simply $t_1\theta$.)

The justification for constructing this conditional as an output expression is as follows. We consider only the GG case: Suppose that the goal

$$F\theta[P_1\theta \leftarrow true] \textit{ and } G\theta[P_2\theta \leftarrow false]$$

has been obtained by GG-resolution from two goals F and G. We would like to show that if the goal is true, the conditional output expression satisfies the desired specification. We assume that the resolvent is true; therefore both $F\theta[P_1\theta \leftarrow true]$ and $G\theta[P_2\theta \leftarrow false]$ are true. In the case that $P_1\theta$ is true, we have that $F\theta$ is also true. Consequently, the corresponding instance $t_1\theta$ of the output expression t_1 satisfies the specification of the desired program. In the other case, in which $P_1\theta$ is false, $P_2\theta$ is false, and the same reasoning allows us to conclude that $t_2\theta$ satisfies the specification of the desired program. In either case we can conclude that the conditional

$$\mathit{if}\ P_1\theta\ \mathit{then}\ t_1\theta\ \mathit{else}\ t_2\theta$$

satisfies the desired specification. By duality, the same output expression can be derived for the AA-resolution, GA-resolution, and AG-resolution.

For example, let $u \cdot v$ denote the operation of inserting u before the first element of the list v, and suppose we have the goal

assertions	*goals*	*outputs* $f(a, b)$
	$head(z) = a$ *and* $tail(z) = b$	z

and we have the assertion

$head(u \cdot v) = u$		

with no output expression; then by GA-resolution, applying the substitution

$$\theta = [u \leftarrow a;\ z \leftarrow a \cdot v]$$

and eliminating the subsentence

$$head(a \cdot v) = a,$$

we obtain the new goal

	(*true and* $tail(a \cdot v) = b$) *and* (*not false*)	$a \cdot v$

which can be reduced to

	$tail(a \cdot v) = b$	$a \cdot v$

by application of the appropriate transformation rules. Note that we have applied the substitution$[u \leftarrow a;\ z \leftarrow a \cdot v]$ to the original output expression z, obtaining the new output expression $a \cdot v$. Therefore, if we can find v such that $tail(a \cdot v) = b$, the corresponding instance of $a \cdot v$ will satisfy the desired specification.

Another example: Suppose we have derived the two goals

assertions	*goals*	*outputs* $max(l)$
	$max(tail(l)) \geq head(l)$ *and* $tail(l) \neq []$	$max(tail(l))$
	$not(max(tail(l)) \geq head(l))$ *and* $tail(l) \neq []$	$head(l)$

Then by GG-resolution, eliminating the subsentence $max(tail(l)) \geq head(l)$, we can derive the new goal

	(*true and* $tail(l) \neq []$) *and* (*not false*) *and* $tail(l) \neq []$)	*if* $max(tail(l)) \geq head(l)$ *then* $max(tail(l))$ *else* $head(l)$

which can be reduced to

	$tail(l) \neq []$	*if* $max(tail(l)) \geq head(l)$ *then* $max(tail(l))$ *else* $head(l)$

THE POLARITY STRATEGY

Not all applications of the resolution rules will produce valuable conclusions. For example, suppose we are given the goal

assertions	*goals*	*outputs*
	$P(c, x)$ *and* $Q(x, a)$	

and the assertion

if $P(y, d)$ *then* $Q(b, y)$		

Then if we apply GA-resolution, eliminating $Q(b, a)$, we can obtain the resolvent

$$(P(c, b) \textit{ and true}) \textit{ and not}(\textit{if } P(a, d) \textit{ then false}),$$

which reduces to the goal

	$P(c, b)$ *and* $P(a, d)$	

However, we can also apply GA-resolution and eliminate $P(c, d)$, yielding the resolvent

$$(\textit{true and } Q(d, a)) \textit{ and not}(\textit{if false then } Q(b, c)),$$

which reduces to the trivial goal

	false	

Finally, we can also apply AG-resolution to the same assertion and goal in two different ways, eliminating $P(c, d)$ and eliminating $Q(b, a)$; both of these applications lead to the same trivial goal *false*.

A *polarity strategy* adapted from Murray [8] restricts the resolution rules to prevent many such fruitless applications. We first assign a *polarity* (either positive or negative) to every subsentence of a given sequent as follows:

(1) each goal is positive;
(2) each assertion is negative;
(3) if a subsentence S has form "*not* α," then its component α has polarity opposite to S;
(4) if a subsentence S has form "α *and* β," "α *or* β," "*for all* x, α," or "*for some* x, β," then its components α and β have the same polarity as S;
(5) if a subsentence S has form "*if* α *then* β," then β has the same polarity as S, but α has the opposite polarity.

For example, the above goal and assertion are annotated with the polarity of each subsentence, as follows:

assertions	*goals*	*outputs*
$(if\ P(y, d)^+\ then\ Q(b, y)^-)^-$		
	$(P(c, x)^+\ and\ Q(x, a)^+)^+$	

The four resolution rules we have presented replace certain subsentences by *true*, and others by *false*. The polarity strategy, then, permits a subsentence to be replaced by *true* only if it has at least one positive occurrence, and by *false* only if it has at least one negative occurrence. For example, we are permitted to apply GA-resolution to the above goal and assertion, eliminating $Q(b, a)$ because $Q(x, a)$, which is replaced by *true*, occurs positively in the goal, and $Q(b, y)$, which is replaced by *false*, occurs negatively in the assertion. On the other hand, we are not permitted to apply GA-resolution to eliminate $P(c, d)$, because $P(y, d)$, which is replaced by *false*, only occurs positively in the assertion. Similarly, we are not permitted to apply AG-resolution between this assertion and goal, whether we eliminate $P(c, d)$ or $Q(b, a)$. Indeed, the only application of resolution permitted by the polarity strategy is the one that led to a nontrivial conclusion.

The deductive system we have presented so far, including the splitting rules, the resolution rules, and an appropriate set of logical transformation rules, has been proved by Murray to constitute a complete system for first-order logic, in the sense that a derivation exists for every valid sentence. (Actually, only the resolution rules and some of the logical transformation rules are strictly necessary.) The above polarity strategy does not interfere with the completeness of the system.

MATHEMATICAL INDUCTION AND THE FORMATION OF RECURSIVE CALLS

Mathematical induction is of special importance for deductive systems intended for program synthesis because it is only by the application of some form of the induction principle that recursive calls or iterative loops are introduced into the program being constructed. The induction rule we employ is a version of the principle of mathematical induction over a well-founded set, known in the computer science literature as "structural induction."

We may describe this principle as follows: In attempting to prove that a sentence of form $F(a)$ holds for an arbitrary element a of some well-founded set, we may assume inductively that the sentence holds for all u that are strictly less than a in the well-founded ordering $<_w$. Thus, in trying to prove $F(a)$, the well-founded induction principle allows us to assume the induction hypothesis

$$for\ all\ u,\ if\ u <_w a\ then\ F(u).$$

In the case that the well-founded set is the nonnegative integers under the usual $<$ ordering, well-founded induction reduces to the familiar complete induction principle: To prove that $F(n)$ holds for an arbitrary nonnegative integer n, we may assume inductively that the sentence $F(u)$ holds for all nonnegative integers u such that $u < n$.

In our inference system, the principle of well-founded induction is represented as a deduction rule (rather than, say, an axiom schema). We present only a special case of this rule here.

Suppose we are constructing a program whose specification is of form

$$f(a) \Leftarrow \textit{find } z \textit{ such that } R(a, z)$$
$$\textit{where } P(a).$$

Our initial sequent is thus

assertions	*goals*	*outputs* $f(a)$
$P(a)$		
	$R(a, z)$	z

Then we can always add to our sequent a new assertion, the induction hypothesis

if $u <_w a$ *then if* $P(u)$ *then* $R(u, f(u))$		

Here, f denotes the program we are trying to construct. The well-founded set and the particular well-founded $<_w$ to be employed in the proof have not yet been determined. If the induction hypothesis is used more than once in the proof, it always refers to the same well-founded ordering $<_w$.

Let us paraphrase: We are attempting to construct a program f such that for an arbitrary input a satisfying the input condition $P(a)$, the output $f(a)$ will satisfy the output condition $R(a, f(a))$. By the well-founded induction principle, we can assume inductively that for every u less than a (in some well-founded ordering) such that the input condition $P(u)$ holds, the output $f(u)$ will satisfy the same output condition $R(u, f(u))$. By employing the induction hypothesis in the proof, recursive calls to f can be introduced into the output expression for $f(a)$.

As we shall see in a later section, we can introduce an induction hypothesis corresponding to any subset of the assertions or goals in our sequent, not just the initial assertion and goal; most of these induction hypotheses are not relevant to the final proof, and the proliferation of new assertions obstructs our efforts to find a proof. Therefore, we employ the following *recurrence strategy* for determining when to introduce an induction hypothesis.

Let us restrict our attention to the case where the induction hypothesis is formed from the initial sequent. Suppose that at some point in the derivation a goal is developed of form

	$R(s, z')$	$t(z')$

where s is an arbitrary term. In other words, the new goal is a precise instance of the initial goal $R(a, z)$ obtained by replacing a by s. This recurrence motivates us to add the induction hypothesis

if $u <_w a$ *then if* $P(u)$ *then* $R(u, f(u))$		

The rationale for introducing the induction hypothesis at this point is that now we can perform GA-resolution between the newly developed goal $R(s, z')$ and the induction hypothesis. The resulting goal is then

	true and
	not if $s \prec_w a$
	    then if $P(s)$
	        then false
		$t(f(s))$

This simplifies (by the application of logical transformation rules) to

	$s \prec_w a$ *and* $P(s)$	$t(f(s))$

Note that a recursive call $f(s)$ has been introduced into the output expression for $f(a)$. By proving the expression $s \prec_w a$, we ensure that this recursive call will terminate; by proving the expression $P(s)$, we guarantee that the argument s of the recursive call will satisfy the input condition of the program f.

The particular well-founded ordering $\prec_w$ to be employed by the proof has not yet been determined. We assume the existence of transformation rules of form

$$u \prec_{w_1} v \Rightarrow true \quad \textit{if } Q(u, v)$$

capable of choosing or combining well-founded orderings applicable to the particular theories under consideration (e.g., numbers, lists, and sets).

Let us look at an example. Suppose we are constructing two programs $div(i, j)$ and $rem(i, j)$ to compute the quotient and remainder, respectively, of dividing a nonnegative integer i by a positive integer j; the specification may be expressed as

$$\begin{aligned}(div(i,j), rem(i,j)) \Leftarrow\ & \textit{find } (y, z) \textit{ such that}\\ & \quad i = y \cdot j + z \textit{ and } 0 \le z \textit{ and } z < j\\ \textit{where } & 0 \le i \textit{ and } 0 < j.\end{aligned}$$

(Note that, for simplicity, we have omitted type requirements such as *integer*(i).) Our initial sequent is then

assertions	*goals*	*outputs* $div(i, j)$	$rem(i, j)$
$0 \le i$ *and* $0 < j$	$i = y \cdot j + z$ *and* $0 \le z$ *and* $z < j$	y	z

Here, the inputs i and j are constants, for which we can make no substitution; y and the output z are variables.

Assume that during the course of the derivation we develop the goal

	$i - j = y_1 \cdot j + z$ *and* $0 \le z$ *and* $z < j$	$y_1 + 1$	z

This goal is a precise instance of the initial goal

$$i = y \cdot j + z \textit{ and } 0 \le z \textit{ and } z < j$$

obtained by replacing i by $i-j$. Therefore, we add as a new assertion the induction hypothesis.

if $(u_1, u_2) \prec_w (i, j)$ *then if* $0 \leq u_1$ *and* $0 < u_2$ *then* $u_1 = div(u_1, u_2) \cdot u_2 + rem(u_1, u_2)$ *and* $0 \leq rem(u_1, u_2)$ *and* $rem(u_1, u_2) < u_2$			

Here, $\prec_w$ is an arbitrary well-founded ordering, defined on pairs because the desired program f has a pair of inputs.

We can now apply GA-resolution between the goal

	$i-j = y_1 \cdot j + z$ *and* $0 \leq z$ *and* $z < j$	y_1+1	z

and the induction hypothesis; the unifying substitution θ is

$$[u_1 \leftarrow i-j;\ u_2 \leftarrow j;\ y_1 \leftarrow div(i-j, j);\ z \leftarrow rem(i-j, j)].$$

The new goal is

	true and *not* (*if* $(i-j, j) \prec_w (i, j)$ *then if* $0 \leq i-j$ *and* $0 < j$ *then false*)	$div(i-j, j)+1$	$rem(i-j, j)$

which reduces to

	$(i-j, j) \prec_w (i, j)$ *and* $0 \leq i-j$ *and* $0 < j$	$div(i-j, j)+1$	$rem(i-j, j)$

Note that the recursive calls $div(i-j, j)$ and $rem(i-j, j)$ have been introduced into the output entry.

The particular well-founded ordering $\prec_w$ to be employed in the proof has not yet been determined. It can be chosen to be the $<$ ordering on the first component of the pairs, by application of the transformation rule

$$(u_1, u_2) \prec_{N1} (v_1, v_2) \Rightarrow \textit{true} \quad \textit{if } u_1 < v_1 \textit{ and } 0 \leq u_1 \textit{ and } 0 \leq v_1.$$

A new goal

	$i-j < i$ *and* $0 \leq i-j$ *and* $0 \leq i$ *and true* *and* $0 \leq i-j$ *and* $0 < j$	$div(i-j, j)+1$	$rem(i-j, j)$

is produced; this goal ultimately reduces to

	$j \leq i$	$div(i-j, j)+1$	$rem(i-j, j)$

In other words, in the case that $j \leq i$, the outputs $div(i-j, j)+1$ and $rem(i-j, j)$ satisfy the desired program's specification. In the next section, we give the full derivation of these programs.

In our presentation of the induction rule, several limitations were imposed for simplicity but are not actually essential:

(1) In the example we considered, the only skolem functions in the initial

sequent are the constants corresponding to the program's inputs, and the only variables are those corresponding to the program's outputs; the sequent was of form

assertions	*goals*	*outputs* $f(a)$
$P(a)$		
	$R(a, z)$	z

In forming the induction hypothesis, the skolem constant a is replaced by a variable u and the variable z is replaced by the term $f(u)$; the induction hypothesis was of form

if $u <_w a$ *then if* $P(u)$ *then* $R(u, f(u))$		

However, if there are other skolem functions in the initial sequent, they too must be replaced by variables in the induction hypothesis; if there are other variables in the initial sequent, they must be replaced by new skolem functions. For example, suppose the initial sequent is of form

$$\begin{aligned} f(a) \Leftarrow\ & \textit{find } z \textit{ such that} \\ & \quad \textit{for all } x_1, \\ & \quad \textit{for some } x_2, \\ & \quad R(a, z, x_1, x_2) \\ & \textit{where } P(a). \end{aligned}$$

Then the initial sequent is of form

assertions	*goals*	*outputs* $f(a)$
$P(a)$		
	$R(a, z, g_1(z), x_2)$	z

where $g_1(z)$ is the skolem function corresponding to x_1. The induction hypothesis is then of form

if $u <_w a$ *then if* $P(u)$ *then* $R(u, f(u), v, g_2(u, v))$		

Here, the skolem function $g_1(z)$ has been replaced by the variable v, and the variable x_2 has been replaced by a new skolem function $g_2(u, v)$.

(2) One limitation to the recurrence strategy was that the induction hypothesis was introduced only when an entire goal is an instance of the initial goal. In fact, the strategy can be extended so that the hypothesis is introduced when some subsentence of a goal is an instance of some subsentence of the initial goal, because the resolution rule can then be applied between the goal and the induction hypothesis. This extension is straightforward.

(3) A final observation: The induction hypothesis was always formed directly from the initial sequent; thus, the theorem itself was proved by induction. In later sections we extend the rule so that induction can be applied to lemmas that are stronger or more general than the theorem itself. This extension also accounts for the formation of auxiliary procedures in the program being constructed.

Some early efforts toward incorporating mathematical induction in a resolution framework were made by Darlington [4]. His system treated the induction principle as a second-order axiom schema rather than as a deduction rule; it had a limited ability to perform second-order unifications.

A COMPLETE EXAMPLE: FINDING THE QUOTIENT OF TWO INTEGERS

In this section, we present a complete example that exploits most of the features of the deductive synthesis approach. Our task is to construct programs $div(i, j)$ and $rem(i, j)$ for finding the integer quotient of dividing a nonnegative integer i by a positive integer j. Portions of this synthesis have been used to illustrate the induction principle in the previous section.

Our specification is expressed as

$$\begin{aligned} (div(i,j), rem(i,j)) \Leftarrow\ & \textit{find } (y, z) \textit{ such that} \\ & i = y \cdot j + z \textit{ and } 0 \le z \textit{ and } z < j \\ \textit{where } 0 \le i \textit{ and } 0 < j. & \end{aligned}$$

(For simplicity, we again omit type conditions, such as *integer*(i), from this discussion.) Our initial sequent is therefore

assertions	*goals*	*outputs* $div(i,j)$	*outputs* $rem(i,j)$
1. $0 \le i$ *and* $0 < j$	2. $i = y \cdot j + z$ *and* $0 \le z$ *and* $z < j$	y	z

(Note that we are enumerating the assertions and goals.)

In presenting the derivation we sometimes apply simple logical and algebraic transformation rules without mentioning them explicitly. We assume that our background knowledge includes the two assertions

3. $u = u$ 4. $u \le v$ *or* $v < u$			

Applying the *andsplit rule* to assertion 1 yields the new assertions

5. $0 \le i$ 6. $0 < j$			

Assume we have the following transformation rules that define integer multiplication:

$$0 \cdot v \Rightarrow 0$$

$$(u + 1) \cdot v \Rightarrow u \cdot v + v.$$

Applying the first of these rules to the subexpression $y \cdot j$ in goal 2 yields

	7. $i = 0 + z$ *and* $0 \leq z$ *and* $z < j$	0	z

The unifying substitution in deriving goal 7 is

$$\theta = [y \leftarrow 0;\ v \leftarrow j];$$

applying this substitution to the output entry y produced the new output 0.

Applying the numerical transformation rule

$$0 + v \Rightarrow v$$

yields

	8. $i = z$ *and* $0 \leq z$ *and* $z < j$	0	z

The GA-resolution rule can now be applied between goal 8 and the equality assertion 3, $u = u$. The unifying substitution is

$$\theta = [u \leftarrow i;\ z \leftarrow i]$$

and the eliminated subexpression is $i = i$; we obtain

	9. $0 \leq i$ *and* $i < j$	0	i

By applying GA-resolution again, against assertion 5, $0 \leq i$, we obtain

	10. $i < j$	0	i

In other words, we have found that in the case that $i < j$, the output 0 will satisfy the specification for the quotient program and the output i will satisfy the specification for the remainder program.

Let us return our attention to the initial goal 2,

$$i = y \cdot j + z \textit{ and } 0 \leq z \textit{ and } z < j.$$

Recall that we have a second transformation rule

$$(u + 1) \cdot v \Rightarrow u \cdot v + v$$

for the multiplication function. Applying this rule to goal 2 yields

	11. $i = y_1 \cdot j + j + z$ *and* $0 \leq z$ *and* $z < j$	$y_1 + 1$	z

where y_1 is a new variable. Here, the unifying substitution is

$$\theta = [y \leftarrow y_1{+}1;\ u \leftarrow y_1;\ v \leftarrow j];$$

applying this substitution to the output entry y produced the new output $y_1{+}1$ in the *div* program.

The transformation rule

$$u = v + w \Rightarrow u - v = w$$

applied to goal 11 yields

	12. $i-j = y_1 \cdot j + z$ *and* $0 \le z$ *and* $z < j$	$y_1 + 1$	z

Goal 12 is a precise instance of the initial goal 2,

$$i = y \cdot j + z \textit{ and } 0 \le z \textit{ and } z < j,$$

obtained by replacing the input i by $i-j$. (Again, the replacement of the dummy variable y by y_1 is not significant.) Therefore, the following induction hypothesis is formed:

13. *if* $(u_1, u_2) \prec_w (i, j)$ *then if* $0 \le u_1$ *and* $0 < u_2$ *then* $u_1 = div(u_1, u_2) \cdot u_2 + rem(u_1, u_2)$ *and* $0 \le rem(u_1, u_2)$ *and* $rem(u_1, u_2) < u_2$			

Here, $\prec_w$ is an arbitrary well-founded ordering.

By applying GA-resolution between goal 12 and the induction hypothesis, we obtain the goal

	14. *true and not* (*if* $(i-j, j) \prec_w (i, j)$ *then if* $0 \le i-j$ *and* $0 < j$ *then false*)	$div(i-j, j) + 1$	$rem(i-j, j)$

Here, the unifying substitution is

$$\theta = [u_1 \leftarrow i-j;\ u_2 \leftarrow j;\ y_1 \leftarrow div(i-j, j);\ z \leftarrow rem(i-j, j)]$$

and the eliminated subexpression is

$$i-j = div(i-j, j) \cdot j + rem(i-j, j) \textit{ and } 0 \le rem(i-j, j) \textit{ and } rem(i-j, j) < j.$$

Note that the substitution to the variable y_1 has caused the output entry $y_1 + 1$ to be changed to $div(i-j, j) + 1$ and the output entry z to be replaced by $rem(i-j, j)$. The use of the induction hypothesis has introduced the recursive calls $div(i-j, j)$ and $rem(i-j, j)$ into the output.

Goal 14 reduces to

	15. $(i-j, j) \prec_w (i, j)$ *and* $0 \le i-j$ *and* $0 < j$	$div(i-j, j) + 1$	$rem(i-j, j)$

The particular ordering $\prec_w$ has not yet been determined; however, it is chosen to be the $<$ ordering on the first component of the pairs, by application of the transformation rule

$$(u_1, u_2) \prec_{N1} (v_1, v_2) \Rightarrow \textit{true} \quad \textit{if } u_1 < v_1 \textit{ and } 0 \le u_1 \textit{ and } 0 \le v_1.$$

A new goal is produced:

	16. $i-j < i$ *and* $0 \le i-j$ *and* $0 \le i$ *and* $0 \le i-j$ *and* $0 < j$	$div(i-j, j) + 1$	$rem(i-j, j)$

Note that the conditions of the transformation rule caused new conjuncts to be added to the goal.

By application of algebraic and logical transformation rules, and GA-resolution with the assertion 5, $0 \leq i$, and assertion 6, $0 < j$, goal 16 is reduced to

	17. $j \leq i$	$div(i-j, j) + 1$	$rem(i-j, j)$

In other words, we have learned that in the case that $j \leq i$, the outputs $div(i-j, j) + 1$ and $rem(i-j, j)$ satisfy the specification of the *div* program. On the other hand, in deriving goal 10 we learned that in the case that $i < j$, 0 and i are satisfactory outputs. Assuming we have the assertion 4

$$u \leq v \textit{ or } v < u,$$

we can obtain the goal

	18. $not(i < j)$	$div(i-j, j) + 1$	$rem(i-j, j)$

by GA-resolution.

The final goal

	19. *true*	*if* $i < j$ *then* 0 *else* $div(i-j, j) + 1$	*if* $i < j$ *then* i *else* $rem(i-j, j)$

can then be obtained by GG-resolution between goals 10 and 18. The conditional expressions have been formed because both goals have a corresponding output entry. Because we have developed the goal *true* and a corresponding primitive output entry, the derivation is complete. The final programs

```
div(i, j) ⇐ if i < j
              then 0
              else div(i-j, j) + 1
```

and

```
rem(i, j) ⇐ if i < j
              then i
              else rem(i-j, j)
```

are obtained directly from the final output entries.

THE FORMATION OF AUXILIARY PROCEDURES

We have remarked that mathematical induction need not be restricted to apply only to the initial assertion and goal but may legitimately be applied to any subset of the assertions and goals in the sequent. In fact, when induction is applied in this more general setting, *auxiliary procedures* may be introduced into the program being constructed. For example, in constructing a program *sort* to order a list, we might introduce an auxiliary procedure *merge* to insert a number in its place in an ordered list of numbers. In this section we develop the extended form of the induction principle that accounts for the formation of auxiliary procedures. We begin with a description of the recurrence strategy that applies to this extended induction.

Assume that we are in the process of constructing a program $f(a)$ whose specification is of form

$$f(a) \Leftarrow \textit{find } z \textit{ such that } R(a, z)$$
$$\textit{where } P(a).$$

Then our initial sequent is of form

assertions	*goals*	*outputs* $f(a)$
$P(a)$	$R(a, z)$	z

Let goal A be any goal obtained during the derivation of $f(a)$, and assume that goal A is of form

A:		$R'(a, z')$	$t'(z')$

Suppose that by applying deduction rules successively to goal A and to the assertions $P'_1(a), P'_2(a), \ldots, P'_k(a)$ of the sequent, we obtain a goal B of form

B:		$R'(s, z'')$	$t''(z'')$

where s is an arbitrary term. (For simplicity, we assume that no goals are required other than those derived from goal A, and that none of the k required assertions have associated output entries.)

In summation, we have developed a new goal (goal B) that is a precise instance of the earlier goal (goal A), obtained by replacing the input a by the term s. This recurrence motivates us to define an auxiliary procedure $\mathit{fnew}(a)$ whose output condition is goal A; we then hope to achieve goal B by a recursive call to the new procedure.

Let us be more precise. The specification for $\mathit{fnew}(a')$ is

$$\mathit{fnew}(a') \Leftarrow \textit{find } z' \textit{ such that } R'(a', z')$$
$$\textit{where } P'(a').$$

Here, the input condition $P'(a')$ is $P'_1(a')$ *and* $P'_2(a')$ *and* $\cdots$ *and* $P'_k(a')$. If we succeed in constructing a program that meets this specification, we can employ it as an auxiliary procedure of the main program $f(a)$.

Consequently, at this point we add a new output column for $\mathit{fnew}(a')$ to the sequent, and we introduce the new rows

	assertions	*goals*	*outputs* $f(a)$	$\mathit{fnew}(a')$
	$P'(a')$			
A':		$R'(a', z')$	$t'(\mathit{fnew}(a))$	z'

Note that in these rows we have replaced the input constant a by a new constant a'. This step is logically necessary; adding the induction hypothesis without renaming the constant can lead to false results. The second row (goal A') indicates that if we succeed in constructing $\mathit{fnew}(a')$ to satisfy the above specification, then $f(a)$ may be computed by a call $t'(\mathit{fnew}(a))$ to the new procedure.

By introducing the procedure *fnew*(a') we are able to call it recursively. In other words, we are now able to form an induction hypothesis from the assertion $P'(a')$ and the goal $R'(a', z')$, namely,

if $u' \prec_{w'} a'$ *then if* $P'(u')$ *then* $R'(u', fnew(u'))$			

If this assertion is employed during a proof, a recursive call to *fnew* can be introduced into the output column for *fnew*(a'). The well-founded ordering $\prec_{w'}$, corresponding to *fnew*(a') may be distinct from the ordering $\prec_w$ corresponding to $f(a)$.

Note that we do not begin a new sequent for the derivation of the auxiliary procedure *fnew*; the synthesis of the main program $f(a)$ and the auxiliary procedure *fnew*(a') are both conducted by applying derivation rules to the same sequent. Those rows with output entries for *fnew*(a') always have the expression $t'(fnew(a))$ as the output entry for $f(a)$.

Suppose we ultimately succeed in obtaining the goal *true* with primitive output entries t and t':

		outputs	
assertions	*goals*	$f(a)$	*fnew*(a')
	true	t	t'

Then the final program is

$$f(a) \Leftarrow t$$

and

$$fnew(a') \Leftarrow t'.$$

Note that although the portion of the derivation leading from goal A to goal B serves to motivate the formation of the auxiliary procedure, it may actually have no part in the derivation of the final program; its role has been taken over by the derivation of goal B′ from goal A′.

It is possible to introduce many auxiliary procedures for the same main program, each adding a new output column to the sequent. An auxiliary procedure may have its own set of auxiliary procedures. An auxiliary procedure may call the main program or any of the other procedures; in other words, the system of procedures can be "mutually recursive."

If we fail to complete the derivation of an auxiliary procedure *fnew*(a'), we may still succeed in finding some other way of completing the derivation of $f(a)$ without using *fnew*, by applying deduction rules to rows that have no output entry for *fnew*(a').

To illustrate the formation of auxiliary procedures, we consider the synthesis of a program *cart*(s, t) to compute the cartesian product of two (finite) sets s and t, i.e., the set of all pairs whose first component belongs to s and whose second component belongs to t. The specification for this program is

$$\begin{aligned}&\mathit{cart}(s, t) \Leftarrow \mathit{find}\ z\ \mathit{such\ that}\\ &\qquad\qquad z = \{(a, b)\colon a \in s\ \mathit{and}\ b \in t\}.\end{aligned}$$

The initial sequent is then

	assertions	*goals*	*outputs* $\mathit{cart}(s, t)$
		$z = \{(a, b)\colon a \in s\ \mathit{and}\ b \in t\}$	z

(Note that this specification has no input condition, except for the type condition *isset*(s) and *isset*(t), which we omit for simplicity.)

We denote the empty set by $\{\}$. If u is a nonempty set, then *choice*(u) denotes some particular element of u, and *rest*(u) denotes the set of all other elements. We assume that the transformation rules concerning finite sets include:

$$u \in v \Rightarrow \mathit{false} \quad \mathit{if}\ v = \{\}$$

$$u \in v \Rightarrow u = \mathit{choice}(v)\ \mathit{or}\ u \in \mathit{rest}(v) \quad \mathit{if}\ v \neq \{\}$$

$$\{u\colon \mathit{false}\} \Rightarrow \{\}$$

$$\{u\colon P\ \mathit{or}\ Q\} \Rightarrow \{u\colon P\} \cup \{u\colon Q\}$$

$$\mathit{rest}(u) <_{s_1} u \Rightarrow \mathit{true} \quad \mathit{if}\ u \neq \{\}$$

$$\{u\colon u = v\} \Rightarrow \{v\} \quad \text{(where u does not occur in v)}$$

We will not reproduce the complete derivation, but only those portions that concern the formation of auxiliary procedures.

By application of deduction rules to the initial sequent, we obtain the goal

A:

	$z' = \{(a, b)\colon a = \mathit{choice}(s)\ \mathit{and}\ b \in t\}$	$\mathit{if}\ s = \{\}$ $\mathit{then}\ \{\}$ $\mathit{else}\ z' \cup \mathit{cart}(\mathit{rest}(s), t)$

By applying several deductive rules to this goal alone, we obtain the new goal

B:

	$z'' = \{(a, b)\colon a = \mathit{choice}(s)\ \mathit{and}\ b \in \mathit{rest}(t)\}$	$\mathit{if}\ t = \{\}$ $\mathit{then}\ \{\}$ $\mathit{else\ if}\ s = \{\}$ $\quad \mathit{then}\ \{\}$ $\quad \mathit{else}\ (\mathit{choice}(s), \mathit{choice}(t)) \cup$ $\qquad \mathit{cart}(\mathit{rest}(s), t) \cup z''$

This goal is a precise instance of the earlier goal; consequently, our recurrence strategy motivates us to form an auxiliary procedure *cartnew*(s, t) having the earlier goal as its output specification, i.e.,

$$\mathit{cartnew}(s', t') \Leftarrow \{(a, b)\colon a = \mathit{choice}(s')\ \mathit{and}\ b \in t'\}.$$

We therefore introduce an additional output column corresponding to the new procedure, and we add to the sequent the row

	assertions	goals	outputs $cart(s, t)$	outputs $cartnew(s', t')$
A′:		$z' = \{(a, b)\text{: } a = choice(s')$ *and* $b \in t'\}$	*if* $s = \{\}$ *then* $\{\}$ *else* $cartnew(s, t) \cup cart(rest(s), t)$	z'

The induction hypothesis corresponding to this goal is then

if $(u', v') \prec_{w'} (s', t')$ *then* $cartnew(u', v') = \{(a, b)\text{: } a = choice(u')$ *and* $b \in v'\}$			

By applying deduction rules to the new goal, we obtain the goal

B′:		$z'' = \{(a, b)\text{: } a = choice(s')$ *and* $b \in rest(t')\}$	*if* $s = \{\}$ *then* $\{\}$ *else* $cartnew(s, t) \cup cart(rest(s), t)$	*if* $t' = \{\}$ *then* $\{\}$ *else* $(choice(s'), choice(t')) \cup z''$

Applying GA-resolution between this goal and the induction hypothesis, and simplying by transformation rules, we obtain the goal

	$(s', rest(t')) \prec_{w'} (s', t')$	*if* $s = \{\}$ *then* $\{\}$ *else* $cartnew(s, t) \cup cart(rest(s), t)$	*if* $t' = \{\}$ *then* $\{\}$ *else* $(choice(s'), choice(t')) \cup cartnew(s', rest(t'))$

Note that a recursive call has now appeared in the output entry for the auxiliary procedure *cartnew*. By further transformation, the well-founded ordering $\prec_{w'}$ is chosen to be $\prec_{s_2}$, defined by

$$(u_1, u_2) \prec_{s_2} (v_1, v_2) \quad \text{if } u_2 \text{ is a proper subset of } v_2.$$

The final program obtained from this derivation is

```
cart(s, t) ⇐ if s = {}
             then {}
             else cartnew(s, t) ∪
                  cart(rest(s), t)

cartnew(s′, t′) ⇐ if t′ = {}
                  then {}
                  else (choice(s′), choice(t′)) ∪
                       cartnew(s′, rest(t′)).
```

There are a few extensions to the method for forming auxiliary procedures that we will not describe in detail:

(1) We have been led to introduce an auxiliary procedure when an entire goal was found to be an instance of a previous goal. As we remarked in the section on

mathematical induction, we can actually introduce an auxiliary procedure when some subsentence of a goal is an instance of some subsentence of a previous goal.

(2) Special treatment is required if the assertions and goal incorporated into the induction hypothesis contain more than one occurrence of the same skolem function. We do not describe the formation of such an induction hypothesis here.

(3) To complete the derivation of the auxiliary procedure, we may be forced to weaken or strengthen its specification by adding input or output conditions incrementally. We do not present here the extension of the procedure-formation principle that permits this flexibility.

GENERALIZATION

In performing a proof by mathematical induction, it is often necessary to generalize the theorem to be proved, so as to have the advantage of a stronger induction hypothesis in proving the inductive step. Paradoxically, the more general statement may be easier to prove. If the proof is part of the synthesis of a program, generalizing the theorem can result in the construction of a more general procedure, so that recursive calls to the procedure will be able to achieve the desired subgoals. The recurrence strategy we have outlined earlier provides a strong clue as to how the theorem is to be generalized.

We have formed an auxiliary procedure when a goal is found to be a precise instance of a previous goal. However, in some derivations it is found that the new goal is not a precise instance of the earlier goal, but that both are instances of some more general expression. This situation suggests introducing a new auxiliary procedure whose output condition is the more general expression, in the hope that both goals may be achieved by calls to this procedure.

Let us be more precise. Suppose we are in the midst of a derivation and that we have already developed a goal A, of form

	assertions	*goals*	*outputs* $f(a)$
A:		$R'(a, s_1, z_1)$	$t_1(z_1)$

where s_1 is an arbitrary term. Assume that by applying deduction rules only to goal A and some assertions $P'_1(a), P'_2(a), \ldots, P'_k(a)$, we obtain a goal B, of form

B:		$R'(a, s_2, z_2)$	$t_2(z_2)$

where s_2 is a term that does not match s_1. Thus, the new goal (goal B) is not a precise instance of the earlier goal (goal A). Hence, if an induction hypothesis is formed for goal A itself, the resolution rule cannot be applied between goal B and the induction hypothesis.

However, both goals A and B may be regarded as instances of the more general expression $R'(a, b', z')$, where b' is a new constant: goal A is obtained by replacing b' by s_1, and goal B is obtained by replacing b' by s_2. This suggests that we attempt to establish a more general expression (goal A′) hoping that the proof of goal A′ will contain a subgoal (goal B′) corresponding to the original goal B, so that the induction hypothesis resulting from goal A′ will be strong enough to establish goal B′.

The new goal A′ constitutes the output condition for an auxiliary procedure, whose specification is

$$\begin{aligned}&fnew(a', b') \Leftarrow \mathit{find}\ z'\ \mathit{such\ that}\ R'(a', b', z')\\ &\quad \mathit{where}\ P'(a').\end{aligned}$$

(Here, $P'(a')$ is the conjunction $P_1'(a')$ *and* $P_2'(a')$ $\cdots$ *and* $P_k'(a')$.) Consequently, we introduce a new output column to the sequent, and we add the new assertion

assertions	*goals*	*outputs* $f(a)$	$fnew(a', b')$
$P'(a')$			

and the new goal

A′:		$R'(a', b', z')t_1(fnew(a, s_1))$	z'

(Note again that it is logically necessary to replace the input constant a by a new constant a'.) Corresponding to this assertion and goal we have the induction hypothesis

if $(u', v') <_{w'} (a', b')$ *then if* $P'(u')$ *then* $R(u', v', fnew(u', v'))$			

There is no guarantee that we will be able to develop from goal A′ a goal B′ such that the resolution rule can be applied between goal B′ and the induction hypothesis. Nor can we be sure that we will conclude the derivation of *fnew* successfully. If we fail to derive *fnew*, we may still complete the derivation of *f* in some other way.

We illustrate the generalization process with an example that also serves to show how program-synthesis techniques can be applied as well to *program transformation* (see, e.g., Burstall and Darlington [3]). In this application we are given a clear and concise program, which may be inefficient; we attempt to derive an equivalent program that is more efficient, even though it may be neither clear nor concise.

We are given the program

```
reverse(l) ⇐ if l = []
                then []
                else reverse(tail(l))<>[head(l)]
```

for reversing the order of the elements of a list l. Here, *head*(l) is the first element of a nonempty list l and *tail*(l) is the list of all but the first element of l. Recall that $u <> v$ is the result of appending two lists u and v, [] denotes the empty list, and $[w]$ is the list whose sole element is w. As usual, we omit type conditions, such as *islist*(l), from our discussion.

This *reverse* program is inefficient, for it requires many recursive calls to *reverse* and to the append procedure <>. We attempt to transform it to a more efficient version. The specification for the transformed program *rev*(l) is

$$rev(l) \Leftarrow \mathit{find}\ z_1\ \mathit{such\ that}\ z_1 = reverse(l).$$

The initial sequent is thus

	assertions	*goals*	*outputs* $rev(l)$
A:		$z_1 = reverse(l)$	z_1

The given *reverse* program is not considered to be a primitive. However, we admit the transformation rules

$$reverse(u) \Rightarrow [] \quad \mathit{if}\ u = []$$

and

$$reverse(u) \Rightarrow reverse(tail(u)) <> [head(u)] \quad \mathit{if}\ u \neq [];$$

obtained directly from the *reverse* program.

We assume that the transformation rules we have concerning lists include:

$$head(u \cdot v) \Rightarrow u$$

$$tail(u \cdot v) \Rightarrow v$$

$$[u] \Rightarrow u \cdot []$$

$$(u \cdot v = []) \Rightarrow \mathit{false}$$

(where $u \cdot v$ is the result of inserting u before the first element of the list v; it is the Lisp *cons* function)

$$u <> v \Rightarrow v \quad \mathit{if}\ u = []$$

$$u <> v \Rightarrow u \quad \mathit{if}\ v = []$$

$$u <> v \Rightarrow head(u) \cdot (tail(u) <> v) \quad \mathit{if}\ u \neq []$$

$$(u <> v) <> w \Rightarrow u <> (v <> w)$$

$$tail(l) \prec_L l \Rightarrow \mathit{true} \quad \mathit{if}\ l \neq []$$

Applying transformation rules to the initial goal, we obtain a subgoal

B:		$z_2 = reverse(tail(l)) <> [head(l)]$	*if* $l = []$ *then* $[]$ *else* z_2

This goal is not a precise instance of goal A. However, both goals may be regarded as instances of the more general expression

$$z' = reverse(l') <> m'.$$

Goal A is obtained by replacing l' by $tail(l)$ and m' by $[]$ (because $u <> [] = u$), and goal B is obtained by replacing l' by $tail(l)$ and m' by $[head(l)]$. This suggests that we attempt to construct an auxiliary procedure having the more general expression as an output condition; the specification for this procedure is

$$revnew(l', m') \Leftarrow \mathit{find}\ z'\ \mathit{such\ that}\ z' = reverse(l') <> m'.$$

Consequently, we introduce a new output column to the sequent, and we add the new goal

A′:

assertions	goals	outputs $rev(l)$	outputs $revnew(l', m')$
	$z' = reverse(l') <> m'$	$revnew(l, [])$	z'

The induction hypothesis corresponding to this goal is then

if $(u', v') \prec_{w'} (l', m')$ *then* $revnew(u', v') = reverse(u') <> v'$		

By applying deduction rules to the goal A′, we eventually obtain

B′:

assertions	goals	outputs $rev(l)$	outputs $revnew(l', m')$
	$z'' = reverse(tail(l')) <> (head(l') \cdot m')$	$revnew(l, [])$	*if* $l' = []$ *then* m' *else* z''

We succeed in applying the resolution rule between this goal and the induction hypothesis.

Ultimately, we obtain the final program

```
rev(l) ⇐ revnew(l, [])
revnew(l′, m′) ⇐  if l′ = []
                  then m′
                  else revnew(tail(l′), head(l′)·m′).
```

This program turns out to be more efficient than the given program *reverse*(l); it is essentially iterative and employs the insertion operation · instead of the expensive append operation <>. In general, however, we have no guarantee that the program produced by this approach will be more efficient than the given program. A possible remedy is to include efficiency criteria explicitly in the specification of the program. For example, we might require that the *rev* program should run in time linear to the length of l. In proving the theorem obtained from such a specification, we would be ensuring that the program constructed would operate within the specified limitations. Of course, the difficulty of the theorem-proving task would be compounded by such measures.

Some generalizations are quite straightforward to discover. For example, if goal A is of form $R'(a, 0, z_1)$ and goal B is of form $R'(a, 1, z_2)$, this immediately suggests that we employ the general expression $R'(a, b', z')$. Other generalizations may require more ingenuity to discover. In the *reverse* example, for instance, it is not immediately obvious that $z_1 = reverse(l)$ and $z_2 = reverse(tail(l)) <> [head(l)]$ should both be regarded as instances of the more general expression $z' = reverse(l') <> m'$.

Our strategy for determining how to generalize an induction hypothesis is distinct from that of Boyer and Moore [2]. Their system predicts how to generalize a goal before developing any subgoals in our approach, recurrences between a goal and its subgoals suggest how the goal is to be generalized.

COMPARISON WITH THE PURE TRANSFORMATION-RULE APPROACH

Recent work (e.g., Manna and Waldinger [7], as well as Burstall and Darlington [3]) does not regard program synthesis as a theorem-proving task, but instead adopts the basic approach of applying transformation rules directly to the given specification. What advantage do we obtain by shifting to a theorem-proving approach, when that approach has already been attempted and abandoned?

The structure we outline here is considerably simpler than, say, our implemented synthesis system DEDALUS, but retains the full power of that system. DEDALUS required special mechanisms for the formation of conditional expressions and recursive calls, and for the satisfaction of "conjunctive goals" (of form "*find z such that* $R_1(z)$ *and* $R_2(z)$"). It could not treat specifications involving quantifiers. It relied on a backtracking control structure, which required it to explore one goal completely before attention could be passed to another goal. In the present system, these constructs are handled as a natural outgrowth of the theorem-proving process. In addition, the foundation is laid for the application of more sophisticated search strategies, in which attention is passed back and forth freely between several competing assertions and goals. The present framework can take advantage of parallel hardware.

Furthermore, the task of program synthesis always involves a theorem-proving component, which is needed, say, to prove the termination of the program being constructed, or to establish the input condition for recursive calls. (The Burstall-Darlington system is interactive and relies on the user to prove these theorems; DEDALUS incorporates a separate theorem prover.) If we retain the artificial distinction between program synthesis and theorem proving, each component must duplicate the efforts of the other. The mechanism for forming recursive calls will be separate from the induction principle; the facility for handling specifications of the form

find z such that $R_1(z)$ *and* $R_2(z)$

will be distinct from the facility for proving theorems of form

for some z, $R_1(z)$ *and* $R_2(z)$;

and so forth. By adopting a theorem-proving approach, we can unify these two components.

Theorem proving was abandoned as an approach to program synthesis when the development of sufficiently powerful automatic theorem provers appeared to flounder. However, theorem provers have been exhibiting a steady increase in their effectiveness, and program synthesis is one of the most natural applications of these systems.

ACKNOWLEDGMENTS

We would like to thank John Darlington, Chris Goad, Jim King, Neil Murray, Nils Nilsson, and Earl Sacerdoti for valuable discussions and comments. Thanks are due also to Patte Wood for aid in the preparation of this manuscript.

REFERENCES

1. BLEDSOE, W.W. Non-resolution theorem proving. *Artif. Intell. J. 9*, (1977), 1-35.
2. BOYER, R.S., AND MOORE, JS. Proving theorems about LISP functions *J. ACM 22*, 1 (Jan. 1975), 129-144.
3. BURSTALL, R.M., AND DARLINGTON, J. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 44-67.
4. DARLINGTON, J.L. Automatic theorem proving with equality substitutions and mathematical induction. *Machine Intell. 3* (Edinburgh, Scotland) (1968), 113-127.
5. GREEN, C.C. Application of theorem proving to problem solving. In *Proc. Int. Joint Conf. on Artificial Intelligence* (Washington D.C., May 1969), 219-239.
6. HEWITT, C. Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. Ph.D. Diss., M.I.T., Cambridge, Mass., 1971.
7. MANNA, Z., AND WALDINGER, R. Synthesis: dreams $\Rightarrow$ programs. *IEEE Trans. Softw. Eng. SE-5*, 4 (July 1979), 294-328.
8. MURRAY, N. A proof procedure for non-clausal first-order logic. Tech. Rep. Syracuse Univ., Syracuse, N.Y., 1978.
9. NELSON, G., AND OPPEN, D.C. A simplifier based on efficient decision algorithms. In *Proc. 5th ACM Symp. Principles of Programming Languages* (Tucson, Ariz., Jan. 1978), pp. 141-150.
10. NILSSON, N.J. A production system for automatic deduction. *Machine Intell. 9*, Ellis Horwood, Chichester, England, 1979.
11. NILSSON, N.J. *Problem-solving methods in artificial intelligence.* McGraw-Hill, New York, 1971, pp. 165-168.
12. ROBINSON, J.A. A machine-oriented logic based on the resolution principle. *JACM 12*, 1 (Jan. 1965), 23-41.
13. WALDINGER, R.J., AND LEE, R.C.T. PROW: A step toward automatic program writing. In *Proc. Int. Joint Conf. on Artificial Intelligence* (Washington D.C., May 1969), pp. 241-252.
14. WILKINS, D. QUEST—A non-clausal theorem proving system. M.Sc. Th., Univ. of Essex, England, 1973.

Received December 1978; revised September 1979