Stanford Artificial Intelligence Laboratory
Memo AIM-320

Computer Science Department
Report No. STAN-CS-78-690

November 1978

A DEDUCTIVE APPROACH TO PROGRAM SYNTHESIS

by

Zohar Manna
Artificial Intelligence Lab
Stanford University

Richard Waldinger
Artificial Intelligence Center
SRI International

Research sponsored by

National Science Foundation
Office of Naval Research
Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-320

November 1978

Computer Science Department
Report No. STAN-CS-78-690

A DEDUCTIVE APPROACH TO PROGRAM SYNTHESIS

by

Zohar Manna
Artificial Intelligence Lab
Stanford University

Richard Waldinger
Artificial Intelligence Center
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.

This research was supported in part by the National Science Foundation under Grants MCS 76-83655 and MCS 78-02591, by the Office of Naval Research under Contracts N00014-76-C-0687 and N00014-75-C-0816, by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, and by the United States-Israel Binational Science Foundation.

MOTIVATION

The early work in program synthesis relied strongly on mechanical theorem-proving techniques. The work of Green [1969] and Waldinger and Lee [1969], 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 [1975]), 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 [1977] and Manna and Waldinger [1977]), on the other hand, has abandoned the theorem-proving approach, and has relied instead on the direct application of transformation or rewriting rules to the program's specifications; 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 will 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 will 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: . . .}, 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

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

Here, the *input condition*

$$integer(n)\ and\ 0 \le n$$

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

$$integer(z)\ and\ z^2 \le n < (z+1)^2$$

describes the relation the output z is intended to satisfy.

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

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

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.

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

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

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

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

Thus, in this paper we limit our discussion to the synthesis of applicative programs, which yield an output but produce no side effects. To derive a program from such a specification, we attempt to prove a theorem of the form

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

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,

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 [1969]). We will denote a sequent by a table with three columns: assertions, goals, and output. Each row in the sequent has the form

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

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 instance 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 will denote constants or tuples of constants by a, *b,* c, . . . , *n* and variables or tuples of variables by u, v, w, . . . , 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 $t_i(a, e)$] satisfies the specification.

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 "dummys," 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

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

we construct the initial sequent

Assertions	*Goals*	*output*
$P(a)$		
	$R(a, z)$	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. Quantifiers have been removed by the usual skolemization procedure (see, *e.g.,* Nilsson [1971]). The output z is a variable, for which we can make substitutions; the input a is a constant.

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."

In the remainder of this paper we outline the deductive rules of our system, and we present two complete examples illustrating the application of the system 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:

the ***andsplit** rule*

assertions	*goals*	*output*
F and G		*t*
F		*t*
G		*t*

Similarly, we have the ***orsplit** rule*

assertions	*goals*	*output*
	F or G	*t*
	F	*t*
	G	*t*

and the *ifsplit rule*

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

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

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

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

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

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

$$u|0 \Rightarrow true \qquad \text{if } integer(u) \text{ and } u \neq 0$$

expresses that every nonzero integer divides zero.

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

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

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

A transformation rule

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

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

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

For example, we might include the rule

$x + 0 \Rightarrow x$ *if number(x)*

but not the rule

$x \Rightarrow x + 0$ *if number(x).*

Assertions and goals are affected differently by transformation rules. Suppose

$r \Rightarrow s$ *if* P

is a transformation rule and $F(r')$ is an assertion such that its subexpression r' 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 [1965]) of r and r'. (We rename the variables of $F(r')$, if necessary, to insure 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

if $P\theta$ *then* $F(s)\theta$

to our sequent.

For example, suppose we have the assertion

$a \in l$ *and* $a \neq 0$

and we apply the transformation rule

$u \in v \Rightarrow u = head(v)$ *or* $u \in tail(v)$ *if islist(v) 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

if islist(l) and $l \neq [\,]$
then $(a = head(l)$ *or* $a \in tail(l))$ *and* $a \neq 0$.

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 $F(r')$ 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* $F(s)\theta$" is false; then the corresponding instance of $P\theta$ is true, and the corresponding instance of $F(s)6$ is false. Recall that $F(r)\theta$ and $F(r')\theta$ are identical. Then, by the transformation rule, the corresponding instance of $F(r)\theta$, *i.e.* of $F(r')\theta$, is false. We know that if any instance of $F(r')$ is false, the corresponding instance of t satisfies the given specification. Hence, because some instance of $F(r')\theta$ is false, the corresponding instance of $t\theta$ is the desired output,

In our deduction rule notation, we write

assertions	*goals*	*output*
$F(r')$		t
if $P\theta$ *then* $F(s)\theta$		$t\theta$

The corresponding dual deduction rule for goals is

assertions	*goals*	*output*
	$F(r')$	t
	$P\theta$ *and* $F(s)\theta$	$t\theta$

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

For example, suppose we have the goal

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

and we apply the transformation rule

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

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

	$(integer(a)$ and $a \neq 0)$ and $(true$ and $b \mid 0)$	ot 1

which can be further transformed to

	$integer(a)$ 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 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. In general, efficient procedures for particular subtheories may be represented as transformation rules (see, e.g., Bledsoe [1977] or Nelson and Oppen [1978].)

Transformation rules play the role of the "antecedent theorems" and "consequent theorems" of PLANNER (Hewitt [1971]). For example, a consequent theorem that we might write as

to prove $f(u) = f(v)$
prove $u = v$

can be represented by the transformation rule

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

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 $not(f(a) = f(b))$ the simpler assertion $not(a = b)$.)

RESOLUTION

The original resolution principle (Robinson [1965]) applied only to a sentence in conjunctive normal form. However, the ability to deal with sentences not in this form is essential if resolution and mathematical induction are to coexist happily within the same framework. The version of resolution we employ does not require the sentences to be in conjunctive normal form.

Assume our sequent contains two assertions of form $F(P_1)$ and $G(P_2)$, where P_1 and P_2 are subsentences of these assertions 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

$$F(true)\theta \text{ or } G(false)\theta,$$

and add it to the sequent. (Here, $F(true)$ denotes the result of replacing P_1 by true in $F(P_1)$. Of course, we may need to do the usual renaming to ensure that $F(P_1)$ and $G(P_2)$ 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 $F(P_1)$ and $G(P_2)$ may be reversed.

For example, suppose our sequent contains the assertions

if (P(x) and Q(b)) then R(x)

and

P(a) 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

(if true then R(a)) 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

$(not\ P_1)\ or\ Q$

and

$P_2\ or\ R$

the new assertion

$Q\theta\ or\ R\theta$,

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

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

which reduces to the same conclusion

$Q\theta\ or\ R\theta$

as the original resolution rule.

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

$F(true)\theta\ or\ G(false)\theta$

holds,

A "non-clausal" resolution rule similar to ours has been developed by Murray [1978]. Other such rules have been proposed by Wilkins [1973] and Nilsson [1977].

THE RESOLUTION RULES

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

the *AA-resolution rule*

assertions	*goals*
$F(P_1)$	
$G(P_2)$	
$F(true)\theta$ *or* $G(false)\theta$	

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 *CC-resolution rule*

assertions	*goals*
	$F(P_1)$
	$G(P_2)$
	$F(true)\theta$ *and* $G(false)\theta$

-the *GA-resolution rule*

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

the *AC-resolution rule*

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

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 $F(P_1)$ and $G(P_2)$ have output expressions t_1 and t_2, respectively, the resolvent will have the output expression

$$if\ P_1\theta\ then\ t_1\theta\ else\ t_2\theta.$$

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

$$F(true)\theta\ and\ G(false)\theta$$

has been obtained by GG-iesolution from two goals $F(P_1)$ and $G(P_2)$. We would like to show that if this goal is true, the conditional output expression satisfies the desired specification. We assume that the resolvent is true; therefore both $F(true)\theta$ and $G(false)\theta$ are true. In the case that $P_1\theta$ is true, we have that $F(P_1)\theta$ is identical to $F(true)\theta$, and therefore is 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

$$if\ P_1\theta\ then\ t_1\theta\ else\ t_2\theta$$

satisfies the desired specification. By duality, the same output expression can be derived for 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	*gods*	*output*
	$head(z) = a$ *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

	max(tail(l)) $\geq$ *head(l)* *and tail(l)* $\neq$ *[ ]*	*max(tail(l))*
	not(max(tail(l)) I’ 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 ant-l 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

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

and the assertion

assertions		
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 [1978] restricts the resolution rules to prevent many such fruitless applications.

We first assign a polarity (either positive (+) or negative (-) or both) to every subsentence of a given sequent, as follows:

- each goal is positive
- each assertion is negative
- if a subsentence S has form "*not* α", then its component α has polarity opposite to S
- 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
- 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*	*output*
$(if\ P(y,d)^{+}\ then\ Q(b,y)^{-})^{-}$	$(P(c,x)^{+}\ and\ Q(x,a)^{+})^{+}$	

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 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, constitutes 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(n) holds for every 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 $\prec$. Thus, in trying to prove F(a), the well-founded induction principle allows us to assume the induction hypothesis

$$\textit{for all } u, \textit{ if } u \prec a \textit{ then } F(u).$$

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 every 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

$$\begin{array}{ll} f(a) \Leftarrow & \textit{find } z \textit{ such that} \\ & \quad \textit{for some } y,\ R(a, y, z) \\ \quad \textit{where } P(a), & \end{array}$$

Then our initial sequent is

assertions	*goals*	*output*
$P(a)$	$R(a, y, z)$	z

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

if $u \prec a$ *then if* $P(u)$ *then* $R(u,\ g(u),\ f(u))$		

Here, f denotes the program we are trying to construct, and g is a new Skoiem function corresponding to the variable y. The well-founded set and the particular well-founded ordering $\prec$ to be employed in the proof have not yet been determined.

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 $R(a, y, f(a))$, for some y; or, equivalently, $R(a,$ g(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 $R(u,$ *g(u), flu)).*

in general, we could 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 would not be relevant to the final proof, and the proliferation of new assertions would obstruct 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 derived from the initial assertion and goal. Suppose that $Q(a, y,\ z)$ is some subsentence of the initial goal; then that goal may be written

$$R(Q(a,\ y,\ z)).$$

Suppose further that at some point in the derivation an assertion or goal of form

$$S(Q(t,\ y',\ z'))$$

is developed, where t is an arbitrary term and y' and z' are distinct variables. In other words, the newly developed assertion or goal has a subsentence $Q(t, y', z')$ that is a precise instance of a subsentence $Q(a, y, z)$ of the initial goal. This recurrence motivates us to add the induction hypothesis

```
if $u \prec a$
then if $P(u)$
    then $R(Q(u, g(u), f(u)))$.
```

The rationale for introducing the induction hypothesis at this point is that now we can perform resolution between the induction hypothesis and the newly developed assertion or goal $S(Q(t, y', z'))$, eliminating the subexpression $Q(t, g(t), f(t))$. In fact, we do not need to introduce the induction hypothesis unless the original subexpression $Q(a, y, z)$ and the recurrrent subexpression $Q(t, y', z')$ have the same polarity, either both positive or both negative. For the subexpression $Q(u, g(u), f(u))$ in the inductive assertion always has polarity opposite to the subexpression $Q(a, y, z)$ of the initial goal; and the induction hypothesis cannot be resolved against the newiy developed assertion or goal unless the eliminated subexpressions $Q(u, g(u), f(u))$ and $Q(t, y', z')$ have opposite polarity, by the polarity strategy for resolution.

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

```
rem(i, j) ⇐ find z such that
               for some y,
               $i = y \cdot j + z$ and $0 \le z$ and $z < j$
   where $0 \le i$ and $0 < j$.
```

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

assertions	*goals*	*outputs*
$0 \le i$ *and* $0 < j$	$i = y \cdot j + z$ *and* $0 \le z$ *and* $z < j$	z

Here, the inputs i and j are constants, for which we can make no substitution; 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* $z < j$	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, taking $Q(i, j, y, z)$ to be the initial goal itself, we add as a new assertion the induction hypothesis

$\textit{if } (u_1, u_2) \prec (i, j)$ $\textit{then if } 0 \le u_1 \textit{ and } 0 < u_2$ $\textit{then } u_1 = g(u_1, u_2) \cdot u_2 + rem(u_1, u_2)$ $\textit{and } 0 \le rem(u_1, u_2) \textit{ and } rem(u_1, u_2) < u_2$		

Here, g is a new Skolem function corresponding to the variable y, and $\prec$ is an arbitrary well-founded ordering. Note that $\prec$ is to be defined on pairs because the desired program f has a pair of inputs.

We can now apply GA-resolution between the goal

	$i-j = y_1 \cdot j + z \textit{ and } 0 \le z \textit{ and } z < j$	z

and the induction hypothesis; the unifying substitution θ is

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

The new goal is

	$\textit{true and}$ $\textit{not } (\textit{if } (i-j, j) \prec (i, j)$ $\textit{then if } 0 \le i-j \textit{ and } 0 < j$ $\textit{then false})$	$rem(i-j, j)$

which reduces to

	$(i-j, j) \prec (i, j) \textit{ and}$ $0 \le i-j \textit{ and } 0 < j$	$rem(i-j, j)$

Note that the recursive call *rem*($i-j$, j) has been introduced into the output entry.

The particular well-founded ordering $\prec$ to be employed in the proof has not yet been determined. To choose the ordering requires special transformation rules, which describe known well-founded orderings and ways of combining them. In this case, the ordering $\prec$ is chosen to be the $<$ ordering on the first component of the pairs, by application of the transformation rule

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

A new goal

	$i-j < i$ *and* $0 \le i-j$ *and* $0 \le i$ *and* *true and* $0 \le i-j$ *and* $0 < j$	*rem*($i-j$, j)

is produced; this goal ultimately reduces to

	$j \le i$	*rem*($i-j$, j)

In other words, in the case that $j \le i$, the output *rem*($i-j$, j) satisfies the desired program's specification.

In a later section we will give the full derivation of the related program that finds the integer quotient of two integers.

We will not discuss here the more general case, where a newly developed assertion or goal has a subsentence that is an instance of a subsentence not of the initial goal, but of some intermediate goal or assertion; this situation accounts for the introduction of "auxiliary procedures" to be called by the program under construction. We will also not discuss the case where the new subsentence is not a precise instance of the earlier subsentence, but where both are instances of a somewhat more general sentence.

Some early efforts toward incorporating mathematical induction in a resolution framework were made by J. L. Darlington [1968]. 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 cieductive synthesis approach. Our task is to construct a program *div*(i, j) for finding the integer quotient of dividing a nonnegative integer i by a positive integer j. Our specification is expressed as

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

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

assertions --	*goats*	*output*
1. $0 \le i$ *and* $0 < j$	2. $i = y \cdot j + z$ *and* $0 \le z$ *and* $z < j$	y

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

in presenting the derivation we will 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

The unifying substitution in deriving goal 7 is

$$e = [\, y \leftarrow 0;\ v \leftarrow j \,];$$

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

The GA-resolution rule can now be appiied 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

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

	10. $i < j$	0

in other words, we have found that in the case that i < j, the output 0 will satisfy the specification for the quotient 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. $i = y_1 \cdot j + j + z \textit{ and } 0 \leq z \textit{ and } z < j$	y_1+1

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 z produced the new output y_1+1.

The transformation rule

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

applied to goal 11 yields

	12. $i-j = y_1 \cdot j + z \textit{ and } 0 \leq z \textit{ and } z < j$	y_1+1

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

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

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. $\textit{if } (u_1, u_2) \prec (i, j)$ $\textit{then if } 0 \le u_1 \textit{ and } 0 < u_2$ $\textit{then } u_1 = div(u_1, u_2) \cdot u_2 + h(u_1, u_2) \textit{ and}$ $0 \le h(u_1, u_2) \textit{ and } h(u_1, u_2) < u_2$		

Here, h is a Skolem function corresponding to the variable z, and $\prec$ is an arbitrary well-founded ordering.

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

	14. $\textit{true and}$ $\textit{not } (\textit{if } (i-j, j) \prec (i, j)$ $\textit{then if } 0 \le i-j \textit{ and } 0 < j$ $\textit{then false})$	$div(i-j, j)+1$

. Here, the unifying substitution is

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

and the eliminated subexpression is

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

Note that the substitution to the variable y_1 has caused the output entry y_1+1 to be changed to $div(i-j, j)+1$. The use of the induction hypothesis has introciuced the recursive call $div(i-j, j)$ into the output.

Goal 14 reduces to

	15. $(i-j, j) \prec (i, j) \textit{ and } 0 \le i-j \textit{ and } 0 < j$	$div(i-j, j)+1$

The particular ordering $\prec$ 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

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

A new goal is produced:

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

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, $0 \le i$, and assertion 6, $0 < j$, goal 16 is reduced to

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

In other words, we have learned that in the case thatj $\le i$, the output $div(i-j, j)+1$ satisfies the specification of the *div* program. On the other hand, in deriving goal 10 we learned that in the case that $i < j$, 0 is a satisfactory output. Assuming we have the assertion 4

$u \le v$ *or* $v < u$,

we can obtain the goal

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

by GA-resolution.

The final goal

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

can then be obtained by GG-resolution between goals 10 and 18. The conditional expression has 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 program

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

is obtained directly from the final output entry.

Note that the same proof could be used to derive a remainder program as well as a quotient program. The specification of the remainder program

```
rem(i, j) <= find z such that
                  for some y,
                  i = y·j + z and 0 ≤ z and z < j
   where 0 ≤ i and i < j
```

yields the same initial assertion and goal as the quotient program, except that the initial output entry is *z* instead of *y*. The succeeding output entries are changed accordingly. The final remainder program is then

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

We used steps from the derivation of this program to illustrate the formation of recursive calls in the section on mathematical induction.

ANOTHER COMPLETE EXAMPLE: Finding the Last Element of a List

In this example, we apply the same techniques to derive a list-processing program. Our discussion here will be a bit more brisk than in the preceding section.

Our task is to construct a program last(l) to find the last element of a **nonempty** list l. Our specification is

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

Recall that $u{<>}v$ is the result of appending two lists u and v, $[w]$ is the list whose sole element is w, and $[\,]$ denotes the empty list. Again, we omit type conditions, such as $islist(l)$, from our discussion.

Our initial sequent-is

assertions	*goals*	*output*
4.1 $\neq [\,]$	2. $l = y{<>}[z]$	z

Let us assume that our subject knowledge includes the assertion

3. u = u		

and the transformation rules

$$[\,]{<>}u \Rightarrow u$$

$$(u \cdot v){<>}w \Rightarrow u \cdot (v{<>}w)$$

$$w = u \cdot v \Rightarrow w \neq [\,] \textit{ and } head(w) = u \textit{ and } tail(w) = v$$

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

$$tail(u) \prec_L u \text{ 3 } true \quad \textit{if } u \neq [\,].$$

The first two rules constitute the definition of the append function $<>$; the third expresses the uniqueness of the decomposition of a list into a head and a tail; the fourth provides the meaning of the abbreviation $[u]$; and the final rule defines a welt-founded ordering $\prec_L$ over the lists.

The first transformation rule

$$[\,]<>u \Rightarrow u$$

can be applied to the initial goat 2,

$$l = y<>[z];$$

the unifying substitution is

$$\theta = [\, y \leftarrow [\,];\ u \leftarrow [z]\,]$$

and the resulting goal is

	4. $l = [z]$	z

Applying the two rules

$$[u] \Rightarrow u \bullet [\,]$$

and

$$w = u \bullet v \Rightarrow w \neq [\,] \textit{ and } head(w) = u \textit{ and } tail(w) = v$$

yields

	5. $l \neq [\,]$ *and* $head(l) = z$ *and* $tail(l) = [\,]$	z

Applying GA-resolution between goat 5 and assertion 1, $l \neq [\,]$, produces the goal

	6. $head(l) = z$ *and* $tail(l) = [\,]$	z

Applying GA-resolution again, between goat 6 and assertion 3, u = u, produces the goat

	7. $tail(l) = []$	$head(l)$

Here, the unifying substitution is

$$e = [\, z \leftarrow head(l);\ u \leftarrow head(l)\ ]$$

and the eliminated subexpression is $head(l) = head(l)$. Note that the substitution has caused the output entry z to be *replaced* by $head(l)$. We have learned that in the case where $tail(l)$ is empty the output $head(l)$ satisfies the specification for *last*.

Returning to the initial goal 2,

$$l = y <> [z], \text{--}$$

we can apply the second transformation rule

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

to the subexpression $y <> [z]$. The unifying substitution is

$$\theta = [\, u \leftarrow y_1;\ v \leftarrow y_2;\ w \leftarrow [z];\ y \leftarrow y_1 \cdot y_2\ ]$$

and the resulting goat is

	8. $l = y_1 \cdot (y_2 <> [z])$	z

Applying the transformation rule

$$w = u \cdot v \Rightarrow w \neq [] \ and\ head(w) = u \ and\ tail(w) = v$$

yields

	9. $l \neq []\ and\ head(l) = y_1\ and\ tail(l) = y_2 <> [z]$	z

Next, applying GA-resolution between goal 9 and assertion 1, $l \neq []$, and then between the resulting goal and assertion 3, u = u, we obtain

	10. $rail(l) = y_2 <> [z]$	z

Note that goal 10 is a precise instance of our initial goat 2, I $= y <> [z]$, obtained by replacing l by $tail(l)$; therefore, the following induction hypothesis is formed:

11. *if* $u \prec l$ *then if* $u \neq$ [I *then* $u = g(u) <> [last(u)]$		

Here, $\prec$ is an arbitrary well-founded ordering and g is a Skolem function corresponding to the variable y.

We can now apply GA-resolution between goal 10 and the induction hypothesis, . assertion 1 1. The unifying substitution is

$$\theta = [\ u \leftarrow tail(t);\ y_2 \leftarrow g(tail(l));\ z \leftarrow last(tail(l))\ ]$$

and the eliminated subexpression is

$$tail(l) = g(tail(l)) <> [last(tail(l))];$$

we obtain

	12. *true and not(if* $tail(l) \prec l$ *then if* $tail(l) \neq []$ *then false)*	$last(tail(l))$

which reduces to

	13. $tail(l) \prec l$ *and* $tail(l) \neq []$	$last(tail(l))$

Note that the unifying substitution caused the introduction of the recursive call *last(tail(l))* in the output entry.

The rule

$$tail(u) \prec_L u \Rightarrow true \qquad if\ u \neq [\,]$$

suggests taking the well-founded ordering $\prec$ to be $\prec_L$; we derive

	14. $l \neq [\,]$ *and* $tail(l) \neq [\,]$	$last(tail(l))$

which reduces to

	15. $tail(l) \neq [\,]$	$last(tail(l))$

after GA-resoiuion with assertion 1, $l \neq [\,]$.

We have deduced that in the case where $tail(l) \neq [\,]$, the output $last(tail(l))$ satisfies the specification; on the other hand, from goal 7 we know that in the case where $tail(l) = [\,]$, $head(l)$ is a satisfactory output. Combining these two goals by GG-resolution, we obtain

	16. *true*	*if* $tail(l) = [\,]$
		then $head(l)$
		else $last(tail(l))$

Because we have derived the goal *true* with a corresponding primitive output entry, our derivation is complete. The final program, extracted from the final output entry, is

```
last(l) <= if tail(l) = []
             then head(l)
             else last(tail(l)).
```

Note that the same proof could be used to derive a program $front(l)$ to remove the last element from a **nonempty** list l. The specification for *front* is

```
front(l) <= find y such that
              for some z, l = y<>[z]
```

where $l \neq [\,]$.

This specification yields the same initial assertion and goal as the *last* program, except that the initial output entry is y instead of z. The succeeding output entries are changed accordingly, and the final program derived is

```
front(l) <=  if tail(l) = []
             then []
             else head(l)·front(tail(l)).
```

APPLICATION TO PROGRAM TRANSFORMATION

Our program synthesis techniques can be applied as well to the transformation of programs. in this application, we are given a clear and concise program for a certain task, which may be inefficient; we derive a more efficient equivalent program, which may be neither clear nor concise (see Burstall and Darlington [1977]).

To transform a given program, we regard the program itself as the specification of a new program. For example, suppose we are given the program

```
rev(l)  <= if l = []
             then []
             else rev(tail(l)) <> [head(l)]
  where islist(l)
```

for reversing the order of the elements of a list *l*. This program is inefficient, for it requires many recursive calls to rev and to the append program <>. The specification for the transformed program *revnew*(*l*) is then

```
revnew(l) <= find z such that z = rev(l)
  where islist(l).
```

The initial sequent is thus

assertions	*goals*	*output*
1. *islist*(*l*)	2. z = rev(l)	z

We admit the new transformation rules

$$rev(u) \Rightarrow [\,] \quad \textit{if } u = [\,]$$

and

$$rev(u) \; 3 \; rev(tail(u)) <> [head(u)] \quad \textit{if } u \neq [\,];$$

these rules are obtained directly from the given program.

in such a derivation, the given program *rev* is not regarded as a primitive construct of the target language. For efficiency purposes, we may also choose to regard the append function <> as nonprimitive.

Applying our synthesis techniques, we can obtain the following new program for reversing a list:

```
revnew(l) <= revnew2(l, []),
```

where

```
revnew2(l, m) <= if l = []
                    then m
                    else revnew2(tail(l), head(l)•m ).
```

The derivation involves the formation of auxiliary procedures and the use of generalization, which we do not discuss in this paper.

The new program is more efficient than the given program *rev(l)*; it is essentially iterative and does not employ the expensive <> operation. In general, however, unless we introduce additional efficiency criteria, we cannot ensure that the program we obtain is more efficient than the given program.

COMPARISON WITH THE PURE TRANSFORMATION-RULE APPROACH

Recent work (*e.g.*, Manna and Waldinger [1977], as well as Burstall and Darlington [1977]) 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. That system 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 relied on a backtracking control structure, that 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.

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 Burstail-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.

The two complete examples in this paper have been chosen to illustrate the advantages of the new approach; both were beyond the capabilities of the DEDALUS system.

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:

Bledsoe, W. W. [1977], *Non-resolution theorem proving,* Artificial Intelligence Journal, Vol. 9, pp. 1-35.

Boyer, R. S. and J S, Moore [Jan, 1975], *Proving theorems about LISP functions,* JACM, Vol. 22, pp. 129-144.

Burstall, R. M. and J. Darlington [Jan. 1977], *A transformation system for developing recursive programs,* JACM, Vol. 24, No. 1, pp. 44-67.

Darlington, J. L, [1968], *Automatic theorem proving with equality substitutions and mathematical induction,* Machine intelligence 3, Edinburgh, Scotland, pp. 113-127.

Green, *C. C,* [May 1969], *Application of theorem proving to problem solving,* Proceedings of the International Joint Conference on Artificial Intelligence, Washington DC, pp. 2 1 Q-239,

Hewitt, C. [Apr. 1971], *Description and theoretical analysis (using schemata) of PLANNER:* A *language for proving theorems and manipulating models in a robot,* Ph.D. thesis, MIT, Cambridge, MA.

Manna, Z. and R. Waldinget [Nov. *1977*], *Synthesis: dreams $\Rightarrow$ programs,* Technical Report, Computer Science Dept., Stanford University, Stanford, CA and Artificial Intelligence Center, SRI International, Menlo Park, CA.

Murray, N. [1978], *A proof procedure for non-clausal first-order logic,* Technical Report, Syracuse University, Syracuse, NY.

Nelson, G. and D. C. Oppen [Jan, **1978],** *A simplifier based on efficient decision algorithms,* Proceedings of the Fifth ACM Symposium *on* Principles of Programming Languages, Tuscon, AZ, **pp. 141-I 50.**

Nilsson, N. J. [1971], *Problem-solving methods in artificial intelligence,* McGraw-Hill Book Co., New York, NY **[pp. 165-168].**

Nilsson, N. J, [Aug. 1977], *A production system for automatic deduction,* Technical Report, SRI International, Menlo Park, CA.

Robinson, J. A. [Jan, 1965], *A machine-oriented logic based on the resolution principle,* **JACM, Vol. 12, No. 1, pp. 23-41.**

Waldinger, R. J. and R. C. T. Lee [May **1969],** *PROW: a step toward automatic program writing,* Proceedings of the International Joint Conference on Artificial Intelligence, Washington, DC, pp. 241-262.

Wilkins, D. [*1973* **],** *QUEST--a non-clausal theorem proving system,* M.Sc. thesis, University of Essex, England,