

A DEDUCTIVE APPROACH TO PROGRAM SYNTHESIS

Zohar Manna
Artificial Intelligence Laboratory
Stanford University

Richard Waldinger
Artificial Intelligence Center
SRI International

ABSTRACT

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.

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 [1976]), 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 [1979]), 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. This paper is an abbreviated version of a Stanford University and SRI International technical report, which includes more detailed discussion and some complete examples to illustrate the application of the method.

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 $\{x: \dots\}$, and the descriptor *find z such that*...

For example, to specify a program to compute the remainder of dividing a nonnegative integer i by a positive integer j , we would write

$$\begin{aligned} \text{rem}(i, j) &\Leftarrow \text{find } z \text{ such that} \\ &\quad \text{for some } y, \\ &\quad i = y \cdot j + z \text{ and } 0 \leq z \text{ and } z < j \\ &\quad \text{where } 0 \leq i \text{ and } 0 < j \end{aligned}$$

Here, the *input condition*

$$0 \leq i \text{ and } 0 < j.$$

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

$$\begin{aligned} &\text{for some } y, \\ &i = y \cdot j + z \text{ and } 0 \leq z \text{ and } z < j \end{aligned}$$

describes the relation the output z is intended to satisfy. (Note that, for simplicity, we omit type requirements such as *integer*(i).)

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

$$f(a) \Leftarrow \text{find } z \text{ such that } R(a, z)$$

where $P(a)$.

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

for all a ,
if $P(a)$
then for some z , $R(a, z)$.

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* $A_1, A_2, \dots, A_m$, and the *goals* $G_1, G_2, \dots, G_n$. 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 , $A(a, x)$
then for some x , $G(a, x)$

where a denotes all the constants of the sequent, x denotes all the free variables, $A(a, x)$ denotes the conjunction of all the assertions $A_i(a, x)$, and $G(x, a)$ denotes the disjunction of all the goals $G_j(a, x)$. (In general, we will denote constants or tuples of constants by $a, b, c, \dots, n$ and variables or tuples of variables by $u, v, w, \dots, z$.) 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," which 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

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

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.

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: if our sequent contains a goal of form F or G , we can introduce the two goals F and G into our sequent; and the *ifsplit* rule: if our sequent contains a goal of form *if* F *then* G , we can introduce the new assertion F and the new goal G .

There is no *orsplit* rule for assertions or *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.

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

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

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

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

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

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

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

Assertions and goals are affected differently by transformation rules. Suppose

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

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

$$\text{if } P\theta \text{ then } F(s)\theta$$

to our sequent.

For example, suppose we have the assertion

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

and we apply the transformation rule

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

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

$$\text{if } \text{islist}(l) \text{ and } l \neq []$$

$$\text{then } (a = \text{head}(l) \text{ or } a \in \text{tail}(l)) \text{ and } a \neq 0.$$

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 .

The corresponding dual deduction rule for goals, in our deduction-rule notation, is

assertions	goals	output
	$F(r')$	t
	$P\theta \text{ and } F(s)\theta$	$t\theta$

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

For example, suppose we have the goal

	$a z \text{ and } b z$	$z+1$
--	------------------------	-------

and we apply the transformation rule

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

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

	$(\text{integer}(a) \text{ and } a \neq 0) \text{ and } (\text{true and } b 0)$	$0+1$
--	---	-------

which can be further transformed to

	$\text{integer}(a) \text{ and } a \neq 0 \text{ and } b 0$	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]). Some transformation rules effect the removal of quantifiers. Other rules play the role of the "antecedent theorems" and "consequent theorems" of PLANNER (Hewitt [1971]).

RESOLUTION

The original resolution principle (Robinson [1965]) applied only to sentences in conjunctive normal form. The version we employ does not make this requirement.

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

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

and add it to the sequent. (Here, $F(\text{true})$ 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

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

and

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

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

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

Therefore, the AA-resolution rule allows us to eliminate the subexpression " $P(a) \text{ and } Q(b)$ " and derive the conclusion

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

which reduces to

$R(a)$

by application of the appropriate transformation rules.

A "nonclausal" resolution rule similar to ours has been developed by Murray [1978]. Other such rules have been proposed by Wilkins [1973] and Nilsson [1979].

THE RESOLUTION RULES

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

assertions	goals
$F(P_1)$	
$G(P_2)$	
$F(\text{true})\theta$ or $G(\text{false})\theta$	

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

By duality, we can regard goals as negated assertions; consequently, we obtain a "GG-resolution rule", a "GA-resolution rule", and an "AG-resolution rule" as corollaries of the AA-resolution rule.

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 $t_2\theta$.

For example, suppose we have derived the two goals

	$\text{max}(\text{tail}(l)) \geq \text{head}(l)$ and $\text{tail}(l) = []$	$\text{max}(\text{tail}(l))$
	$\text{not}(\text{max}(\text{tail}(l)) \geq \text{head}(l))$ and $\text{tail}(l) = []$	$\text{head}(l)$

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

	$(\text{true and tail}(l) = [])$ and $(\text{not}(\text{false}) \text{ and } \text{tail}(l) = [])$	<i>if $\text{max}(\text{tail}(l)) \geq \text{head}(l)$ then $\text{max}(\text{tail}(l))$ else $\text{head}(l)$</i>
--	---	---

which can be reduced to

	$\text{tail}(l) = []$	<i>if $\text{max}(\text{tail}(l)) \geq \text{head}(l)$ then $\text{max}(\text{tail}(l))$ else $\text{head}(l)$</i>
--	-----------------------	---

A *polarity strategy* adapted from Murray [1978] restricts the application of the resolution rules. 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.)

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.

We may describe this principle as follows: In attempting to prove that a sentence of form $F(a)$ 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 $<$. Thus, in trying to prove $F(a)$, the well-founded induction principle allows us to assume the induction hypothesis

for all u , if $u < a$ 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 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

$f(a) \Leftarrow$ find z such that
for some y , $R(a, y, z)$
where $P(a)$.

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

$if\ u < a$ $then\ if\ P(u)$ $then\ R(u, g(u), f(u))$		
---	--	--

Here, f denotes the program we are trying to construct, and g is a new skolem function corresponding to the variable y . The well-founded set and the particular well-founded ordering $<$ 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 $R(u, g(u), f(u))$.

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 $u < a$
 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 $Q(t, g(t), f(t))$.

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

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

Our initial sequent is then

assertions	goals	outputs
$0 \leq i$ and $0 < j$	$i = y \cdot j + z$ and $0 \leq z$ and $z < j$	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 $0 \leq z$ and $z < j$	z
--	--	-----

This goal is a precise instance of the initial goal

$i = y \cdot j + z$ and $0 \leq z$ and $z < j$

obtained by replacing i by $i - j$. Therefore, taking $Q(i, j, z)$ to be the initial goal itself, we add as a new assertion the induction hypothesis

if $(u_1, u_2) < (i, j)$ then if $0 \leq u_1$ and $0 < u_2$ then $u_1 = g(u_1, u_2) \cdot u_2 + rem(u_1, u_2)$ and $0 \leq rem(u_1, u_2)$ and $rem(u_1, u_2) < u_2$		
--	--	--

Here, g is a new skolem function corresponding to the variable y , and $<$ is an arbitrary well-founded ordering. Note that $<$ 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

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

	<i>true and</i> <i>not (if $\langle i-j, j \rangle < \langle i, j \rangle$</i> <i>then if $0 \leq i-j$ and $0 < j$</i> <i>then false)</i>	<i>rem(i-j, j)</i>
--	---	--------------------

which reduces to

	$\langle i-j, j \rangle < \langle i, j \rangle$ and $0 \leq i-j$ and $0 < j$	<i>rem(i-j, j)</i>
--	---	--------------------

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

The particular well-founded ordering $<$ 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 $<$ is chosen to be the $<$ ordering on the first component of the pairs. Ultimately, the following goal is obtained:

	$j \leq i$	<i>rem(i-j, j)</i>
--	------------	--------------------

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

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.

CONCLUSION

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.

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 Defense Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, and by the United States-Israel Binational Science Foundation.

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. 219-239.
- 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. Waldinger [July 1979], *Synthesis: dreams $\Rightarrow$ programs*, IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, (to appear).
- 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. 141-150.
- Nilsson, N. J. [1971], *Problem-solving methods in artificial intelligence*, McGraw-Hill Book Co., New York, NY [pp. 165-168].
- Nilsson, N. J. [June 1979], *A production system for automatic deduction*, Machine Intelligence 9, Chichester, England (to appear).
- 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. 241-252.
- Wilkins, D. [1973], *QUEST--a non-clausal theorem proving system*, M.Sc. thesis, University of Essex, England.