KNOWLEDGE AND REASONING IN PROGRAM SYNTHESIS

Technical Note 98

November 1974

By: Zohar Manna
       Applied Mathematics Department
       Weizmann Institute of Science
       Rehovot, Israel

       Richard Waldinger
       Artificial Intelligence Center
       Stanford Research Institute
       Menlo Park, California   94025

KNOWLEDGE AND REASONING IN PROGRAM SYNTHESIS

ZOHAR MANNA

Applied Mathematics Department
Weizmann Institute of Science
Rehovot, Israel

and

RICHARD WALDINGER

Artificial Intelligence Center
Stanford Research Institute
Menlo Park, California, U. S. A.

ABSTRACT

Program synthesis is the construction of a computer program from given specifications. An automatic program synthesis system must combine reasoning and programming ability with a good deal of knowledge about the subject matter of the program. This ability and knowledge must be manifested . both procedurally (by programs) and structurally (by choice of representation).

We describe some of the reasoning and programming capabilities of a projected synthesis system. Special attention is paid to the introduction of conditional tests, loops, and instructions with side effects in the program being constructed. The ability to satisfy several interacting goals simultaneously proves to be important in many contexts. The modification of an already existing program to solve a somewhat different problem has been found to be a powerful approach.

We illustrate these concepts with hand simulations of the synthesis of a number of pattern-matching programs. Some of these techniques have already been implemented, some are in the course of implementation, while others seem equivalent to well-known unsolved problems in artificial intelligence.

TABLE OF CONTENTS

I. INTRODUCTION

In this paper we describe some of the knowledge and the reasoning ability that a computer system must have in order to construct computer programs automatically. We believe that such a system needs to embody a relatively small class of reasoning and programming tactics combined with a great deal of knowledge about the world. These tactics and this knowledge are expressed both procedurally (i.e., explicitly in the description of a problem-solving process) and structurally (i.e., implicitly in the choice of representation). We consider the ability to reason as central to the program synthesis process, and most of this paper is concerned with the incorporation of common-sense reasoning techniques into a program synthesis system. However, symbolic reasoning alone will not suffice to synthesize complex programs; therefore other techniques are necessary as well, such as

- The construction of "almost correct" programs that must be debugged (cf. Sussman [1973]).
- The modification of an existing program to perform a somewhat different task (cf. Balzer [1972]).
- The use of "visual" representations to reduce the need for deduction (cf. Bundy [1973]).

We regard program synthesis as a part of artificial intelligence. Many of the abilities we require of a program synthesizer, such as the ability to represent knowledge or to draw common-sense conclusions from facts, we would also expect from a natural language understanding system or a robot problem solver. These general problems have been under study by researchers for many years, and we do not expect that they will all be solved in the near future. However, we still prefer to address those problems rather than restrict ourselves to a more limited program synthesis system without those abilities.

Thus, although implementation of some of the techniques in this paper has already been completed, others require further development before a complete implementation will be possible. We imagine the knowledge and reasoning tactics of the system to be expressed in a PLANNER-type language (Hewitt [1972]); our own implementation is in the QLISP language (Reboh and Sacerdoti [1973]). Further details on the implementation are discussed in Section V-A.

Part II of the paper gives the basic techniques of reasoning for program synthesis. They include the formation of conditional tests and loops, the satisfaction of several simultaneous goals, and the handling of instructions with side effects. Part III applies the techniques of Part II to synthesize a nontrivial "pattern-matcher" that determines if a given expression is an instance of a given pattern. Part IV demonstrates the modification of programs. We take the pattern matcher we have constructed in Part III and adapt it to construct a more complex program: a "unification algorithm" that determines if two patterns have a common instance. In Part V we give some of the historical background of automatic program synthesis, and we compare this work with other recent efforts.

II. FUNDAMENTAL REASONING

In this section we will describe some of the reasoning and pro-
gramming tactics that are basic to the operation of our proposed syn-
thesizer. These tactics are not specific to one particular domain;
they apply to any programming problem. In this class of tactics, we
include the formation of program branches and loops and the handling
of statements with side effects.

A.   Specification and Tactics Language

We must first say something about how programming problems are
to be specified. In this discussion we consider only correct and
exact specifications in an artificial language. Thus, we will not
discuss input-output examples (cf. Green et al. [1974], Hardy [1974]),
traces (cf. Biermann et al. [1973]), or natural language descriptions
as methods for specifying programs; nor will we consider interactive
specification of programs (cf. Balzer [1972]). Neither are we limit-
ing ourselves to the first-order predicate calculus (cf. Kowalski
[1974]). Instead, we try to introduce specification constructs that
allow the natural and intuitive description of programming problems.
We therefore include constructs such as

        Find x such that $P(x)$
and the ellipsis notation, e.g.,

        A[1], A[2], ... , A[n] .
Furthermore, we introduce new constructs that are specific to certain
subject domains. For instance, in the domain of sets we use

        {x| $P(x)$}
for "the set of all x such that $P(x)$". As we introduce an example we
will describe features of the language that apply to that example.
Since the specification language is extendible, we can introduce new
constructs at any time.

We use a separate language to express the system's knowledge and
reasoning tactics. In the paper, these will be expressed in the form
of rules written in English. In our implementation, the same rules
are represented as programs in the QLISP programming language. When
a problem or goal is presented to the system, the appropriate rules
are summoned by "pattern-directed function invocation" (Hewitt [1972]).

In other words, the form of the goal determines which rules are applied.

In the following two sections we will use a single example, the synthesis of the set-theoretic union program, to illustrate the formation both of conditionals and of loops. The problem here is to compute the union of two finite sets, where sets are represented as lists with no repeated elements.

Given two sets, s and t, we want to express

union(s t) = {x|x∈s or x∈t}

in a LISP-like language. We expect the output of the synthesized program to be a set itself. Thus

union((A B) (B C)) = (A B C).

We do not regard the expression {x|x∈s or x∈t} itself as a proper program: the operator { |...} is a construct in our specification language but not in our LISP-like programming language. We assume that the programming language does have the following functions:

head(ℓ) = the first element of the list ℓ.

Thus head((A B C D)) = A.

tail(ℓ) = the list of all but the first element of the list ℓ.

Thus tail((A B C D)) = (B C D).*

add(x s)= the set consisting of the element x and the elements of the set s.

Thus add(A (B C D)) = (A B C D)

whereas add(B (B C D)) = (B C D).

empty(s) is true if s is the empty list, and

false otherwise.

Our task is to transform the specifications for union into an equivalent algorithm in this programming language.

We assume the system has some basic knowledge about sets, such as the following rules:

(1)   x ∈ s   is false if empty(s).

(2)   s is equal to add(head(s) tail(s)) if ~empty(s).

(3)   x ∈ add(s t)   is equivalent to (x = s or x ∈ t).

(4)   {x|x ∈ s} is equal to s.

---

* Since sets are represented as lists, head and tail may be applied to sets as well as lists. Their value then depends on our actual choice of representation.

(5)   {x|x=a or Q(x)} is equal to <u>add</u>(a {x|Q(x)}).

We also assume that the system knows a considerable amount of propo-
sitional logic, which we will not mention explicitly.

Before proceeding with our example we must discuss the formation
of conditional expressions.

B.   <u>Formation of Conditional Expressions</u>

In addition to the above constructs, we assume that our programm-
ing language contains conditional expressions of the form

        (if p then q else r) = r if p is false

                              q otherwise.

The conditional expression is a technique for dealing with uncertain-
ty. In constructing a program, we want to know if condition p is true
or not, but in fact p may be true on some occasions and false on others,
depending on the value of the argument. The human programmer faced
with this problem is likely to resort to "hypothetical reasoning": he
will assume p is false and write a program r that solves his problem in
that case; then he will assume p is true and write a program q that
works in that case; he will then put the two programs together into a
single program

        (if p then q else r).

Conceptually he has solved his problem by splitting his world into two
worlds: the case in which p is true and the case in which p is false.
In each of these worlds, uncertainty is reduced. Note that we must
be careful that the condition p on which we are splitting the world
is computable in our programming language; otherwise, the conditional
expression we construct also will not be computable (cf. Luckham and
Buchanan [1974]).

We can now proceed with the synthesis of the union function. Our
specifications were

        <u>union</u>(s t) = {x|x ∈ s or x ∈ t}.

We begin to transform these specifications using our rules. Rule (1)
applies to the subexpression x ∈ s, generating a subgoal, <u>empty</u>(s). We
cannot prove s in empty - this depends on the input -- and therefore
this is an occasion for a hypothetical world split. (We know that
<u>empty</u>(s) is a computable condition because <u>empty</u> is a primitive in
our language.) In the case in which s is empty, the expression

$\{x | x \in s \text{ or } x \in t\}$

therefore reduces to

$\{x | \text{ false or } x \in t\}$,

or, by propositional logic,

$\{x | x \in t\}$.

Now rule (4) reduces this to t, which is one of the inputs to our program and therefore is itself an acceptable program segment in our language.

In the other world--the case in which s is not empty--we cannot solve the problem without resorting to the recursive loop formation mechanism, which is the subject of the next section. However, we know at this point that the program will have the form

$\underline{\text{union}}(s \ t) = \text{if } \underline{\text{empty}}(s)$

then t

else ....

where the else clause will be whatever program segment we construct for the case in which s is not empty.

C.   Formation of Loops

The term "loop" includes both iteration and recursion; however, in this paper we will only discuss recursive loops (cf. Manna and Waldinger [1971]). Intuitively, we form a recursive call when, in the course of working on our problem, we generate a subgoal that is identical in form to our top-level goal. For instance, suppose our top-level goal is to construct the program $\underline{\text{reverse}}(\ell)$, that reverses the elements of the list ℓ (e.g., $\underline{\text{reverse}}(A \ (B \ C) \ D) = (D \ (B \ C) \ A)$). If in the course of constructing this program we generate the subgoal of reversing the elements of the list $\underline{\text{tail}}(\ell)$, we can use the program we are constructing to satisfy this subgoal. In other words we can introduce a recursive call $\underline{\text{reverse}}(\underline{\text{tail}}(\ell))$ to solve the subsidiary problem. We must always check that a recursive call doesn't lead to an infinite recursion. No such infinite loop can occur here because the input $\underline{\text{tail}}(\ell)$ is "shorter" than the original input ℓ.

Let us see how this technique applies to our union example. Continuing where we left off in the discussion of conditionals, we attempt to expand the expression

$$\{x | x \in s \text{ or } x \in t\}$$

in the case in which s is not empty. Applying rule (2) to the sub-expression s, we can expand our expression to

$$\{x | x \in \underline{\text{add}}(\underline{\text{head}}(s) \ \underline{\text{tail}}(s)) \text{ or } x \in t\}.$$

This is transformed by rule (3) into

$$\{x | x = \underline{\text{head}}(s) \text{ or } x \in \underline{\text{tail}}(s) \text{ or } x \in t\}.$$

Using rule (5), this reduces to

$$\underline{\text{add}}(\underline{\text{head}}(s) \ \{x | x \in \underline{\text{tail}}(s) \text{ or } x \in t\}).$$

If we observe that

$$\{x | x \in \underline{\text{tail}}(s) \text{ or } x \in t\}$$

is an instance of the top-level subgoal, we can reduce it to

$$\underline{\text{union}}(\underline{\text{tail}}(s) \ t).$$

Again, this recursive call leads to no infinite loops, since tail(s) is shorter than s. Our completed union program is now

$$\underline{\text{union}}(s \ t) = \text{if } \underline{\text{empty}}(s)$$
$$\text{then } t$$
$$\text{else } \underline{\text{add}}(\underline{\text{head}}(s) \ \underline{\text{union}}(\underline{\text{tail}}(s) \ t)).$$

As presented in this section, the loop formation technique can only be applied if a subgoal is generated that is a special case of the top-level goal. We shall see in the next section how this restriction can be relaxed.

D. Generalization of Specifications

When proving a theorem by mathematical induction, it is often necessary to strengthen the theorem in order for the induction to "go through." Even though we have an apparently more difficult theorem to prove, the proof is facilitated because we have a stronger induction hypothesis. For example, in proving theorems about LISP programs, the theorem prover of Boyer and Moore [1973] often automatically generalizes the statement of the theorem in the course of a proof by induction.

A similar phenomenon occurs in the synthesis of a recursive program. It is often necessary to strengthen the specifications of a program in order for that program to be useful in recursive calls. We believe that this ability to strengthen specifications is an essential part of the synthesis process, as many of our examples will show.

For example, suppose we want to construct a program to reverse a list. A good recursive reverse program is

reverse(ℓ) = rev(ℓ ())

where

rev(ℓ m) = if empty(ℓ)

        then m

        else rev(tail(ℓ) head(ℓ)·m).

Here

    () is the empty list

    x·ℓ is the list formed by inserting x before the first

        element of the list ℓ (e.g., A·(B C D) = (A B C D)).

Note that rev(ℓ m) reverses the list ℓ and appends it onto the list m, e.g.,

rev((A B C) (D E)) = (C B A D E).

This is a good way to compute reverse: it uses very primitive LISP functions and its recursion is such that it can be compiled without use of a stack. However, writing such a program entails writing the function rev, which is apparently more general and difficult to compute than reverse itself, since it must reverse its first argument as a subtask. The synthesis of this reverse function involves generalizing the original specifications of reverse into the specifications of rev.

The reverse function requires that the top-level goal be generalized in order to match the lower level goal. Another way to strengthen the specifications is to propose additional requirements for the program being constructed. For instance, suppose in the course of the synthesis of a function f(x), we generate a subgoal of the form P(f(a)), where f(a) is a particular recursive call. Instead of proving P(f(a)), it may be easier to strengthen the specifications for f(x) so as to also satisfy P(f(x)) for all x. This step may require that we actually modify portions of the program f that have already been synthesized in order to satisfy the new specification P. The recursive call to the modified program will then be sure to satisfy P(f(a)). This process will be illustrated in more detail during the synthesis of the pattern matcher in Part III.

E.   Conjunctive Goals

The problem of solving conjunctive goals is the problem of synthe-
sizing a program that satisfies several constraints simultaneously.
The general form for this problem is

Find z such that P(z) and Q(z).

The conjunctive goals problem is difficult because, even if we
have methods for solving the goals

Find z such that P(z)

and

Find z such that Q(z)

independently, the two solutions may not merge together nicely into
a single solution. Moreover, there seems to be no way of solving
the conjunctive goal problem in general; a method that works on one
such problem may be irrelevant to another.

We will illustrate one instance of the conjunctive goals problem:
the solution of two simultaneous linear equations. Although this prob-
lem is not itself a program synthesis problem, it could be rephrased
as a synthesis problem. Moreover the difficulties involved and the
technique to be applied extend also to many real synthesis problems,
such as the pattern-matcher synthesis of Part III. Suppose our prob-
lem is the following:

Find $\langle z_1, z_2 \rangle$ such that
$2z_1 = z_2 + 1$ and
$2z_2 = z_1 + 2$.

Suppose further that although we can solve single linear equations
with ease, we have no built-in package for solving sets of equations
simultaneously. We may try first to find a solution to each equation
separately. Solving the first equation, we might come up with

$\langle z_1, z_2 \rangle = \langle 1,1 \rangle$,

whereas solving the second equation might give

$\langle z_1, z_2 \rangle = \langle 2,2 \rangle$.

There is no way of combining these two solutions. Furthermore, it
doesn't help matters to reverse the order in which we approach the
two subgoals. What is necessary is to make the solution of the first
goal as general as possible, so that some special case of the solu-
tion might satisfy the second goal as well. For instance, a "general"

solution to the first equation might be

$$\langle 1 + w, 1 + 2w \rangle \text{ for any } w.$$

This solution is a generalization of our earlier solution $\langle 1,1 \rangle$. The problem is how to find a special case of the general solution that also solves the second equation. In other words, we must find a w such that

$$2(1 + 2w) = (1 + w) + 2.$$

This strategy leads us to a solution.

Of course the method of generalization does not apply to all conjunctive goal problems. For instance, the synthesis of an integer square-root program has specifications

Find z such that

z is an integer and

$z^2 \leqslant x$ and

$(z + 1)^2 > x$,

where $x \geqslant 0$.

The above approach of finding a general solution to one of the conjuncts and plugging it into the others is not effective in this case.

F.   Side Effects

Up to now we have been considering programs in a LISP-like language: these programs return a value but effect no change in any data structure. In the next two sections we will consider the synthesis of programs with "side effects" that may modify the state of the world.

For instance, a LISP-like program to sort two variables x and y would return as its value a list of two numbers, either (x y) or (y x), without altering the contents of x and y. On the other hand, a program with side effects to sort x and y might change the contents of x and y.

In order to indicate that a program with side effects is to be constructed, we provide a specification of form

Achieve P.

This construct means that the world is to be changed so as to make P true. For instance, if we specify a program

Achieve x = y,

we intend that the program actually change the value of x or y, say by an assignment statement. However, if we specify

Find x such that x = y,

the program constructed would return the value of y, but would not change the value of x or y.

Many of the techniques we used in the synthesis of LISP-like programs also apply to the construction of programs with side effects. In particular, we can use pattern-directed function invocation to retrieve tactical knowledge. The synthesis of the program in the following example has the same flavor as our earlier union example, but involves the introduction of side effects.

The program sort(x y) to be constructed is to sort the values of two variables x and y. For simplicity we will use the statement interchange(x y) to exchange the values of x and y, instead of the usual sequence of assignment statements. Our specification will be simply
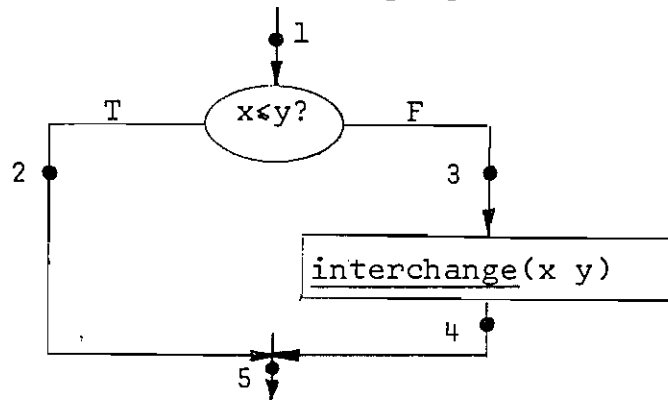
Achieve x ≤ y.

Strictly speaking, we should include in the specification the additional requirement that the set of values of x and y after the sort should be the same as before the sort. However, we will not consider such compound goals until section H, and we can achieve the same effect by requiring that the interchange statement be the only instruction with side effects that appears in the program.

The first step in achieving a goal is to see if it is already true. (If a goal is a theorem, for instance, we do not need to construct a program to achieve it.) We cannot prove x≤y, but we can use it as a basis for a hypothetical world split. This split corresponds to a conditional expression in the program being constructed. In flowchart notation the conditional expression is written as a program branch:
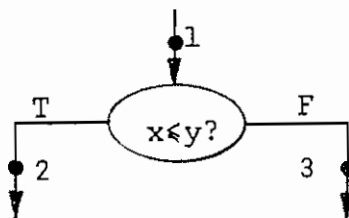
At point 2 our goal is already achieved. At point 3 we know that
~(x≤y), i.e., x>y. To achieve x≤y, it suffices to establish x<y,
but this may be achieved by executing interchange(x y). Thus we
have x≤y in both worlds, and the final program is therefore:

```
                        │ 1
                        ▼
         T       ╭─────────╮       F
   ┌─────────────│  x≤y?   │─────────────┐
   │             ╰─────────╯             │
 2 ●                                   3 ●
   │                                     │
   │                                     ▼
   │                      ┌───────────────────────┐
   │                      │   interchange(x y)    │
   │                      └───────────────────────┘
   │                                   4 ●
   │                                     │
   └──────────────────┬──────────────────┘
                    5 ●
                      ▼
```

    This example introduced no difficulties that our LISP-like pro-
gram synthesis techniques could not handle. However, in general, pro-
grams with side effects must be given special treatment because of the
necessity for representing changes in the world. It is important to
be able to determine whether a given assertion is always true at a
given point in a program. To this end we study the relationship be-
tween assertions and program constructs in the next section.

G. Assertions and Program Constructs

    Suppose a program contains an assignment statement

```
           1 │
             ▼
       ┌───────────┐
       │   x←y     │
       └───────────┘
           2 ●
             ▼
```
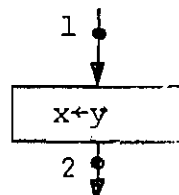
and we wish to determine if x≤3 at point 2. In order to do this it
suffices to check if what we know at point 1 implies that y≤3. In
general, to determine an assertion of form P(x) at point 2, check
P(y) at point 1. We will say that the assertion P(y) is the result
of "passing back" the assertion P(x) from point 2 to point 1. (This is
precisely the process outlined by Floyd [1967] and Hoare [1969] -
see also Manna [1974] - in reference to program verification.)

    Furthermore, if our program contains the instruction
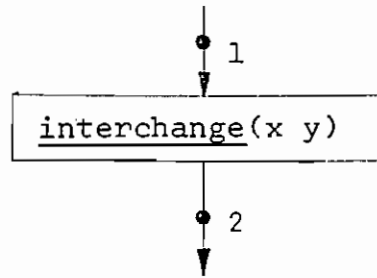
and we wish to establish x≤y at point 2 we must check if y≤x at
point 1.   In general, an assertion of form P(x y) results in an
assertion of form P(y x) when passed back over interchange(x y).

Suppose the program being constructed contains a branch
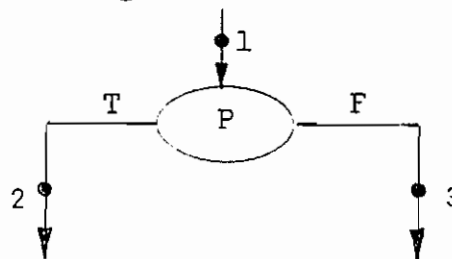
To determine if an assertion Q is true at point 2, it suffices to
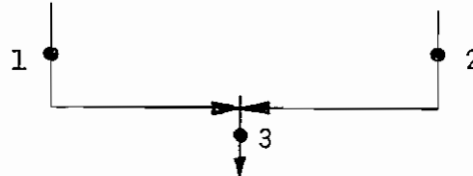check whether

        Q if P

(i.e., $P \supset Q$) is true at point 1.   In order to determine if R is
true at point 3, it suffices to check whether

        R if ~P

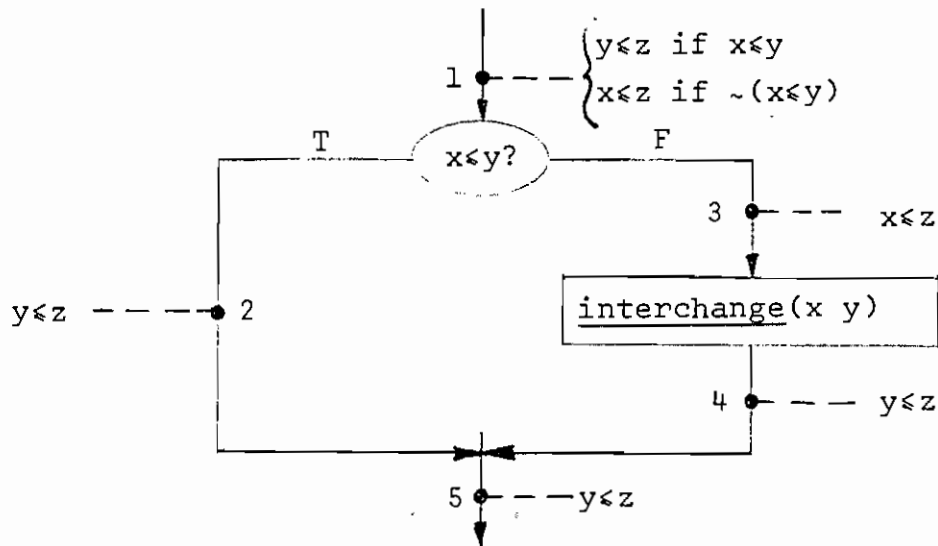(i.e., $\sim P \supset R$) is true at point 1.

Suppose two control paths join in the program being constructed.

Thus to determine if assertion P is true at point 3, it is sufficient
to check that P be true at both point 1 and point 2.

Assertions may be passed back over complex programs.   For in-
stance, let us pass the assertion y≤z back over the program sort(x y)
which we constructed in the previous section.

By combining the methods that we have just introduced for passing assertions back over program constructs, we can see that in order to establish y≤z at point 5, it is necessary to check that (y≤z if x≤y) and (x≤z if ~(x≤y)) are true at point 1.

Often the specification of a program will require the simultaneous satisfaction of more than one goal. As in the case of conjunctive goals in LISP-like programs, the special interest of this problem lies in the inter-relatedness of the goals. The techniques of this section will now be applied to handle the interaction between goals.

H.   Simultaneous Goals

A simultaneous goal problem has the form

    Achieve P and Q.

Sometimes P and Q will be independent conditions, so that we can achieve P and Q simply by achieving P and then achieving Q. For example, if our goal is

    Achieve x = 2 and y = 3,

the two goals x=2 and y=3 are completely independent. In this section, however, we will be concerned with the more complex case in which P and Q interact. In such a case we may make P false in the course of achieving Q.

Consider for example the problem of sorting three variables x,y, and z. We will assume that the only instruction we can use is the subroutine sort(u v), described in the previous section, which sorts

two variables. Our goal is then

Achieve $x \leqslant y$ and $y \leqslant z$.

We know that the program sort(u v) will achieve a goal of form $u \leqslant v$.
If we apply the straightforward technique of achieving the conjunct
$x \leqslant y$ first, and then the conjunct $y \leqslant z$, we obtain the program

sort(x y)

sort(y z).

However, this program has a bug in that sorting y and z may disrupt the
relation $x \leqslant y$: if z is initially the smallest of the three, in inter-
changing y and z we make y less than x. Reversing the order in which
the conjuncts are achieved is useless in this case.

There are a number of ways in which this problem may be resolved.
One of them involves the notion of program modification. (cf. Sussman
[1973]). The general strategy is as follows: to achieve P and Q simul-
taneously, first write a program to achieve P; then modify that program
to achieve Q as well. The essence of this strategy, then, lies in a
technique of program modification.

Let us see how this strategy applies to the simple sort problem.
The specification is

Achieve $x \leqslant y$ and $y \leqslant z$.

It is easy to achieve $x \leqslant y$; the program sort(x y) will do that immediately.
We must now modify the program sort(x y) to achieve $y \leqslant z$ without disturb-
ing the relation $x \leqslant y$ we have just achieved. In other words, we would
like to "protect" the relation $x \leqslant y$. We have seen that simply achieving
$y \leqslant z$ after achieving $x \leqslant y$ is impossible without disturbing the protected
relation. Therefore we will pass the goal $y \leqslant z$ back to the beginning of
the program sort(x y) and try to achieve it there, where there are no
protected relations.

We have seen in the previous section that the goal $y \leqslant z$ passed back
before the program sort(x y) results in two goals

(i) $y \leqslant z$ if $x \leqslant y$

and

(ii) $x \leqslant z$ if $\sim(x \leqslant y)$.

Both of these goals must be achieved before applying sort(x y). We can
achieve (i) by applying sort(y z). (This will achieve $y \leqslant z$ whether or not
$x \leqslant y$.) Our program so far is thus
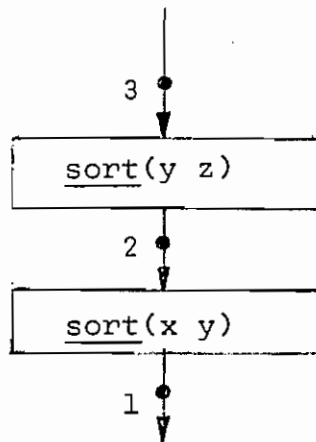
We still need to achieve goal (ii) at point 2; however, we will pass
this goal back to point 3 so as not to endanger our protected relation
(y≤z if x≤y) at point 2, which we have just achieved. Passing goal
(ii) back to point 3, using the techniques of Section G, yields two
more goals

    (iii) x ≤ z if [~(x ≤ y) and y ≤ z]

and

    (iv) x ≤ y if [~(x ≤ z) and ~(y ≤ z)].

Both of these goals must be achieved at point 3. We can achieve (iii)
simply by inserting the instruction sort(x z) before point 3.
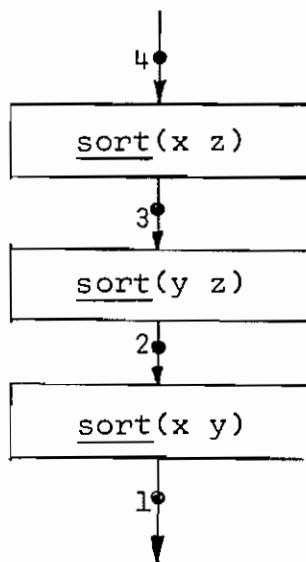
Apparently we must still achieve (iv) at point 3. However, since we have just executed <u>sort</u>(x z), we know x≤z at point 3, and so goal (iv) is already satisfied vacuously. Thus the desired program is

<u>sort</u>(x z)

<u>sort</u>(y z)

<u>sort</u>(x y).

If the subgoals are pursued in a different order, different variations on this program are obtained.

The program modification strategy seems to be a fairly general approach to the simultaneous goal problem. It also is a powerful program synthesis technique in general, as we will see when we develop the unification algorithm in Part IV.

This concludes the presentation of our basic program synthesis techniques. In the next part we will show how these same techniques work together in the synthesis of a more complex example.

III. PROGRAM SYNTHESIS: THE PATTERN-MATCHER

We will present the synthesis of a simple pattern-matcher to show how the concepts discussed in the previous section can be applied to a non trivial problem. Later, in Part IV, we shall show how we can construct a more complex program, the unification algorithm of Robinson [1965], by modifying the program we are about to synthesize. We must first describe the data structures and primitive operations involved in the pattern-matching and unification problems.

A.   Domain and Notations

The main objects in our domain are expressions and substitutions.

1.   Expressions

Expressions are atoms or nested lists of atoms; e.g., (A B (X C) D) is an expression. An atom may be either a variable or a constant. (In our examples we will use A,B,C,... for constants and U,V,W,... for variables.) We have basic predicates atom, var and const to distinguish these objects:

$$\text{atom}(\ell) \equiv \ell \text{ is an atom,}$$
$$\text{var}(\ell) \equiv \ell \text{ is a variable,}$$

and $\text{const}(\ell) \equiv \ell$ is a constant.

To decompose an expression, we will use the primitive functions head(ℓ) and tail(ℓ), defined when ℓ is not an atom.

head(ℓ) is the first element of ℓ,

tail(ℓ) is the list of all but the first element of ℓ.

Thus

$$\text{head}(((A\ (X)\ B)\ C\ (D\ X))) = (A\ (X)\ B),$$
$$\text{tail}(((A\ (X)\ B)\ C\ (D\ X))) = (C\ (D\ X)).$$

We will abbreviate head(ℓ) as ℓ_1 and tail(ℓ) as ℓ_2.

To construct expressions we have the '·' function: if ℓ is any expression and m is a nonatomic expression, $\ell \cdot m$ is the expression with ℓ inserted before the first element of m. For example

$$A \cdot (B\ C\ D) = (A\ B\ C\ D),$$
$$(A\ (X)\ B) \cdot (C\ (D\ X)) = ((A\ (X)\ B)\ C\ (D\ X)).$$

The predicate occursin(x ℓ) is true if x is an atom that occurs in expression ℓ at any level, e.g.,

occursin(A (C (B (A) B) C)) is true,

but

occursin(X Y) is false.

Finally, we will introduce the predicate constexp(ℓ), which is true if ℓ is made up entirely of constants. Thus

constexp((A (B) C (D E))) is true,

but

constexp(X) is false.

Note that constexp differs from const in that constexp may be true for nonatomic expressions.

2.   Substitutions

A substitution replaces certain variables of an expression by other expressions. We will represent a substitution as a list of pairs. Thus

(<X (A B)> <Y (C Y)>)

is a substitution.

The instantiation function inst(s ℓ) applies the substitution s to the expression ℓ. For example, if

s = (<X (A B)> <Y (C Y)>)

and

ℓ = (X (A Y) X)

then

inst(s ℓ) = ((A B) (A (C Y)) (A B)).

Note that the substitution is applied by first replacing all occurrences of X simultaneously by (A B) and then all occurrences of Y simultaneously by (C Y). Thus, if

s' = (<X Y> <Y C>),

then

inst(s' ℓ) = (C (A C) C).

The empty substitution Λ is represented by the empty list of pairs. Thus, for any expression ℓ,

inst(Λ ℓ) = ℓ.

We regard two substitutions s_1 and s_2 as equal (written $s_1 = s_2$) if and only if

inst(s_1 ℓ) = inst(s_2 ℓ)

for every expression ℓ. Thus

        (<X Y> <Y C>)

and

        (<X C> <Y C>)

are regarded as equal substitutions.

    We can build up substitutions by using the functions pair and °
(composition):   If v is a variable and t an expression, pair(v t) is
the substitution that replaces v by t; i.e.,

        pair(v t) = (<v t>).

If s_1 and s_2 are two substitutions, $s_1°s_2$ is the substitution with
the same effect as applying s_1 followed by s_2.   Thus

        inst($s_1°s_2$ ℓ) = inst(s_2 inst(s_1 ℓ)).

For example, if

        s_1 = (<X A> <Y B>)

and

        s_2 = (<Z C> <X D>)

then

        $s_1°s_2$ = (<X A> <Y B> <Z C>).

Note that for the empty substitution Λ

        Λ°s = s°Λ = s

for any substitution s.

B.   The Specifications

    The problem of pattern-matching may be described as follows.   We
are given two expressions, pat and arg.   While pat can be any expres-
sion, arg is assumed to contain no variables; i.e., constexp(arg) is
true.   We want to find a substitution z that transforms pat into arg,
i.e., such that

        inst(z pat) = arg.

We will call such a substitution a match.   If no match exists, we want
the program to return the distinguished constant NOMATCH.   For example,
if

        pat is (X A (Y B))

and

        arg is (C A (D B)),

we want the program to find the match

        (<X C> <Y D>).

On the other hand, if

> pat is (X A (X B))

and

> arg is (B A (D B)),

then no substitution will transform pat into arg, because X cannot be matched with both B and D. Therefore the program should yield NOMATCH.

This version of the pattern-matcher is simpler than the pattern-matching algorithms usually implemented in programming languages because of the absence of "sequence" or "fragment" variables. Our variables must match exactly one expression, whereas a fragment variable may match any number of expressions. Thus if

> pat is (X Y Z) and
>
> arg is ((A B) C (A B)),

X and Z must be bound to (A B), and Y must be bound to C. If

> pat is (X Y) and
>
> arg is (A B C),

no match is possible at all, while if X and Y were fragment variables, four matches would be possible.

In mathematical notation the specifications for our pattern-matcher are:

Goal 1:

> match(pat arg) =
> Find z such that inst(z pat) = arg
> else z = NOMATCH

where "Find z such that P(z) else Q(z)" means find a z such that P(z) if one exists; otherwise, find a z such that Q(z).

The above specifications do not completely capture our intentions; for instance, if

> pat is (X Y), and
>
> arg is (A B),

then the substitution

> z = (<X A> <Y B> <Z C>)

will satisfy our specifications as well as

> z = (<X A> <Y B>).

We have neglected to include in our specifications that no substitutions should be made for variables that do not occur in pat. We will call a match that satisfies this additional condition a most general match.

An interesting characteristic of the synthesis we present is that even if the user does not require that the match found be most general, the system will strengthen the specifications automatically to imply this condition, using the method outlined in Section II-D. Therefore we will begin the synthesis using the weaker specifications.

C.   The Synthesis: The Base Cases

Rather than list all the knowledge we require in a special section at the beginning, we will mention a rule only when it is about to be used. Furthermore, if a rule seems excessively trivial we will omit it entirely. The general strategy is to first work on

Goal 2:   Find z such that inst(z pat) = arg.

If this is found to be impossible (i.e., if it is proven that no such z exists), we will work on

Goal 3:   Find a z such that z = NOMATCH;

which is seen to be trivially satisfied by taking z to be NOMATCH.

Thus, from now on we will be working primarily on Goal 2. However, in working on any goal we devote a portion of our time to showing that the goal is impossible to achieve. When we find cases in which Goal 2 is proven impossible, we will automatically return NOMATCH, which satisfies Goal 3.

We have in our knowledge base a number of rules concerning inst, including

Rule 1:   inst(s x) = x for any substitution s
                if constexp(x)

Rule 2:   inst(pair(v t) v) = t
                if var(v).

We assume that these rules are retrieved by pattern-directed function invocation on Goal 2. Rule 1 applied only in the case that constexp(pat) and pat = arg. We cannot prove either of these conditions; their truth or falsehood depends on the particular inputs to the program. We use these predicates as conditions for a hypothetical world-split. In the case that both of these conditions are true, Rule 1 tells us that any substitution is a satisfactory match. We will have occasion to tighten the specifications of our program later; as they stand now, we will simply return "any", so as not to restrict

our choice. The portion of the program we have constructed so far
reads   match(pat arg) =
                if constexp(pat)
                then if pat = arg
                        then any
                        else ...
                else ...

On the other hand, in the case constexp(pat) and pat ≠ arg, Rule
1 tells us that

        inst(z pat) ≠ arg

for any z. Hence we are led to try to satisfy Goal 3 and return
NOMATCH.

We now consider the case

        ~constexp(pat).

Rule 2 establishes the subgoal

        var(pat).

This is another occasion for a hypothetical world-split. When var(pat)
is true, the program must return pair(pat arg); the program we have
constructed so far is

        match(pat arg) =
                if constexp(pat)
                then if pat = arg
                        then any
                        else NOMATCH
                else if var(pat)
                        then pair(pat arg)
                        else ....

Hencefore we assume ~var(pat).   Recall that we have been assuming also
that ~constexp(pat).   To proceed we make use of the following addition-
al knowledge about the function inst:

Rule 3: inst(s x·y) = inst(s x)·inst(s y) for any substitution s.

This rule applies to our Goal 2 if pat = x·y for some expressions x
and y. We have some additional knowledge about expressions in gene-
ral:

Rule 4: $u = u_1 \cdot u_2$
          if ~atom(u).

Recall that u_1 is an abbreviation for <u>head</u>(u) and u_2 is an abbreviation for <u>tail</u>(u).

<u>Rule 5</u>: u ≠ v·w for any v and w

if <u>atom</u>(u).

Using Rule 4, we generate a subgoal

~<u>atom</u>(<u>pat</u>).

Since we have already assumed ~<u>constexp</u>(<u>pat</u>) and ~<u>var</u>(<u>pat</u>), we can actually prove ~<u>atom</u>(<u>pat</u>) using knowledge in the system. Therefore <u>pat</u>=<u>pat</u>$_1$·<u>pat</u>$_2$ and using Rule 3 our Goal 2 is then reduced to

<u>Goal 4</u>:   Find z such that <u>inst</u>(z <u>pat</u>$_1$)·<u>inst</u>(z <u>pat</u>$_2$) = <u>arg</u>.

We now make use of some general list-processing knowledge.

<u>Rule 6</u>:   To prove x·y = u·v, prove x = u and y = v.

Applying this rule, we generate a subgoal to show that

<u>arg</u> = u·v

for some u and v. Applying Rule 4, we know this is true with u = <u>arg</u>$_1$ and v = <u>arg</u>$_2$ if

~<u>atom</u>(<u>arg</u>).

This is another occasion for a hypothetical world-split.

Thus, by Rule 6, in the case that ~<u>atom</u>(<u>arg</u>), our subgoal reduces to

<u>Goal 5</u>:   Find z such that

<u>inst</u>(z <u>pat</u>$_1$) = <u>arg</u>$_1$

and

<u>inst</u>(z <u>pat</u>$_2$) = <u>arg</u>$_2$.

We will postpone treatment of this goal until after we have considered the other case, in which

<u>atom</u>(<u>arg</u>)

holds. In this case Rule 5 tells us that

<u>inst</u>(z <u>pat</u>$_1$)·<u>inst</u>(z <u>pat</u>$_2$) ≠ <u>arg</u>

for any z. Hence, our goal is unachievable in this case, and we can return NOMATCH.

The program so far is

```
match(pat arg) =
     if constexp(pat)
     then if pat = arg
          then any
          else NOMATCH
     else if var(pat)
          then pair(pat arg)
          else if atom(arg)
               then NOMATCH
               else ...
```

For the as yet untreated case neither pat nor arg is atomic. Henceforth using Rule 4 we assume that pat is $pat_1 \cdot pat_2$ and arg is $arg_1 \cdot arg_2$.

D.   The Synthesis: The Inductive Case

We will describe the remainder of the synthesis in less detail, because the reader has already seen the style of reasoning we have been using. Recall that we had postponed our discussion of Goal 5 in order to consider the case in which arg is atomic. Now that we have completed our development of that case, we resume our work on Goal 5:

Find z such that
$\underline{inst}(z\ pat_1) = arg_1$, and
$\underline{inst}(z\ pat_2) = arg_2$.

This is a conjunctive goal, and is treated analogously to the goal in the simultaneous linear equations example (Section II-E): The system will attempt to solve the first conjunct, using a recursive call to the pattern-matcher itself.

The interaction between the two conjuncts is part of the challenge of this synthesis. It is quite possible to satisfy each conjunct separately without being able to satisfy them both together. For example, if pat = (X X) and arg = (A B) then pat_1 = X, pat_2 = (X), arg_1 = A and arg_2 = (B). Thus z = (<X A>) satisfies the first conjunct, z = (<X B>) satisfies the second conjunct, but no substitution will satisfy both conjuncts because no substitution can match X against both A and B. Some mechanism is needed to ensure that the expression assigned to a

variable in solving the first conjunct is the same as the expression
assigned to that variable in solving the second conjunct.

There are several ways to approach this difficulty. For instance,
the programmer may satisfy the two conjuncts separately and then at-
tempt to combine the two substitutions thereby derived into a single
substitution. Or he may actually replace those variables in pat_2
that also occur in pat_1 by whatever expressions they have been matched
against, before attempting to match pat_2 against arg_2. Or he may
simply pass the substitution that satisfied the first conjunct as a
third argument to the pattern-matcher in working on the second conjunct.
The pattern-matcher must then check that the matches assigned to vari-
ables are consistent with the substitution given as the third argument.

We will examine in this section how a system would discover the
second of these methods. A similar system could also discover the
third method. We will not consider the first method here because it
is not easily adapted to the unification problem.

Our strategy for approaching the conjunctive goal is as follows.
We will consider the first conjunct independently:

Goal 6: Find z such that $inst(z\ pat_1) = arg_1$.
If we find a z that satisfies this goal, we will substitute that z
into the second conjunct, giving

Goal 7: Prove $inst(z\ pat_2) = arg_2$.
If we are successful in Goal 7, we are done; however, if we fail, we
will try to generalize z. In other words, we will try to find a broader
class of substitutions that satisfy Goal 6 and from these select one
that also satisfies Goal 7. This is the method we introduced to solve
conjunctive goals in Section II-E.

Applying this strategy, we begin work on Goal 6. We first use a
rule that relates the construct

      Find z such that P(z)

to the construct

      Find z such that P(z) else Q(z).

Rule 7: To find z such that P(z),
      it suffices to find z_1 such that $P(z_1)$ else $Q(z_1)$
                                 for some predicate Q,
      and $\sim Q(z_1)$.

This rule, applied to Goal 6 causes the generation of the subgoal
Goal 8:   Find z_1 such that $\underline{inst}(z_1\ \underline{pat}_1) = \underline{arg}_1$ else $Q(z_1)$,
            and $\sim Q(z_1)$.
The first conjunct of this subgoal matches the top-level Goal 1, where
$Q(z_1)$ is z_1 = NOMATCH. This suggests establishing a recursion at this
point, taking

$\qquad z_1 = \underline{match}(\underline{pat}_1\ \underline{arg}_1)$.

Henceforth we will use z_1 as an abbreviation for $\underline{match}(\underline{pat}_1\ \underline{arg}_1)$.
Termination is easily shown, because both $\underline{pat}_1$ and $\underline{arg}_1$ are proper sub-
expressions of $\underline{pat}$ and $\underline{arg}$, respectively. It remains to show, accord-
ing to Rule 7, that $z_1 \neq$ NOMATCH. This causes another hypothetical
world-split: in the case z_1 = NOMATCH (i.e., no substitution will
cause $\underline{pat}_1$ and $\underline{arg}_1$ to match), we can show that no substitution can
cause $\underline{pat}$ and $\underline{arg}$ to match either, and hence can take z = NOMATCH.

   We have thus constructed the following new program segment

$\qquad z_1 \leftarrow \underline{match}(\underline{pat}_1\ \underline{arg}_1)$
$\qquad$ if z_1 = NOMATCH
$\qquad$ then NOMATCH
$\qquad$ else ...

We have used z_1 as a program variable to improve readability. The
actual program constructed would use $\underline{match}(\underline{pat}_1\ \underline{arg}_1)$ itself in place
of z_1.

   On the other hand, if $z_1 \neq$ NOMATCH, we know that z_1 satisfies the
first conjunct (Goal 6). Thus, in keeping with the conjunctive goal
strategy, we try to show that z_1 satisfies the second conjunct (Goal 7),
as well, i.e.,

$\qquad \underline{inst}(z_1\ \underline{pat}_2) = \underline{arg}_2$.

However, we fail in this attempt; in fact we can find sample inputs
$\underline{pat}$ and $\underline{arg}$ that provide a counter-example to Goal 7 (e.g., $\underline{pat}$=(A X),
$\underline{arg}$=(A B), z_1=A). Thus we go back and try to generalize our solution
to Goal 6.

   We already have a solution to Goal 6: we know $\underline{inst}(z_1\ \underline{pat}_1)=\underline{arg}_1$.
We also can deduce that $\underline{constexp}(\underline{arg}_1)$, because we have assumed
$\underline{constexp}(\underline{arg})$. Hence Rule 1 tells us that

$\qquad \underline{inst}(z_2\ \underline{arg}_1) = \underline{arg}_1$

for any substitution z_2. Therefore
$$\underline{\text{inst}}(z_2 \ \underline{\text{inst}}(z_1 \ \underline{\text{pat}}_1)) = \underline{\text{arg}}_1,$$
i.e.,
$$\underline{\text{inst}}(z_1{}^{\circ}z_2 \ \underline{\text{pat}}_1) = \underline{\text{arg}}_1$$
for any substitution z_2. Thus having one substitution z_1 that satisfies Goal 6, we have an entire class of substitutions, of form $z_1{}^{\circ}z_2$, each of which satisfies Goal 6. These substitutions may be considered to be "extensions" of z_1; although z_1 itself may not satisfy Goal 7, perhaps some extension of z_1 will.

The above reasoning is straightforward enough to justify, but further work is needed to motivate a machine to pursue it.

It remains now to find a single z_2 so that $z_1{}^{\circ}z_2$ satisfies Goal 7, i.e.,

Goal 9: Find z_2 such that $\underline{\text{inst}}(z_1{}^{\circ}z_2 \ \underline{\text{pat}}_2) = \underline{\text{arg}}_2$,
or equivalently,

Find z_2 such that $\underline{\text{inst}}(z_2 \ \underline{\text{inst}}(z_1 \ \underline{\text{pat}}_2)) = \underline{\text{arg}}_2$.
applying Rule 7, we establish a new goal

Goal 10: Find z_2 such that $\underline{\text{inst}}(z_2 \ \underline{\text{inst}}(z_1 \ \underline{\text{pat}}_2)) = \underline{\text{arg}}_2$ else $Q(z_2)$,
and $\sim Q(z_2)$

The first conjunct of this goal is an instance of our top-level goal, taking $\underline{\text{pat}}$ to be $\underline{\text{inst}}(z_1 \ \underline{\text{pat}}_2)$, $\underline{\text{arg}}$ to be $\underline{\text{arg}}_2$, and $Q(z_2)$ to be z_2 = NOMATCH. Thus we attempt to insert the recursive call
$$z_2 \leftarrow \underline{\text{match}}(\underline{\text{inst}}(z_1 \ \underline{\text{pat}}_2) \ \underline{\text{arg}}_2)$$
into our program at this point. (Again, the introduction of z_2 as a program variable is for notational simplicity.) However, we must first establish
$$\sim Q(z_2),$$
i.e.,
$$z_2 \neq \text{NOMATCH}.$$
We cannot prove this: it is true for some examples and false for others. Therefore we split on this condition.

In the case $z_2 \neq$ NOMATCH Goal 10 is satisfied. Thus z_2 also satisfies Goal 9, and $z = z_1{}^{\circ}z_2$ satisfies Goal 7.

Our program so far is

```
match(pat arg) =
    if constexp(pat)
    then if pat = arg
        then any
        else NOMATCH
    else if var(pat)
        then pair(pat arg)
        else if atom(arg)
            then NOMATCH
            else z₁ ← match(pat₁ arg₁)
                if z₁ = NOMATCH
                then NOMATCH
                else z₂ ← match(inst(z₁ pat₂) arg₂)
                    if z₂ = NOMATCH
                    then ...
                    else z₁°z₂.
```

E.   The Synthesis: The Strengthening of the Specifications

We have gone this far through the synthesis using the weak specifications, i.e., without requiring that the match found be most general. In fact, the match found may or may not be most general depending on the value taken for the unspecified substitution "any" produced in the very first case. The synthesis is nearly complete. However, we will be unable to complete it without strengthening the specifications and modifying the program accordingly. We now have only one case left to consider. This is the case in which

$$z_2 = \text{NOMATCH}$$

i.e.,

$$\text{match}(\text{inst}(z_1 \ pat_2) \ arg_2) = \text{NOMATCH}.$$

This means that no substitution w satisfies

$$\text{inst}(w \ \text{inst}(z_1 \ pat_2)) = arg_2,$$

or, equivalently

$$\text{inst}(z_1°w \ pat_2) \neq arg_2 \text{ for every substitution } w.$$

This means that no substitution of form $z_1°w$ could possibly satisfy

$$\text{inst}(z_1°w \ pat) = arg.$$

We here have a choice: we can try to find a substitution s not of form $z_1{}^\circ w$ that satisfies

$$\underline{inst}(s\ \underline{pat}_1) = \underline{arg}_1$$

and repeat the process; or we could try to show that only a substitution s of form $z_1{}^\circ w$ could possibly satisfy

$$\underline{inst}(s\ \underline{pat}_1) = \underline{arg}_1,$$

and therefore we can take z=NOMATCH.

Pursuing the latter course, we try to show that the set of substitutions s of form $z_1{}^\circ w$ is the entire set of solutions to

$$\underline{inst}(s\ \underline{pat}_1) = \underline{arg}_1.$$

In other words, we show that for any substitution s,

$$\text{if } \underline{inst}(s\ \underline{pat}_1) = \underline{arg}_1 \text{ then } s = z_1{}^\circ w \text{ for some } w.$$

This condition is equivalent to saying that z_1 is a most general match. We cannot prove this about z_1 itself; however, since z_1 is $\underline{match}(\underline{pat}_1\ \underline{arg}_1)$ it suffices to add the condition to the specifications for $\underline{match}$, as described in Section II-D. The strengthened specifications now read

```
Find z such that
      {inst(z pat) = arg and
        for all s [if inst(s pat) = arg
                    then s = z°w for some w]}
else z = NOMATCH.
```

Once we have strengthened the specifications it is necessary to go through the entire program and see that the new, stronger specifications are satisfied, modifying the program if necessary. In this case no major modifications are necessary; however, the assignment

$$z \leftarrow any$$

that occurs in the case in which $\underline{pat}$ and $\underline{arg}$ are equal and constant is further specified to read

$$z \leftarrow \Lambda.$$

Our final program is therefore

```
match(pat arg) =
    if constexp(pat)
    then if pat = arg
        then Λ
        else NOMATCH
    else if var(pat)
        then pair(pat arg)
        else if atom(arg)
            then NOMATCH
            else z₁ ← match(pat₁ arg₁)
                if z₁ = NOMATCH
                then NOMATCH
                else z₂ ← match(inst(z₁ pat₂) arg₂)
                    if z₂ = NOMATCH
                    then NOMATCH
                    else z₁°z₂.
```

The above pattern-matcher is only one of many pattern-matchers
that can be derived to satisfy the same specifications. In pursuing
the synthesis the system has made many choices; some of the alterna-
tive paths result in a failure to solve the problem altogether, where-
as other paths result in different, possibly better programs.

IV. PROGRAM MODIFICATION: THE UNIFICATION ALGORITHM

In general, we cannot expect a system to synthesize an entire com-
plex program from scratch, as in the pattern-matcher example. We would
like the system to remember a large body of programs that have been
synthesized before, and the method by which they were constructed.
When presented with a new problem, the system should check to see
if it has solved a similar problem before.   If so, it may be able
to adapt the technique of the old program to make it solve the new
problem.

There are several difficulties involved in this approach. First,
we cannot expect the system to remember every detail of every synthe-
sis in its history. Therefore, it must decide what to remember and
what to forget. Second, the system must decide which problems are
similar to the one being considered, and the concept of similarity
is somewhat ill-defined. Third, having found a similar program, the
system must somehow modify the óld synthesis to solve the new problem.
We will concentrate only on the last of these problems in this discus-
sion. We will illustrate a technique for program modification as ap-
plied to the synthesis of a version of Robinson's [1965] unification
algorithm.

A.   The Specifications

Unification may be considered to be a generalization of pattern-
matching in which variables appear in both pat and arg. The problem
is to find a single substitution (called a "unifier") that, when ap-
plied to both pat and arg, will yield identical expressions. For
instance, if

        pat = (X A)

and

        arg = (B Y),

then a possible unifier of pat and arg is

        (<X B> <Y A>).

The close analogy between pattern-matching and unification is
clear. If we assume that the system remembers the pattern-matcher we
constructed in Sections III-B through III-E and the goal structure
involved in the synthesis, the solution to the unification problem is
greatly facilitated.

The specifications for the unification algorithm, in mathematical notation, are

```
unify(pat arg) =
Find z such that inst(z pat) = inst(z arg)
else z = NOUNIFY
```

B.   The Analogy with the Pattern-Matcher

For purposes of comparison we rewrite the match specifications:

match(pat arg) =

Find z such that $inst(z\ pat) = arg$

else z = NOMATCH.

In formulating the analogy, we identify unify with match, pat with pat, the arg in unify(pat arg) with arg, and $inst(z\ arg)$ also with arg, NOMATCH with NOUNIFY. In accordance with this analogy, we must systematically alter the goal structure of the pattern-matcher synthesis. For example, Goal 5 becomes modified to read

Find z such that

$inst(z\ pat_1) = inst(z\ arg_1)$ and

$inst(z\ pat_2) = inst(z\ arg_2)$.

In constructing the pattern-matcher, we had to break down the synthesis into various cases. We will try to maintain this case structure in formulating our new program. Much of the savings derived from modifying the pattern-matcher instead of constructing the unification algorithm from scratch arises because we do not have to deduce the case splitting all over again.

A difficult step in the pattern-matcher synthesis involved the strengthening of the specifications for the entire program. We added the condition that the match found was to be "most general." In formulating the unification synthesis, we will immediately strengthen the specifications in the analogous way. The strengthened specifications read

```
unify(pat arg) =
Find z such that
   {inst(z pat) = inst(z arg) and
    for all s [if inst(s pat) = inst(s arg)
              then s = z°w for some w]}
else z = NOUNIFY.
```

Following Robinson, we will refer to a unifier satisfying the new condition as a "most general-unifier."

Note that this alteration process is purely syntactic; there is no reason to assume that the altered goal structure corresponds to a valid line of reasoning. For instance, simply because achieving Goal 2 in the pattern-matching program is useful in achieving Goal 1 does not necessarily imply that achieving the corresponding Goal 2' in the unification algorithm will have any bearing on Goal 1'. The extent to which the reasoning carries over depends on the soundness of the analogy. If a portion of the goal structure proves to be valid, the corresponding segment of the program can still remain; otherwise, we must construct a new program segment.

C.   The Modification

Let us examine the first two cases of the unification synthesis in full detail, so that we can see exactly how the modification process works. In the pattern-matcher, we generated the subgoal (Goal 2)

Find z such that inst(z pat) = arg.

The corresponding unification subgoal is

Find z such that inst(z pat) = inst(z arg).

In the pattern-matcher we first considered the case constexp(pat) where pat=arg. In this case the corresponding program segment will return Λ. This segment also satisfies the modified goal in this case, because

inst(Λ pat) $\doteq$ inst(Λ arg).

The system must also check that Λ is a most general unifier, i.e.,

for all s [if inst(s pat) = inst(s arg)

then s = $\Lambda°$w for some w].

This condition is easily satisfied, taking w=s. Thus, in this case, the program segment is correct without any modification.

The next case does require some modification. In the pattern-matcher, when constexp(pat) is true and pat≠arg, z is taken to be NOMATCH. However, in this case in the unification algorithm we must check that

inst(s pat) ≠ inst(s arg),

i.e.,

pat ≠ inst(s arg)

for any s, in order to take z=NOUNIFY. Since for the unification problem $\underline{\text{arg}}$ may contain variables, this condition cannot be satisfied. We must therefore try to achieve the specifications in some other way.
In this case (where $\underline{\text{constexp}}(\underline{\text{pat}})$), the specifications of the unification algorithm reduce to

> Find z such that
>
> {$\underline{\text{pat}}$ = $\underline{\text{inst}}(z \ \underline{\text{arg}})$ and
>
>   for all s [if $\underline{\text{pat}}$ = $\underline{\text{inst}}(s \ \underline{\text{arg}})$
>
>   then s = $z°w$ for some w]}
>
> else z = NOUNIFY.

These specifications are precisely the specifications of the pattern-matcher with $\underline{\text{pat}}$ and $\underline{\text{arg}}$ reversed; consequently, we can invoke $\underline{\text{match}}(\underline{\text{arg}} \ \underline{\text{pat}})$ at this point in the program.

The balance of the modification can be carried out in the same manner. The derived unification algorithm is

> $\underline{\text{unify}}(\underline{\text{pat}} \ \underline{\text{arg}})$ =
>
>   if $\underline{\text{constexp}}(\underline{\text{pat}})$
>
>   then if $\underline{\text{pat}}$ = $\underline{\text{arg}}$
>
>   then Λ
>
>   else $\underline{\text{match}}(\underline{\text{arg}} \ \underline{\text{pat}})$
>
>   else if $\underline{\text{var}}(\underline{\text{pat}})$
>
>   then if $\underline{\text{occursin}}(\underline{\text{pat}} \ \underline{\text{arg}})$
>
>   then NOUNIFY
>
>   else $\underline{\text{pair}}(\underline{\text{pat}} \ \underline{\text{arg}})$
>
>   else if $\underline{\text{atom}}(\underline{\text{arg}})$
>
>   then $\underline{\text{unify}}(\underline{\text{arg}} \ \underline{\text{pat}})$
>
>   else $z_1 \leftarrow \underline{\text{unify}}(\underline{\text{pat}}_1 \ \underline{\text{arg}}_1)$
>
>   if z_1 = NOUNIFY
>
>   then NOUNIFY
>
>   else $z_2 \leftarrow \underline{\text{unify}}(\underline{\text{inst}}(z_1 \ \underline{\text{pat}}_2) \ \underline{\text{inst}}(z_1 \ \underline{\text{arg}}_2))$
>
>   if z_2 = NOUNIFY
>
>   then NOUNIFY
>
>   else $z_1°z_2$.

Recall that $\underline{\text{occursin}}(\underline{\text{pat}} \ \underline{\text{arg}})$ means that $\underline{\text{pat}}$ occurs in $\underline{\text{arg}}$ as a subexpression.

The termination of this program is considerably more difficult to prove than was the termination of the pattern-matcher. However, the construction of the unification algorithm from the pattern-matcher is much easier than the initial synthesis of the pattern-matcher itself.

Note that the program we have constructed contains a redundant branch. The expression

    if pat = arg
    then Λ
    else match(arg pat)

could be reduced to

    match(arg pat).

Such improvements would not be made until a later optimization phase.

V.   DISCUSSION

A.   Implementation

Implementation of the techniques presented in this paper is under-way. Some of them have already been implemented. Others will require further development before an implementation will be possible.

We immagine the rules, used to represent reasoning tactics, to be expressed as programs in a PLANNER-type language. Our own implementation is in QLISP (Reboh and Sacerdoti [1973]). Rules are summoned by pattern-directed function invocation.

World-splitting has been implemented using the context mechanism of QLISP, which was introduced in QA4 (Rulifson et al. [1972]). The control-structure necessary for the hypothetical worlds, which involves an actual splitting of the control path as well as the assertional data base, is expressed using the multiple environments (Bobrow and Wegbreit [1973]) of INTERLISP (Teitelman [1974]). Although the world-splitting has been implemented, we have yet to experiment with the various strate-gies for controlling it.

The existing system is capable of producing simple programs such as the union function, the program to sort two variables from Part II, or the loop-free segments of the pattern-matcher from Part III.

The generalization of specifications (Sections II-D and III-E) is a difficult technique to apply without its going astray. We will develop heuristics to regulate it in the course of the implementation. Similarly, our approach to conjunctive goals (Section II-E) needs further explication.

B.   Historical Context and Contemporary Research

Early work in program synthesis (e.g. Simon [1963], Green [1969], Waldinger and Lee [1969]), was limited by the problem-solving capabili-ties of the respective formalisms involved (the General Problem Solver in the case of Simon, resolution theorem proving in the case of the others). Our paper on loop formation (Manna and Waldinger [1971]) was set in a theorem-proving framework, and paid little attention to the implementation problems.

It is typical of contemporary program synthesis work not to at-tempt to restrict itself to a formalism; systems are more likely to write programs the way a human programmer would write them. For

example, the recent work of Sussman [1973] is modelled after the de-
bugging process. Rather than trying to produce a correct program at
once, Sussman's system rashly goes ahead and writes incorrect programs
which it then proceeds to debug. The work reported in Green et al.
[1974] attempts to model a very experienced programmer. For example,
if asked to produce a sort program, the system recalls a variety of
sorting methods and asks the user which he would like best.

The work reported here emphasizes reasoning more heavily than the
papers of Sussman and Green. For instance, in our synthesis of the
pattern-matcher we assumed no knowledge about pattern-matching itself.
Of course we do assume extensive knowledge of lists, substitutions,
and other aspects of the subject domain.

Although Sussman's debugging approach has influenced our treat-
ment of program modification and the handling of simultaneous goals,
we tend to rely more on logical methods than Sussman. Furthermore,
Sussman deals only with programs that manipulate blocks on a table;
therefore he has not been forced to deal with problems that are
more crucial in conventional programming, such as the formation of
conditionals and loops.

The work of Buchanan and Luckham [1974] (see also Luckham and
Buchanan [1974]) is closest to ours in the problems it addresses.
However, there are differences in detail between our approach and
theirs:

The Buchanan-Luckham specification language is first-order pre-
dicate calculus; ours allows a variety of other notations. Their me-
thod of forming conditionals involves an auxiliary stack; ours uses
contexts and the Bobrow-Wegbreit control structures. In the Buchanan-
Luckham system the loops in the program are iterative, and are specified
in advance by the user as "iterative rules," whereas in our system the
(recursive) loops are introduced by the system itself when it recog-
nizes a relationship between the top-level goal and a subgoal. The
treatment of programs with side effects is also quite different in
the Buchanan-Luckham system, in which a model of the world is maintain-
ed and updated, and assertions are removed when they are found to con-
tradict other assertions in the model. Our use of contexts allows the

system to recall past states of the world and avoids the tricky problem of determining when a model is inconsistent. It should be added that the implementation of the Buchanan-Luckham system is considerably more advanced than ours.

C. Conclusions and Future Work

We hope we have managed to convey in this paper the promise of program synthesis, without giving the false impression that automatic synthesis is likely to be immediately practical. A computer system that can replace the human programmer will very likely be able to pass the rest of the Turing test as well.

Some of the approaches to program synthesis that we feel will be most fruitful in the future have been given little emphasis in this paper because they are not yet fully developed. For example, the technique of program modification, which occupied only one small part of the current paper, we feel to be central to future program synthesis work. The retention of previously constructed programs is a powerful way to acquire and store knowledge. Furthermore program optimization (cf. Darlington and Burstall [1973]) and program debugging are just special cases of program modification.

Another technique that we believe will be valuable is the use of more visual or graphic representations, that convey more of the properties of the object being discussed in a single structure. A mathematician will often informally use a diagram instead of a symbolic representation to help himself find a proof. The theorem-proving systems of Gelernter [1963] (in geometry) and Bundy [1973] (in algebra), for example, use diagram-like notations to facilitate proofs. We suspect that program synthesis would also benefit from such notations. For instance, we have found that the synthesis of the pattern-matcher could be made shorter and more intuitive by the introduction of the substitution notation of mathematical logic. If we represent an expression P as $P(x_1,\ldots,x_n)$, where $x_1,\ldots,x_n$ is the complete list of the variables that occur in P, then $P(t_1,\ldots,t_n)$ is the result of substituting variables x_i by terms t_i in P. We can then formulate the problem of pattern-matching as follows:

Let $\underline{pat}$ = $pat(x_1,\ldots,x_n)$.

Find z such that

    if $\underline{arg}$ = $pat(t_1,\ldots,t_n)$ for some $t_1,\ldots,t_n$

    then z = $\{<x_1\ t_1>,\ldots,<x_n\ t_n>\}$

    else z = NOMATCH.

Note that this specification includes implicitly the restriction that the match found be a most general match, because each of the variables x_i actually occurs in $\underline{pat}$. Therefore, the specifications do not need to be strengthened during the course of the synthesis.

We hope to experiment with visual representations in a variety of applications. Clearly, while the reasoning required is simplified by the use of pictorial notation, the handling of innovations such as the ellipsis notation ("...") in an implementation is correspondingly more complex.

VI. BIBLIOGRAPHY

1. Balzer, R. M. (September 1972), "Automatic Programming," Institute Technical Memo, University of Southern California/Information Sciences Institute, Los Angeles, Cal.

2. Biermann, A. W., R. Baum, R. Krishnaswamy and F. E. Petry (October 1973), "Automatic Program Synthesis Reports," Computer and Information Sciences Technical Report TR-73-6, Ohio State University, Columbus, Ohio.

3. Bobrow, D. G. and B. Wegbreit (August 1973), "A Model for Control Structures for Artificial Intelligence Programming Languages," Adv. Papers 3d. Intl. Conf. of Artificial Intelligence, 246-253, Stanford University, Stanford, Ca.

4. Boyer, R. S. and J S. Moore (Aug. 1973), "Proving Theorems about LISP Functions," Adv. Papers 3d. Intl. Conf. on Artificial Intelligence, 486-493, Stanford University, Stanford, Ca.

5. Buchanan, J. R. and D. C. Luckham (March 1974), "On Automating the Construction of Programs," Memo, Stanford Artificial Intelligence Project, Stanford, Ca.

6. Bundy, A. (August 1973), "Doing Arithmetic with Diagrams," Adv. Papers 3d. Intl. Conf. on Artificial Intelligence, 130-138, Stanford University, Stanford, Ca.

7. Darlington, J. and R. M. Burstall (August 1973), "A System which Automatically Improves Programs," Adv. Papers 3d. Intl. Conf. on Artificial Intelligence, 479-485, Stanford University, Stanford, Ca.

8. Floyd, R. W. (1967), "Assigning Meanings to Programs," Proc. of a Symposium in Applied Mathematics, Vol. 19, (J. T. Schwartz, ed.), Am. Math. Soc., 19-32.

9. Gelernter, H. (1963), "Realization of a Geometry-Theorem Proving Machine," in Computers and Thought, McGraw-Hill, 134-152.

10. Green, C. C. (May 1969), "Application of Theorem Proving to Problem Solving," Proc. Intl. Joint Conf. on Artificial Intelligence, 219-239, Washington, D.C.

11. Green, C. C., R. J. Waldinger, D. R. Barstow, R. Elschlager, D. B. Lenat, B. P. McCune, D. E. Shaw, and L. I. Steinberg (August 1974), "Progress Report on Program-Understanding Programs," Memo, Stanford Artificial Intelligence Project, Stanford, Ca.

12. Hardy, S. (July 1974), "Automatic Induction of LISP Functions," AISB Summer Conference, 50-62, University of Sussex, Brighton, England.

13. Hewitt, C. (April 1972), "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," AI Memo No. 251, Project MAC, M.I.T., Cambridge, Mass.

14. Hoare, C. A. R. (October 1969), "An Axiomatic Basis for Computer Programming," C. ACM, Vol. 12, No. 10, 576-580, 583.

15. Kowalski, R. (March 1974), "Logic for Problem Solving," Memo No. 75, Department of Computational Logic, University of Edinburgh, Edinburgh.

16. Luckham, D. and J. R. Buchanan (March 1974), "Automatic Generation of Programs Containing Conditional Statements," Memo, Stanford Artificial Intelligence Project, Stanford, Ca.

17. Manna, Z. (1974), Mathematical Theory of Computation, McGraw-Hill.

18. Manna, Z. and R. Waldinger (March 1971), "Toward Automatic Program Synthesis," Comm. ACM, Vol. 14, No. 3, 151-165.

19. McCarthy, J. (1962), "Towards a Mathematical Science of Computation," Proc. IFIP Congress 62, North Holland, Amsterdam, 21-28.

20. Reboh, R. and E. Sacerdoti (August 1973), "A Preliminary QLISP Manual," Tech. Note 81, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, Ca.

21. Robinson, J. A. (January 1965), "A Machine-Oriented Logic Based on the Resolution Principle," J. ACM, Vol. 12, No. 1, 23-41.

22. Rulifson, J. F., J. A. Derksen, and R. J. Waldinger (November 1972), "QA4: A Procedural Calculus for Intuitive Reasoning," Tech. Note 73, Artificial Intelligence Group, Stanford Research Institute, Menlo Park, Ca.

23. Simon, H. A. (October 1963), "Experiments with a Heuristic Com-
    uter," J. ACM, Vol. 10, No. 4, 493-506.   .

24. Sussman, G. J. (August 1973), "A Computational Model of Skill Ac-
    quisition," Ph.D. Thesis, Artificial Intelligence Laboratory,
    M.I.T., Cambridge, Mass.

25. Teitelman, W. (1974), INTERLISP Reference Manual, Xerox, Palo
    Alto, Ca.

26. Waldinger, R. J., and R. C. T. Lee (May 1969), "PROW: A Step
    Toward Automatic Program Writing," Proc. Intl. Joint Conf. on
    Artificial Intelligence, 241-252, Washington, D.C.