# 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

## Abstract

Prograin 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.

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.

## 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 PLAINER-type language (Hewitt [1972]); our own implementation is in the QLISP language (Reboh and Sacerdotl [1973]). Further details on the implementation are discussed in Section III-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. In Part III we give some of the historical background of automatic program synthesis, and we compare this work with other recent efforts.

A longer version *of* this paper (to appear in the Artificial Intelligence Journal) will apply the techniques of Part II to synthesize two "pattern matching" programs of some complexity.

## 11 - Fundamental Reasoning

In this section we will describe some of the reasoning and programming tactics that *are* basic to the operation of our proposed synthesizer. 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, Specifications 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-putput examples (cf. Green et al. [19743, Hardy [1974]), traces (cf. Blermann 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 limiting ourselves to the first-order predicate calculus (cf. Kowalski [19743). 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[13, Af2J, ... , A{n3 .

Furthermore, we introduce new constructs that are specific to certain subject domains. For instance, In the domain of sets we use

[xj 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 [19723). 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 rlfiments.

**Given two sets, s and t, we want to express**
**union(s t) = {x|xεs or xεt}**
**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|xεs or xεt} 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 ε s is false if empty(s).**
**(2) s is equal to add(head(s) tail(s)) if ~empty(s)**
**(3) x ε add(s t) is equivalent to (x = s or x ε t).**
**(4) {x|x ε s} is equal to s.**
**(5) {x|x=a or Q(x)} is equal to add(a{x|Q(x)}).**

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

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

## B. Formation of Conditional Expressions

In addition to the above constructs, we assume that our programming 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 uncertainty. 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 la 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

union(s t) = {x|x € s or x e tj.

We begin to transform these specifications using our rules. Rule (1) applies to the subexpression x e *s,* generating a subgoal, empty(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 empty(s) is a computable condition because empty is a primitive in our language.) In the case in which s is **empty, the expression**

**{x|x ε s or x ε t}**

**therefore reduces to**

**{x| false or x ε t},**

**or, by propositional logic,**

**{x|x ε t}.**

Now rule (4) reduces this to t, which is one *of* the inputs to our program and therefore is itself

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

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

```
union(s t) = if 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 reverse(l), that reverses the elements of the list l (e.g., 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 tail(t), we can use the program we are constructing to satisfy this subgoal. In other words we can Introduce a recursive call reverse(t*il(J&)) 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 tail(l) 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

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

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

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

This is transformed by rule (3) into

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

Using rule (5), this reduces to

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

If we observe that

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

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

union(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

```
union(s t) = if empty(s)
             then t
             else add(head(s) union(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.

I) 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 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

$x \cdot l$ is the list formed by inserting x before the first element of the list $l$ (e.g., $A \cdot (B\ C\ D) = (A\ B\ C\ D)$).

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 Btack. However, wrltlng such a program enta11a 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.

Tne averse 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 is illustrated in more detail during the synthesis of the pattern matcher in the full version of this paper.

The same recursion-introduction mechanism has been found Independently by Burstall and Darlington (1975).

E. Conjunctive Goals

The problem of solving conjunctive goals Is the problem of synthesizing 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 goa Is

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 problem is not itself a program synthesis problem, 1t could be rephrased as a synthesis problem. Moreover the difficulties in*vol*ved and the technique to be applied extend also to many real synthesis problems, such as the pattern-matcher synthesis of the full paper. Suppose our problem 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 equat ion 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. Whet is necessary is to make the solution of the first goal as general as possible, so that some special case of the solution might satisfy the second goal as well. For instance, a "general" solution to the first equation might be

$\langle 1 + w, 1 + 2w \rangle$ 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 al1 conjunctive goal problems For instance, the synthesis of an integer square-root program has specifications

Find z such that

z is an integer and

$z^2 \leq x$ and

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

where $x \geq 0$.

The above approach of find1ng a genera1 so1ution to one of the conJunets 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 I.ISP-like language: these programs return n value but effect no change in any data si rueture. In the next two sections we wi11 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, 1f 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.

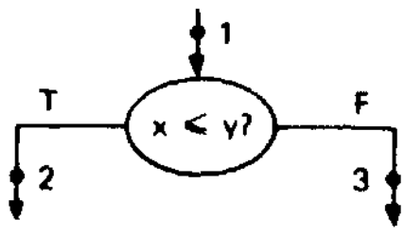
Many of the techniques we used In the synthesis of IJSP-like programs also apply to the construction of programs with side effects. In particular, we can use pattern-directed function invocation to retrieve tactical knowledge. The synthesis of the program In the following example has the same flavor 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

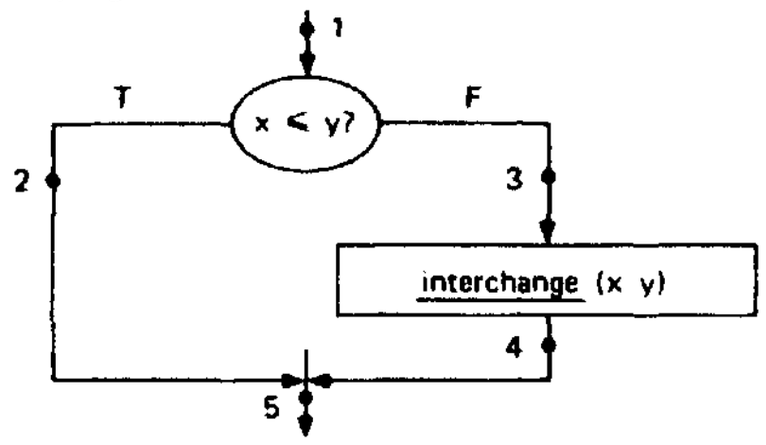
Achieve x & y.

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 reouiring 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 $x \pm y$, 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:



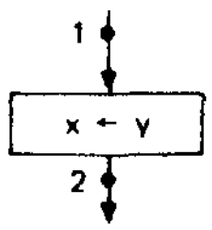

At point 2 our goal is already achieved. At point 3 we know that ~(x≤y), i.e., x>y. To achieve x≤y, it suffices to establish x<y, but this may be achieved by executing interchange(x y) Thus we have x≤y in both worlds, and the final program is therefore:




This example introduced no difficulties that our LISP-like program synthesis techniques could not handle. However, in general, programs 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 between assertions and program constructs in the next section.

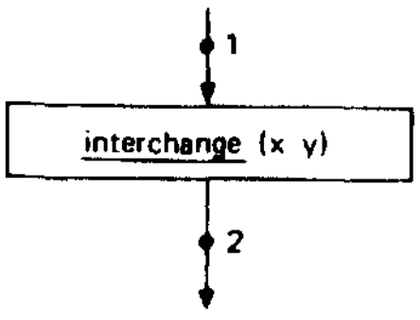
## G. Assertions and Program Constructs

Suppose a program contains an assignment statement



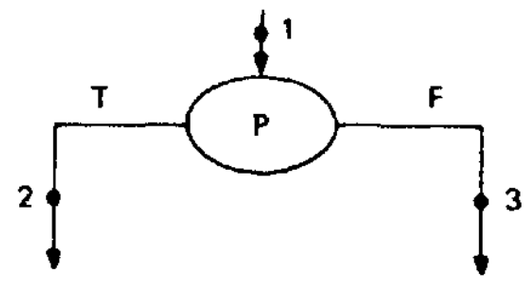

and we wish to determine if x<3 at point 2. In order to do this it suffices to check if what we know at point 1 implies that y<3. In general, to determine an assertion of form P(x) at point 2, check P(y) at point 1. We will say that the assertion P(y) is the result of "passing back" the assertion P(x)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

Furthermore, if our program contains the instruction




and we wish to establish x≤y at point 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

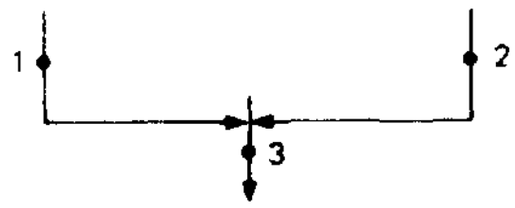
Q if P

(i.e., P ⊃ 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

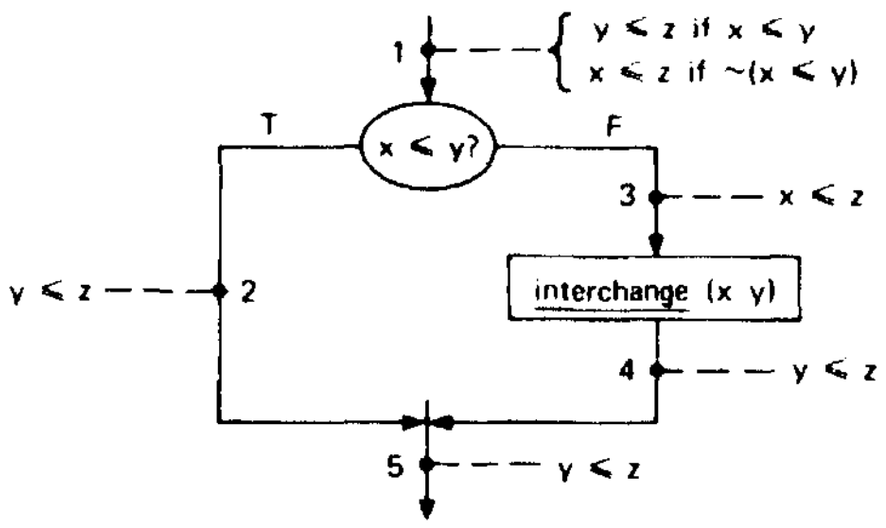
(i.e., ~P ⊃ R) is true at point 1.

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 instance, 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 MSP-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<y and y<z

We know that the program sort(u v) will achieve a goal of form u' v. If we apply the straightforward technique of achieving the conjunct x<y first, and then the conjunct y<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<y: if z is initially the smallest of the three, in interchanging 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 simultaneously, 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≤y and y≤z.

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

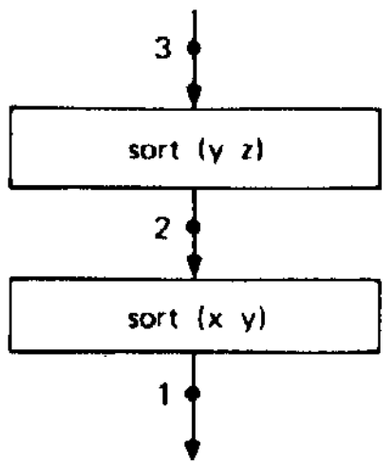
We have seen in the previous section that the goal y≤z passed back before the program sort(x y) results in two goals

(i) y≤z if x≤y

and

(ii) x≤z if ~(x≤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≤z whether or not x≤y.) Our program so far is thus




We still need to achieve goal (ii) at point 2; we can achieve this goal simply by inserting the instruction sort(x z) at that point. This insertion is seen not to violate any protected relation. Our final program is thus

```
sort(y z)
sort(x z)
sort(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. An expanded treatment of this strategy is contained in Waldinger [1975].

This concludes the presentation of our basic program synthesis techniques. In the longer version of this paper we show how these same techniques work together in the synthesis of some more complex examples.

III. DISCUSSION

A. Implementation

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

We imagine 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 assertlonal data base, is expressed using the multiple environments (Bobrow and Wegbreit [1973]) of INTERLISP (Teltelman [1974]) Although the world-splitting has been implemented, we have yet to experiment with the various strategies for controlling it.

The existing system is capable of producing simple programs such as the union function, the program to sort three variables from Part II, or

the loop-free segments of the pattern-matcher from the full version of this paper.

The generalization of specifications (Sections II-D) 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 11-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 capabilities 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 11971]) 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 attempt 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 debugging 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, relying on knowledge more than reasoning in producing a program.

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 assume 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 treatment 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 [19741) 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 predicate calculus; ours allows a variety of other notations. Their method of forming conditionals involves an auxiliary stack; ours uses contexts and the Bobrow-Wegbrelt 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 recognises 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 maintained and updated, and assertions *are* removed when they *are* found to *contradict* 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 immidately 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 .


## Acknowledgements

We wish to thank Robert Boyer, Nachum Dershowitz, Bertram Raphael, and Georgia Sutherland for giving detailed critical readings of the manuscript. We would also like to thank Peter Deutsch, Richard Fikes, Akira Fusaoka, Cordell Green and his students, Irene Greif, Carl Hewitt, Shmuel Katz, David Luckham, Earl Sacerdoti, and Ben Wegbrelt for conversations that aided in formulating the ideas in this paper. The set-theoretic expression handler is based on the work of Jan Derksen. We would also like to thank Linda Katuna, Claire Collins and Hanna Zies for typing many versions of this manuscript.

This research was primarily sponsored by the National Science Foundation under grants GJ-36146 and GK-35493.

## References


Balzer, R.M. "Automatic Programming," InstItute Technical Memo, University of Southern California/In formation Sciences Institute, Los Angeles, California, (September 1972).

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

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

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

Buchanan, JR. and D.C. Luckham, "On Automating the Construction of Programs," Memo, Stanford Artificial Intelligence Project, Stanford, California, (March 1974).

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

Burstall, R.M. and J. Darlington, "Some Transformations for Developing Recursive Programs," Proceedings International Conference on Reliable Software, Los Angeles, California, 482-492, (April 1975)

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

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

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

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

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

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

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

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

Kowalskl, R., "Logic for Problem Solving," Memo No. 75, Department of Computational Logic, University of Edinburgh, Edinburgh.

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

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

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

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

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

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

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

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

Sussman, G.J., "A Computational Model of Skill Acquisition," Ph.D. Thesis, Artificial Intelligence Laboratory, M.I.T , Cambridge, Massachusetts, (August 1973).

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

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

Waldinger, R.J., "Achieving Several Goals Simultaneously," Technical Note, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California (to appear).