

THE AUTOMATIC SYNTHESIS OF SYSTEMS OF RECURSIVE PROGRAMS

ZOHAR MANNA
Artificial Intelligence Lab
Stanford University
Stanford, Ca.

RICHARD WALDINGER
Artificial Intelligence Center
Stanford Research Institute
Menlo Park, Ca.

Abstract

A technique is presented for constructing a program from given specifications. The basic approach is to transform the specifications repeatedly, according to certain rules, until the desired program is produced. Two important transformation rules are those responsible for introducing conditional expressions and recursion into the target program. These transformations have been introduced in previous publications, and are discussed here briefly.

Often, to construct a recursive program it is necessary to define other *auxilliary programs* to achieve certain subtasks of the main task. The formation of such systems of auxilliary programs is specially emphasized in this paper.

The program synthesis techniques we discuss have been incorporated into a running system called SYNSYS. This system accepts high-level specifications expressed in mathematical notation, and produces recursive programs in pure LISP. The transformations are represented in the system as programs in the QLISP language, and are summoned by pattern-directed function invocations. The synthesis of two programs produced by the system are presented.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, by the National Science Foundation under Grant DCR72-03737 A01, by the Office of Naval Research under Contracts N00014-76-C-0687 and N00014-75-C-0816; and by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, Stanford Research Institute, or the U.S. Government.

I. Introduction

In this paper we describe a system that attempts to construct a program to meet given specifications. The system, called SYNSYS, accepts specifications expressed in a high-level mathematical notation and produces recursive, side-effect-free LISP programs. The basic approach is to find a sequence of transformations which, when applied successively to the output specification, yield a sequence of equivalent descriptions leading to the desired program.

The SYNSYS system has been partially described in an earlier paper (Manna and Waldinger [1977]). In that discussion we emphasized the mechanisms for introducing conditional expressions and recursion into the program being produced. The exposition in this paper is self-contained, but our principal concern here will be the formation of *systems* of recursive programs.

The earlier paper also contained a more complete description of the SYNSYS implementation. The transformation rules of the system are written in QLISP (Wilber [1976]), an extension of INTERLISP with pattern-directed function invocation and backtracking. The two examples we present have been performed automatically by SYNSYS.

In Section II we describe our basic approach. We then (Section III) present the first example, the synthesis of the Euclidean algorithm. Section IV provides a discussion of the method for introducing auxilliary functions, which is then illustrated by the construction of a two-function system for computing Cartesian products (Section V). A final section outlines some of the system's limitations, and our future research plans.

II. Basic Approach

There are many constructs that are valuable in expressing the specifications of a program, but that are not likely to appear as features in a programming language. For example, in specifying a program to compute the greatest common divisor of two integers x and y , we provide the *input specification*

$$x \geq 0 \text{ and } y \geq 0 \text{ and } (x \neq 0 \text{ or } y \neq 0),$$

and the *output specification*

$$\text{gcd}(x \ y) \leq \max\{z : z|x \text{ and } z|y\}.$$

The output specification requires that we find the greatest integer that divides both x and y . The input specification states that we can expect both arguments to be nonnegative and at least one to be nonzero; if both x and y are zero, the output specification is not defined. The set constructor $\{z : \dots\}$ is a valuable aid in expressing the output specification, but it is not a primitive construct in most programming languages because it is not always computable.

Similarly, to specify the program to find the maximum element of a list of numbers, we may supply the input specification

$$\neg \text{empty}(x)$$

(the input list is not empty) and the output specification

$\text{maxl}(x) \leftarrow \text{find } z \in x \text{ and } z : z \geq \text{all}(x)$

(find an element of the given list greater than or equal to all the elements of the list). Here, the constructs "find $z : \dots$ " and " $\text{all}(x)$ " are not primitives of any programming language, but are useful components of a specification language.

The goal of a synthesis system is to transform the output specifications into an equivalent description that employs only primitive constructs. The resulting *primitive program* will then be executable.

Our basic approach is to supply our system with a large set of *transformation rules*, which transform one description into another, equivalent one. We attempt to find a sequence of these rules which, applied to the output specification, will yield a sequence of equivalent descriptions leading to the desired primitive program. Some of these rules express the semantics of the subject domain (such as facts about lists, sets, numbers and the underlying logic). For example, the rule

$u|v \Rightarrow \text{true} \quad \text{if } v = 0$

represents the fact that every integer divides zero. Other rules express knowledge about the programming language and programming techniques (e.g. the uses of conditional expressions and recursive calls).

A given transformation can only be applied to a description that matches a characteristic pattern. For instance, the above rule can only be applied to descriptions of the form $u|v$. However, even if the pattern matches the description, a rule may have conditions that prevent it from being applied; the sample rule above can be applied only if the value of the expression that matches v is known to be zero. This logical condition is imposed because the rule could not be applied legitimately if it were violated. Other conditions may be imposed because of strategic considerations; we know that it is unwise to apply the rule when the condition is violated, even though the application would be logically legitimate. For example, the rule

$u|v \text{ and } u|w \Rightarrow u|v \text{ and } u|\text{rem}(w v)$

(the common divisors of v and w are the same as the common divisors of v and $\text{rem}(w v)$) has the logical condition

$v \neq 0$

and the strategic condition that we *not* be able to prove

$w < v$.

Although the rule could be applied correctly when $w < v$, in that case $\text{rem}(w v)$ reduces to w , and the original expression " $u|v$ and $u|w$ " would reappear.

During the process of transforming the output specification into a primitive program, we generate a sequence of intermediate descriptions of the desired program. In searching for the next rule to apply, we may find several rules whose patterns match the current description. Most of these rules will be discarded because their conditions will not be satisfied. Of the rules that remain, the system will apply one. If that application fails to lead to a primitive program, backtracking may occur so that the other rules may be attempted instead. The synthesis is complete when a primitive program is generated.

In order to examine the synthesis process in more detail, we present the complete construction of a familiar algorithm.

III. Example 1: The Euclidean Algorithm

In this example we illustrate the construction of the classical Euclidean algorithm for computing the greatest common divisor $\text{gcd}(x y)$ of two nonnegative integers x and y .

Recall that the specifications for the program are:

input specification: $x \geq 0$ and $y \geq 0$ and $(x \neq 0 \text{ or } y \neq 0)$

(the arguments are both nonnegative and at least one of them is nonzero), and

output specification: $\text{gcd}(x y) \leftarrow \text{max}\{z : z|x \text{ and } z|y\}$

(the output is the greatest integer that divides both x and y).

The set constructor $\{u : \dots\}$ is admitted to our specification language but is not a primitive of our programming language. We must find a sequence of transformations to produce an equivalent description of the output that does not use the set constructor or any other nonprimitive construct. This description will be the desired primitive program. In what follows we will exhibit a successful sequence of transformations, but we will not always indicate how the next transformation at a given stage was found, or which sequences were attempted and discarded before the successful sequence was found.

Among the transformations we may apply, let us assume we have the following rules which express properties of division useful in developing the desired program:

For any integers u, v , and w

- (1) $u|v \Rightarrow \text{true} \quad \text{if } v = 0$
(any integer divides zero)
- (2) $u|v \text{ and } u|w \Rightarrow u|v \text{ and } u|\text{rem}(w v) \quad \text{if } v \neq 0$
(the common divisors of v and w are the same as those of v and $\text{rem}(w v)$), and
- (3) $\text{max}\{u : u|v\} \Rightarrow v \quad \text{if } v > 0$
(any positive integer is its own greatest divisor).

In applying these transformations, we will produce a sequence of goals; the first will be derived directly from the output specification, and the last will be the desired program itself.

Goal 1: Compute $\text{max}\{z : z|x \text{ and } z|y\}$,

for any x and y satisfying the input specification. The transformation (2) above,

$u|v \text{ and } u|w \Rightarrow u|v \text{ and } u|\text{rem}(w v) \quad \text{if } v \neq 0$,

applies directly to Goal 1, yielding

Goal 2: Compute $\text{max}\{z : z|x \text{ and } z|\text{rem}(y x)\}$
and prove $x \neq 0$.

Note that the condition $v \neq 0$ attached to the transformation rule produced a condition $x \neq 0$ to be proved in the resulting goal. We cannot prove or disprove this condition — it may be true for some inputs and false for others — so we will consider separately the case in which it is false. This case analysis will yield a test on the condition $x = 0$ in the final program.

Case: $x = 0$

We cannot achieve Goal 2 here, so we examine alternate transformations to apply to Goal 1. The rule (1),

$$u|v \Rightarrow \text{true} \quad \text{if } v = 0$$

and the basic logical transformation

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

yield

Goal 3: Compute $\max\{u : u|y\}$.

Here, the transformation rule (3) applies, producing

Goal 4: compute y
and prove $y > 0$.

It is straightforward to prove $y > 0$, because $y \geq 0$ and $(x \neq 0 \text{ or } y \neq 0)$, by the input specification, but $x = 0$ by our case assumption. Consequently, we have reduced the problem in this case to the task of computing y , which involves no nonprimitive constructs. The desired program may simply output

y .

We have thus completed one branch of our case analysis; the corresponding branch of the program is

```
gcd(x y) <= if x = 0
             then y
             else ...
```

We now turn to the alternate possibility.

Case $x \neq 0$

The case analysis was introduced in attempting to satisfy the condition $x \neq 0$ of Goal 2. This condition is satisfied in this case, and Goal 2 is reduced to

Goal 5: Compute $\max\{z : z|x \text{ and } z|rem(y x)\}$.

Goal 5 is an instance of our output specification (i.e. Goal 1), with x and y replaced by x and $rem(y x)$. This suggests achieving Goal 5 by a recursive call to $gcd(x rem(y x))$, because the gcd program is intended to satisfy all instances of its output specification. However, the program is only guaranteed to work if its input specification is satisfied. To insure that the recursive call $gcd(x rem(y x))$ will compute the desired result, we must prove that the input specification is satisfied by the arguments of the recursive call, i.e.

Goal 6: Prove $x \geq 0$ and $rem(y x) \geq 0$
and $(x \neq 0 \text{ or } rem(y x) \neq 0)$.

We will call Goal 6 the *input condition* for the recursive call $gcd(x rem(y x))$.

Furthermore, in introducing a recursive call we must be concerned with the termination of the final program. In other words, we must ensure that an infinite sequence of recursive calls cannot occur in any computation of the program. To prove termination we employ the concept of the well-founded set, one whose elements are ordered in such a way that no infinite decreasing sequence can exist. (The nonnegative integers, for

example, constitute a well-founded set under the usual greater-than ordering. The integers, on the other hand, do not.)

To prove the termination of a program $f(x)$ with one recursive call $f(t)$, we must find a well-founded set W_f with ordering $>_f$ such that

x and t belong to W_f
and $x >_f t$.

If a infinite sequence of recursive calls were to occur, the corresponding arguments would constitute an infinitely decreasing sequence of elements of W_f , contradicting the well-foundedness of W_f .

Thus, to show the termination of the recursive call $gcd(x rem(y x))$ in the program $gcd(x y)$, we must achieve the following *termination condition*:

Goal 7: Find a well-founded set W_{gcd} with ordering $>_{gcd}$ such that
 $(x y) \in W_{gcd}$ and $(x rem(y x)) \in W_{gcd}$
and $(x y) >_{gcd} (x rem(y x))$.

We cannot satisfy Goal 7; in fact, in the case that $y < x$, $rem(y x) = y$, and there is no well-founded ordering $>$ such that $(x y) > (x y)$. Because we cannot show the termination of the recursive call $gcd(x rem(y x))$ that was introduced in an attempt to achieve Goal 5, we look for an alternate approach to achieve this goal.

The logical transformation

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

applied to Goal 5 yields

Goal 8: Compute $\max\{z : z|rem(y x) \text{ and } z|x\}$.

Goal 8 is again an instance of our output specification, Goal 1, and we therefore attempt to achieve it by the recursive call $gcd(rem(y x) x)$. Actually, the SYNSYS system reaches this goal in a slightly different manner to be discussed below. The input condition for this recursive call is

Goal 9: $rem(y x) \geq 0$ and $x \geq 0$ and
 $(rem(y x) \neq 0 \text{ or } x \neq 0)$,

and the termination condition is

Goal 10: Find a well-founded set W_{gcd} and ordering $>_{gcd}$ such that
 $(x y) \in W_{gcd}$ and $(rem(y x) x) \in W_{gcd}$
and $(x y) >_{gcd} (rem(y x) x)$.

Goal 9 is satisfied immediately: in this case x is positive and $rem(y x)$ is always nonnegative. To achieve Goal 10, we take W_{gcd} to be the set of all pairs of nonnegative integers ordered by the usual $>$ ordering applied to the first component; this suffices because $x > rem(y x)$. The proposed recursive call is therefore successful in achieving Goal 8 and the program can output

$gcd(rem(y x) x)$

We have succeeded in transforming all the goals into primitive program segments. The final program, formed directly from the outlined program segments is

$$\text{gcd}(x\ y) \leftarrow \begin{array}{l} \text{if } x = 0 \\ \text{then } y \\ \text{else } \text{gcd}(\text{rem}(y\ x)\ x) \end{array}.$$

This is a recursive version of the algorithm Euclid provided for computing the greatest common divisor.

Recall that when we fail to achieve Goal 5,

$$\text{Compute } \max\{z : z|x \text{ and } z|\text{rem}(y\ x)\}$$

by a recursive call, we applied a logical transformation

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

to the goal and then successfully introduced a recursion. Actually, the SYNSYS system does not represent the commutativity of "and" by a transformation; that property is built into the underlying QLISP system. Normally, in QLISP, a function f applied to several arguments $u_1, u_2, \dots, u_n$ is represented internally as $f\langle u_1\ u_2 \dots u_n \rangle$, where $\langle u_1\ u_2 \dots u_n \rangle$ is an ordered tuple. An expression such as

$$u_1 \text{ and } u_2 \text{ and } \dots \text{ and } u_n,$$

however, is represented internally as

$$\text{and}\{u_1\ u_2 \dots u_n\},$$

where $\{u_1\ u_2 \dots u_n\}$ is an unordered set, because the order and multiplicity of the arguments does not influence the value of the expression. In attempting to introduce a recursive call, the system tried to match the subexpression $\text{and}\{x\ \text{rem}(y\ x)\}$ of Goal 5 against the subexpression $\text{and}\{x\ y\}$ of Goal 1. The QLISP pattern matcher discovered two distinct matches, pairing x and y with x and $\text{rem}(y\ x)$ or pairing x and y with $\text{rem}(y\ x)$ and x . When the first match failed to lead to a satisfactory recursion, the second match was attempted automatically.

The mechanisms employed to construct the gcd program are examined more closely in our earlier paper (Manna and Waldinger [1977]). A similar mechanism to ours for introducing conditional tests into synthesized programs has been implemented by Warren [1976]. A facility for initiating a case analysis in a mathematical proof is included in the theorem proving system of Bledsoe and Tyson [1977]. Our recursion-introduction device is the same as the "folding" rule of Burstall and Darlington [1975]; their system is interactive, however, and they require that the user be responsible for establishing the input and termination conditions.

IV. Introducing Auxilliary Functions

In the preceding example we introduced a recursive call when we discovered that a subgoal was an instance of our output specification, the top-level goal. In other words, we found that the subgoal is of form

$$\text{Compute } \alpha(t(x)),$$

where the top-level goal is of form

$$f(x) \leftarrow \text{Compute } \alpha(x).$$

If the input specification is $P(x)$, then a recursive call $f(t(x))$ could be introduced to achieve the subgoal "Compute $\alpha(t(x))$ ", provided we could prove the *input condition*

$$P(t(x))$$

and the *termination condition*

Find a well-founded set W_f with ordering $>_f$ such that $x \in W_f$ and $t(x) \in W_f$ and $x >_f t(x)$.

Actually, in attempting to introduce recursion we compare the subgoal not only with the top-level goal, but also with each of the intermediate subgoals. If we discover that our subgoal is of form

$$\text{Compute } \beta(t(x))$$

where

$$\text{Compute } \beta(x)$$

is an intermediate subgoal, then we introduce a new auxilliary function g whose output specification is

$$g(x) \leftarrow \text{Compute } \beta(x).$$

The subgoal "Compute $\beta(t(x))$ " might then be achieved by a recursive call $g(t(x))$.

Of course to introduce a recursive call to an auxilliary function, we must establish the appropriate input and termination conditions. The input specification for the auxilliary program is not the same as the input specification $P(x)$ for the entire program f . For, suppose that in developing the subgoal "Compute $\beta(x)$," we have made several case assumptions "Case $R_1(x)$," ..., "Case $R_n(x)$ ". Then the conditions $R_1(x)$, ..., $R_n(x)$ will be the tests of conditional expressions in the final program f , and must be true if control is to reach the call to $g(x)$. Thus, we may expect that the conjunction $Q(x)$ of all these tests with the original input specification, i.e.

$$P(x) \text{ and } R_1(x) \text{ and } \dots \text{ and } R_n(x),$$

will be true of the input to g . Moreover, the correctness of g may depend on the truth of the above conjunction $Q(x)$, which we therefore take as the input specification for g . The input condition for a recursive call $g(t(x))$ is then $Q(t(x))$, i.e.

$$P(t(x)) \text{ and } R_1(t(x)) \text{ and } \dots \text{ and } R_n(t(x)).$$

In the simple case that the auxilliary function g does not call the main function f , the termination condition for a recursive call $g(t(x))$ is analogous to the termination condition for a recursive call to f , i.e. we must

Find a well-founded set W_g with ordering $>_g$ such that $x \in W_g$ and $t(x) \in W_g$ and $x >_g t(x)$.

The general case, in which g may also call f , is more complicated and will not be considered here.

In the next section we illustrate the formation of auxiliary functions in a concrete example.

V. Example 2: The Cartesian Product

A problem that requires the formation of a simple system of recursive programs is the computation of the Cartesian product $\text{cart}(X \ Y)$ of two (finite) sets X and Y . This is set of all pairs $(x \ y)$ such that $x \in X$ and $y \in Y$. The output specification is therefore

Goal 1: Compute $\{(x \ y) : x \in X \text{ and } y \in Y\}$.

There is no input specification, because the program is intended to apply to any two sets. Again, the set constructor $\{z : \dots\}$ is regarded as a nonprimitive component of our specification language. We include in our programming language the following primitive operations:

U	set union
$\{t_1, t_2, \dots, t_n\}$	the finite set of elements $t_1, t_2, \dots, t_n$
$\{\}$	the empty set
$\text{empty}(U)$	the test for emptiness
$\text{head}(U)$	a specific element of a nonempty set U
$\text{tail}(U)$	the set of elements of U other than $\text{head}(U)$
$(u \ v)$	the pair with elements u and v

and if-then-else and recursion. Let us assume that the transformations we can use in transforming the goal include

- (1) Membership expansion
 $u \in U \Rightarrow \text{if } \text{empty}(U) \text{ then } \text{false} \text{ else } u = \text{head}(U) \text{ or } u \in \text{tail}(U);$
- (2) Empty-set introduction
 $\{u : \text{false}\} \Rightarrow \{\};$
- (3) Union introduction
 $\{u : P(u) \text{ or } Q(u)\} \Rightarrow \{u : P(u)\} \cup \{u : Q(u)\};$ and
- (4) Equality elimination
 $\{u : u = t\} \Rightarrow t,$
 where u and t are expressions with no variables in common.

Applying membership expansion rule (1) to the subexpression $x \in X$ of Goal 1 yields

Goal 2: Compute $\{(x \ y) : [\text{if } \text{empty}(x) \text{ then } \text{false} \text{ else } x = \text{head}(X) \text{ or } x \in \text{tail}(X)] \text{ and } y \in Y\}$.

Applying the logical if-then-else distribution rule

$$f(\text{if } P \text{ then } t_1 \text{ else } t_2) \Rightarrow \text{if } P \text{ then } f(t_1) \text{ else } f(t_2),$$

yields

Goal 3: Compute $\text{if } \text{empty}(X) \text{ then } \{(x \ y) : \text{false and } y \in Y\} \text{ else } \{(x \ y) : (x = \text{head}(X) \text{ or } x \in \text{tail}(X)) \text{ and } y \in Y\}$.

We next apply the logical transformation

$$\text{false and } P \Rightarrow \text{false}$$

to the then-clause, and the and-or distribution rule

$$(P \text{ or } Q) \text{ and } R \Rightarrow (P \text{ and } R) \text{ or } (Q \text{ and } R)$$

to the else-clause, to obtain

Goal 4: Compute $\text{if } \text{empty}(X) \text{ then } \{(x \ y) : \text{false}\} \text{ else } \{(x \ y) : (x = \text{head}(X) \text{ and } y \in Y) \text{ or } (x \in \text{tail}(X) \text{ and } y \in Y)\}$.

The empty-set introduction rule (2) applies to the then-clause of Goal 4 and the union-introduction rule (3) applies to the else-clause, to produce

Goal 5: Compute $\text{if } \text{empty}(X) \text{ then } \{\} \text{ else } \{(x \ y) : x = \text{head}(X) \text{ and } y \in Y\} \cup \{(x \ y) : x \in \text{tail}(X) \text{ and } y \in Y\}$.

The subexpression $\{(x \ y) : x \in \text{tail}(X) \text{ and } y \in Y\}$ is an instance of our output specification,

$$\{(x \ y) : x \in X \text{ and } y \in Y\};$$

this suggests replacing this subexpression by a recursive call

$$\text{cart}(\text{tail}(X) \ Y).$$

There is no input condition for the proposed recursive call because the desired program has no input specification. The termination condition,

Goal 6: Find a well-founded set W_{cart} and ordering $>_{\text{cart}}$ such that $(x \ y) \in W_{\text{cart}}$ and $(\text{tail}(X) \ Y) \in W_{\text{cart}}$ and $(X \ Y) >_{\text{cart}} (\text{tail}(X) \ Y)$.

is solved by taking W_{cart} to be the set of all pairs of finite sets, ordered by the usual proper containment relation on the first component. This ordering is sufficient because X properly contains $\text{tail}(X)$.

Goal 5 is therefore reduced to

Goal 7: Compute $\text{if } \text{empty}(X) \text{ then } \{\} \text{ else } \{(x \ y) : x = \text{head}(X) \text{ and } y \in Y\} \cup \text{cart}(\text{tail}(X) \ Y)$.

We still have to remove the remaining set-constructor from the else-clause. To simplify the exposition we will reduce this subexpression in a separate series of starred subgoals

Goal 7*: Compute $\{(x \ y) : x = \text{head}(X) \text{ and } y \in Y\}$.

Applying the member expansion rule (1) to the subexpression $y \in Y$ yields

Goal 8*: Compute $\{(x \ y) : x = \text{head}(X) \text{ and } [\text{if } \text{empty}(Y) \text{ then } \text{false} \text{ else } y = \text{head}(Y) \text{ or } y \in \text{tail}(Y)]\}$.

Applying the if-then-else distribution rule and the union-introduction rule (3) produces

Goal 9*: Compute if $empty(Y)$
 then $\{(x\ y) : x = head(X) \text{ and } false\}$
 else $\{(x\ y) : x = head(X) \text{ and } y = head(Y)\}$
 $\cup \{(x\ y) : x = head(X) \text{ and } y \in tail(Y)\}$.

The then-clause of Goal 9* is transformed by the logical rule

$P \text{ and } false \Rightarrow false$,

and the empty-set introduction rule (2). The first component of the else-clause is transformed by the equality-elimination rule (4). The resulting goal is

Goal 10*: Compute if $empty(Y)$
 then $\{\}$
 else $\{head(X) \ head(Y)\}$
 $\cup \{(x\ y) : x = head(X) \text{ and } y \in tail(Y)\}$.

Now the subexpression

$\{(x\ y) : x = head(X) \text{ and } y \in tail(Y)\}$

is an instance of Goal 7*. This suggests that we introduce an auxiliary function $cart2(X\ Y)$ to achieve Goal 7*, and replace the above subexpression by the recursive call $cart2(X\ tail(Y))$. To establish that this proposed recursive call is legitimate, we must prove its input and termination conditions.

Even though the main program $cart(X\ Y)$ had no input specification, the auxiliary function $cart2(X\ Y)$ is applied in the else-branch of the program, for which it is known that X is not empty. This condition $\neg empty(X)$ is therefore the input specification for $cart2(X\ Y)$. Thus, the input condition for the recursive call $cart2(X\ tail(Y))$ is simply

Goal 11*: $\neg empty(X)$.

This condition is satisfied immediately, because it is identical to the input specification for $cart2(X\ Y)$.

The termination condition for the recursive call $cart2(X\ tail(Y))$ is

Goal 12*: Find a well-founded set W_{cart2} with ordering $>_{cart2}$ such that
 $(X\ Y) \in W_{cart2}$ and $(X\ tail(Y)) \in W_{cart2}$
 and $(X\ Y) >_{cart2} (X\ tail(Y))$.

Goal 12* can be achieved by taking W_{cart2} to be the set of all pairs of finite sets, with the usual proper containment ordering applied to its second component.

Having established both the input and termination conditions, we are justified in introducing the recursive call $cart2(X\ tail(Y))$ into Goal 10*, yielding

Goal 13*: Compute if $empty(Y)$
 then $\{\}$
 else $(head(X) \ head(Y)) \cup cart2(X\ tail(Y))$.

Because Goal 13* is composed entirely of primitive constructs, we have succeeded in constructing the auxiliary function

$cart2(X\ Y) \Leftarrow$ if $empty(Y)$
 then $\{\}$
 else $(head(X) \ head(Y)) \cup cart2(X\ tail(Y))$

This program computes the Cartesian product of $\{head(X)\}$ with Y .

The auxiliary function $cart2(X\ Y)$ is intended to satisfy Goal 7*, a subexpression of Goal 7. We can therefore replace that subexpression by a call to the new function, yielding

Goal 14: Compute if $empty(X)$
 then $\{\}$
 else $cart2(X\ Y) \cup cart(tail(X) \ Y)$.

Now Goal 14 contains no nonprimitives and we have succeeded in constructing the desired program

$cart(X\ Y) \Leftarrow$ if $empty(X)$
 then $\{\}$
 else $cart2(X\ Y) \cup cart(tail(X) \ Y)$.

VI. Limitations and Future Research

The preceding example illustrates construction of the simplest form of auxiliary function, which does not call the main function. The general case is more difficult, and is beyond the capabilities of our current system because of the complexity of the termination condition.

We introduce a recursive call only when a subgoal is an instance of a higher-level subgoal. For some problems it may be necessary to generalize the higher-level subgoal to force the match to occur. This situation is analogous to proving a mathematical theorem by mathematical induction: it is often necessary to generalize the theorem to be proved, so that the induction hypothesis will be strong enough for the induction step to be proved. Some such generalizations have been performed automatically by the theorem-proving system of Boyer and Moore [1975], and the program synthesis system of Darlington [1975].

Our current SYNSYS only constructs pure LISP programs, which produce an output value but do not have any side-effects. Systems by Warren [1974] and Waldinger [1977] can produce programs with side-effects, but they cannot introduce recursive or iterative loops into these programs. We intend to integrate both abilities into a single system.

VII. References

- Bledsoe, W. W., and M. Tyson [1977], *Typing and proof by cases in program verification*, in *Machine Intelligence 8: Machine Representations of Knowledge*, (E. W. Elcock and D. Michie, editors), John Wiley & Sons, New York, N.Y. (to appear).
- Boyer, R. S., and J. S. Moore [Jan. 1975], *Proving theorems about LISP functions*, JACM, Vol. 22, No. 1, pp. 129-144.
- Burstall, R. M. and J. Darlington [April 1975], *Some transformations for developing recursive programs*, Proceedings of the International Conference on Reliable Software, Los Angeles, Ca., pp. 465-472.

- Darlington, J. [July 1975], *Applications of program transformation to program synthesis*, Colloques IRIA on Proving and Improving Programs, Arc-et-Senans, France, pp. 133-144.
- Manna, Z. and R. Waldinger [August 1977], *The automatic synthesis of recursive programs*, Proceedings of the SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, Rochester, N.Y.
- Waldinger, R. J. [1977], *Achieving several goals simultaneously*, in *Machine Intelligence 8: Machine Representations of Knowledge*, (E. W. Elcock and D. Michie, editors), John Wiley & Sons, New York, N.Y. (to appear).
- Warren, D. H. D. [June 1974], *WARPLAN: A system for generating plans*, Technical Note, Dept. of Computational Logic, University of Edinburgh, Edinburgh, Scotland.
- Warren, D. H. D. [July 1976], *Generating conditional plans and programs*, Proceedings of Conference on Artificial Intelligence and Simulation on Behaviour, Edinburgh, Scotland, pp. 344-354.
- Wilber, B. M. [Mar. 1976], *A QLISP reference manual*, Technical note, Stanford Reserch Institute, Menlo Park, Ca.