THE AUTOMATIC SYNTHESIS OP SYSTEMS OF RECURSIVE PROGRAMS

ZOHAR MANNA
Artificial Intelligence Lab
Stanford University
Stanford, Ca.

RICHARD WALDINCER
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 MDA901- 76-C-0206, *by the National Science Foundation under Grant* DCK72-03737 *A0\, by the Office of Naval Research under Contracts* A/00014-76-C-06S7 *and* A/000 M-75-C-08I6; *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

Jn 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 111) 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. Basio 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*

$$x \; z \; 0 \text{ and } y \; z \; 0 \text{ and } \{x * 0 \text{ or } y * 0),$$

and the *output specification*

$$gcd(x\ y) \text{ <- } max\{z : z\backslash x \text{ and } z\backslash y) .$$

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* : ...J *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

-i *empty(x)*

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

maxl(x) <• find *z* € *x* and *z* : *z z 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; ..." and *"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 •> *true* 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 it 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 and *u\w* »> *u\v* and *u\rem(ui v)*

(the common divisors of *v* and *w* are the same as the common divisors of *v* and *rem(iu v))* has the logical condition

v * 0

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

$w < v$.

Although the rule could be applied correctly when $w < v_t$ in that case *rem{w v)* reduces to *w,* and the original expression *"u\v* and *u* 1 n/[M] 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 *gcd(x y)* of two nonnegative integers *x* and *y.*

Recall that the specifications for the program are:

input specification: *x* > 0 and) ^ 0 and *(x* * 0 or *y* * 0)

(the arguments are both nonneg^ative and at least one of them is nonzero), and

output specification: *gcd(x y)* <«* *max{z : z\x* and *z\y]*

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

The set constructor *{u* : ...} 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* »> *true* if *v* » 0
(any integer divides zero)
(2) *u* | *v* and *u* | *w* *> *u* | *v* and *u* \ *rem(w v)* *if v* * 0
(the common divisors of *v* and *w* are the same as those of *v* and *rem{tu v))*> and
(3) *max{u* : *u\v}* *> *v* 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 *max{z* : *z\x* and *z\y],*

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

u|v and *u\w* -> *u\v* and *u\rem(xu v)* if *v* * 0 ,

applies directly to Coal 1, yielding

Coal 2: Compute *max\z : z\x* and *z\rem(y x)}*
and prove X K O .

Note that the condition *v* * 0 attached to the transformation rule produced a condition *x* * 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 Coal 1. The rule (1),

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

and the basic logical transformation

P and *true* ■> P

yield

Coal 3: Compute $max\{u : u \mid y\}$.

Here, the transformation rule (3) applies, producing

Coal 4: compute *y*
and prove *y* > 0 .

It is straightforward to prove *y* > 0, because *y* > 0 and *(x* * 0 or *y* * 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=O

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

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

Goal 5 is an instance of our output specification (i.e. Coal 1), wjth *x* and *y* replaced by *x* and *rcm{y x).* This suggests achieving Goal 5 by a recursive call ro *gcd(x remiy 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 $(x \neq 0$ 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 *prognm.* 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..*

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

Coal 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 and <£-> Q^and P

applied to Goal 5 yields

Coal 8: Compute *max{i* : *z\rcm{y x)* 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

Coal 9: $rem(y\ x) \geq 0$ and $x \geq 0$ and
$(rem(y\ x) \neq 0$ or $x \neq 0)$.

and the termination condition *is*

Coal 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* , 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 *$* 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

$$gcd(x\ y) \Leftarrow \text{if } x = 0 \text{ then } y \text{ else } gcd(rem(y\ x)\ x)$$

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,

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

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 / applied to several arguments $u_{lt}\ u_2\ .\ .\ .\ u_n$ is represented internally as $f<u_1\ u_2\ \ldots\ u_n>$, where $<u_1\ u_2\ \ldots\ u_n>$ is an ordered tuple. An expression such as

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

however, is represented internally as

$$and\{u_1\ u_2\ \ldots\ u_n\},$$

where $\{u_l\ u_2\ \ldots\ u_n\}$ 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 *and{x rem(y x)}* of Coal 5 against the subexpression *and\x y)* of Goal 1. The QLISP pattern matcher discovered two distinct matches, pairing *x* and *y* with *x* and *remiy x)* or pairing *x* and *y* with *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.

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

Compute a(r(x)),

where the top-level goal is of form

f(x) <- Compute a(x).

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

?(t(x))

and the termination condition

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

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

Compute B*(t(x))*

where

Compute *0(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 B(r(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 *?{x)* for the entire program /. For, suppose that in developing the subgoal "Compute B(x)," we have made several case assumptions "Case $R_,(x)$," .... **"Case $R_n(x)$".** Then the conditions Ri(x), ... $R_n(x)$ will be the tests of conditional expressions in the final program /, and must be true if control is to reach the call to *g(x).* Thus, we may expect that the conjunction Qj[.v) of all these tests with the original input specification, i.e.

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

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(f(x)), i.e.

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

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

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

The general case, in which *g* may also call/, 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 Produot

A problem that requires the formation of a simple system of recursive programs is the computation of the Cartesian product *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 : ...} is regarded as a nonprimitive component of our specificaton language. We include in our programming language the following primitive operations;

∪	set union
$\{t_1, t_2, ..., t_n\}$	the finite set of elements r1, r_2,..., t
{ }	the empty set
empty(U)	the test for emptiness
head(U)	a specific element of a nonempty set U
tail(U)	the set of elements of U other than *head(V)*
(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
$u \in U$ => if *empty*(U)
then *false*
else u = *head*(U) or $u \in$ *tail*(U) ;

(2) Empty-set introduction
{u : *false*} => { } ;

(3) Union introduction
{u : P(u) or Q(u)} => {u : P(u)} ∪ {u : Q(u)} ; and

(4) Equality elimination
{u : u = t} => 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 $\{(x\ y) : [\text{if } empty(x) \text{ then } false \text{ else } x = head(X) \text{ or } x \in tail(X)] \text{ and } y \in Y\}$.

Applying the logical if-then-else distribution rule

/(if P then f, else t_2) «> *i(* P thenar,) *elsef*$(t_2)_t$

yields

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

We next apply the logical transformation

false and P »> *false*

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

(P or Q) and R => (P and R) or (Q and R)

to the else-clause, to obtain

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

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

The subexpression $\{(x^*\ y) : x\ e\ tail(X)$ and y e 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

cart(tail{\) 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 wpll-frmnrlpri ser. W_{cart}
and ordering $>_{cart}$ such that
$(x\ y) \in W_{cart}$ and $(tail(X)\ Y) \in W_{cart}$
and $(X\ Y) >_{cart} (tail(X)\ Y)$.

is solved by taking W_{rnrl} 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 *tail(X).*

Goal 5 is therfore reduced to

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

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

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

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

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

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

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

P and *false* -> *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 $\{(head(X)\ head(Y))\}$
$\cup\ \{(x\ y) : x = head(X)$ and $y \in tail(Y)\}$.

Now the subexpression

$\{(x\ y) : x = head(X)$ 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 auxilhary function *ca,rt2(X* Y) is applied in the else-branch of the program, for which it is known that X is not empty. This condition -. *empty(X)* is therefore the input specification for *cart2(X* Y). Thus, [he 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 c<zr'2(X Y).

The termination condition for the recursive call car/2(X *tail(Y))* is

Goal 12*: Find a well-founded set W_{rafll} 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_{eaft2} 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

$cart2(X\ Y) <=$ 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 *car:2(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 $car:2(X\ Y) \cup cart(tail(X)\ Y)$

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

$cart(X\ Y)$ <= 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 auxilliary 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 *8c* 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-HI

Manna, Z. and R. Waldinger [August 1977], *The automatic synthesis of recursive programs,* Proceedings of the SICART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, Rochester, N.Y.

Waldinger, R. J. [1977], *Achieving several goals simultaneously,* in *Machine Intelligence* S: *Machine Representations of Knowledge,* (E. W. Elcock and D. Michie, editors), John Wiley *8c* 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, Edingurgh, 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.

SISP/I AN INTERACTIVE SYSTEM ABLE TO SYNTHESIZE FUNCTIONS FROM EXAMPLES

Jean-Pierre JOUANNAUD FRANCE
Maitre-Assistant a 1'Institut de Programination
Universite Paris VI
A, Place Jussieu
75005 PARIS

Gerard GU1HO FRANCE
Maitre de Conference - Laboratoire de Recherche en Informatique
Universite Paris Sud
91405 ORSAY

Jean-Pierre TREUIL FRANCE
Chercheur - Laboratoire de Recherche en Informatique
Universite Paris Sud
91405 ORSAY

The research presented in this paper is supported by IklA-CESORI under contract number 76.

ABSTRACT

SISP/i is an interactive system whose goal is the automatic inference of LISP functions from a finite set of examples {(x., f(x.))) where x. is a list belonging to the domain of the function f we want to infer. SISP/I is able to infer the recursive form of many linear recursive functions and its stop-condition. SISP/I tries to work with one example only. When it fails, it asks for new ones: using then a method of generating new partial subproblems, SISP/I is able to perfect its generated recursive function until it gets a correct one.

I. INTRODUCTION

In this paper we describe the system SISP/I whose goal is the automatic inference of LISP functions from a finite set of examples {(x., I(x.))}, where x. is a list belonging to the domain of the function f we want to infer.

The problem originates from a more general one: how to build a "Learning-Question-Answering-System" (L.Q.A.S.) using a functional method to provide an answer to any given question. The method we propose in SISP/I is naturally well adapted to the L.Q.A.S. we are. developping (6.1, 17 1.

In the field of "Automatic Programming from Examples", an important piece of recent work is THESYS by SUMMERS L5J. The major result of this work, is the following: using a small number of well chosen examples ((NIL, f(NIL)), ((A), f((A)))...} THESYS is able to infer a recursive expression $ equivalent to f for every x belonging to the domain of f.

Only a small class of functions can however be obtained by Summers's method, which works by looking for a recurence relation between representative predicates p. of the given input structure and for a recurence relation between the map functions m. providing the given outputs from the given inputs. Then, using a fixed point theorem, *V* is constructed .

Although Summers's method is very powerful it has four important drawbacks:

1.- The constructed expression <*p* is necessarily recursive: for instance the identity function will be infered by *V* (x) » if X * NIL then NIL
else CONS (CAR(x), ^(CDR(x)))

2.- THESYS needs well chosen examples, which in particular must contain the stop condition of the recursive function *V*. For instance, the construction of the function REVERSE requires the following set of examples:
{(NIL + NIL), ((A) - (A)), ((A B) - (B A)),
((A B C) + (C B A))}

3.- The function to be constructed has to present only one "iterative level[11]. For instance, THESIS fails to construct a correct function corresponding to the example: (P Q R S) -> (P P Q P Q R P Q R S).

4.- When THESYS has to solve a difficult problem, it does not try to generate a partial, simpler problem for which it could either find a correct solution or perhaps use a knowledge previously stored in a data base by the system itself. Thus, THESYS cannot be efficiently used in a L.Q.A.S. without important modifications.

The method we propose in this paper is very different in particular, it has the built capacity to use a Professor in interactive mode. It does not lie yet on any theorical groundwork, but allows us to overcome some of the previous drawbacks, although new ones appear:

- recursion is not automatically infered by the synthesis algorithm; for instance, using the example ((A B C) -* (A B C)), STSP/1 infers the function <p *v*p* (x) = x for any x.
- for some "simple" functions, S'ISP/1 needs only one example (x, f(x)).

In the case where a recursive expression is infered, the stop condition is then found by SISP/I itself. However the. list x must be long enough to be representative of the function f. For instance, REVERSE is obtained using the only example ((A B C D) + (U C B A)), but is not obtained with ((A B C) > (C B A)).

- when the function f is "more complicated" SISP/I fails to construct a correct function with only one example and it then tries to work with two examples.
- when the function f is "much more complicated", SISP/I generates a new partial simpler problem (y> &(y)) where y is defined in terms of x and g(y) is defined in terms of f(x). To solve this new problem, SISP/I sometimes needs a new example (x^1, f(x')) which is used to deduce an example (y'» g(y'))« The interaction is only used in the sense of asking for new examples, when necessary. SISP/I is thus extensible and has the potentiality to use a self constructed knowledge data base.

Some objections can be raised to our interactive method:

- when a function f needs several examples to be infered, the professor sometimes has to give an appropriate sequence of examples.
- we do not exactly know the class of functions which SISP/I is able to infer. However, it seems to be much larger than THESYS one. For instance ((P Q R S)-»(P P Q P Q R P Q R S)) is infered by SISP/I using only one example whereas the HALF function ((P Q R S T U) + (P Q R)), which is infered by THESYS, requires two examples by SISP/I. In fact,

we hope that SISP will be able to infer a larger class of linear recursive functions.

II. GENERAL DESCRIPTION OF THE METHOD

1.- L§ngu§ge

SISP/I infers functions defined on character strings "ABCD..." which will be represented by the list (A B C D...).

SISP/1 synthetizes LISP-functions built with the following basic functions, described here by examp1es:

LCAR: (A B C D) -* (A) CDR: (A B C D) -> (B C D)
LRAC: (A B C D) -+ (D) RDC: (A B C D) -> (A B C)
CONC: (A B), (C D) ■> (A B C D)
CONCT: (A B), (C D), (E F) + (A B C D E F)
PREF: (B C), (A B C D) •> (A) L Prefix of (B C) in (A B C D)]
SUFF: (B C), (A B C D) ■+ (D) l Suffix of (B C) in (A B C D)j

and a control structure using COND and NULL.

2*~ Notion_of_tYp_e

A type is a set of lists which can be defined by rules which are summarised as follows [6 1:

a) the set of known inputs "x" and the set of outputs "$f(x)^M$" of the function f to be synthetized are types.

b) if X is a type and f a LISP function, then the set of outputs of f restricted to X as input is a type.

c) if Y is a type and g a LISP function then the set X of x such as g(x) C- Y is a type.

3«~ Segment^].i_on_pattern

Let f be a function to be synthetized and (x, f(x)) an example of "input-output" of this function.

SISP/1 uses a general heuristic to create an expression of the function:

a) segmentation of strings x and y = f(x) into three consecutive segments such that:

CONCT (px, c, sx) ■+ X
CONCT (py, c, $_S$y) -► y

where c denotes the larger string common to x and y, px and py denote the prefixs of c in x and y, sx and sy denote the suffixs of c in x and y.

b) building of relations between these segments.

A "Segmentation Pattern" of (x,y), for all x and y, is defined as the network shown in figure I.

We can see on this network:

- seven nodes representing types respectively associated to the strings x,y,c,px,py,sx,sy.
- twelve relations between nodes. Each relation consists of a function and a scheme (I,» I«, •••> I -> J) which indicates the input nodes I_1, I_2,..., I_n in this order and the output node J. This order is represented on the network by a double arrow.

Note that functions FX, FY, GPX, GPY, GSX, GSY are built by SISP/1 using the basic functions LCAR, CDR, LRAC, RDC, and the composition rule. They are choosen of the less possible complexity (the smallest number of basic functions).

In some cases, the segmentation pattern is simpler:

- when one or several strings are empty (NIL), .. the associated nodes are suppressed from the pattern.
- when two strings are equal, the associated nodes are joined together. For instance, if x and y are the same, the pattern is reduced to one node; if x and y have no common part, the pattern is reduced to only two nodes.

4*"" §Y.£lt}£sis fron}_on£_exam£le

The synthesis consists of three steps:

a) SISP/1 generates a network (called a "Segmentation Structure") by the following process:

(1) Generate the segmentation pattern of (x,y).The generation gives the two sets of pairs: {(px» py), (c,py), (sx, py)} Kpx,sy), (c,sy), (sx, sy)}

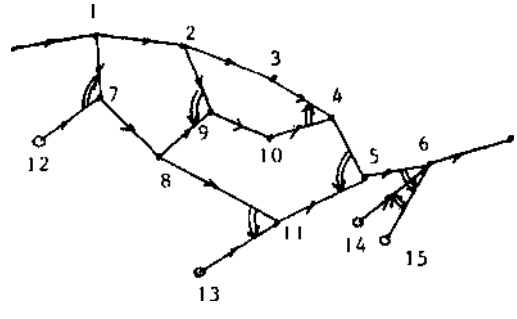
(2) As long as py and sy are not empty, choose one pair in each set by a heuristic way; for each of these pairs, rename it as (x,y) and go to step 1.

b) SISP/1 looks at the segmentation structure for a lattice in which the minimal and final nodes are respectively X and Y (that is x and y types). This lattice is stepwise constructed using Algorithm 1, defined as follows:

Definitions:

- LAT is the constructed part of the lattice at any step (except in the final step, LAT is not a lattice).
- an incomplete node of LAT is a node such that the relation ending at this node (in LAT) owns some entries which are not connected to X. These nodes are called unsatisfied entries.
- BEG (Z) is the set of nodes in LAT which are less than Z and which are not unsatisfied entries.
- P is a "path" from BEG (U) to V, where U and V are nodes of LAT, if P is an oriented path starting from one node belonging to BEG (U) and ending at V. This path may contain incomplete nodes together with their unsatisfied entries*

Example of LAT:



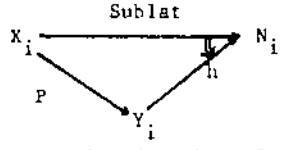
Nodes 6, 7, 11 are incomplete nodes
Nodes 12, 13, 14, 15 are unsatisfied entries
All others nodes are complete nodes.
BEG (9) - {X, 1, 2, 7, 8}

Algorithm 1:

1. LAT «- X
2. Look for a path P between X and Y.
3. Add path P to LAT.
4. Lf there is no incomplete node in LAT then stop
   else select the minimal one and call it N. (It can be demonstrated that Algorithm 1 generates a set of incomplete nodes which is totaly ordered on LAT).

5. Let Y_i be one unsatisfied entry of node N_i
6. Look for a path P between $BEG(N_i)$ and Y_i. Let X_j be the origin of P on $BEG(N_i)$ and try to detect a recursivity between X_j and Y_i using Algorithm 2.
7. Go to step 3.

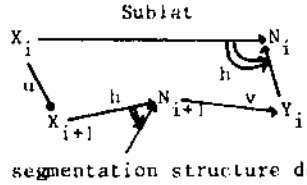
It follows from algorithm 1 that when Algorithm 2 is called, a part of LAT has the following structure:



where sublat is restricted to be a lattice, and Y_i is the previous unsatisfied node.

Algorithm 2:

1. If no path from X_j to N_i (in sublat) matches a subpath of P (in the sense of an identical sequence of operators) then stop algorithm 2,
   else let $X_{j+1} \to N_{j+1}$ be the subpath of P which has been matched.
   The above structure is changed to:



2. If the segmentation structure does not contain a lattice starting in X_{j+1}, ending in N_{j+1} and analogous to sublat then stop algorithm 2 else assume the following recursion:

$$X_j \xrightarrow{\varphi} N_i$$

$$\varphi(x) = h\ [\text{sublat}\ (x),\ v(\varphi(u(x)))]$$

or $X_j \xrightarrow{\varphi} N_i$

$$\varphi(x) = h[v(\varphi(u(x))),\ \text{sublat}(x)]$$

depending on the order of arguments of h.

3. Find a primitive stop condition of the recursive function φ as follows: match the operators of path P from X_{j+1} to N_{j+1} in the segmentation structure, then from X_{j+2} to N_{j+2} and so on, until it fails. Assume it fails from X_j to N_j. Find a path w from X_j to N_j. The primitive stop condition is assumed to be:

   if $X \in X_j$, then $w(X)$

4. Reduce the primitive stop condition as follows:
   a) remove Sublat from LAT.
   b) if a relation from N_k to N_{k+1} for every k, $j \le k < i$, cannot be found in the segmentation structure, then set $P \leftarrow (X_j \xrightarrow{\varphi} N_i)$ and stop algorithm 2 else let r be the found relation.
   c) find k, the smaller non negative integer such that $w_k = r^k(w(x))$ is not a fixed point of equation:

$$h\ [\text{sublat}\ (u^k(x)),\ v(r(w_k(x)))] = w_k(x)$$

for every $x \in X_j$.

   d) Set $P \leftarrow (X_j \xrightarrow{\varphi} N_i)$

$$\text{with } \varphi(x) = \begin{cases} \text{if } x \in u^k(X_j) \text{ then } z_k(x) \\ \text{else } h\ [\text{sublat}\ (x),\ v(\varphi(u(x)))] \end{cases}$$

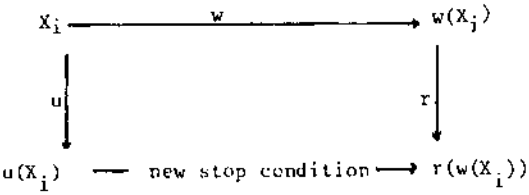
   where z_k is solution of the following equation:

$$z_k(u^k(x)) = r^k(w(x)) \text{ for every } x \in X_j$$

Remark : To reduce the primitive stop condition, the following process is iterated:
Suppose the last stop condition is

if $x \in X_j$ then $w(x)$

Using the functions u and r, it is possible to calculate $\varphi(x)$, $x \in X_j$, as shown on the following figure:



that is: $\varphi(x) = h\ [\text{sublat}\ (x),\ v(\varphi(u(x)))]$
$= h\ [\text{sublat}\ (x),\ v(r(w(x)))] = w(x)$

which means, if the correct answer w(x) is obtained that w(x) is a fixed point of the last equation.

It thus follows that there exists a function z such that:

$$z(u(x)) = r(w(x)) \text{ for every } x \in X_j$$

Thus now the stop condition is:

if $x \in u(X_j)$ then $z(x)$

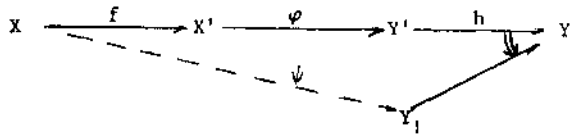
5.- Synthesis from two examples

Let us suppose that after a first example, SISP/1 generates a function which fails on a second example. Let the two examples be (x,y) and (x',y').

The principle is always to build a structure from the generation of segmentation-patterns; SISP/1 here generates the segmentation patterns associated with the initial pairs (x,x'),(y,y'),(x,y),(x',y') and goes on in the same way as in the first method.

SISP/1 then tries to find a three parts splitted path from X to Y:
- a path from X to X'.
- the function φ itself from X' to Y'.
- a path from Y' to Y.

Remarks: - using this technique, SISP/1 looks explicitely for a recursive form of the function φ.
- when unsatisfied nodes are remaining in the path, SISP/1 generates sub-problems which are to be solved either by algorithm 1 or again by using one more example when algorithm 1 fails [3 bis]. For instance, let Y_1 be a remaining unsatisfied node in the path:



SISP provides the following expression of φ:

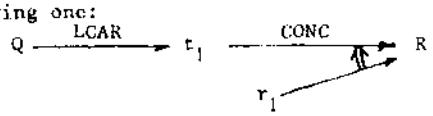
$$\varphi(x) = \begin{cases} \underline{\text{if}}\ x \in X'\ \underline{\text{then}}\ w(x) \\ \underline{\text{else}}\ h\,[\varphi(f(x)),\ \psi(x)] \end{cases}$$

where ψ is a sub-problem to be solved by SISP/1 and where $w(x)$ is the function which has been found by algorithm 1 working on only the example (x',y'). This stop condition can then be reduced as explained in algorithm 2.

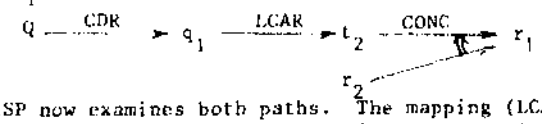
III. PRACTICAL EXAMPLES

1. Let us use our method to find the REVERSE function. The input (A B C D E) is given to SISP/1: it does not know the answer and asks the Professor who returns: (E D C B A). SISP analyses input and output and generates the segmentation structure indicated in figure 3.

SISP looks then for a path from the Question Q containing the list (A B C D E) to the answer R containing (E D C B A) and finds the following one:



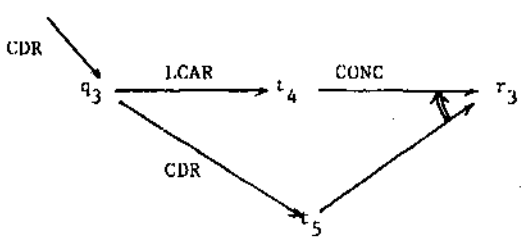
Looking for the unsatisfied entries, SISP finds r_1. It looks again for a path from BEG(R) to r_1, and finds the following one:



SISP now examines both paths. The mapping (LCAR-CONC) of the first one matches into the mapping (CDR-LCAR-CONC) of the second one. This statement is sufficient to infer a recursive expression φ:

$$\varphi(x) = \text{CONC}\,[\varphi(\text{CDR}(x)),\ \text{LCAR}\ (x)]$$

SISP has to still find the stop condition. Matching the three operators CDR, LCAR, CONC with the structure, it remarks that it can apply CDR on type q_3 giving t_5 but cannot apply LCAR on type t_5. SISP/1 thus knows that the stop condition is to be found in this part of the structure:



SISP/1 tries now to find a new binding of the path from t_5 to the unsatisfied entry of r_3 which happen here to be identical to t_5. It finds the trivial one and generates the primitive stop condition:

$$\underline{\text{if}}\ x \in t_5\ \underline{\text{then}}\ x$$

SISP now has to reduce the stop condition, using the following mappings:

$$q_i \xrightarrow{\text{CDR}} q_{i+1} \qquad r_i \xrightarrow{\text{RDC}} r_{i+1}$$

Assuming that CDR ((E)) = NIL $\xrightarrow{\varphi}$ RDC((E)) = NIL is the first reduced stop condition, SISP calculates now $\varphi(x)$ for every x belonging to t_5:

$$\varphi(t) = \text{CONC}\,[\varphi(\text{CDR}((E))),\ \text{LCAR}((E))]$$
$$= \text{CONC}\,[\varphi(\text{NIL}),\ (E)]$$

using the new stop condition: $\varphi(E) = \text{CONC}\,[\text{NIL},\ (E)] = (E)$ which here gives the correct answer.

The process cannot be performed further, because CDR(CDR((E))) does not exist. The function generated by SISP is thus:

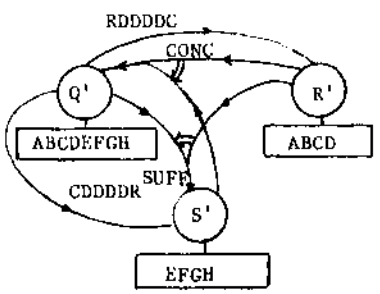
$$\varphi(x) = \begin{cases} \underline{\text{if}}\ x = \text{NIL}\ \underline{\text{then}}\ \text{NIL} \\ \underline{\text{else}}\ \text{CONC}\ (\varphi(\text{CDR}(x)),\ \text{LCAR}\ (x)) \end{cases}$$

that is the usual REVERSE.

2. The second example we display now needs more material than the first one since two couples (input - output) are necessary. Let HALF be the function to synthetize:

- first couple: (A B C D E F G H) → (A B C D)

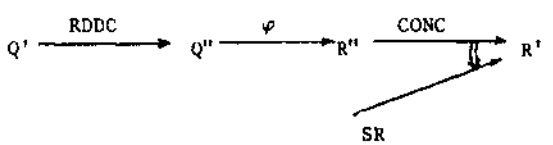
From this first example, SISP/1 constructs the following structure:



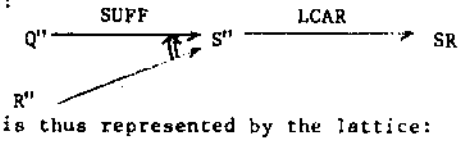
The relation φ:Q' → R' found here is thus $\varphi(x)$ = RDDDDC(x).

The professor gives now the following input: (A B C D E F). SISP uses φ to answer (A B), which is false. The professor then gives the correct answer: (A B C D E F) → (A B C).

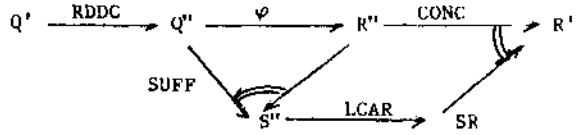
SISP then generates the structure displayed on figure 3. Assuming that R" can be obtained from Q" using the correct function φ to be synthetized, SISP, as explained before, uses the following path from Q' to R':



The problem is now to find a path from BEG (R') to SR. The simplest one which is found here is:



φ is thus represented by the lattice:



it follows that $\varphi(x)$ = CONC [φ(RDDC(x)), LCAR(SUFF [φ(RDDC(x)), RDDC(x)])] with the trivial stop condition:

if x ∈ Q" then RDDDC(x)

SISP now has to reduce this stop condition using the following mappings:

Q' —RDDC→ Q" and R' —RDC→ R"

Assuming that RDDC ((A B C D E F)) = (A B C D) —φ→ RDC ((A B C D)) = (A B)

SISP computes now $\varphi(x)$ for every x belonging to Q":

φ((A B C D E F)) = CONC [φ(RDDC((A B C D E F)), LCAR(SUFF [φ(RDDC((A B C D E F)), RDDC((A B C D E F))])]

= CONC [φ((A B C D)), LCAR(SUFF [φ((A B C D)),(A B C D)])]

= CONC [(A B), LCAR(SUFF [(A B), A B C D)])]

= CONC [(A B), (C)] = (A B C)

which is the correct answer.

The primitive stop condition can thus be reduced to:

if x ∈ RDDC(Q') then RDDC(x)

where RDDC is the solution of the following equation on z

z(RDDC(x)) = RDC (RDDDC(x)) for every x ∈ Q".

This process is recursively applied and stops when RDDC ((A B)) = NIL. At this step, we obtain the function HALF defined as follows:

$$\varphi(x) = \begin{cases} \underline{\text{if}}\ \text{RDDC}(x) = \text{NIL}\ \underline{\text{then}}\ \text{RDC}(x) \\ \underline{\text{else}}\ \text{CONC}\ [\varphi(\text{RDDC}(x)), \text{LCAR}(\text{SUFF}\ [\varphi(\text{RDDC}(x)), \text{RDDC}(x)])] \end{cases}$$

IV. LIMITS AND PROSPECTS OF THE METHOD

1.- Prospects

SISP/1 is already able to synthetize most of the functions given in SUMMERS [5] and HEDRICKS [2] in particular it synthetizes the following ones by using algorithm I:

(A B C D E) → (E D C B A)
(A B C D E) → (A X B X C X D X E X)
(A B C D E) → (A A B B C C D D E E)
(A B C D E) → (A A B A B C A B C D A B C D E)

By using two or more examples it synthetizes:

(A B C D E F G) → (A A G G B B F F C C E E D D)
(A B C D E F G H) → (A B C D)
(A B C D E) → (E D C B A E D C B E D C E D E)
(A B C D) → (D C B A C B A B A A D C B C B B D C C D)
(A B C D E) → (A B B C C C D D D D E E E E E)
(A B C D E F G H) → (D C B A H G F E)
(A B) → (A A A A A A A A)(cube of the entry length)
(A B C D) → (A A A A A A A A) (half square): such a way is not always easy to use, as we shall see now:

- assume SISP has to synthetize the HALF function using the previous example (A B C D E F G H) → (A B C D). Algorithm I fails and the professor gives as second example:

(B C D E F G) → (B C D)

SISP here generates the following HALF function:

$$\varphi(x) = \begin{cases} \underline{\text{if}}\ x = \text{NIL}\ \underline{\text{then}}\ \text{NIL} \\ \underline{\text{else}}\ \text{CONC}\ [\text{LCAR}(x), \varphi(\text{CDR}(\text{RDC}(x)))] \end{cases}$$

which is much simpler than previous HALF function. This simplicity was however found by the professor who gave better examples.

- assume now that SISP has to synthetize a function using the example (A B C D E) → (A B B C C C D D D D E E E E E). Algorithm I fails and the professor gives as second example : (A B C D) → (A B B C C C D D D D). SISP generates the following functions:

$$\varphi(x) = \begin{cases} \underline{\text{if}}\ x \in Q"\ \underline{\text{then}}\ R" \\ \underline{\text{else}}\ \text{CONC}\ [\varphi(\text{RDC}(x)), \psi(x)] \end{cases}$$

where $\psi(x)$ is bound to the following subproblem:

(A B C D E) → (E E E E E)

Algorithm I fails then to provide a correct function ψ and the professor now has to give the two particularly well choosen examples:

(B C D E) → (B C C D D D E E E E)
(B C D) → (B C C D D D)

they allow SISP to generate a new appropriate example in order to synthetize a correct ψ:

$$\psi(x) = \begin{cases} \underline{\text{if}}\ x = \text{NIL}\ \underline{\text{then}}\ \text{NIL} \\ \underline{\text{else}}\ \text{CONC}\ [\psi(\text{CDR}(x)), \text{LRAC}(x)] \end{cases}$$

the stop condition of φ is then found:

if x = NIL then NIL

the generated function φ will thus be given by the linear recursive system Φ

$$\Phi: \begin{cases} \varphi(x) = \underline{\text{if}}\ x = \text{NIL}\ \underline{\text{then}}\ \text{NIL}\ \underline{\text{else}}\ \text{CONC}\ [\varphi(\text{RDC}(x)), \psi(x)] \\ \psi(x) = \underline{\text{if}}\ x = \text{NIL}\ \underline{\text{then}}\ \text{NIL}\ \underline{\text{else}}\ \text{CONC}\ [\psi(\text{CDR}(x)), \text{LRAC}(x)] \end{cases}$$

These two last examples show the main importance of good examples. We hope however that it would be possible to use "bad examples" joined together with a unification process, in order to improve the given "bad examples".

2.- Limits

- with the exception of stop-condition, the functions generated by SISP do not use predicates in their definition . Thus the function:

if "length of x is even" then reverse (x) else x

cannot be synthetized by SISP. This problem is attacked in [6].

- SISP/1 requires a good sequence of example in order to use the second technique. They have to be of decreasing length and consecutive.
- SISP/1 only works on atomic lists.

V. CONCLUSION

In summary, the described method consists of constructing a structure from an adequate set of

examples $\{(x_i, f(x_i))\}$ and in extracting from the structure a lattice which represents an expression of the function f.

SISP is a LISP program working on PDP 10 using VLISP 10 [1].

Future developments will tend to make SISP able to:

- define and store self contained problems in its memory,
- recognize that a partial problem has already been encountered and solved,
- improve the professor's bad examples in order to be able to solve partial problems which have never before been encountered,
- synthetize n-any functions (the two presented techniques can easily be generalized).

VI. BIBLIOGRAPHY

[1] GREUSSAY P., at all: "VLISP 10. Reference manuel". Rapport interne Univ. Vincennes.

[2] HEDRICKS C.L.: "Learning Production System from examples", Artificial Intelligence Journal n° 7, January 1976.

[3] JOUANNAUD J.P., GUIHO G., TREUIL J.P., COALLAND P.: "SISP, système interactif de synthèse de programmes à partir d'exemples". Publication de l'Institut de Programmation, Mars 1977.

[3bis] JOUANNAUD J.P., GUIHO G.: "Inference of functions with an interactive system" to be published in Machine Intelligence n° 9 edited by D. MITCHIE.

[4] POHL Ir.: "Bi-Directional and Heuristic Search in Path Problems", Thesis, Computer Science Dept. Stanford University, 1969.

[5] SUMMERS P.D.: "Program Construction from examples". Phd. Thesis, December 1975.

[6] TREUIL J.P., JOUANNAUD J.P., GUIHO G.: "Une méthode d'apprentissage de concepts, Colloque APCET: Panaroma de la Nouveauté informatique en France, Gif s/Yvette, Novembre 1976.

[7] TREUIL, J.P., JOUANNAUD J.P., GUIHO,G.: "L.Q.A.S., un système question-réponses basé sur l'apprentissage et la synthèse de programmes à partir d'exemples. Publication de l'Institut de Programmation, Mars 1977.

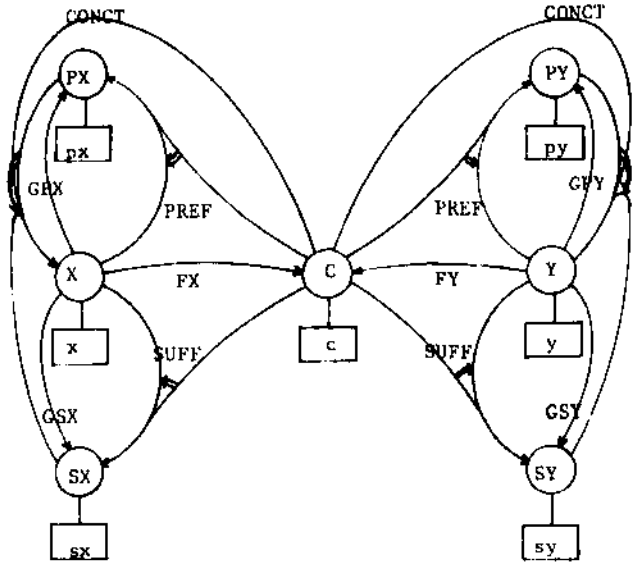


Figure 1: Segmentation pattern associated to (x,y).

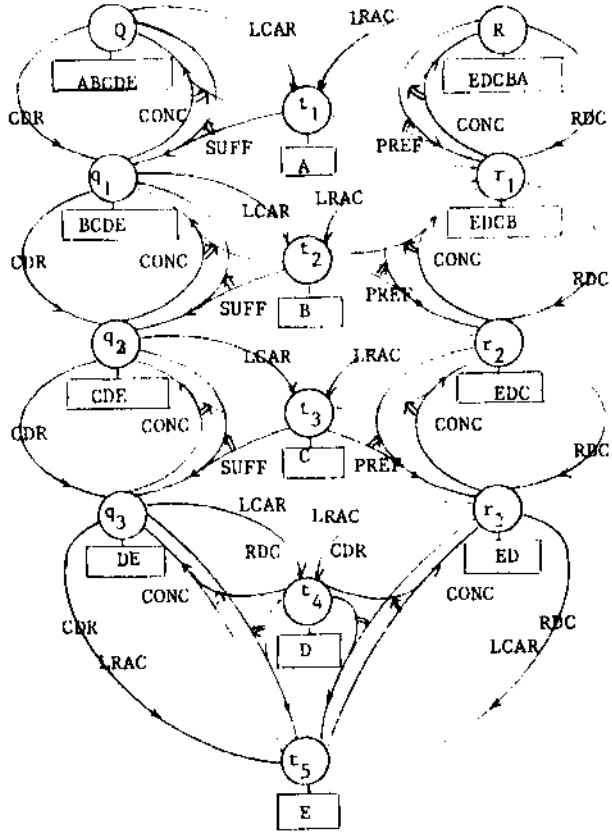


Figure 2: Segmentation structure associated to the REVERSE function.

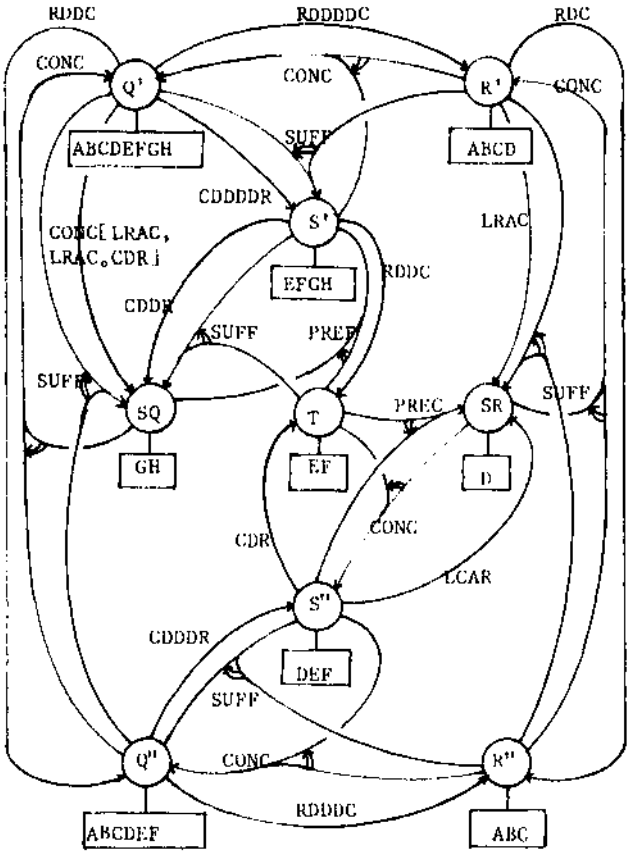


Figure 3: Structure associated to the HALF function.