

CHARACTERIZATION OF A CLASS OF FUNCTIONS SYNTHESIZED FROM EXAMPLES BY A SUMMERS LIKE METHOD USING A "B.M.W." MATCHING TECHNIQUE.*

J.P. JOUANNAUD
Centre de Recherche en Informatique de Nancy
UER de Mathématiques
Université de NANCY I
54037 NANCY CEDEX

Y. KODRATOFF
GR 22 du CNRS
Institut de Programmation, T 55-65
Université Pierre et Marie Curie (Paris VI)
4, place Jussieu
75230 PARIS CEDEX 05

Summary :

This paper describes first a program synthesis from examples method using a SUMMERS like methodology together with a sophisticated pattern matching technique, the BOYER-MOORE-WEGBREIT algorithm. We then characterize the class of functions synthesized from a restricted BMW algorithm. Our methodology is founded on three grounds. The first one is a precise characterization of the list domains fitting the synthesis from examples problem. The second one is SUMMERS' technique of input-output examples transformation into computational traces. The third one is the description by a restricted program scheme of the functions in the class. It seems to us that the proofs methods easily extend to the general class.

1. INTRODUCTION

The synthesis of functions from a set $\{x_i \rightarrow F(x_i)\}$ of input-output examples has been extensively studied [1,2,4,5,6,7,8,9,10,14,15,16] our purpose here is to study a powerful extension of SUMMER's method [16] using a pattern matching technique we shall call the BOYER-MOORE-WEGBREIT algorithm.

The principles of this technique are discussed in [3,8,9,10,17].

We recall briefly this method which is detailed in [8,9,10]. However we think that it is not enough to describe an algorithm, especially in our field : we must aim at the characterization of the class of functions which can be synthesized. A recent work of SMITH [15] describes such an investigation for a restricted SUMMER's method. The goal of this paper is to design more adapted tools which allow simpler proofs of our theorems. We are able to characterize the functions synthesized by a restricted BMW algorithm and prove a linear growth property of the functions in this class. We mean that our results shall extend to the general case.

2. SYNTHESIS METHODOLOGY

We recall briefly, in an intuitive way, the methodology relative to one variable functions [8,9,10,16]. We are concerned with LISP unary functions defined on lists. Before the synthesis process itself takes place, two transformations of the input-output values are necessary.

The first one transforms these values into a set of computational traces expressed with the set $\{\text{cons}, \text{car}, \text{cdr}, \text{atom}\}$ of primitive functions. The second one transforms the computational traces into a partial function, defined on subdomains of F .

In order to make our notations and results clearer, we shall describe the synthesis of an ad'hoc function F described by the following set of examples :

$x_0 = (A(B)C) \quad F(x_0) = (B) ;$
 $x_1 = (A(B(C(D)E))) \quad F(x_1) = (C \ C \ B) ;$
 $x_2 = (A(B(C(D(E(F)G)))) \quad F(x_2) = (C \ C \ D \ C \ B) ;$
 $x_3 = (A(B(C(D(E(F(G(H)I)))))) \quad F(x_3) = (C \ C \ C \ D \ C \ B)$

2.1. Notations and definitions

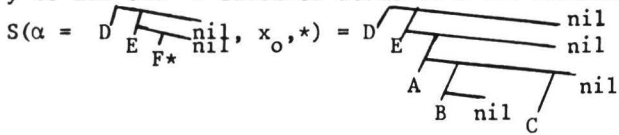
Let L be the set of LISP lists. A list x of L may be viewed as a cons-tree, each leaf of which is an atom.

For instance, $x_0 = (A(B)C)$ or $x_0 =$  nil

* Partly supported by IRIA-SESORI.

Let $|x|$ be the complexity of the list x , i.e. the number of leafs in the associated cons-tree.

Following KNUTH [11], we shall denote by $S(x, y, *)$ the list obtained by substitution of y to all the $*$ -labelled atoms of x . For instance



It is well known that the order associated to the substitution relation is a partial ordering of L .

Definition : A reduction function is a finite composition of car and cdr. Let R be the set of reduction functions. The length $|f|$ of $f \in R$ is the number of car and cdr in f .

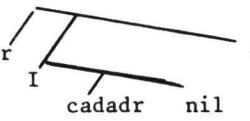
Notice that the triplet $\langle R, \text{composition rule}, I \rangle$, where I denotes the identity function, is isomorphic to the free monoid constructed on the alphabet $\{\text{car}, \text{cdr}\}$.

Definition : Let A_i be the i th occurrence of the atom A in the list x and u the unique reduction function which verifies $u(x) = A_i$. Then u is said to be the functional name of A_i in x . For instance, the name of F_* in α is cadadr .

Definition : If $v \in R$ is a right factor of $u \in R$, then $u v^{-1}$ denotes the function of R such that $u = (u v^{-1})v$. Let $u \text{ modulo } v$ be the function $u v^{-k}$ provided v^k is a right factor of u iff $0 \leq k' \leq k$.

Example : $u = \text{caadadr}$, $v = \text{cadr}$, $u v^{-1} = \text{caadr}$, $u \text{ modulo } v = \text{car}$.

Definition : Given a set of terminal functions (Generally $R \cup \{\text{nil}\}$) a cons-expression is a cons-tree, each leaf of which is a terminal function or nil.

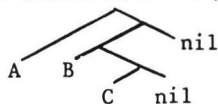
Example : $f = \text{car}$  caddr nil

Notice that nil may be viewed either as an atom or as a function.

Let ω be the undefined atom, $D(f)$ the domain of the function f , i.e. the set $\{x | x \in L, f(x) \neq \omega\}$

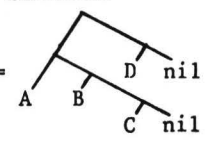
Definition : The adapted structure [15] $\text{ad}(f)$ of a function $f \in R$ is the smallest $x \in D(f)$.

For example, $\text{ad}(\text{caddr}) =$



Notice that the atoms of $\text{ad}(f)$ do not have any importance at all, i.e. $\text{ad}(f)$ is in fact a list structure.

Definition : The adapted structure $\text{ad}(F)$ of a set $F = \{f_i\}$, $f_i \in R \forall i$, is the smallest $x \in D(F) = \bigcap_i D(f_i)$.

For example, $\text{ad}\{\text{caddr}, \text{cddr}\} =$ 

In the following, we shall suppose that no atoms of $\text{ad}(F)$ bear the same name.

2.2. Computational traces

A main step in SUMMER's method is to transform the sequence $\{x_i \rightarrow F(x_i)\}$ of input-output examples in a sequence $\{p_i(x) \rightarrow f_i(x)\}$ of computational traces.

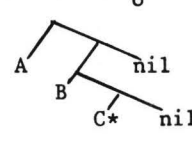
2.2.1. Ascending linear domains.

The further step of synthesis requires the existence of recurrence relations between the predicates p_i . It is thus natural to study the input domains that possess this property.

Definition : Given $x_0 \in L$, $\alpha \in L$, α owning a single $*$ -labelled atom, the set $\{x_0, x_{i+1} = S(\alpha, x_i, *)\}$ is called ascending linear domain $\mathcal{A}(x_0, \alpha)$.

In fact, we change the names of the atoms in order to avoid several occurrences of the same atom in x_i .

For instance, if $x_0 =$ 

and $\alpha =$ 

$\mathcal{A}(x_0, \alpha)$ is the set $\{(A(B)C), (A(B(C(D)E))), (A(B(C(D(E(F)G))))), \dots\}$

Notice that the partial ordering on L is total on $\mathcal{A}(x_0, \alpha)$.

We now have to exhibit a sequence p_i of predicates which verify :

$p_i(x_i) = \underline{T}$, $p_i(x_j) = \underline{F}$ iff $j > i$, $p_i(x_j) = \omega$ iff $j < i$, where $\underline{T}$ and $\underline{F}$ denote the logical values true and false.

Let us call b the function name of the $*$ -labelled atom in α and suppose that we know a reduction π such that $\text{atom. } \pi(x_0) = \underline{T}$.

It is obvious that $b(x_{i+1}) = x_i$ and it follows that the sequence $p_i = \text{atom}.\pi.b^i$ verifies

$p_i(x_i) = \underline{T}$. We still need to find the "good" function π , i.e. the function such that p_i verifies the two remaining conditions $p_i(x_j) = \underline{F}$ iff $j > i$ and $p_i(x_j) = \omega$ iff $j < i$.

Definition : An atom in a list x is said to be in the direction of $f \in R \setminus \{I\}$ iff its functional name has the form $f'.f$ where f' is a proper right factor of f , i.e. $f = f''.f'$, $f'' \neq I$.

For instance, nil_* in $x_0 = A \begin{array}{c} \swarrow \searrow \\ B \quad \text{nil}_* \end{array} C \text{nil}$ is in the direction of cadr : its functional name is $\text{cadr} = \text{cdr}.\text{cadr}$.

The intuitive meaning of this concept is the following : if we compute $\text{cadr}(x_0)$ we obtain $B \text{nil}_*$ which is not yet an atom. We cannot compute $\text{cadr}.\text{cadr}(x_0)$ which is not defined, but we start the computation and keep computing as long as we do not reach an atom : $\text{cdr}.\text{cadr}(x_0) = \text{nil}_*$ is an atom ; we therefore stop the computation at this point.

Property of the ascending linear domains : given an ascending linear domain $\mathcal{A}(x_0, \alpha)$ and the name b of the $*$ -labelled atom of α , let π be the functional name of the atom of x which is in the direction of b . Then $\{p_i = \text{atom}.\pi.b^i\}$ is a sequence of predicates for the domain.

Proof : $\pi = b'.b^P$, where $b = b''.b'$ and $b'' \neq I$.
- As $\text{atom}.\pi(x_0) = \underline{T}$ by definition of π , we

know that $p_i(x_i) = \underline{T}$, $\forall i$
- $j < i$: $p_i(x_j) = \text{atom}.\pi.b^i(x_j) = \text{atom}.b'.b^P.b^{i-j}(x_0)$
 $= \text{atom}.b'.b^{i-j-1}.b''.b'.b^P(x_0) = \text{atom}.b'.b^{i-j-1}.b''.\pi(x_0) = \omega$

because $\pi(x_0)$ is an atom and $b'' \in R \setminus \{I\}$.

- $j > i$: $p_i(x_j) = \text{atom}.\pi.b^i(x_j) = \text{atom}.\pi(x_{j-i})$.

Let us compute
 $p_j(x_j) = \underline{T} = \text{atom}.\pi.b^j(x_j) = \text{atom}.b'.b^{j-i-1}.b''.\pi(x_{j-i})$

It is obvious that if $p_i(x_j) = \text{atom}.\pi(x_{j-i}) = \omega$ or $\underline{T}$ then

$p_j(x_j) = \omega$ which is false. Thus $p_i(x_j) = \underline{F}$.

Notice that the sequence of predicates verifying these previous conditions is not unique in the sense that π is not unique in some particular cases.

Example : Let us consider the previous set $\mathcal{A}(x_0, \alpha)$. We obtain $b = \text{cadr}$ and $\pi = \text{cdadr}$.

Thus $p_0 = \text{atom}.\text{cdadr}$, $p_1 = \text{atom}.\text{cdadadr}$, $p_2 = \text{atom}.\text{cdadadadr}$. The match of p_i and p_{i+1} succeeds and we obtain :

$$p_{i+1}(x) = p_i(\text{cadadr } x)$$

More generally, $p_i = \text{atom}.\pi.b^i$ and $p_{i+1} = p_i.b$

2.2.2. Fragments

The output $F(x_i)$ is taken in its cons-tree form. Each atom, except nil , of this cons-tree is replaced by its functional name in x_i . The process is unambiguous iff there are not several atoms (except nil) bearing the same name in any x_i . Each obtained cons-expression $f(x_i)$ is then turned into a partial function $f_i(x)$ by the substitution of x to each x_i of $f(x_i)$. The function f_i is called the i th fragment of F . Notice that f_i is defined for any $x \geq x_i$ and equal to $f(x_i)$ when $x = x_i$.

Example :

$$F(x_0) = B \text{nil} \Rightarrow f(x_0) = \text{caadr} \text{nil} \Rightarrow$$

$$f_0(x) = \text{caadr} \text{nil}$$

$$F(x_1) = C \begin{array}{c} \swarrow \searrow \\ C' \quad B \text{nil} \end{array}$$

$$f_1(x) = \text{caadadr} \begin{array}{c} \swarrow \quad \swarrow \quad \swarrow \quad \searrow \\ x \quad \text{caadadr} \quad \text{caadr} \quad \text{nil} \end{array}$$

$$F(x_2) = C \begin{array}{c} \swarrow \quad \swarrow \quad \swarrow \quad \searrow \\ C' \quad D \quad C' \quad B \text{nil} \end{array}$$

$$f_2(x) = \text{caadadr} \begin{array}{c} \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \searrow \\ x \quad \text{caadadr} \quad \text{caadadadr} \quad \text{caadadr} \quad \dots \end{array}$$

$$F(x_3) = C \begin{array}{c} \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \searrow \\ C' \quad C' \quad E \quad D \quad C' \quad B \text{nil} \end{array} \quad \dots \quad \text{caadr} \text{nil}$$

$$f_3(x) = \text{caadadr} \begin{array}{c} \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \searrow \\ x \quad \text{caadadr} \quad \text{caadadr} \quad \text{caadadadadr} \quad \text{caadadadr} \quad \text{caadadr} \quad \text{caadr} \quad \text{nil} \end{array}$$

2.2.3. Approximations

The input-output sequence $\{x_i \rightarrow f(x_i)\}$ has been changed into a sequence of predicates and fragments $\{p_i(x) \rightarrow f_i(x)\}$ which is embedded into an if... then... else... function in order to obtain the following n-approximation of F :

```

if  $p_0(x)$  then  $f_0(x)$  else
if  $p_1(x)$  then  $f_1(x)$  else...
if  $p_n(x)$  then  $f_n(x)$  else  $\omega$ .

```

Notice that the names of the atoms are now lost we therefore cannot synthesize functions depending on these names, like the function which is true iff all the names of the atoms are different.

Our problem now is to get a finite expression of the ω -approximation.

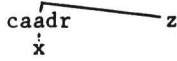
2.3. The synthesis technique

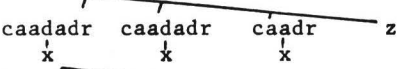
The principle of the method consists in detecting a repetitive computation on the successive traces.

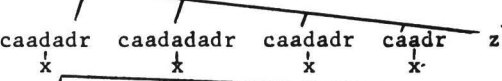
Predicates : We have the recurrence relation
 $(1) p_{i+1}(x) = p_i(\text{caddr } x)$.

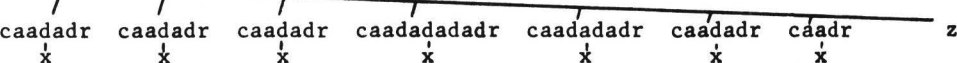
Fragments : We attempt to match $f_i(x)$ and $f_{i+1}(x)$ using the BMW algorithm. When it succeeds, we use its substitutions to write recurrence relations proved to be equivalent to programs by SUMMER's main synthesis theorem [16] or its extensions [9].

The classical pattern matching techniques [13] do not succeed to match the preceding sequence of fragments. In this case, a first order pattern matching attempts to use generalization techniques together with matches between subtrees, i.e. the BMW algorithm. It works as follows : A first generalization leads to replace the nil atom by a variable z in each fragment :

$f_0(x)$ becomes 

$f_1(x)$ becomes 

$f_2(x)$ becomes 

$f_3(x)$ becomes 

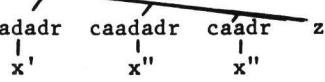
The matching is now possible : Each fragment is matched with the first subtree of the successive fragment, provided the following set of substitutions :

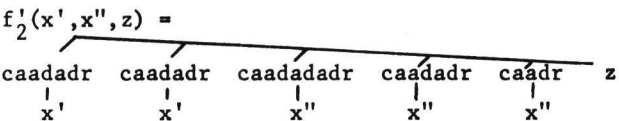
tutions :

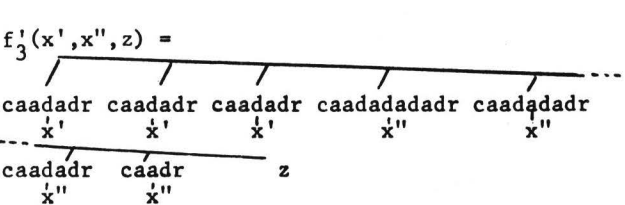
$\{x \rightarrow x, x \rightarrow \text{cadr } x, z \rightarrow \text{caadr } z\}$.

Since the variable x receives two different substitutions, a new generalization is necessary : the variable x becomes x' if it undergoes the first substitution and x'' if it undergoes the second substitution, and the fragment $f_i(x)$ becomes $f'_i(x', x'', z)$:

$f'_0(x', x'', z) =$ 

$f'_1(x', x'', z) =$ 

$f'_2(x', x'', z) =$ 

$f'_3(x', x'', z) =$ 

The matching between two consecutive fragments now succeeds and we find the following recurrence relations :

$$f'_{i+1}(x', x'', z) = h_i[x', f'_i(x', (\text{cadr } x''), g_i(x'', z))]$$

(2)

$$h_i(x', y) = (\text{cons}(\text{caadadr } x')y)$$

$$g_i(x'', z) = (\text{cons}(\text{caadr } x'')z)$$

Notice that $\{h_i\}$ and $\{g_i\}$ are two sequences of fragments associated to the sequence of predicates p_{i+1} . In the present case, h_i and g_i are

constant cons-expressions. Sometimes h_i and g_i are not constant. When it is the case, we use repetitively the same algorithm until we get

constant h and g .

In the following, we shall restrict ourselves to a one step BMW algorithm (the above example behaves in this way).

Since we use matches between subtrees, the succeeding match is generally not unique. One should not worry about this: we shall prove that all the matches lead to equivalent programs.

From the relations (1) and (2) a program is derived using the following extension [9] of SUMMER's Synthesis Theorem (MST):

Theorem: If the infinite sequences $\{p_i\}$ and $\{f_i\}$ verify the recurrence relations:

$$\begin{aligned} p_{i+1}(x) &= p_i(b(x)) \\ f_i(x) &= f_i'(x^n, x, \text{nil}^m) \text{ with } x^n = x, \dots, x \text{ } n \text{ times} \\ &\text{and } \text{nil}^m = \text{nil}, \dots, \text{nil } m \text{ times.} \\ f_{i+1}'(x, xx, z) &= h[\vec{x}, \vec{f}_i'(b'(x), b(xx), \vec{g}(x, z))] \\ &\text{where } b'(x) \text{ is } b_1'(x_1), \dots, b_n'(x_n), \vec{g}(x, z) \text{ is } \\ &g_1(x, z), \dots, g_m(x, z) \quad e \in R - \{I\}, b = e^q, \\ &b_i' = e^{r_i}, 0 \leq r_i \leq q, 0 < q. \end{aligned}$$

h and g are constant cons-expressions then the ω -approximation associated to the computational traces sequence $\{p_i(x) \rightarrow f_i(x)\}$ is equivalent to the program:

$$\begin{aligned} F(x) &\leftarrow F'(x^n, x, \text{nil}^m) \\ F'(x', xx, z) &\leftarrow \text{If atom}(xx) \text{ then } f_0'(x, z) \\ &\quad \text{else } h[\vec{x}, F'(b'(x), b(xx), \vec{g}(x, z))] \end{aligned}$$

The xx variable, defined as the domain variable, is introduced to take into account the recursion on $p_i(x)$. One realizes its importance when

we look back to the previous example: neither x' nor x'' have the same relation as xx :

$$\begin{aligned} F(x) &\leftarrow F'(x, x, \text{nil}) \\ F'(x', x'', xx, z) &\leftarrow \text{If atom}(caddr xx) \text{ then} \\ &\quad (\text{cons}(caddr x'') z) \\ &\quad \quad h(x', x'', xx, F'(x', (cadr x''), \\ &\quad \quad \quad (caddr xx), g(x', x'', xx, z))) \\ h(x', x'', y) &\leftarrow (\text{cons}(caaddr x') y) \\ g(x', x'', z) &\leftarrow (\text{cons}(caadr x'') z) \end{aligned}$$

In the following, the symbol F used as an example will always denote the above function.

Remark: Given a function, the recurrence relations are supposed to begin with the first example of the function. For the sake of readability, we shall rule out the trivial cases where the first examples of the function's behaviour (i.e. a finite number of them) have particular properties.

3. THE SYNTHESIS ALGORITHM AND THE RESTRICTED PROGRAM SCHEME.

Let us recall that we are only concerned with unary LISP functions given by a sequence $\{x_i \rightarrow F(x_i)\}$ of examples, the set $\{x_i\}$ of inputs belonging to an ascending linear domain $\mathcal{A}(x_0, \alpha)$.

Synthesis algorithm

1. Compute the functional name b of the unique $*$ -labelled atom of α .
2. Compute the functional name π of the unique atom in x_0 which is in the direction of b .
3. Compute the fragments f_i .
4. Look for recurrence relations between consecutive fragments using the BMW algorithm restricted to one step.
5. If recurrence relations are found then use the MST. Our present aim is to precise which functions can be synthesized and how many examples are needed.

Let us call $\mathcal{F}\mathcal{A}$ the set of functions which can be synthesized. If $F \in \mathcal{F}\mathcal{A}$, obviously F verifies the program scheme used in the MST. Our purpose is to prove the reciprocal. Following SMITH [15], we shall first introduce some very natural restrictions on the program scheme which come from the fragment computation technique.

Program scheme 0:

F is a cons-tree whose terminal functions g_i belong to $R \cup \{\text{nil}\}$

Restriction: $\forall g_1, g_2, g_1 = v.g_2 \Rightarrow v = I$.

Notice that this restriction comes very naturally: It means that g_1 and g_2 are terminal functions of fragments (Since F is constant, $f_i = F$, $\forall i$) and thus $g_1(x_i)$ and $g_2(x_i)$ are both atoms.

It follows that $v.g_2(x_i)$ is defined iff $v = I$.

Program scheme 1:

$$\begin{aligned} F(x) &\leftarrow F'(x, x, \text{nil}) \\ F'(x, xx, z) &\leftarrow \text{If } p_0(xx) \text{ then } f_0'(x, z) \\ &\quad \text{else } h[\vec{x}, F'(c(x), b(xx), \vec{g}(x, z))] \end{aligned}$$

In fact x, c and z in the F' definition should be vectors as shown by the example. We do not use vectors in order to lighten the notations but this simplification implies no loss of generality.

We shall call $\mathcal{F}\mathcal{S}$ the set of functions which can be computed by an instance of this restricted schema.

Restrictions :

- 1/ $\exists e \in R - \{I\}, \exists o < q, \exists o \leq r \leq q$ such that $b = e^q, c = e^r$.
- 2/ $p_o = \text{atom}, \pi, \pi = b'b^p, p \geq o, b = b''b', b'' \neq I$.
- 3/ f'_o is a cons-tree whose terminal functions $f'_{o,i}$ belong to $R - \{I\} \cup \{\text{nil}\}$ when applied to x and equal I when applied to z .
- 4/ h is a cons-tree whose terminal functions h_i belong to $R - \{I\} \cup \{\text{nil}\}$ when applied to x and equal I when applied to F' .
- 5/ g is a cons-tree whose terminal functions g_i belong to $\{R\} - \{I\} \cup \{\text{nil}\}$ when applied to x and equal I when applied to z .
- 6/ Let X_o be the set of functions defined below which lead to a computation on x_o starting from a computation $F(x_k)$:

$$\forall f, g \in X_o, f = v.g \Rightarrow v = I.$$
- 7/ Let $\mathcal{A}$ be the set of functions defined below which lead to a computation on α starting from a computation $F(x_k)$:

$\forall f \in \mathcal{A} \Rightarrow f \neq I$ and $\forall f, g \in \mathcal{A}, f = v.g \Rightarrow v = I$.

These two last restrictions mean that functions of X_o and $\mathcal{A}$ come from fragments. It follows that if g belongs to X_o , then $g(x_o)$ is an atom. Thus $v.g(x_o)$ is defined iff $v = I$.

$X_o = \{\pi\} \cup \bigcup_i \{f'_{o,i}.c^k.b^{-k} \mid o \leq k, b^k \text{ is a right factor of } f'_{o,i}.c^k\}$

$\bigcup_i \{g_i.c^j.b^{-k} \mid o \leq j < k, b^k \text{ is a right factor of } g_i.c^j\}$

$\bigcup_i \{h_i.c^j.b^{-k} \mid o \leq j < k, b^k \text{ is a right factor of } h_i.c^j\}$

$\mathcal{A} = \{b\} \cup \bigcup_i \{f'_{o,i}.c^j \text{ modulo } b \mid o \leq j, r \neq q\},$

where r and q are defined in restriction 1

$\bigcup_i \{g_i.c^j \text{ modulo } b \mid o \leq j\}$

$\bigcup_i \{h_i.c^j \text{ modulo } b \mid o \leq j\}$

Example : In the above example (section 2) c was a two-dimensional vector (c_1, c_2) . Its first component, c_1 , is associated with the terminal functions which compute on x' and c_2 is associated with the terminal functions which compute

on x'' .

$\pi = \text{cdadr} ; \bigcup_i \{f'_{o,i}.c^k.b^{-k}\} = \{\text{caadr}, \text{car}\} ;$
 $\bigcup_i \{g_i.c^j.b^{-k}\} = \{\text{car}\} ; \bigcup_i \{h_i.c^j.b^{-k}\} = \text{car} ; .$

Thus $X_o = \{\text{car}, \text{caadr}, \text{cdadr}\}$

$b = \text{cadadr} ; \bigcup_i \{f'_{o,i}.c^j \text{ modulo } b\} = \{\text{caadr}, \text{car}\} ;$
 $\bigcup_i \{g_i.c^j \text{ modulo } b\} = \{\text{caadr}, \text{car}\} ; \bigcup_i \{h_i.c^j \text{ modulo } b\} = \{\text{car}\} ; .$

Thus $\mathcal{A} = \{\text{car}, \text{caadr}, \text{cadadr}\}.$

Let us call $\mathcal{F}_1$ the set of functions which can be computed by a program which is an instance of this restricted scheme and $\mathcal{F}$ the set $\mathcal{F}_o \cup \mathcal{F}_1$.

Theorem : $\mathcal{F} = \mathcal{F}$.

Before proving this theorem, we shall first express the fragment f_k in terms of the functions f'_o, h and g which appear in the program scheme.

Lemma 1 : $f_k(x) = h[x, h[c(x), \dots, h[c^{k-1}(x), f'_o[c^k(x), g(c^{k-1}(x), \dots, g(c(x), g(x, \text{nil})) \dots)] \dots]]]$

This lemma is easy to prove by induction on k . As a consequence, the set of terminal functions of f_k is $\{h_i.c^j, f'_{o,i}.c^k, g_i.c^j \mid j \leq k\}$.

First part of the proof : $\mathcal{F} \subseteq \mathcal{F}$

Two cases may occur : the computed fragments are constant or not. If they are constant, the program scheme o is used and the restriction holds, else scheme 1 is used and we have to prove the restrictions 1 to 7.

Lemma 2 : Restriction 1 holds.

Proof : The matching of the fragments provides a function $c \in R$ such that $\forall i > o, j < i, c^j(x_i) \neq \omega$. This implies that either $c = I$ and lemma 2 is proved with $e = b, q = 1$ and $r = o$, or $c \neq I$ and c and b "compute in the same direction".

It follows that $b|c| = c|b|$.

As b and c belong to the free monoid constructed on the alphabet $\{\text{car}, \text{cdr}\}$ they are powers of the same $e \in R$. Thus $b = e^q$ and $c = e^r$. Moreover if $q < r$, then for k large enough $c^{k-1}(x_k)$ would be ω since $b^{k-1}(x_k) = x_1$.

Restriction 2 holds because $\mathcal{A}(x_o, \alpha)$ is an ascending linear domain.

Restriction 3, 4, 5 come from the working technique of the BMW algorithm.

Lemma 3 : $\forall f \in X_o, \text{atom } f(x_o) = \underline{I}$.

Restriction 6 follows easily.

Lemma 4 : $\forall f \in \mathcal{A}$, atom. $f(\alpha) = \underline{T}$.

Restriction 7 follows then easily and the first part of the theorem is proved.

Second part of the proof : $\mathcal{P}\mathcal{F} \subset \mathcal{J}\mathcal{A}$. Three steps are needed.

First step.

We have to exhibit an ascending linear domain (x_0, α) . This domain should be the "smallest" domain on which the synthesis of the known function F is possible. Given $F \in \mathcal{P}\mathcal{F}$, we naturally take as x_0 and α :

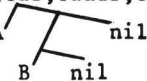
$$x_0 = \text{ad}(X_0), \quad \alpha = \text{ad}(\mathcal{A})$$

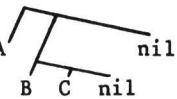
This has a meaning, because :

Lemma 5 : X_0 and $\mathcal{A}$ are finite sets.

The proof is simple.

Example : for the example F we have $X_0 = \{\text{car}, \text{caadr}, \text{cdadr}\}$, $\mathcal{A} = \{\text{car}, \text{caadr}, \text{cadadr}\}$,

It follows that : $x_0 = A$ 

$\alpha = A$ 

Notice that the new x_0 is simpler than the previous one which contained useless information. To define $\mathcal{F}(x_0, \alpha)$, we need a $*$ -labelled atom in α , the functional name of which is b : it is thus the atom C .

Second step.

Given the set $\{x_i\}$ of inputs, it is now possible to compute the set $\{x_i \rightarrow F(x_i)\}$ of examples and the set $\{p_i'' \rightarrow f_i''\}$ of computational traces.

We shall now prove that the sets $\{p_i \rightarrow f_i\}$ and $\{p_i'' \rightarrow f_i''\}$ are identical, where $p_i = \text{atom}.\pi. b^i$ and f_i is defined in lemma 1.

Lemma 6 : Given a set G of reduction functions and $x = \text{ad}(G)$, if $\text{atom}.f(x) = F$ then there exists $g \in G$ such that $g = v.f, v \in R - \{I\}$.

Lemma 7 : If $G = \{g_i\}$ and $\forall g_1, g_2 \in G, g_1 = v.g_2 \Rightarrow v = I$, then $\text{atom}.g_i(\text{ad}(G)) = \underline{T}$.

Lemma 8 : $p_i'' = \text{atom}.\pi.b^i$.

Lemma 9 : $\forall_k, f_k'' = f_k$.

Third step.

We now may match the consecutive fragments. We know recurrence relations between these fragments : they derive from the given program F . Nevertheless, several matches may succeed and

lead to different programs with program scheme 0 or 1. We thus have to prove that all these different matches lead to equivalent programs, although a finite number of examples are used to compute the fragments and the recurrence relations.

Notice that there is no problem if F is given by a program which is an instance of the program scheme 0. In this case, F will be synthesized with the same program. Difficulties appear when F is given by a program which is an instance of the program scheme 1.

Before solving this problem, we shall prove a property of the class $\mathcal{P}\mathcal{F}$.

Property 1 : $F \in \mathcal{P}\mathcal{F} \Rightarrow |F(x_k)|$ is a constant.

Property 2 : $F \in \mathcal{P}\mathcal{F}_1 \Rightarrow |F(x_k)|$ is a constant or a linear function of $|x_k|$.

Lemma 10 : If F is an instance of the program scheme 1 and F'' an instance of the program scheme 0 which take the same values in x_0 and x_1 , then F and F'' are equivalent. The synthesis process of F needs two examples : $x_0 \rightarrow F(x_0)$ and $x_1 \rightarrow F(x_1)$.

Lemma 11 : If F and F'' are two instances of the program scheme 1, which take the same values in x_0, x_1 and x_2 , then F and F'' are equivalent. The synthesis process of F needs three examples $x_0 \rightarrow F(x_0), x_1 \rightarrow F(x_1)$ and $x_2 \rightarrow F(x_2)$.

Proof : A detailed proof is rather technical and can be found in [7 bis]. Its principle is : characterize the links between the two sets $\{\pi, b, c, f_0', h, g\}$ and $\{\pi, b, c'', f_0'', h, g\}$ (which define F and F'') when F and F'' have the same values on x_0, x_1 and x_2 . These links enable us to prove easily that the sequences of fragments $\{f_k'\}$ and $\{f_k''\}$ are identical.

We have thus proved that any function in $\mathcal{P}\mathcal{F}_0$ is synthesized from 2 examples and in $\mathcal{P}\mathcal{F}_1$ from 3 examples.

4. CONCLUSION

We have obtained a characterization of the class of functions synthesized from examples, when a restricted BMW algorithm is used. Moreover, we have proved an important property of the functions in this class : the existence of a deep relation between the number of examples needed for their synthesis and their growth speed. These above results, obtained for the classes $\mathcal{P}\mathcal{F}$ and $\mathcal{P}\mathcal{F}_1$, may be extended to the class $\mathcal{P}\mathcal{F}_n$ defined by the use of an n -step BMW algorithm. A key step for this proof is the

obtention of an equivalence result between an instance of the program scheme n and an instance of the program scheme $m \leq n$ which take the same values for the inputs $x_0, x_1, \dots, x_{n+1}$.

A further interesting problem should be to obtain an intuitive characterization of the class $\mathcal{P}_n = \bigcup_n \mathcal{P}_n$. We mean that there is a strong analogy between the properties of the polynomials and those of $\mathcal{P}_n$.

The BMW algorithm has been implemented in LISP by E. PAPON (unpublished "stage de D.E.A." Univ. Paris-Sud).

REFERENCES

- [1] BIERMANN A.W. : The inference of regular LISP programs from examples, I.E.E.E. Trans. on Systems, Man and Cybernetics Vol. SMC - 8, (1978), 585-600.
- [2] BIERMANN A.W., SMITH D.R. : "The hierarchical synthesis of LISP scanning programs", Information Processing 77, B. GILCHRIST ed. (North Holland, 1977) pp. 41-45.
- [3] BOYER R.S., MOORE J.S. : "Proving theorems about LISP functions", JACM-22, vol 1, (1975) pp. 129-144.
- [4] HARDY J. : "Synthesis of LISP functions from examples", Proc. 4th IJCAI (1975) pp. 268-273.
- [5] JOUANNAUD J.P., GUIHO G., TREUIL J.P. : "SISP/1, an interactive system able to synthesize functions from examples", Proc. 5th IJCAI (1977).
- [6] JOUANNAUD J.P., GUIHO G. : "Inference of functions with an interactive system", Machine Intelligence 9, D. MICHIE ed., 1979.
- [7a] JOUANNAUD J.P., KODRATOFF Y. : "A methodology for two variables functions synthesis from examples", Pub. CRIN Univ. Nancy (1979) (On request to the author).
- [7b] JOUANNAUD J.P., KODRATOFF Y. : "Characterization of a class of functions synthesized from examples by a SUMMERS like method using a "BMW" mathing technique" Pub. CRIN Univ. Nancy (1979).
- [8] JOUANNAUD J.P., KODRATOFF Y. : "Quelques méthodes de synthèse automatique de programmes à partir d'exemples", Journées IRIA-SESORI Synthèse, Manipulation et Transformation de Programmes (1978) pp. 215-239.
- [9] KODRATOFF Y. : "A class of functions synthesized from a finite number of examples and a LISP program scheme". Pub. Institut de programmation (1979) and Int. J. of Comp. and Inf. Sci. Vol 8, n° 6, (1979)
- [10] KODRATOFF Y., FARGUES J. : "A sane algorithm for the synthesis of LISP functions from examples problems : The BOYER and MOORE algorithm", Proc. AISB meeting Hambourg (1978) pp. 169-175.
- [11] KNUTH D.E., BENDIX P.B. : "Word problems in universal algebras" in Computational problems in abstract algebra, J. LEECH ed. (1970), pp. 263-297.
- [12] CATHY J. Mc, ABRAHAMS P.W., EDWARDS D.J., HART T.P., LEVIN M.E. : LISP 1.5. programmer's manual (M.I.T. Press, 1962).
- [13] ROBINSON J.A. : "A machine oriented logic based on the Resolution Principle", J. ACM (1965), 12, 23-41.
- [14] SHAW D., WARTOUT W.S., GREEN C. : Inferring LISP programs from example problems". Proc 4th IJCAI (1975), pp. 268-273.
- [15] SMITH J.P. : "A class of synthesizable LISP programs", Report CS-1977-4, Dpt. of Computer Sciences, DUKE University, 1978.
- [16] SUMMERS P.D. : "A methodology for LISP program construction from examples", J. ACM (1977), 24, 161-175.
- [17] WEGBREIT B. : "Goal directed program transformations", IEEE SE-2 (1976), 69-80.