PAPE : AN ON-LINE SYSTEM FOR INFERING PROCEDURES FROM SETS OF THEIR TRACES

F-Y. VILLEMIN

Departement de Mathematiques et Informatique
CNAM, 292 rue Saint-Martin
75141 PARIS CEDEX 03 (FRANCE)

ABSTRACT

Many learning-by-example problems can be viewed as infering structured non recursive procedures from sets of their traces knowing the basic functions and predicates, used to write them. The on-line system PAPE does this as follows, an example (data ♦ trace) is given. PAPE first checks whether the already learned procedure gives the same trace with the given input data. If not, the trace is transformed into a procedure body (in $0(t^p)$ time, where t it is the length of the trace, and, p an integer depending upon the procedure and the basic functions and predicates), and then merges this body with the actual one to get the new procedure body.

0/ INTRODUCTION

Many problems of learning from examples (e.g.: drawing objects on a screen [v *2]* , complex actions of an industrial robot, commands of a computer system, queries of a database...) can be viewed as infering non-recursive procedures from sets of their traces knowing the functions and the predicates used to write them.

These applications suggest that we can restrict ourselves to non-recursive procedures written in PASCAL- like language in which :

- there is no composition of functions in an assignment,and, of a function and a predicate (of a loop, or, of a conditional).

- all constants are introduced by an assignment (x := constant).

- the sets of identifiers of variables, and, of formal parameters of the procedures are disjoint.

- all procedures to synthesize are supposed structured.

1/ THE SYSTEM PAPE

We describe here a program named PAPE [V 1], which, like BIERMANN and KRISHNASWAMY'S program AUTOPROGRAMMER [BiK] , is an on-line system learning a non-recursive procedure (a structured one for PAPE) from a set of this traces, while SIKLOSSY and SYKES' program SYN [sis] infers a recursive procedure from a single trace. But, unlike AUTOPROGRAMMER which requires from its programmer symbolic traces (sequences of assignments) and, the predicates for loops and conditionals at the places they are used, PAPE, like SYN, works automatically once the set of traces and the basic functions and predicates have been given.

An example E for PAPE is a 4-tuples E = (N, D, R, T) where N is the name of the procedure to synthesize, D the input data (actual parameters), R the output data, and, T the trace (a set of pairs of a variable and its value) free of errors.

All examples are supposed correct.

PAPE is mainly composed of two functions Al and A2.

It works as follows :

An example E is proposed.

-If E is the first one, PAPE creates a procedure-head with identifier N and Al (E) as a body.

-If not, PAPE starts with an execution of a call of N with D as actual parameter. If the trace obtained matches T, then PAPE stops (awaiting for some other example if any). Otherwise the partial body A1 (E) is compared and merged with the actual body of N by A2.

Al has a worst-case time complexity in $0(t^p)$ where t is the length of the trace T, and, p is an integer depending upon the procedure to synthesize, and, the sets of functions, and, of predicates.

In the following the behaviour of PAPE will be described on an example :

Suppose we wish to learn the procedure MULTIPLY which performs the multiplication of two positive integers using F - *(_* ♦ (addition), pred (predecessor function), /2 (integral division by 2), * 2 (multiplication by 2) J and P = {= 0, even J respectively as sets of functions, and, of predicates .

We have in mind a procedure of the form :

```
Procedure MULTIPLY (M1, M2/N) ;
loc Z ;
X := M1; Y := M2; Z := 0 ;
while   X ≠ 0 do
    if even X then X := X/2 ; Y := Y * 2
              else X := X-1 ; Z := Z + Y
    fi
od
N:=Z
corp
```

2/ THE SYNTHESIS FUNCTION A1

A1 is composed of three functions. Their behaviours will be shown on an example :

E_0 = (MULTIPLY,(18,4),(N=72),((X,18)(Y,4)(Z,0)(X,9)(Y,8)(X,8)(Z,8)(X,4)(Y,16)(X,2)(Y,32)(X,1)(Y,64)(X,0)(Z,72)(N,72))).

- First A11 generates a set of symbolic traces together with assertions (conjunctions of true predicates) : each pair (X_j,a_j) of T is replaced by the set of all assignments X_j:=Fm($X_1,\ldots,X_n$) when $F_m(b_1,\ldots,b_n) = a_j$,$b_i$ being the value of X_i before (X_j, a_j).

If we call M1 and M2 the formal input parameters, we get :

A1(E_0)=(X:=M1 {even(X)} ; Y:=M2 {even(X)∧even(Y)} Z:=0 {even(X)∧even(Y)∧Z=0} ; X:=X/2 {even(Y)∧Z=0} ; (Y:=Y+Y,Y:=Y*2,Y:=pred X) {even(Y)∧Z=0} ; (X:=X+Z,X:=pred X, X:=Y) {even(X)∧even(Y)∧Z=0} ; (Z:=Z+Y, Z:=Z+X, Z:=Y, Z:=X) {even(X)∧even(Y)∧even(Z)} ; (Y:=Y+Z, Y:=Y+Y, Y:=Z+Z, Y:=Y*2, Y:=Z*2) {even(X)∧even(Y)∧even(Z)} X:=X/2 {even(X)∧even(Y)∧even(Z)} ;(Y:=Y+Y, Y:=Y*2) {even(X)∧even(Y)∧even(Z)} ;(X:=X/2, X:=pred X) {even(Y)∧even(Z)} ;(Y:=Y+Y, Y:=Y*2) {even(X)∧even(Z)} ; (X:=X/2,X:=pred X) {X=0 ∧ even(Y)∧even(Z)} ; Z:=Z+Y {X=0∧even(Y)∧even(Z)} N:=Z {X=0∧even(Y)∧even(Z)})

This is a purely combinational process.

- Then A12 yields a single symbolic trace. This done by using :

a/ The "nice sequence" algorithm [V 1]. (A sequence is said "nice" if it has a partition in which each part has a common initial subpart and the remaining is either empty, or a nice sequence, or a sequence of patterns having at least one occurrence in some other part. This notion captures the syntactical properties of the symbolic traces of an iteration). The assertions being not modified we get :
(X:=M1; Y:=M2; Z:=0; X:=X/2; (Y:=Y+Y,Y:=Y*2); X:=pred X; Z:=Z+Y; X:=X/2;(Y:=Y+Y,y:=Y*2);X:=X/2; (Y:=Y+Y, Y:=Y*2);(X:=pred X, X:=X/2);(Y:=Y+Y,Y:=Y*2); X:=pred X; Z:=Z+Y; N:=Z)
This algorithm does not decide when sequences are equally "nice". Heuristics are used to get a single trace.

b/ The "regularity" heuristic [V 1] (patterns are choosen to give either the longuest, or, the more repeating subsequences) :X:=X/2 is prefered to X:=pred X since X:=X/2 is always followed by (Y:=Y+Y, Y:=Y*2), and, X:=pred X by Z:=Z+Y.

c/ The "insane" heuristic [Sik, V 1] (the most general schema is choosen, e.g. : the one having the more occurrences of variables): Y:=Y+Y is prefered to Y:=Y*2.

- Finally A13 introduces loops by first cutting A cutting A12(A1(E)) into the initial, the "nice" and the final sequences, and, retaining only assertion between subparts :
((X:=M1; Y:=M2; Z:=0) {even(X)∧even(Y)∧Z=0} ; ((X:=X/2; Y:=Y+Y) {even(Y)∧Z=0} ; (X:=pred X; Z:=Z+Y)) {even (X)∧even(Y)∧even(Z)} ; ((X:=X/2; Y:=Y+Y)) {even (X)∧even(Y)∧even(Z)} ; ((X:=X/2; Y:=Y+Y)) {even(X)∧even(Y)∧even(Z)} ; ((X:=X/2; Y:=Y+Y) {even(Y) ∧even(Z)} ; (X:=pred X ; Z:=Z+Y)) {X=0∧even(Y)∧even(Z)} ; (N:=Z) {X=0∧even(Y)∧even(Z)}) then, loops are introduced, by using the "pumping lemma" heuristic [Mic, V 1] (an iteration S_0 ; while do S od is infered from occurrences of subsequences of the form S_0;S,...,S) in two steps starting from the inside, the predicate being the largest common predicate included into all the assertion preceding the subparts and into the complement of the assertion following the last occurrence of this subpart :
A1(E_0) =(X:=M1 ; y:=M2; Z:=0){ even(X)∧even(Y)∧Z=0} ; while ¬X=0 do X:=X/2; Y:=Y+Y;while ¬even(X) do X :=pred X; Z:=Z+Y od od {X=0∧even(Y)∧even(Z)}; N:=Z)

3/ THE MERGING FUNCTION A2

A2 is composed of three functions, only A21 is fully worked out :

- First A21 looks to places where A1(E)differs from P, the actual body of N, and, makes a partition of both A1(E) and P between common and different parts, and, then infers a new body for N. There are three cases :

a/ They differ on a assertion, or, a predicate of a loop, or, of a conditional, A21 replaces it P by their disjunction.

b/ They differ on a sequence of assertions and instructions, starting by an assertion, A21 replaces this part in P by a conditional with predicate being the difference of the two assertions, and its assertion being the largest common one. Suppose the body P is : P=(X:=M1; Y:=M2; Z:=0; {even(Y)∧ Z=0} ; while ¬ X=0 do X:=pred X; Z:=Z+Y; while even(X) do X:=X/2; Y:=Y+Y od od {X=0 ∧even(Y)∧even(Z)} ; N:=Z)
This body P together with A1(E_0), from above,yields: A21(P,A1(E_0))=(X:=M1; Y:=M2; Z:=0 {even(Y)∧ Z=0} ; if even(X) then while ¬ X=0 do X:=X/2; Y:=Y+Y; while even(X)do X:=pred x; Z:=Z+Y od od else while X=0 do X:=pred X; Z:=Z+Y; while even(X) do X:=X/2 ; Y:=Y+Y od od fi {X=0 ∧even(Y)∧even(Z)} ; N:=Z)

c/ They differ on a sequence of instructions and assertions, not starting by an assertion : this is a filtering error due to A12. This is taken care by A22, a backtracking error recovery function (on all examples looked after, we haven't met such an error).

- Then A23 "normalizes" the body of N, by doing a symbolic execution and another program inference to get a "better looking" body.

4/ CONCLUSIONS

We can prove that the procedure M synthesized by PAPE (with A 23) from a set of examples $E_0, E_1, \ldots$ has the following properties :

- For any example E_i of the set, the execution of M with the input data D_i produces the trace T_i of E_i.

- Let N be the actual procedure to synthesize, consider any infinite enumeration of the examples of N, $E_0, E_1, \ldots$, then there is a finite k such that if M is synthesized from the set $E_0, E_1, \ldots, E_k$. Then for any i > k, M executed with D_i produces T_i of E_i.(But it is impossible to determine k).

So far only parts of PAPE have been implemented : A 11, A 12, A 13 and A 21. Many procedures in arithmetics have being partially synthesized. On the example given here, to get A 1(E_0) took 5.7. seconds on an IBM 370/168 in LISP 1.5. The full system described here is being implemented in V-LISP on a DEC VAX 11/780.

ACKNOWLEDGEMENT

We thank Ms A. ALMANDIN for the typing.

5/ REFERENCES

BiK BIERMANN A.W. and KRISHNASWAMY R. :

Constructing programs from example computations - IEEE Trans, on Software Engineering V SE-2 - N- 3 (1976).

Mic MICLET L,

inference de grammaires regulieres - These de Docteur-Ingenieur, ENST - PARIS, DEC 199

Sik SIKLOSSY L.

The synthesis of programs from their properties and the insane heuristic in Proc. 3rd Texas Conf. on Computing Systems - Austin, Texas, 1975.

SiS SIKLOSSY L. and SYKES D.A.

Automatic programming synthesis from examples problems in 4th IJCAI, Tbilissi, URSS, 1975.

V 1 VILLEMIN F-Y :

Le systeme PAPE - rapport CRIP-02-CNAM,1981

V 2 VILLEMIN F-Y :

Le systeme AGE - rapport CRIP-03-CNAM,1981

APPENDIX

The "nice sequence" algorithm, on A 11(E_0) above, behaves as follows :

- it first scans the sequence, ignoring the assertions, looking for the first set of assignment, i.e : (Y:=Y+Y, Y:=Y*2, Y:-pred X).

then checks whether a subset of it has repeating occurrences in the sequence, here (Y:=Y+Y, Y:=Y*2). This subset is setted apart. If it has more than one element, it checks whether there is a single repeating subsequence formed with either the preceeding, or the succeeding assignment, or both, and so on, until nothing repeats, or, comes up with this single subsequence.

- and then, does the same for the next set.

At the end, only setted apart subsets are retained, and, repeating subsequences are parenthesized.