AUTOMATIC PROGRAMMING FROM DATA TYPES DECOMPOSITION PATTERNS

Christian Gresse

LRI Bat. 490 Universite PARIS-SUD F 91405 ORSAY CEDEX FRANCE

ABSTRACT

A running system is presented which provides powerful support to the user while designing programs .
This system automatically constructs program schemas by decomposition of the datas involved in the specification. Datas are specified within a knowledge-base as data types having associated decomposition patterns.
The user may combine different data-decomposition strategies with any of its decomposition patterns. So it is possible to build several program schemas for the same problem.
Once a program schema is built the user can instantiate or adapt it by hand. He can also apply the system repetitively until he obtains primitive problems which can be solved directly.

Key-words : Running system. Data types. Decomposition patterns. Powerful support. Progam schema. Strategies of decomposition.

I INTRODUCTION

In the attempt to automatically construct a program from its specification two main approaches have been proposed.

- The first method acts in a formal framework and uses a deductive mecanism to solve the problem. The desired program can be obtained by proving a theorem in a constructive way [M.W 80], or by applying the Knuth-Bendix completion procedure to a rewrite system [DER 82] .

- The second method is based on a large, amount of knowledge that one can have about some aspects of programming. This knowledge base contains a lot of rules and facts which allow gradual refinement and transformation of a specification into an efficient program [BAR 79] [B.O.R 81].

However as previously noticed [BAR 83] , none of these approaches is really satisfying. The first one can't avoid the combinatorial explosion inherent to this kind of mecanism. The second one, lacking of relevant rules for algorithm creation can only synthesize programs from algorithmic specifications. A third method can be tried in partially automating the software development and providing a powerful aid to the programmer. Such an example is the Programmer's Apprentice [WAT 82] designed to be midway between an improved programming methodology and an automatic synthesis system.

The system presented in this paper, called the Program Builder (PB), belongs to this third method. It relies heavily on a top-down and modular approach and constructs automatically a program schema by decomposing the initial problem with respect to the data structures involved in its specification. This program schema introduces new sub-problems and eventually some guards which must be verified to validate these sub-problems.
Decomposition is done by processing structured data objects independantly of the semantics of the problem : so in some cases the program schema may be irrelevant. In the more favourable cases the user of the PB can instantiate or adapt the schema by hand. He can also run the PB to apply the same decomposition process to the sub-problems. The process ends once primitive problems which can be solved directly, appear.

II GENERAL STRUCTURE OF THE PB.

2.1 The data type knowledge-base.

The PB deals with problems which can be specified in the following way :
given a collection x,y,z,... of objects belonging to types X,Y,Z,... find one object t of type T such that t = f(x,y,z,...)
All types are stored in a data type knowledge-base. Each type is specified as an algebraic data type with an operation part and an axiom part.
e.g type string [integer]

Operations :
?EMPTY? _ : string[integer] → bool
FMPTY : → string[integer]
HEAD _ : string[integer] → integer
TAIL _ : string[integer] → string[integer]
CONS _ _ : integer string[integer] → string[integer]
With the above presentation and with the axioms of the type we can see that a string of integers can be empty or can be split into an integer (HEAD) and a string of integers (TAIL)
So we can write

```
CASE   ?EMPTY?(T) ; T=EMPTY
FCASE  ELSE       ; T=CONS(HEAD(T),TAIL(T))
```

Such a schema is called a decomposition pattern of the type string[integer]
Suppose we add to the type string[integer] new operations

```
SINGLE _ which tests if the string contains a single
         integer
UNIT   _ which builds a string from a single element
FIRST  _ and SECOND _ which split the string into
         two halves
CONC   _ which concatenates two strings
we obtain another decomposttion pattern
CASE    ?EMPTY?(T) ; T=EMPTY
        SINGLE(T)  ; T=UNIT(HEAD(T))
FCASE   ELSE       ; T=CONC(FIRST(T),SECOND(T))
More formally the presentation of a type includes
predicates P, constructors C and selectors S. Then
a decomposition pattern can be written :
CASE      P0(T)     ; T=C0
          P1(T)     ; T=C1(S1(T),S2(T),...)
          ..............................
FCASE     ELSE      ; T=Cn(Sα(T),Sß(T),...)
C0 is a constant and the following relations hold
vPi=true Pi∧Pj=false if i≠j
Each type of the data type base may have any number
of decomposition patterns
```

2.2 The strategies of decomposition

Two strategies are implemented by the PB.

a. Decomposition of an input data

```
If x has the following decomposition pattern
      CASE   P0(x)    ; x=C0
      FCASE  ELSE     ; x=C1(S1(x),S2(x))
(t:T)<-f(x:X,y:Y,...) can be rewritten into :

t=f(if P0(x) then C0 else C1(S1(x),S2(x),y,...))(i)
t=if P0(x) then f(C0,y,..)else f(C1(S1(x),S2(x),
y,..)) (ii) and finally
t=if P0(x) then g(C0) else h(S1(x),S2(x),y..) (iii)
where g and h are new problems

Now suppose that the selector S2 has the signature
S2 _ : X → X
if asked the PB introduces a recursive call by ap-
plying f to S2(x) and constructs the final program:
t=if P0(x) then g(C0) else k(S1(x),f(S2(x)),y..)
g and k are the new sub-problems to be solved.

Note : As S2 is a selector : S2(x) < x according to
a well-founded ordering on X, the program will thus
terminate.

The PB writes programs expressed in an algol-like
language. This language allows assignments, recur-
sion and conditionnals (if then else)
A program is composed of two parts :
- a specification part includes the header of the
  program and the declaration of all the variables
  used
- a body part
Exemple :
Suppose we want to construct the SORT program :
"Sort a string of integers"
      C:string[ integer]<-SORT(S:string[ integer])
Applying the first decomposition pattern of the
type string[ integer] to the input data S the PB
produces the program
Header :
      (C:string[ integer])<-SORT(S:string[ integer])
Variables :
      $N1 : integer
      $S1 : string[ integer]
Body : if ?EMPTY?(S) then C:=$Problem1(EMPTY)
       else   $N1:=HEAD(S);
              $S1:=TAIL(S);
              C:=$Problem2($N1,SORT($S1))
       fi
$N1 and $S1 are new variables created by the PB.
The problems prefixed by $ are the new problems to
be solved.
The program schema produced is an insertion sort
schema : $Problem1 can be solved directly (identity
problem), $Problem2 is the INSERT problem.

Suppose now we apply the second decomposition pa-
ttern, we obtain :
Header :
     (C:string[ integer])<-SORT(S:string[ integer])
Variables :
     $N1 : integer
     $S1,$S2 : string[ integer]
Body :
     if ?EMPTY?(S) then C=$Problem1(EMPTY)
     else if SINGLE(S) then $N1:=HEAD(S);
                            C=$Problem2($N1)
          else
                            $S1:=FIRST(S);
                            $S2:=SECOND(S)
                              C:=$Problem2(SORT
                                 ($S1),SORT($S2))
          fi
     fi
FIRST and SECOND have the same output type as SORT
so the PB introduces two recursive calls and pro-
duces a merge sort algorithm.

Note : This kind of decomposition is very similar
to those obtained by the transfer paradigm [BAR83]
or the divide and conquer paradigm also called the
decompose, solve and compose paradigm [SMI 82].
The decomposition operators are the selectors of
the input data type and composition is done by the
new problems introduced.
```

b. Decomposition of an output variable

```
In this case the composition operators are the cons-
tructors of the output data type. Guards are in-
troduced by the PB to determine which constructor
is to be used. The decomposition operators are the
new sub-problems to be solved.

Exemple :
Applying the first decomposition pattern of type
string [integer] to the output variable C of the
SORT problem the PB constructs :

Header :(C:string[integer])<-SORT(S:string[ integer])
Variables : $N1:integer
            $S1:string[ integer]
Body :      if $G1(S) then C:=EMPTY
            else $N1:=$Problem1(S);
                 $S1:=$Problem2(S,$N1);
                   C:=CONS($N1,SORT($S1))
            fi

A guard $G1 has to be computed and it is easy to
see that the solution of $Problem1 is equivalent to
searching the smallest element of S (problem MIN).
$Problem2 is the operation which deletes an integer
from a string
```

This program schema is a selection sort algorithm. Construction of $Problem1 whith the PB is straightforward when the same decomposition pattern is applied to the input variable S : In this case the new problem $Problem3 can be viewed as a primitive problem (finding the smaller of two integers). We have'nt room enough to describe the. whole program but the computation of $N1 would be :

$N1:=#Problem3(HEAD(S),MIN(TAIL(S))

2.3 Implementation of the PB.

It is implemented in Maclisp on a Milltics system. A programmer can use it interactivi1y via the display-oriented editor Emacs.
During the same session, while constructing a program he may combine different alternatives :
-Apply the first or the second strategy
-Ask for a recursive or a non recursive program
-Choose any of the possible decomposition patterns of a type
-Add a new decomposition pattern and select it
-Declare a new problem as an operation on an abstract data type
So it is possible to build quickly several different program schemas for a same problem.
Once a program schema has been built the user can instantiate it by renaming the entitles prefixed by *$*. He may also adapt it by modifying the code produced or by solving directly a problem.
The PB is a building block of a more general software development environment. Such an environment is being designed and built at Orsay University in the ASSPRO project [GRE 82], and PB is part of that project.

111 CONCLUSION

An overview has been presented of a system for the construction of programs from decomposition of their data. In its current implementation the PB is able to construct various programs processing structured objects as strings, trees, lists,... We are continuing to extend our work by trying to automatically compute the guards and generate the specifications of the new subproblems.

For this purpose we have to deal with the formal specification of the initial problem (e.g SORT can be specified as "Ordered(C) and Bag(C)= Bag(S)") and to enrich the axiom part of the presentation of the abstract data types stored in the knowledge-base.

ACKNOWLEDGMENTS :

I would like to thank Michel Bidoit and Gerard Guiho for several helpful discussions concerning this research.

REFERENCES

[1] [BAR 79]: Barstow D. "Knowledge-based Program Construction" Elsevier North Holland 1979.

[2] [BAR 83]: Barstow D. "The Roles of Knowledge and Deduction in Algorithm Creation" in "Automatic Program Construction Techniques" (A. Bierman, G. Guiho, Y. Kodratoff editors) Mac Millan 1983.

[3] [BIB 80]: Bibel W."Syntax-Directed, Semantics-Supported Program Synthesis" Art Intell 14,3 October, 1980, pp. 243-262.

[4] [B.O.R 81]: Bartels U., W. Olthoff, P. Raulefs "An Expert System for Implementing Abstract Sorting Algorithms on Parameterized Abstract Data Types" IJCA1-81. Vancouver, Canada.

[5] [DER 82]: Dershowitz N.; "Computing with Rewrite Systems" Internal Report, Urbana Champaign, 1982.

[6] [ORE 82]: Gresse C. "The ASSPRO Project" Internal Report, LRI Orsay, December 1982.

[7] [M.W 80]:Manna Z., R. Waldinger, "A Deductive Approach to Program Synthesis" ACM TOPLAS 2,1 1980, pp. 90-121.

[8] [SMI 82]: Smith I). "Top-Down Synthesis of Simple Divide and Conquer Algorithms" Internal Report. Naval Postgraduate School, Monterey November, 1982.

[9] [WAT 82]: Waters R. "The Programmer's Apprentice : Knowledge-Based Program Editing" IEEE Trans on Soft Eng Vol SE-8 NI, January, 1982.