

A SYSTEM WHICH SYNTHESIZES ARRAY-MANIPULATING PROGRAMS FROM SPECIFICATIONS

M. BIDOIT, C. GRESSE, G. GUIHO
Laboratoire de Recherche en Informatique
Bâtiment n° 490
Université de Paris-Sud
91405 ORSAY (France)

A system is presented for constructing array-manipulating programs from given specifications. The system accepts high level specifications and produces recursive programs in an Algol-like language.

Restriction to a specific domain and use of powerful techniques such as first-order matching or generalization make our system very efficient. It has been implemented in LISP on a small computer and has been tested for a large number of examples.

The general techniques we use, are briefly discussed and the synthesis of one program produced is presented.

Key-words : Array-manipulating programs - High-level specifications - First order matching - Generalization - Program synthesis.

1. INTRODUCTION

Although program synthesis is a recent development, there are already a great number of approaches (1-7).

Our approach is similar to that of Manna and Waldinger [5] but we restrict ourselves to the synthesis of array-manipulating programs. We think that this restriction will limit backtracking.

The principle of this kind of synthesis is : We provide the specifications of the problem to be solved in a high-level language (not compilable in a conventional way). The system works as a transformation program in order to obtain a final program written in a compilable language. For this purpose, the system uses heuristics and rewriting rules to eliminate non-compilable terms which occur in the specification language. The synthesis is then achieved by use of matching and generalization. This system has been implemented on a small computer (MITRA 125) in LISP and has been tested for a large number of examples.

2. LANGUAGES

2.1 The specification language

The language level should be high enough to represent the abstract types close to the concepts used by the programmer in his description of the problem. In our restriction to the

array type, the constructors and selectors are simple and few. Here are some constructions we use as specification language.

Note that for all the examples the system knows that T is an input array with bounds A and B and that I is a subscript. So input specifications like in the DEDALUS system of Manna and Waldinger are not necessary.

a. Find I between A and B such that $\mathcal{P}$ (I, input variables)

Examples :

- Find I between A and B such that $T(I)=Z$
- Find I between A and B such that $T(I) \geq \text{ALL } T[A..B]$

b. Test if $\mathcal{P}$ (input variables)

Examples :

- Test if $Z < \text{ALL } T[A..B]$
- Test if for all I between A and B-1 $T(I) \leq T(I+1)$

c. Construct $\text{TAB}[B1..B2]$ such that $\text{TAB} = \text{constructor}$ (I, input variables and $\mathcal{P}$ (I, input variables) from $T[A..B]$

Example :

- $\text{INSERT}(T, A, B, Z)$. We want to insert an integer Z in a sorted array
- construct $\text{TAB}[B1..B2]$ such that $\text{TAB} = \text{CONC}(T[A..I], Z, T[I+1..B])$ and $\text{ALL } T[A..I] \leq Z$ and $\text{ALL } T[I+1..B] > Z$ from $T[A..B]$

d. Compute the number of elements $T[I]$ such that $\mathcal{P}(I, \text{input variables})$

Example :

- the number of elements $T[I]$ such that $T[I] \geq Z$.

e. Two of the upper constructions can be combined to make a more complex one

Example :

- Find I between A and B such that (the number of elements $T[J]$ such that $T[J] \geq T[I] = K$.

Although we use few constructions, the possibility of combining them makes our program more flexible in the sense that we can use different specifications for solving the same problem.

Example : Finding the maximum element of an array can be specified both by :

- Find I between A and B such that

$T[I] \geq \text{All } T[A..B]$

- Find I between A and B such that (the number of $T[J]$ such that $T[J] > T[I] = 0$.

These different specifications lead to different programs

f. etc...

2.2 The output language

The output language is an Algol-like language and we extensively use recursion and basic operators relevant to array structure.

For the INSERT example, the output will be of the form :

```
INSERT (T,A,B,Z) = INSERTGEN (T,A,B,Z,A)
INSERTGEN (T,A,B,Z,C) = if C > B : TAB = CONC(T[A..B],Z)
ifnotif  $\neg(T(C) \leq Z)$  : TAB = CONC(T[A..C-1],Z,T[C..B])
else INSERTGEN (T,A,B,Z,C+1)
endif
```

3. REWRITING RULES AND HEURISTICS

There are two kinds of heuristics. Some of them are related to abstract types which are used and some others are related to general programming principles.

3.1 The first class of heuristics deals with the constructors and selectors of abstract types [12]. Informally, the heuristics use selectors to achieve a decomposition of the problem and then constructors to solve the original problem. In our implementation, the only type we use is the array type so this family of heuristics is restricted. It corresponds to the case formation rule (see Manna [5] or Wegbreit [8]) :

- If the array is empty, then no operation is necessary
- If not, we try to deal with one particular element of the array (generally the first one) and then look at the rest of the array.

3.2 The second class of heuristics deals with general knowledge on programming. The basic principles are similar to those used in Wegbreit [8].

- Formation of recursive calls.

During the synthesis, the first class of heuristics defines subgoals of the initial problem. If

by matching, we notice that a subgoal is an instance of a previous one or an instance of the initial goal, we infer a recursive call. It is then necessary to verify a well-founded set condition.

- Generalization : (Insane heuristic) (Wegbreit [8], Kodratoff [4]). In the previous process, when matching fails ; it is possible in most cases to generalize the goal used. With this generalization, matching becomes effective and a recursive call can be generated.

- Subsidiary programs : (Manna and Waldinger [5]) In the case formation rule, the inferred conditions can be non-elementary (i.e. it contains some specification language terms). Then a subproblem has to be introduced at this point.

4. TECHNICALS TOOLS

4.1 First order matching

This matching is mainly used for the detection of recursive calls. The algorithm implemented here is from Huet [9]. Note that recursive calls are always terminal.

4.2 Second order matching

A second order matching is necessary, in principle, in order to select the transformation rules to be applied. In fact, in our present implementation we don't need a 'sophisticated' algorithm and we use a simple key-word detection algorithm similar to pattern-directed invocation.

4.3 Theorem prover

To solve the subgoals induced by the case formation rule it is often necessary to use a theorem prover.

It is surprising that the necessary proofs are often very simple. So we have implemented a very "naive" theorem prover which has been sufficient for all our examples.

5. ONE EXAMPLE OF SYNTHESIS : INSERT

The input array $T[A..B]$ is sorted and we want to insert an integer Z in T such that the output array $TAB[B1..B2]$ remains sorted.

We provide the following input to our system

```
-INSERT (T,A,B,Z)
-CONSTRUCT TAB[B1..B2] such that
TAB = CONC (T[A..I],Z,T[I+1..B]) and All( $T[A..I] \leq Z$ 
and All  $T[I+1..B] > Z$  from  $T[A..B]$ )
-For All  $I$  between  $A$  and  $B-1$   $T(I) \leq T(I+1)$ 
```

* The first rule, when applied, transforms the problem into a first goal :

(1) (Find I between A and B such that
All $T[A..I] \leq Z$ and All $T[I+1..B] > Z$
then (construct $TAB[B1..B2]$ such that
 $TAB = \text{CONC}(T[A..I],Z,T[I+1..B])$)

* At first the system considers only the first part of the goal and transforms in into

(Find I between A and B such that

All $T[A..I] \leq Z$ then (verify
 All $T[I+1..B] > Z$) then (construct ...)
 The system now applies the decomposition rule to
 the first sub-goal and generates a segment of
 the desired final program and three new subgoals.
 if $A > B$ then (2)
 ifnotif $\neg(T(A) \leq Z)$ then (3)
 else (4)
 where (2), (3), (4) are the following subgoals.

- (2) $I+B$ then....unchanged....
- (3) $I+A-1$ then....unchanged....
- (4) (Find I between $A+1$ and B such that
 All $T[A+1..I] \leq Z$) then (Verify All $T[I+1..B] > Z$)
 then (Construct $TAB[B1..B2]$ such that
 $TAB = \text{CONC}(T[A..I], Z, T[I+1..B])$)

The system considers the goal (4) : Matching (4)
 with (1) fails because A can't be matched simul-
 taneously with $A+1$ (Find...) and A (Construct...)
 So the system introduces a new generalized prob-
 lem : INSERTGEN(T, A, B, Z, C) with :

INSERT(T, A, B, Z) $\Leftarrow$ INSERTGEN(T, A, B, Z, A)
 The input for INSERTGEN is :
 - (Find I between C and B such that All $T[C..B] \leq Z$)
 then (Verify All $T[I+1..B] > Z$) then (Construct
 $TAB[B1..B2]$ such that $TAB = \text{CONC}(T[A..I], Z, T[I+1..B])$)
 - For All I between A and $B-1$ $T(I) \leq T(I+1)$

* Applying the decomposition rule leads to
 if $C > B$ then (2) ifnotif $\neg(T(C) \leq Z)$ then (3) else (4)

- (2) $I+B$ then....unchanged....
- (3) $I+C-1$ then unchanged....
- (4) (Find I between $C+1$ and B such that...)

* Now matching succeeds because A is matched
 with A and C with $C+1$. So (4) is achieved by a
 recursive call to INSERTGEN($T, A, B, Z, C+1$)

* (2) and (3) are rewritten as
 (2') (Verify All $T[B+1..B] > Z$) then (Construct
 $TAB[B1..B2]$ such that $TAB = \text{CONC}(T[A..B], Z, T[B+1..B])$)
 (3') (Verify All $T[C..B] > Z$) then (Construct
 $TAB[B1..B2]$ such that $TAB = \text{CONC}(T[A..C-1], Z, T[C..B])$)
 Verifications are very easy since in
 (2') $T[B+1..B] = \emptyset$ and any property holds for every
 element of an empty array
 (3') the theorem prover uses the two conditions :
 $T(C) > Z$ (introduced by the case formation rule)
 $T[A..B]$ is sorted (input specification)
 * So (2') and (3') are rewritten as
 (2'') $TAB = \text{CONC}(T[A..B], Z)$
 (3'') $TAB = \text{CONC}(T[A..C-1], Z, T[C..B])$

INSERT(T, A, B, Z) $\Leftarrow$ INSERTGEN(T, A, B, Z, A)
 INSERTGEN(T, A, B, Z, C) $\Leftarrow$ if $C > B$: $TAB = \text{CONC}(T[A..B], Z)$
 ifnotif $\neg(T(C) \leq Z)$: $TAB = \text{CONC}(T[A..C-1], Z, T[C..B])$
 else INSERTGEN($T, A, B, Z, C+1$) endif

6. CONCLUSION

This system has been implemented on a small com-
 puter and works efficiently for all examples
 listed below (and some others). The maximum time
 necessary for one example is less than one minu-

te. There are three main reasons for efficiency :
 - the system has a thorough knowledge of the do-
 main and so backtracking is strongly reduced
 - we use a very powerful first order matching
 algorithm
 - the simplicity of the theorem-prover makes it
 very efficient.
 This system, in our opinion, proves that the
 Manna and Waldinger approach is correct and can
 be very powerful in specific domains. This shows
 that studies in the field of automatic program
 synthesis can be relevant for practical applica-
 tions.

Programs constructed :

Representative samples of the programs constructed by
 our system are the following :

- finding the maximum element of an array
- testing if an array is sorted
- testing if a number belongs to an array and finding
 its subscript
- testing if a number is less than every element of an
 array
- testing if an array contains duplicates
- testing if every element of an array is less than every
 element of another
- inserting an element in an array (INSERT)
- reversing an array
- finding an element equal to its subscript
- finding the k^{th} (in increasing order) element of an array
- computing the number of elements greater than a particular one
- testing if two arrays are identical
- computing the intersection of two arrays
- find I between A and B such that All $T[I..B] \geq Z$ and $T[I-1] < Z$
- find I between A and B such that All $T[A..I] \leq Z$ and $T[I+1] > Z$
 (where $T[A..B]$ and $T'[A..C]$ are input arrays).

REFERENCES

- [1] Biermann, Krishnaswamy "Constructing programs from example computation
 IEEE Trans. on Software Eng., vol. 2, n° 3, Sept. 1976.
- [2] Summers P.D. "A methodology for LISP program construction from examples".
 Journal of ACM 24-1, 1977.
- [3] Guinho G., Jouannaud J.P. "Inference of functions with an interactive
 system". Machine Intelligence 9, 1977.
- [4] Kodratoff Y. "Choix d'un programme LISP correspondant à un exemple".
 Congrès AFCET, Reconnaissance des Formes, 1978.
- [5] Manna Z., Waldinger R. "Synthesis : Dreams = Programs". Technical notes 156,
 SRI, 1977.
- [6] Bibel W., Furbach U., Schreiber J.F. "Strategies for the synthesis of
 algorithms". AISB/GI Conference on Artificial Intelligence, Hambourg, 1978.
- [7] Waldinger R. "Constructing programs automatically using theorem proving".
 Ph.D Thesis. Carnegie Mellon, 1969.
- [8] Wegbreit B. "Goal Directed program transformations". IEEE Trans. on Software
 Eng. Vol. 2, n° 2, 1976.
- [9] Huet G. "L'unification dans les langages 1,2,...w". Thèse de Doctorat d'Etat
 Université Paris VII, 1976.
- [10] Greussay P. "Contribution à la définition interprétative et à l'implémenta-
 tion des langages lambda". Thèse de Doctorat d'Etat, Univ. Paris VII, 1977.
- [11] Gresse C. et Guinho G. "Etude comparative des programmes de synthèse de
 programmes". Congrès AFCET Informatique, 1978.
- [12] Guttag J., Horowitz E., Mosser D.R. "The design of data types specification
 Research Report, Information Sciences Institute, Marina del Rey, California.