

A LEARNING SYSTEM ABLE TO SYNTHESIZE ARITHMETICAL FUNCTIONS

François SCHYN

Laboratoire de Recherche en
Informatique
Equipe de Recherche Associée au
CNRS "AL KHOWARIZMI"
Université de Paris Sud
91405 - ORSAY (France)

Gérard GUIHO

Laboratoire de Recherche en
Informatique
Equipe de Recherche Associée au
CNRS "AL KHOWARIZMI"
Université de Paris Sud
91405 - ORSAY (France)

This paper describes a system able to synthesize functions and predicates from examples taken in an arithmetical domain. Numbers are represented as characters strings. They are not represented as bags as in Lenat or Knapman's works. The representations of functions and predicates are sets of Horn clauses and the synthesis lies in the construction of new clauses or in the debugging of the existent ones. The paper describes the concepts which are used in the system and the synthesis process.

Keys Words : Learning, Horn clauses, Program synthesis

I. INTRODUCTION

Program synthesis from examples have been studied quite recently. They can be classified according to the following categories :

- Systems which construct function from non-ambiguous and self-contained examples. That means that all the information which is necessary for the synthesis is contained in the example and that there is no ambiguity in the construction of elementaries functions [see [10], [1], [4], [5]]

- Systems which are considered as the growing of a knowledge base. The examples do not need to be self-contained or non-ambiguous. That is the case with [17], [11], in some sense [8], and more recently [6]

Our system belongs to the second category. It is able to construct some recursive concepts (predicates or functions) as in [8] but uses a more significance and effective representation of numbers than [8] or [6] representation. The representation of numbers which is discovered by the system consists in strings of digits rather than bags as a PEANO representation. The concepts of arithmetical operators (Like successor) are then more effective but they are less obvious to infer.

II. TECHNICAL TOOLS OF THE SYSTEM

II.1 Functions and predicates representation

Functions and predicates are represented as sets of Horn clauses as in First Order Logic. In fact we use an equivalent representation which

more accurately represents a procedural form of clauses.

For instance, if Eq is the Equal predicate, the classical clause

$$\text{Eq}(z, A(x, y)) \leftarrow \text{Eq}(F(x), B(H(x), V(y))) \wedge P(H(x), y) \wedge \text{Eq}(z, V(y))$$

will be represented by

$$z \leftarrow A(x, y) \leftarrow a \leftarrow F(z), b \leftarrow H(x), z \leftarrow V(y), a \leftarrow B(b, z), P(b, y)$$

II.2 Examples

An expert can interact with the system in two different ways.

- Examples of inputs and of the corresponding output are given to the system as in

2258 $\leftarrow$ SUC(2257)

NUMBER (12329)

- An input is given to the system with an interrogation on the output

? $\leftarrow$ SUC(159)

The system tries to answer correctly. After having been told by the expert if it has succeeded, a positive example is generated. If the system gives a wrong output, a negative example is then generated and the expert gives the correct output. For instance let us suppose that the current answer to the question ? $\leftarrow$ SUC(159) is 1510. The expert indicates the correct answer -160- and then both a negative example $\{x=159, y \neq 1510\}$ and a positive example $\{x=159, y=160\}$ are generated.

As [17] and in opposition to [6], we think that negative examples (which are near-miss in our system) are fundamental for a complete function synthesis and, above all, for predicate synthesis.

II.3 Memory

There are in fact three memories.

- A long term memory which contains every clause of the system. They are referred to as names
- An active memory which is a short term memory. This memory will contain the names of functions or predicates which have been recently learnt or used.
- A predicate hierarchy which is a concept hierarchy as in [6]. For each new predicate, we try to find a correct position in the hierarchy and in fact we find more than one possibility. The candidate positions are organized in subhierarchies which are also attached to the name of the predicate.

II.4 Evaluator

Our evaluator is a Prolog like evaluator which includes a sophisticated tool used when the prolog evaluator fails. For instance, during the learning of the successor function, when this learning is incomplete, a Prolog evaluator will answer "False" with the input 2750+SUC(2749) (carry on successor of 9 has not been encountered yet).

A simple answer "False" is not sufficient for us, since we specially want to know the point where the failure is the most relevant. Such a failure point is unique in a Prolog evaluator but it is not so if we change the order of clauses or terms. We need an evaluator which gives the "best" failure point on all these orders (or, one point which is reasonably close to the best one). The evaluator we use in the system is described in [12].

II.5 Locking

A clause is locked if paths exist from the inputs (here x and y) to the output (here z) which allow to calculate the output. (see [4]). Often during the learning process, the generated clauses are not locked and a significant part of the system deals with locking of clauses. Locking of predicates is more frequently used to place the predicate in the hierarchy than to actually construct the predicate.

II.6 Learning heuristics

This paragraph describes the main heuristics which are used in the learning process. Some of them are expanding heuristics, some others deal with reduction or simplification.

II.6.1 Catching

When the evaluator fails on an example, it gives a clause which represents the most relevant failure. Catching is then a partial copy of that clause. Predicates or functions which are compatible with the example are copied. Predicates

which are false in the example become negative in the catching clause. Multiple values are separated, when a value is out the input domain of a function, a new function is created. This last action could be considered as an unavoidable combinatorial explosion but subsumption will reduce it.

II.6.2 Remarks

The "remark" heuristic is used with three elements a clause, an example on this clause and the active memory. It is a combinatorial testing of all concepts which are in the active memory, these concepts being tested on all the values induced by the example.

II.6.3 Unfolding

Our unfolding is the classical one [3]. To unfold a function F (or a predicate) which appears on the right part of a clause is to add to this clause the right part of a clause in which F constitutes the left part. Unfolding is controlled by an example and then the right part of F which is added must be compatible with the example.

II.6.4 Locking by constant

When the three preceding heuristics (catching, remarks, unfolding) have failed to lock the clause. A localization process delivers a set of variables which can be replaced by constants.

II.6.5 Simplification

Near the end of the learning process, many terms become useless, particularly in a recently created clause. The objective of the simplification process is to detect these terms and to get rid of them.

II.6.6 Subsumption [9]

A clause C is said to subsume a clause D if all the positive examples of D are accepted by C. D can be cancelled and all its positive examples become positive examples of C.

II.7 Learning process

As the heuristics can be greatly changed, many different learning processes are possible. The objective of the newly designed system is to study experimentally these different kinds of heuristics or learning processes. For this reason we are presently designing a new version of the system which will allow a more flexible change of heuristics or the learning process. The one which is now described here is very close to the first version of the system. [13].

After the expert introduces an example, the system works in this manner :

- for a function

1. EVALUATION
2. If the example is compatible, it is added as a positive example to the clause which accepted it and the process stops
3. If not the CATCHING heuristic is used
4. If the new clause is not LOCKED try REMARKS
5. If the clause is locked now do SUBSUMPTION, if possible, then SIMPLIFICATION and stop.
6. Try UNFOLDING
7. If the clause is not LOCKED try REMARKS again
8. If the clause is still unlocked, LOCK it by CONSTANT function
9. Try SUBSUMPTION, then do SIMPLIFICATION and stop

- for a predicate

1. EVALUATION
2. If the example is compatible add it to the clause. If there is no subhierarchy, then stop
3. If not, do CATCHING
4. Then do REMARKS
5. If the clause is LOCKED, do an EVALUATION on negative examples. If there is no problem try SUBSUMPTION, SIMPLIFICATION and STOP, else refine the subhierarchy
6. Try UNFOLDING
7. If the clause is not locked, LOCK it by CONSTANT function, try SUBSUMPTION, SIMPLIFICATION and stop
8. Else do an EVALUATION on negative examples, refine the SUBHIEARCHY, try SUBSUMPTION, SIMPLIFICATION and stop

III. Some of the experiments :

The initial knowledge of the system is composed as character strings and some basic functions or predicates.

LAST which gives the last character of a string
 PREF which gives the whole string except the last character

CONC which gives the concatenation of two strings
 NUL which is true is a string is empty

K which is true if the string is a one character string

ST which is always true on strings

Once NB (number) and D (digit) predicates have been learned : the following inputs :

1+SUC(0), 2+SUC(1)...9+SUC(8)
 123+SUC(124), 360+SUC(359), 100+SUC(99)
 will give as output :

```
y+SUC(x) < x+0, y+1, D(x), D(y)
          < x+8, y 9, D(x), D(y)
          < a+PREF(x), b+LAST(x), c+SUC(b)
          y+CONC(a,c), x+CONC(a,b), a+PREF(y)
          c+LAST(y), NB(a), D(b), D(c), D(x), D(y)
          [x= 123, y=124]
```

```
< a+PREF(x), b+LAST(x), c+F(b), d+SUC(a),
  y+CONC(d,c), x+CONC(a,b), d+PREF(y),
  c+LAST(y), NB(a), NB(d), D(b), D(c), D(x),
  D(y), [x=99, y=100] [x=359, y=360]
  < a+PREF(x), d+G(a), c+F(x), y+CONC(d,c)
  d+PREF(y), c+LAST(y), NB(d), D(c), D(x), D(y)
  [x=9, y=10]
```

```
y+F(x) < x+9, y+0, D(x), D(y) [x=9, y=0]
y+G(x) < x+0, y+1, NUL(x), D(y) [x=0, y=1]
```

A complete description of the example can be found in [14].

IV. Conclusion

The system which has been described here presents some particularities

- The learning capacity is greatly related to the expert pedagogy. One of our further interests is to study how the modifications or introductions of heuristics can more or less allow a sophisticated pedagogy.

- The concepts which have been synthesized here are relatively complex. In Lenat's work, the concept of addition is simple because addition is a particularization of union on bags. Our representation of numbers is more realistic. The synthesis of the successor function is constructive and is not a restriction of a previously known concept. However the system is able to infer simple concepts and can be used as a powerful learning-knowledge-base.

References

1. BIERMAN & SMITH : "A production rule mechanism for generating LIPS CODE", Duke University, 1977
2. COLMERAUER & AL : "Un système de communication homme-machine en français", Rapport préliminaire, Groupe de Recherche en I.A. Aix Marseille Luminy, 1972
3. DARLINGTON : "A system which automatically improves programs" IJCAI 3th, 1973
4. GUIHO & JOUANAUD : "Inference of function with an interactive system", Machine Intelligence n° 9, 1977.
5. KODRATOFF : "Choix d'un programme LISP correspondant à un exemple" AFCET, reconnaissance des formes, 1977.
6. KNAPMANN : "Artificial learning", Ph.D. Thesis Edinburgh, 1977.
7. KNOBE & KNOBE : "A method for inferring context free grammars", Information and control 31, pp 123-146 (1976).
8. LENAT : "AM : an artificial intelligence approach to discovery in mathematics or heuristics search", SAIL AIM-286, AI Lab., Stanford University, 1976.
9. NILSON : "Problem solving methods in artificial intelligence", Mc Graw Hill, 1971.
10. SUMMERS : "A methodology for LISP program construction from example", Journal of ACM, 24-1, 1977.
11. SUSSMAN : "A computational model for skill acquisition", Ph.D. MIT, Massachusetts University.
12. MELESE & TREUIL : "Failure analysis in sequence of linear resolutions", Memo LRI n° 30
13. TREUIL & SCHYN : "The system LQAS", LRI, Orsay France, 1978.
14. SCHYN & GUIHO : "A learning system able to synthesize arithmetical functions", Memo LRI n° 29.
15. VERE : "Induction of concepts in the predicates calculus", Proc 7th IJCAI Tbilissi USSR 1975 pp 281-287.
16. WEGBREIT : "Goal directed program transformation", IEEE trans. on software eng. Vol 2 n° 2, 1976.
17. WINSTON : "Learning structural description from example", Ph. D. Mc Graw Hill, N. Y. 1975.