Programming Repetitive Tasks in Prolog by Demonstration

Pierre Robineau
Laboratoire d'Informatique d'Avignon
BP 1228
84911 Avignon cedex 9
pierre.robineau@univ-avignon.fr

Abstract

Most of the recent work in the domain of Programming by Demonstration do not tackle the problem of acquisition of complex program structures like loops. We propose a new algorithm to automate repetitive tasks and to memorize them as Prolog programs. An example of a task is a Prolog goals list. Our method searches for traces of recursive predicates definitions in examples. The Minimum Description Length Principle is the criterion that guide the choice of the algorithm. The construction of a Prolog program automating a task is based on the chosen traces and on constraints due to Prolog syntax and Prolog resolution. Our experimentations with PPD are encouraging.

1 Introduction

Most of the recent work in the domain of Programming by Demonstration (PbD, [Cypher, 1991], [Bocionek and Sassin, 1993], [Lieberman, 1993], [WPbD, 1995]) do not tackle the problem of acquisition of complex program structures like loops. However, they are essential to the automation of many tasks. We propose a new algorithm to automate repetitive tasks and to memorize them as Prolog programs. A so-called repetitive task is a task that is performed by repeting several times the same operations (for example, in a graphical editor, several duplications with rotation of an objet).

PbD is integrated in an application software like word processor or graphical editor. It involves three components :

- The acquisition component that is part of the interface. By interaction with the user, in addition to the example of a task, it acquires various informations that constraint the induction of the program that automate this task. An example is a sequence of commands invoked by the user.
- The algorithm that induces the program automating a task from demonstrations of this task.
- The explanation component that offers a feedback to the user. It is also part of the interface. It gives hints about the interpretation of the example by the system. It also supplies (metaphorical or textual or graphical) descriptions of the induced program.

We are mainly concerned here with the algorithm that induces a Prolog program automating a repetitive task. Nevertheless, we have developed prototypes of an acquisition component and of an explanation component. Both components are briefly described (section 2) in order to indicate how, in our context, limiting errors in the acquired examples and control interactively the induction mechanism. Then we present the algorithm (section 3). Finally we give the first results of our method in section 4.

2 User interface

2.1 Principles

The user interface prototype has two main parts:

- A tool bar grouping the icons of all provided commands.
- A working space (WS) represented by a window containing each object introduced by the user or calculated by a command (each object is distinguished from the others).

In our system, the commands provided to the user are operations in the domain of list, number and symbol processing. On the same principles, we could provide commands applicable to textual or graphical objects.

The user applies commands to objects. A command has as inputs 0 to i objects entered by the user, and produces 0 to j objects. It can succeed or fail. In the latter case, it does not produce output. Each object should be in WS to be used as input of a command. Each output of a tool is an object that is put in WS.

2.2 Acquisition component

The purpose of the developed prototype of an acquisition component is to demonstrate how acquire an example of a repetitive task and the informations sufficient to induce the program automating this task.

When the user goes into PbD mode a new work space WS' is created. The objects used in the example should be entered in WS'. The actions the user performs are memorized until he leaves this mode. During the acquisition of the example of a task, for each new object entered by the user in WS', the system asks him if it is a parameter of the task or if this value is always the same. The answer allows the distinction between the inputs and the constants of the program to be induced. The user can only apply a command

to the objects present in WS'. By this means, the system implicitly obtains the links between the arguments of the commands of the example. When the user leaves the PbD mode, he should indicate the objects to be kept after the task is performed. From this information, the outputs of the program can be deduced.

When the user has left the PbD mode, WS' is closed. At this point, the acquisition component has recorded various informations: the sequence of invoked commands, the links between their arguments, the inputs and outputs of the task, the failure or success of each invoked command. These informations enable the induction of a Prolog program. Once the program is induced, the new tool defined by this program will be integrated in the tool bar and will be allowed to be used in examples of other tasks.

2.3 Explanation component

At any moment the user can control the example under acquisition by editing in a window the recorded commands. In this window, the sequence of operations is detailed in natural language. For each command, the input and output objects, the success or failure are given. The user can correct the example by cancelling the last invoked commands and performing others, for example if the system has misinterpreted a performed command.

Furthermore, once the task is learned, the user can control in another window the induced program. A description in natural language of the task is given with the links between the arguments of the commands composing it, the type of the input and output objects.

3 Inductive construction of Prolog program

3.1 General problem

We recall the definition of a functional logic program (FLP, adapted from ([Bergadano and Gunetti, 1993])). The definition of a predicate P is a FLP iff ($n \geq 0$, n arity of P):

- m ($0 \leq m \leq n$) arguments of P are the inputs of P,
- the n-m other arguments are the outputs of P,
- to a configuration of the inputs of P (i. e. exactly one value is associated to each input of P) corresponds at most one configuration of its outputs (none if P fails for this configuration of its inputs).

Each command provided to the user is a predicate defined by a FLP. We want to create new predicates defined by FLP. We restrict ourselves to the Functional Prolog Programs because our induction algorithm uses the properties of Prolog resolution (i. e. it uses the ordering of the clauses in the program and the ordering of the literals in the clauses).

Thus an example is a Prolog goals list. Inputs of each goal are constants, outputs are constants only if the goal is successfully solved.

We present an example that will allow us to illustrate each step of the algorithm.We aim at programming by demonstration the predicate `reverse` that takes as input a list and returns its reverse as output. We need three goals (figure 1 (a)): `null` that succeeds if its input is the null list; `first_element` that takes a not null list as input and returns the head and the tail of the list; `cons` that takes as input an atom and a list and returns their concatenation. Let the list in figure 1 (b) be the example we give us in order to learn `reverse`.

```
(a)        null([]).
           first_element([X|Y],X,Y).
           cons(X,Y,[X|Y]).

(b)        [null([1,2,3]),
      first_element([1,2,3],1,[2,3]),
      cons(1,[],[1]), null([2,3]),
      first_element([2,3],2,[3]),
      cons(2,[1],[2,1]), null([3]),
      first_element([3],3,[]),
      cons(3,[2,1],[3,2,1]), null([])]
```

Figure 1. An example of `reverse` predicate

We call trace of a predicate P defined by a Functional Prolog Program, the list of the leaves of the proof tree of P (for a configuration of its inputs) ordered as the Prolog resolution mechanism evaluates them.

We can now depict the problem to be solved (figure 2). Knowledge are acquired by the acquisition component. Meta-knowledge are the description of some properties of the commands given to the system by its creator.

Given :

- An example $\mathcal{E}$,
- Knowledge about $\mathcal{E}$,
- Definition of each predicate occuring in $\mathcal{E}$,
- Meta-knowledge about these predicates.

Find :

A predicate P defined by a Functional Prolog Program that, for the same inputs than $\mathcal{E}$, has $\mathcal{E}$ as trace, and that, for other inputs has a trace *similar to* $\mathcal{E}$.

Figure 2. General problem

3.2 Used restrictions

A number of programs verify the general case. In our context, several hypothesis can be made to constraint the search:

- Each example is an example of a repetitive task.
- The examples are error free due to their acquisition through the acquisition component.
- The user should firstly submit simple tasks to the system, and then use the new commands in the examples of more complex tasks.

These restrictions define a strong bias: we consider that a repetitive task can be represented as a loop programmed in Prolog and that only one example is necessary to induce this program. Hence we search for traces of loops in the example. For each trace of a loop, the definition of a predicate is built and the trace replaced in the example by a call to the new predicate. Then the program automating the task is easily completed from the modified example by adding a head to construct a Prolog rule. The definition D of a predicate Q encoding a loop in Prolog has the overall shape indicated in figure 3 (X_i, X_j, X_k are lists of

arguments; ①, ②, ③ are lists of predicates). These loops can be of two types : "While the test fails (respectively succeeds) do". D is composed of two rules D_1 et D_2 that are simulteanously induced. The recursive call is in D_1. Predicates are evaluated before the recursive call (①), others occur after(②). D_2 is the base case. Operations can be done in the base case (③). Depending on the patterns detected in the example, ② and ③ can be missing in a definition.

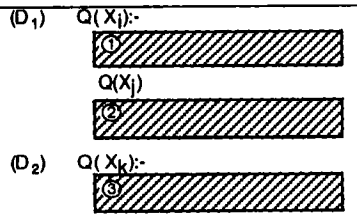

Figure 3. Overall shape of the induced definitions of the recursives predicates

3.3 Inputs of the algorithm

The inputs of the algorithm are the informations extracted by the acquisition component and the meta-knowledge. The meta-knowledge about a command are introduced once for all by the programmer of the software program. There are two kinds of meta-knowledge:

- A command is categorized as a test if it is programmed in such a way that it can fail for some configurations of its inputs, else it is an operation.
- For each argument of each command, its mode, input or output, should be specified.

The meta-knowledge described in figure 4 in Prolog means that `null` is a test and its argument is an input. `first_element` is an operation and its arguments are one input and two outputs. `cons` is an operation and its arguments are two inputs and one output.

```
meta(null,test,[in]).
meta(first_element,ope,[in,out,out]).
meta(cons,ope,[in,in,out]).
```

Figure 4. Meta-knowledge

After acquisition of the example of `reverse`, the following informations have been gathered by the acquisition component:

- the example $\mathcal{E}$ (figure 1 (b)).
- `Res`, a function that memorizes the success or the failure of each goal of $\mathcal{E}$. It returns `success` for each goal except for `null([1,2,3])`, `null([2,3])`, `null([3])` that have failed.
- `Cat`, a function that gives the category of each argument of each goal of $\mathcal{E}$. For outputs, we distinguish the outputs of the task (O) and the intermediate results (X). For inputs, we distinguish constants (C), the inputs of the task (I), and the intermediate inputs (XI). The membership of different categories is deduced from the interactions with the user. The different values of `Cat` for the arguments of the goals of $\mathcal{E}$ respectively are: `I,I,X,X,XI,C,X,XI,XI,X,X,XI,XI,X,XI,XI,X,X,XI, XI,O,XI`
- `Val`, a function that, for each argument of $\mathcal{E}$, gives the associated object (if it exists, otherwise gives `none`).
- `Var`, a function that replaces in $\mathcal{E}$ the object associated to each argument by a variable with regards to links between its arguments deduced from interactions with the user. $\mathcal{E}$ is said variabilized:
  `Var(`$\mathcal{E}$`)=[null(L1), first_element(L1,A1,L2), cons(A1,N,M1), null(L2), first_element(L2,A2,L3), cons(A2,M1,M2), null(L3), first_element(L3,A3,L4), cons(A3,M2,M3), null(L4)]`

3.4 Sketch of the algorithm

Search for traces of loops

Every sub-lists having the structure of the trace of a loop programmed in Prolog are searched in the example. We call repetition several consecutive occurrences of a succession of predicates with the same links between their arguments. Every repetitions are searched in the example. But some of them are not traces of loops. A loop programmed in Prolog as depicted in figure 3 includes a base case. The corresponding test may be at the beginning or at the end of each repetition. The latter case is not implemented in our system, we consequently will not present it. A loop is programmed in Prolog by a recursive definition. If this definition is not in the form of a tail recursivity, then, after the execution of the base case, goals located after the recursive call will be executed, creating an other repetition: two repetitions linked by their arguments are observed. The second does not require a test to terminate.

Three repetitions $\mathcal{R}1$, $\mathcal{R}2$, $\mathcal{R}3$ (figure 5) are found in $\mathcal{E}$. None of these repetitions is linked with another. Only $\mathcal{R}2$ and $\mathcal{R}3$ have a stopping test since `Res` gives for the test immediately following $\mathcal{R}1$ the same value than for the test in $\mathcal{R}1$. $\mathcal{R}2$ and $\mathcal{R}3$ and their respective stopping test are therefore potential traces of loops denoted respectively $\rho\mathcal{L}1$ and $\rho\mathcal{L}2$.

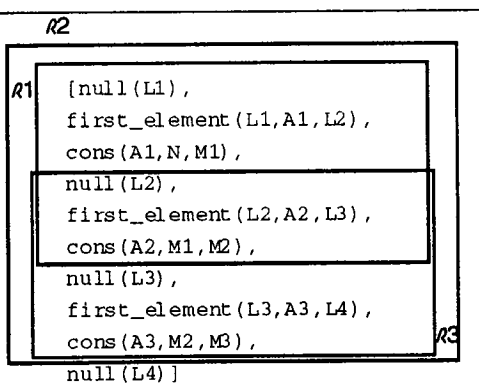

Figure 5. The different repetitions found in the example of `reverse`

Choice of loops to be constructed in Prolog

The simple repetitions and grouped repetitions having the intended structure of the trace of a loop are called potential loops. It remains to choose the best ones. The Minimum Description Length Principle is the criterion that guide the choice of the algorithm.

The information compression allowed by a potential loop is defined as the difference between the number of bits necessary to encode the program automating the task without taking into account this loop and the number of bits necessary to encode the program and the loop (we have used the encoding proposed by Muggleton ([Muggleton, 1988])). In the general case, a number of potential loops can be found. We use a branch-and-bound algorithm that searches a subset of the found potential loops that maximizes the information compression and that not contains two overlapping potential loops. In our example, $\rho\mathcal{L}1$ is chosen because it allows a reduction of about 18 bits while $\rho\mathcal{L}2$ causes an increase of 9 bits.

This approach does not take into account the possible existence of loops themselves included in the body or in the base case of another loop and that would have be chosen at the same time that this latter. This restriction is a consequence of our assumption that the user automates simple tasks and, then, calls them in demonstrations of more complex ones.

Program construction

A set of potential loops has been chosen. Each potential loop PL is the trace of a loop L. D, its definition in Prolog, has to be built.

PL consists of (at most) three parts: two repetitions R1 and R2, and G, the list of goals between them. The structure of D has been described in figure 3. The body of the rule D1 is the pattern of (i. e. the list of goals repeted several times in) R1, followed by a recursive call, followed by the pattern of R2. The body of D2 is the list of goals G.

The arguments of the different literals in D1 and D2 have to be determined.

One can see a potential loop from the point of view of data flows between its different parts (figure 6). I symbolizes inputs of the loop and O its outputs. For the three parts of the potential loop, R1, R2 and G, an incoming arrow represents part of its inputs, an outcoming arrow part of its outputs. A dotted arrow into a repetition elicites transmission of value from an occurrence of the pattern to another: outputs of an occurrence are then inputs of the following.

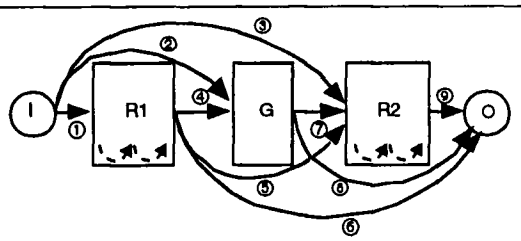

Figure 6. Data flows between the different parts of a loop

The trace of L should be PL for the same inputs than PL. The goals of D1 located before the recursive call, for the same inputs than PL, produce R1, those located after produce R2, and the goals of the body of D2 produce G. The links observed between the arguments of the goals of the trace of L for the same inputs than PL should be the same than those of PL. This constrainst and the Prolog syntax impose us the links between the literals of D.

Finally the definition of the program automating the new command (including calls to the recursive predicates) is built. A program is induced even if no potential loop is found.

In our variabilized example, `reverse` has two arguments: an input (`L1`) and an output (`M3`). Let call `loop` the predicate associated to $\rho\mathcal{L}1$. Each occurrence of the pattern of $\rho\mathcal{L}1$ has two inputs calculated at the previous occurrence (the argument of `null` and the second argument of `cons`). Furthermore, it returns an output (the last argument of `cons`). To return this output, a new input should be added to `loop`. But it is redundant with the second input. Thus only three arguments are necessary. The values of arguments at the call of `loop` by `reverse` must also be fixed. It is sufficient to take values of the first occurrence of the pattern: one is an input of `reverse`, the second is a constant: `[]`. The resulting program is shown in figure 7.

```
reverse(X,Y):-
    loop(X,[],Y).

loop(L,M,R):-
    not(null(L)),
    first_element(L,A,L1),
    cons(A,M,M1),
    loop(L1,M1,R).
loop(L,M,M):-
    null(L).

meta(reverse,ope,[in,out]).
```

Figure 7. The resulting Prolog definition of `reverse` found by our method

Finally the meta-knowledge associated to the new predicate has to be defined. Then the new predicate will be available as the tools provided by the system. We have already found the mode of its arguments. It remains to specify if it is a test or an operation. A new predicate will be classified as a test if tests other than the stopping test of the loop occur in its Prolog definition since it will fail when these tests will fail. In $\mathcal{E}$ the only test is `null`, the resulting meta-knowledge is shown in figure 7.

4 Discussion

This method has been implemented in PPD (Prolog Programming by Demonstration) system ([Robineau, 1995], [Robineau, 1996]). We have experimented the algorithm on various tasks. Each time, the trace of the induced Prolog program is the submitted example and the induced program is the same that the program we have programmed by hand.

member (7,5 s)	[first_element([3,2,1],3,[2,1]), dif(3,1), first_element([2,1],2,[1]), dif(2,1), first_element([1],1,[]), dif(1,1)]	member(X,Y):- first_element(X,T,Q),memb(T,Q,Y). memb(T1,Q1,Y):- dif(T1,Y), first_element(Q1,T2,Q2), memb(T2,Q2,Y). memb(T,Q,Y):-not(dif(T,Y)).
sublist (8,6 s)	[null([1,2]), first_element([1,2],1,[2]), member([3,2,1],1), null([2]), first_element([2],2,[]), member([3,2,1],2), null([])]	sublist(X,Y):-subl(X,Y). subl(X,Y):- not(null(X)), first_element(X,T,Q), member(Y,T), subl(Q,Y). subl(X,Y):-null(X).
factorial (9 s)	[dif(3,1),minus(3,1,2),dif(2,1), minus(2,1,1),dif(1,1),mul(1,2,2), mul(2,3,6)]	factorial(X,Y):-fact(X,Y). fact(X,Y):- dif(X,1),minus(X,1,Z), fact(Z,W),mul(W,X,Y). fact(X,X):-not(dif(X,1)).

Table 8. Experiments with PPD

In our experiments, the tasks are in the domain of list and integer processing: `reverse` that reverse a list, `member` that tests if an element is member of a list, `sublist` that verifies that a list is included in an other one, `factorial` that calculates the factorial of an integer, etc.

Results are shown in table 8. The first column contains the name of the learned tools, the second the submitted example, and the third the induced program. The computation time on a HP 9000/730 workstation is given in brackets. PPD is programmed in an interpreted Prolog.

PPD has been evaluated with users in an informal way. This work has been done with users having programming skill. They have been able to build the commands they have to do in the domain of list processing. But it is clear that our prototype of interface needs to be improved.

Only few hypothesis on provided tools have been included in PPD. Therefore the method is quite general and independant of a particular application.

Our experimentations with PPD are encouraging. We envisioning the refinement of the acquisition and explanation components (more graphical and visual) and the extension of the algorithm to a larger class of programs.

References

[Bergadano and Gunetti, 1993] F. Bergadano and D. Gunetti, "An interactive system to learn functional logic programs", *Proceedings IJCAI'93*, Chambéry, France, 1993.

[Bocionek and Sassin, 1993] S. Bocionek and M. Sassin, "Dialog-based learning (DBL) for adaptative interface agents and Programming-by-Demonstration systems", *Technical report CMU-CS-93-175*, Carnegie Mellon University, 1993.

[Cypher, 1991] A. Cypher, "EAGER : programming repetitive tasks by example", *proceedings ACM CHI'91*, 1991.

[Lieberman, 1993] H. Lieberman, "Mondrian: a teachable graphical editor", in: A. Cypher (éd.), *Watch what I do: Programming by Demonstration*, MIT Press, 1993, 340-358.

[Muggleton, 1988] S. H. Muggleton, "A strategy for constructing new predicates in first order logic", *Proceedings EWSL'88*, 1988.

[Robineau, 1995] P. Robineau, *Vers un système d'apprentissage symbolique flexible et compréhensible pour une aide à la découverte de connaissances*, Thèse de doctorat, Université d'Avignon et des Pays de Vaucluse, France, 1995.

[Robineau, 1996] P. Robineau, "Programmation par démonstration de tâches répétitives : une méthode par induction de programmes Prolog", *Proceedings RFIA'96*, Rennes, France, 1996.

[WPbD, 1995] Proceedings of the Workshop on Programming by Demonstration, *ICML'95*, 1995.