A Program To Deduce Recursive Rules Involving Lists

Debsankha Manik

August 24, 2010

Abstract

Recursive algorithms are very frequently used in prolog, and for good reasons too - they generally make for succinct and readable code. In fact, most of the built-in predicates bundled with swi-prolog dealing with list manipulation are defined recursively. Here we present an example-based model about teaching computers to devise recursive prolog rules involving lists.

1 The Goal

This project is a part of our more ambitious endeavour of achieving artificial intelligence (i.e. strong A.I. in technical parlance) by making computers follow the steps human beings take in order to understand[1] various concepts. For that to happen, one needs to first teach all the tools needed to program in a language to the program itself. Since recursion is such an indispensable tool for programming in prolog, we decided to write a program which would be able to deduce a recursive definition for a predicate involving lists if it is supplied with a set of examples. The set of examples should satisfy certain criteria, which we would discuss later.

1.1 Recursive rules

A recursive definition of a prolog predicate consists of at least two rules[2]. There must be one rule which calls the original predicate, which carries out the actual job of recursion. There should also be at least one exception handling rule preceeding the recursive rule, to ensure that the recursion actually ends at some suitable point. For example, a simple recursive definition of a list predicate is:

```
append(X,[],X).
append(A,B,C):-first(FA,A),first(FB,B),tail(TA,A),tail(TB,B),=(FA,FB),↩
    append(TA,TB,C).
```

(The predicates `first/2` and `tail/2` are predefined predicates. `first(A,B)` is true only if A is the first element of B and `tail(A,B)` is true only if A is the list obtained by removing the first object from B).

This predicate query succeeds if the first argument can be obtained by appending the second one to the third:

[1]It is really tricky to define the term understanding in its full generality. Since we are programming in prolog, we would say that our knowledge base has '*understood*' the concept of a certain predicate if it can return correct results to all possible queries involving that predicate.

[2]Of course, one can cram multiple rules into one using if-then-else constructs, but we are not considering that.

```
? append([a,b,c,d],[a,b],[c,d]).
True
? append([a,b,c],[],[a,b,c]).
True
```

The second rule is the actual recursive rule, which says that a query `append(A,B,C)` will succeed if these two criteria are satisfied:

1. The first element of the first argument is equal to the first element of the third one.
2. The rest of the first argument can be obtained by appending the second argument to the rest of the third argument.

The first rule is the exception handling rule, which succeeds only when the first argument is an empty list and the second argument is equal to the third one. These two rules give a working definition of the append predicate in terms of the basic predicates `first/2` and `tail/2`.

2 Strategy

Now that the goal has been clearly specified, it is time to formulate the procedure. The general structure of recursive rules indicates that the job can be divided into two parts:

1. Finding out what the arguments of the recursive predicate call should be.
2. Deciding when the recursion should stop.

2.1 The recursion step

As demonstrated in 1.1, the recursion step involves applying certain operations on the arguments of the predicate in the LHS so that the recursive precidate call can be made with the new set of arguments. For example, let's consider this query:

```
append([a,b,c,d,e,f],[a,b,c,d],[e,f])
```

Now, when we human beings try to gain an understanding of what the predicate `append/2` actually means from this example, all we can possibly do is look at all possible sublists of the arguments and try to find similarities between those bits and pieces. This activity of *similarity matching* is central to any human learning process in general. Getting back to the example we were talking about, we can readily see that the first element of the first argument [a,b,c,d,e,f] is equal to the first argument of the second element [a,b,c,d] too. Now we fix our attention to the next element of [a,b,c,d,ef] and observe that it can be obtained by applying a tail and a first operation on the second argument succesively:

```
tail([a,b,c,d],[b,c,d]),first([b],[b,c,d]).
```

Now if we keep on finding out the links between the segments of [a,b,c,d,e,f] and the argument lists, we will get the links listed in Table1.

The links are quite interesting for the fragments [a],[b],[c] and [d] - each of them is equal to [tail]+ the preceeding link. Clearly, if we apply tail once to the first and the third arguments, then the links for [b] would be exactly identical to that of [a]. This information, in fact, is sufficient to form the second rule. All we need to do is:

Table 1: **Links for the elements of [[a],[b],[c],[d],[e],[f]]**

Fragment	Link with itself	Link with other lists, # of list
[a]	[first]	[first],2
[b]	[tail,first]	[tail,first],2
[c]	[tail,tail,first]	[tail,tail,first],2
[d]	[tail,tail,tail,first]	[tail,tail,tail,first],2
[e]	[tail,tail,tail,tail,first]	[first],3
[f]	[tail,tail,tail,tail,tail,first]	[tail,first],3

2.1.1 Algorithm for the second rule

1. Find out a partition of the first argument whose elements have the property that the links for one is equal to some fixed quantity (let's call it *diff* from now on) + the corresponding link for the preceeding element. Here, for example, the partition in question is [[a],[b],[c],[d],[e]]; but it could be any valid partition like [[a],[b,c],[d,e,f]] too, depending on the example given and the predefined predicates.

2. Find out the links for the first element of the partition and use then in the second line, before the recursive call. For example, here the only relevant link is between the first element of [a,b,c,d,e,f] and the first element of [a,b,c,d], which is that of equality.

3. Apply the appropriate *diff* on each argument and make the recursive predicate call with the new set of arguments.

2.2 The exception handling step

The exception handling step is about formulating an alternate definition of the predicate for the cases where the recursive definition is not valid. In the example of append/2 we are discussing, such a situation arises when we reach the predicate call:

```
append([e,f],[],[e,f]).
```

Our algorithm outlined in 2.1.1 will fail here for the simple reason that one cannot apply tail to an empty list. So how do we identify these oddities? The answer again lies in the Table 2.1. If we look at the *'Link with other lists'* column for [d] and [e], we will find that the recursive pattern breaks at that very point. This breakdown of the recursive pattern among the links immediately calls for an exception handling rule. The way we actually build up that rule is as follows:

2.2.1 Algorithm for the exception handler

1. The point at which the pattern breaks is identified.

2. The number of *diff's* needed to reach there from the list we started with is noted.

3. The same number of *diff*'s are applied on each of the arguments. (Note that each argument has its seperate *diff*, which can be a null list too).

4. We find out all possible links among the sublists of the lists obtained after appliying the *diff*'s.

5. We get this set of links for the remaining lists for all the examples provided.

6. The intersection of all these sets go into the exception handler.

3 The Examples

As we mentioned a while ago, the set of examples this program would work on needs to satisfy certain criteria due to the very way our algorithm is designed. Those criteria are as follows:

1. The examples should be such that the *diff* can be identified with certainty. For example, `append([a,b,c,d],[a,b],[c,d]).` is not an ideal example for our program. This is so because the recursive pattern breaks at [c] here, so the program cannot check if the *diff* (i.e. [tail]) remains constant from element to element.

2. The examples have to be such that one exception handling rule suffices to uniquely define it.

3. The set of examples must be self-consistent and unambiguous.

This program is capable of finding out the correct rule from even just one rule, but multiple examples might reduce the ambiguity for some cases.

4 About the Code

We used swi-prolog for testing the code, but it should work just fine in other dialects also. As mentioned in the `README`, swi-prolog should be started with command line switches *'`-s init.pl -g init`'* to get the final outcome without manually entering queries.

The program reads the examples from the examples.pl file, so that file may be changed to suit ones needs, so long as the restrictions mentioned in 3 are not violated.

The files in the *`./sunshile_alps/memory/`* directory contains the predefined predicates. This program can use any predicate defined in some file in this directory to build the desired recursive rule.

Acknowledgement

I am grateful to my collaborators - Dibya Chakravorty and Sonali Mohapatra - for helping me get out of many logical nightmares and to our mentor Dr. Ananda Dasgupta for his positive opinions and encouragement.