Tutorial 11

Week of Novemver 28, 2005

Overview

Dear Students,

1. We will examine `findall/3`, `bagof/3`, and `setof/3` which you will find of use on A5.
2. We will look at a data structure example.
3. We will answer general questions about A5.

Sheila

1 findall/3, bagof/3 and setof/3

Acknowledgements to Patrick Blackburn, Johan Bos and Kristina Striegnitz for the material in this section.

There may be many solutions to a query. For example, suppose we are working with the database

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y) :- child(X,Y).

descend(X,Y) :- child(X,Z),
                descend(Z,Y).
```

Then if we pose the query

```
descend(martha,X).
```

there are four solutions (namely `X=charlotte, X=caroline, X=laura, X=rose`).

However Prolog generates these solutions one by one. Sometimes we would like to have all the solutions to a query, and we would like them handed to us in a neat, usable, form. Prolog has three built-in predicates that do this: `findall`, `bagof`, and `setof`. Basically these predicates collect all the solutions to a query and put them in a list, but there are important differences between them, as we shall see.

1.1 findall/3

The query

```
findall(Object,Goal,List).
```

produces a list List of all the objects Object that satisfy the goal Goal. Often Object is simply a variable, in which case the query can be read as: Give me a list containing all the instantiations of Object which satisfy Goal.

Here's an example. Suppose we're working with the above database (that is, with the information about child and the definition of descend). Then if we pose the query

```
findall(X,descend(martha,X),Z).
```

we are asking for a list Z containing all the values of X that satisfy descend(martha,X). Prolog will respond

```
X = _7489
Z = [charlotte,caroline,laura,rose]
```

But Object doesn't have to be a variable, it may just contain a variable that is in Goal. For example, we might decide that we want to build a new predicate fromMartha/1 that is true only of descendants of Martha. We could do this with the query:

```
findall(fromMartha(X),descend(martha,X),Z).
```

That is, we are asking for a list Z containing all the values of fromMartha(X) that satisfy the goal descend(martha,X). Prolog will respond

```
X = _7616
Z = [fromMartha(charlotte),fromMartha(caroline),
                fromMartha(laura),fromMartha(rose)]
```

Now, what happens, if we ask the following query?

```
findall(X,descend(mary,X),Z).
```

There are no solutions for the goal descend(mary,X) in the knowledge base. So findall returns an empty list.

Note that the first two arguments of findall typically have (at least) one variable in common. When using findall, we normally want to know what solutions Prolog finds for certain variables in the goal, and we tell Prolog which variables in Goal we are interested in by building them into the first argument of findall.

You might encounter situations, however, where findall does useful work although the first two arguments don't share any variables. For example, if you are not interested in who exactly is a descendant of Martha, but only in how many descendants Martha has, you can use the follwing query to find out:

```
?- findall(Y,descend(martha,X),Z), length(Z,N).
```

1.2 bagof/3

The findall/3 predicate is useful, but in certain respects it is rather crude. For example, suppose we pose the query

```
findall(Child,descend(Mother,Child),List).
```

We get the response

```
Child = _6947
Mother = _6951
List = [charlotte,caroline,laura,rose,caroline,laura,rose,laura,rose,rose]
```

Now, this is correct, but sometimes it would be useful if we had a separate list for each of the different instantiations of Mother.

This is what bagof lets us do. If we pose the query

```
bagof(Child,descend(Mother,Child),List).
```

we get the response

```
Child = _7736
Mother = caroline
List = [laura,rose] ;

Child = _7736
Mother = charlotte
List = [caroline,laura,rose] ;

Child = _7736
Mother = laura
List = [rose] ;

Child = _7736
Mother = martha
List = [charlotte,caroline,laura,rose] ;
```

```
no
```

That is, bagof is more finegrained than findall, it gives us the opportunity to extract the information we want in a more structured way. Moreover, bagof can also do the same job as findall, with the help of a special piece of syntax. If we pose the query

```
bagof(Child,Mother ^ descend(Mother,Child),List).
```

This says: give me a list of all the values of Child such that descend(Mother,Child), and put the result in a list, but don't worry about generating a separate list for each value of Mother. So posing this query yields:

```
Child = _7870
Mother = _7874
List = [charlotte,caroline,laura,rose,caroline,laura,rose,laura,rose,rose]
```

Note that this is exactly the response that findall would have given us. Still, if this is the kind of query you want to make (and it often is) it's simpler to use findall, because then you don't have to bother explicitly write down the conditions using ^.

Further, there is one important difference between findall and bagof, and that is that bagof fails if the goal that's specified in its second argument is not satisfied (remember, that findall returns the empty list in such a case). So the query bagof(X,descend(mary,X),Z) yields no.

One final remark. Consider again the query

```
bagof(Child,descend(Mother,Child),List).
```

As we saw above, this has four solutions. But, once again, Prolog generates them one by one. Wouldn't it be nice if we could collect them all into one list?

And, of course, we can. The simplest way is to use findall. The query

```
    findall(List,bagof(Child,descend(Mother,Child),List),Z).
```

collects all of bagof's responses into one list:

```
    List = _8293
    Child = _8297
    Mother = _8301
    Z = [[laura,rose],[caroline,laura,rose],[rose],
                      [charlotte,caroline,laura,rose]]
```

Another way to do it is with bagof:

```
    bagof(List,Child ^ Mother ^ bagof(Child,descend(Mother,Child),List),Z).

    List = _2648
    Child = _2652
    Mother = _2655
    Z = [[laura,rose],[caroline,laura,rose],[rose],
                      [charlotte,caroline,laura,rose]]
```

Now, this may not be the sort of thing you need to do very often, but it does show the flexibility and power offered by these predicates.

1.3 setof/3

The setof/3 predicate is basically the same as bagof, but with one useful difference: the lists it contains are ordered and contain no redundancies (that is, each item appears in the list only once).

For example, suppose we have the following database

```
age(harry,13).
age(draco,14).
age(ron,13).
age(hermione,13).
age(dumbledore,60).
age(hagrid,30).
```

Now suppose we want a list of everyone whose age is recorded in the database. We can do this with the query:

```
findall(X,age(X,Y),Out).

X = _8443
Y = _8448
Out = [harry,draco,ron,hermione,dumbledore,hagrid]
```

But maybe we would like the list to be ordered. We can achieve this with the following query:

```
setof(X,Y ^ age(X,Y),Out).
```

(Note that, just like with bagof, we have to tell setof not to generate separate lists for each value of Y, and again we do this with the ^ symbol.)

This query yields:

```
X = _8711
Y = _8715
Out = [draco,dumbledore,hagrid,harry,hermione,ron]
```

Note that the list is alphabetically ordered.

Now suppose we are interested in collecting together all the ages which are recorded in the database. Of course, we can do this with the following query:

```
findall(Y,age(X,Y),Out).

Y = _8847
X = _8851
Out = [13,14,13,13,60,30]
```

But this output is rather messy. It is unordered and contains repetitions. By using setof we get the same information in a nicer form:

```
setof(Y,X ^ age(X,Y),Out).

Y = _8981
X = _8985
Out = [13,14,30,60]
```

Between them, these three predicates offer us a lot of flexibility. For many purposes, all we need is findall. But if we need more, bagof and setof are there waiting to help us out.

2 Prolog Data Structures – Binary Search Tree

We can represent a binary tree of integers as:

```
tree(Root, Left, Right)
```

where `Left` and `Right` are also binary trees such that all elements in `Left` are less than the `Root` and all elements in `Right` are greater than or equal to `Root`. If a tree is `null`, it means that the tree is empty. E.g, :

```
tree(100, tree(50, null, null), tree(75, null, null)).

tree(10, tree(6, tree(4, null, tree(5, null, null)), tree(8, null, null)),
                                        tree(12, null, null)).
```

Write a predicate that traverses a binary tree, generating a sorted list of its contents. I.e., define `traverse( +Tree, -SortedList)` where `Tree` is a binary tree and `SortedList` is the sorted list of members of the tree. E.g.,

```
?- traverse(tree(100, tree(50, null, null), tree(75, null, null)),X).

X = [50, 100, 75] ;
No

?- traverse(tree(10, tree(6, tree(4, null, tree(5, null, null)), tree(8, null, n
ull)),tree(12, null, null)), X).

X = [4, 5, 6, 8, 10, 12] ;
No
```

Solution

```
traverse(null, []).
traverse(tree(Root, Left, Right), SortedList) :-
      traverse(Left, SortedListLeft),
      traverse(Right, SortedListRight),
      append(SortedListLeft,
             [Root | SortedListRight],
             SortedList).
```