Introduction to AI with Prolog

Database and aggregation

Alexey Sery
a.seryj@g.nsu.ru

Department of Information Technologies
Novosibirsk State University

October 6, 2020

Overview

Goals of the lesson

- Define a predicate which causes one or more clauses to be added to or deleted from the Prolog database.
- Define predicates to create and manipulate a database of related facts within the Prolog database.
- To discuss inbuilt predicates that let us collect all solutions to a problem into a single list.

Database

The normal way of placing clauses in the Prolog database is to consult or reconsult one or more files.

A consult directive causes all the clauses in the file to be loaded into the knowledge base to add to those already there. A reconsult directive behaves in a similar way to consult, with one crucial difference. If any clause in the reconsulted file is for the same predicate as a clause already in the database (i.e. their heads have the same functor and arity), all clauses for that predicate in the database are first deleted.

Clauses placed into the knowledge base normally stay there until added to or deleted by a subsequent consult or reconsult directive, or until the user exits from the Prolog system when all clauses are automatically deleted. For most purposes this is entirely sufficient.

Database

However Prolog also has built-in predicates for adding clauses to and deleting clauses from the database which can be useful for more advanced programming in the language. Like many other 'advanced' features, they need to be used with care.

As usual, these built-in predicates can be used either in the body of a rule or in a directive entered at the system prompt. As the user's program and the Prolog database are equivalent, using them in the body of a clause is equivalent to modifying the user's program while it is being used.

Database

SWI-Prolog offers predicates to manage knowledge base during the execution of Prolog programs.

- `assert/1`
- `asserta/1`
- `assertz/1`
- `retract/1`
- `retractall/1`
- `abolish/1`

There are others, of cause. You can become acquainted with them here: https://www.swi-prolog.org/pldoc/man?section=db

Database

If any predicate is to be modified using `assertz`, `retract` etc., it should be specified as **dynamic** by a directive in the user's program. As an example, for predicate `f` with arity 2, the line

```
:- dynamic(f/3).
```

should be added at or near the start of the program and certainly before the first clause that mentions the `f` predicate.

If the dynamic directive is left out, attempts to modify the database are likely to produce error messages such as *"Predicate Protected"* or even *"Predicate Not Defined"*.

Database

Suppose we start with an empty database. So if we give the command:

```
?- listing.
```

we simply get a `true.`; the listing is empty, except for the garbage it always prints for no sane reason. Suppose we now give this command:

```
?- assert(f(x,y)).
```

It succeeds (`assert` commands **always succeed**). But what is important is the side-effect it has on the database. If we now give the command:

```
?- listing.
```

we get the listing:

```
:- dynamic f/2.
f(x,y).
```

Database

We can do the same thing with rules as well. Rules must be enclosed in an additional pair of parentheses.

```
?- assert((f(X,y) :- X > 5)).
```

As you can see, the clause used for the argument should be written without a terminating full stop.

Now we may pose query like:

```
?- f(10, y).
```

and get an answer `true.`

Note that clauses will occur in the knowledge base as many times as they were asserted.

Database

Two main predicates available for deleting clauses from the database are:

- `retract(?Clause)`
- `retractall(?Clause)`

Database

The predicate `retract/1` takes a single argument, which must be a clause, i.e. a fact or a rule. It causes the first clause in the database that matches (i.e. unifies with) `Clause` to be deleted. If the following clauses are in the database

```
dog(jim).
dog(fido).
dog(X).
```

the query

```
?- retract(dog(fido)).
```

will delete the second clause and the further query

```
?-retract(dog(X)).
```

will delete the `dog(jim)` clause, which is the first one of the remaining two clauses to unify with the query. This will leave the `dog(X)` clause in the database. Although unusual, this is a valid Prolog fact which signifies 'everything is a dog'.

Database

The predicate `retractall/1` takes a single argument which must be the head of a clause. It causes every clause in the database whose head matches `Clause` to be deleted. The `retractall` goal always succeeds even if no clauses are deleted. Some examples are:

```
?- retractall(f(_,_)).
```

which deletes all the clauses for the `f/2` predicate, and

```
?- retractall(parent(john,Y)).
```

which deletes all clauses for the `parent/2` predicate which have the atom `john` as their first argument. Note that the directive

```
?- retractall(f).
```

only removes the clauses for predicate `f/0`. To delete all the clauses for predicate `f/2` it is necessary to use

```
?- retractall(f(_,_)).
```

Database: Examples

Let us take an empty database and add some facts and one rule to it:

```
?- assertz(f(x,y)).

?- assertz(f(x,y)).

?- assertz(f(a,b)).

?- assertz((f(X,y) :- g(X,a,b))).
```

Database: Examples

Now if we pose a query `?- listing(f/2).` we will get

```
:- dynamic f/2.

f(x,y).

f(x,y).

f(a,b).

f(X,y) :- g(X,a,b).
```

Database: Examples

Let us now delete one occurrence of the fact `f(x,y)`.

```
retract(f(x,y)).
```

```
f(x,y).
f(a,b).
f(X,y) :- g(X,a,b).
```

Database: Examples

From what is left in database we delete all facts and rules such that their head is matched with `f(X,y)`.

```
retractall(f(X,y)).
```

```
f(a,b).
```

Database: Examples

Consider another example:

```
multab(Scale) :-  member(X,Scale),
                  member(Y,Scale),
                  P is X * Y,
                  assertz(p(X,Y,P)),
                  fail.
```

Database: Conclusions

- You have to define predicate as dynamic in order to use it with `asserta`, `assertz`, `retract` and `retractall`.
- If predicate `p` is not yet defined in the database it will be defined as dynamic when first used in `assert` operation.
- Predicate `abolish/1` removes all clauses of a predicate from the database. All predicate attributes (dynamic, multifile, index, etc.) are reset to their defaults. According to the ISO standard, `abolish/1` can only be applied to dynamic procedures. This is odd, as for dealing with dynamic procedures there is already `retract/1` and `retractall/1`. The `abolish/1` predicate was introduced in DEC-10 Prolog precisely for dealing with static procedures. In SWI-Prolog, `abolish/1` works on static procedures, unless the Prolog flag iso is set to true. **It is advised to use `retractall/1` for erasing all clauses of a dynamic predicate.**

Aggregation

There may be more than one solution to a query. Prolog interpreter, however, returns solutions one by one. Sometimes we would like to have all the solutions to a query at once, and we would like them handed to us on a pretty usable form. Prolog has 3 built-in predicates that do precisely this:

- `findall/3`
- `bagof/3`
- `setof/3`

Basically these predicates collect all the solutions to a query and put them in a list, but there are important differences between them.

Aggregation

The query

```
?- findall(Buffer, Goal, Response).
```

produces a list `Response` of all the objects of a form of `Buffer` that satisfy the goal `Goal`. Often `Buffer` 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.**

If there are no solutions satisfying the `Goal` the `findall` predicate returns an empty list as response.

Aggregation

Suppose we want to get a subset of our little multiplication table we made:

```
?- findall([X,Y], p(9, X, Y), Response).
Response = [[1, 9], [2, 18], [3, 27], [4, 36], [5, 45], [6, 54],
[7, 63], [8|...], [...|...]].

?- findall(9 * X = Y, p(9, X, Y), Response).
Response = [9*1=9, 9*2=18, 9*3=27, 9*4=36, 9*5=45, 9*6=54,
9*7=63, ... * ... = 72, ... = ...].
```

Aggregation

The `bagof/3` predicate 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.

`?- bagof(pair(X,Y), p(X,Y,Z), R).`	`Z = 1, R = [pair(1, 1)] ;`
	`Z = 2, R = [pair(1, 2), pair(2, 1)] ;`
	`Z = 3, R = [pair(1, 3), pair(3, 1)] ;`
	`Z = 4, R = [pair(1, 4), pair(2, 2), pair(4, 1)] ;`
	`Z = 5, R = [pair(1, 5), pair(5, 1)] ;`

Aggregation

The bagof/3 predicate 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.

`?- bagof(X, p(X, Y, Z), R).`	`Y = Z, Z = 1, R = [1] ;`
	`Y = 1, Z = 2, R = [2] ;`
	`Y = 1, Z = 3, R = [3] ;`
	`Y = 1, Z = 4, R = [4] ;`

`?- bagof(X, Y^p(X, Y, Z), R).`	`Z = 1, R = [1] ;`
	`Z = 2, R = [1, 2] ;`
	`Z = 3, R = [1, 3] ;`
	`Z = 4, R = [1, 2, 4] ;`
	`Z = 5, R = [1, 5] ;`
	`Z = 6, R = [1, 2, 3, 6] ;`

Aggregation

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.**

`?- findall(Y, p(X, Y, Z), R).`	`R = [1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9].`
`?- setof(Y, X^Z^p(X, Y, Z), R).`	`R = [1, 2, 3, 4, 5, 6, 7, 8, 9].`

Conclusions

- Dynamic database allows to cache solutions and manipulate facts and rules in runtime.
- Typically it is a bad idea to overuse the database manipulation predicates as it hampers considerably program tracing and debug. Nevertheless, there are scenarios where storing data outside the Prolog stacks is a good option.
- Typically, first consider representing data processed by your program as terms passed around as predicate arguments. If you need to reason over multiple solutions to a goal, consider `findall/3`, and related predicates.