Logisch Programmeren en Zoektechnieken

Ulle Endriss
Institute for Logic, Language and Computation
University of Amsterdam

http://www.illc.uva.nl/~ulle/teaching/prolog/

Advanced Prolog

The purpose of this lecture is to introduce some of the more advanced programming constructs in Prolog not covered in previous lectures:

- Decomposing terms with `=../2`
- Collecting answers: `findall/3` etc.
- Dynamic predicates: `assert/1` and `retract/1`
- Input/output and file handling

Term Decomposition

Recall that a *compound term* consists of a *functor* and one or more *arguments*. (An *atomic term* has no arguments.)

Given a term `T`, the predicate `=../2` (which is defined as an infix operator) can be used to generate a list, the head of which is the functor of `T` and the tail of which is the list of arguments of `T`:

```
?- loves(john,mary) =.. List.
List = [loves, john, mary]
Yes

?- elephant =.. List.
List = [elephant]
Yes
```

Composing Terms

You can also use `=../2` to compose new terms:

```
?- member(X, [f,g,h]), Y =.. [X,a,b].
X = f
Y = f(a, b) ;
X = g
Y = g(a, b) ;
X = h
Y = h(a, b) ;
No
```

This is very useful, because using a variable in the position of a functor would cause a syntax error (for most Prolog systems):

```
?- member(X, [f,g,h]), Y = X(a,b).
ERROR: Syntax error: Operator expected
```

Exercise: Matching Subterms

Write a predicate that, given two terms, succeeds if the first term matches a subterm of the second. Examples:

```
?- subterm(f(a,b), f(g(a,b,c),f(a,b),c)).
Yes

?- subterm(f(X,X), g(f(a,b))).
No

?- subterm(Term, f(g(a,b),a)).
Term = f(g(a, b), a) ;
Term = g(a, b) ;
Term = a ;
Term = b ;
Term = a ;
No
```

Solution

Any term is a subterm of itself (base case). Otherwise, the first term is a subterm of the second if the latter is a compound term and the former is a subterm of one of the arguments of that compound term (recursion step).

```
subterm(Term, Term).


subterm(SubTerm, Term) :-
  Term =.. [_|Arguments],
  member(Argument, Arguments),
  subterm(SubTerm, Argument).
```

This solution assumes that the second argument will be a *ground* term. It would, for instance, succeed for `subterm(f(X),Y)` and then give an error message in case of enforced backtracking (try it!).

Summary: Decomposing Terms with `=../2`

- The built-in predicate `=../2` can be used to de/compose terms. It is declared as a (non-associative) infix operator.
- A goal of the form `Term =.. List` succeeds when `List` is a list, the head of `List` is the functor of `Term`, and the tail of `List` is the list of arguments of `Term`.

  Either `Term` or `List` could be a variable, but not both.

Backtracking and Alternative Answers

Next we are going to see how to collect all alternative answers to a given query (or goal) in a list.

Assume the following program has been consulted:

```
student(ann, 44711, pass).
student(bob, 50815, pass).
student(pat, 41018, fail).
student(sue, 41704, pass).
```

We can get all the answers to a query by forcing Prolog to backtrack. Example:

```
?- student(Name, _, pass).
Name = ann ;
Name = bob ;
Name = sue ;
No
```

Collecting Answers in a List

Instead, the `findall/3` predicate can be used to collect these answers in a single list. Examples:

```
?- findall(Name, student(Name,_,pass), List).
List = [ann, bob, sue]
Yes

?- findall(Name, student(Name,_,dunno), List).
List = []
Yes
```

Specification of findall/3

Schema: `findall(+Template, +Goal, -List)`

Prolog will search for every possible solution to the goal `Goal` (through backtracking). For every solution found, the necessary instantiations to `Template` are made, and these instantiations are collected in the list `List`.

That is, `Template` and `Goal` should share one or more variables. Variables occurring in `Goal` but not in `Template` can have any value (these are not being reported).

Another Example

Here is again our program:

```
student(ann, 44711, pass).
student(bob, 50815, pass).
student(pat, 41018, fail).
student(sue, 41704, pass).
```

An example with a complex goal and a template with two variables:

```
?- Goal = (student(Name,Num,Grade), Num < 50000),
   findall(Name/Grade, Goal, List).
Goal = (student(Name, Num, Grade), Num<50000),
List = [ann/pass, pat/fail, sue/pass]
Yes
```

Collecting Answers with bagof/3

The `bagof/3` predicate is similar to `findall/3`, but now the values taken by variables occurring in the goal but not the template *do* matter and a different list is created for every possible instantiation of these variables. Example:

```
?- bagof(Name/Num, student(Name,Num,Grade), List).
Grade = fail
List = [pat/41018] ;

Grade = pass
List = [ann/44711, bob/50815, sue/41704] ;
No
```

Example with an Unbound Variable

In the following query we say that we are not interested in the value of `Num` (by using the `^` operator), but Prolog will still give alternative solutions for every possible instantiation of `Grade`:

```
?- bagof(Name, Num^student(Name,Num,Grade), List).
Grade = fail
List = [pat] ;

Grade = pass
List = [ann, bob, sue] ;
No
```

Summary: Collecting Answers

- `findall/3` collects all the answers to a given *goal* that match a given *template* in a *list*. Variables not occurring in the template may take different values within the list of answers.
- `bagof/3` is similar, but now a different list is generated for every possible instantiation of the variables not occurring in the template. Use the `Var^` construct to allow for a variable to take different values within the same list of answers.
- The predicate `setof/3` works like `bagof/3`, but duplicates are being removed and the list is being ordered.
- Note that `findall/3` returns an empty list if the goal in the second argument position cannot be satisfied, while `bagof/3` and `setof/3` will simply fail.
- Use these predicates sparingly! They tend to tempt people into writing inelegant and inefficient programs.

Assert and Retract

Prolog evaluates queries with respect to a knowledge base (your program + definitions of built-in predicates). It is possible to *dynamically* add clauses to this knowledge base.

- Executing a goal of the form `assert(+Clause)` will add the clause `Clause` to the Prolog knowledge base.
- Executing `retract(+Clause)` will remove that clause again.
- Using `retractall(+Clause)` will remove *all* the clauses matching `Clause`.

A typical application would be to dynamically create and manipulate a database. In that case the `Clause`s will usually be simple facts.

Database Example

```
?- assert(zoo(monkey)), assert(zoo(camel)).
Yes

?- zoo(X).
X = monkey ;
X = camel ;
No

?- retract(zoo(monkey)).
Yes

?- zoo(X).
X = camel ;
No
```

Dynamic Manipulation of the Program

You can even declare your program predicates as being dynamic and assert and retract clauses for these predicates.

Example: Suppose we have consulted our "big animals" program from the first lecture (see next slide for a reminder) and suppose we have declared `bigger/2` as a dynamic predicate ...

```
?- is_bigger(camel, monkey).
No

?- assert(bigger(camel,horse)).
Yes

?- is_bigger(camel, monkey).
Yes
```

The Big Animals Program

```
:- dynamic bigger/2.

bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).

is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

Fast Fibonacci

A possible application of dynamic predicates is to store previously computed answers, rather than to compute them again from scratch each time they are needed. Example:

```
:- dynamic fibo/2.
fibo(0, 1). fibo(1, 1).
fibo(N, F) :-
  N >= 2, N1 is N - 1, N2 is N - 2,
  fibo(N1, F1), fibo(N2, F2), F is F1 + F2,
  asserta(fibo(N,F):-!). % assert as first clause
```

This is much faster than the standard program for the Fibonacci sequence (= the above program without the last line). However, a solution with just a single recursive call is even better than this one.

Summary: Dynamic Predicates

- Use `assert/1` to add clauses to the knowledge base and use `retract/1` or `retractall/1` to remove them again.
- If the predicate to be asserted or retracted is already in use, then it needs to be declared as being dynamic first (this may work differently for different Prolog systems; in SWI-Prolog use the `dynamic` directive).
- If the order of clauses in the dynamic knowledge base matters, there are further predicates such as `asserta/1` and `assertz/1` that can be used (check the manual for details).
- *Very important:* Be extremely careful when using dynamic predicates! They obfuscate the declarative meaning of Prolog and make it much harder to check programs (the same code will behave differently given different dynamic knowledge bases). Often a sign of poor programming style!

Input/Output and File Handling

We have already seen how to explicitly output data onto the user's terminal using the `write/1` predicate. Example:

```
?- X is 5 * 7, write('The result is: '), write(X).
The result is: 35
X = 35
Yes
```

Now we are also going to see how to read input. In Prolog, input and output from and to the user are similar to input and output from and to files, so we are going to deal with these in one go.

Streams

- In Prolog, input and output happen with respect to two *streams:* the current input stream and the current output stream. Each of these streams could be either the user's terminal (default) or a file.
- The current *output stream* can be changed to `Stream` by executing the goal `tell(+Stream)`. Example:

```
?- tell('example.txt').
Yes
```

- Now `write/1` does not write to the user's terminal anymore, but to the file `example.txt`:

```
?- write('Hello, have a beautiful day!').
Yes
```

- To close the current output stream, use the command `told/0`.

Reading Terms from the Input Stream

- The corresponding predicates for choosing and closing an *input stream* are `see/1` and `seen/0`.
- To read from the current input stream, use the predicate `read/1`. But note that this only works if the input stream is a sequence of *terms*, each of which is followed by a full stop (as in a Prolog program file, for instance).

Reading Terms from a File

```
?- see('students.txt').
Yes
?- read(Next).
Next = student(ann, 44711, pass)
Yes
?- read(Next).
Next = student(bob, 50815, pass)
Yes
?- read(Next).
Next = student(pat, 41018, fail)
Yes
?- read(Next).
Next = end_of_file
Yes
?- seen.
Yes
```

Content of file `students.txt`:

```
% Database of students
student(ann, 44711, pass).
student(bob, 50815, pass).
student(pat, 41018, fail).
```

Example with User Input

Consider the following program:

```
start :-
  write('Enter a number followed by a full stop: '),
  read(Number),
  Square is Number * Number,
  write('Square: '),
  write(Square).
```

After compilation, it works as follows:

```
?- start.
Enter a number followed by a full stop: 17.
Square: 289
Yes
```

Summary: Input/Output and File Handling

- Input and output use the concept of *streams*. The default input/output stream is the user's terminal.
- Main predicates:
  - `see/1`: choose an input stream
  - `seen/0`: close the current input stream
  - `tell/1`: choose an output stream
  - `told/0`: close the current output stream
  - `read/1`: read the next term from the input stream
  - `write/1`: write to the output stream
- Using `read/1` only works with text files that are sequences of Prolog terms. To work with arbitrary files, have a look at the predicate `get/1`.
- There are many more input/output predicates in the manual.