Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 11

ML examples III: A Prolog interpreter

Topics covered:

- Prolog terms: parsing and printing
- Unification and Prolog backtracking
- Prolog examples

Overview

Prolog is another ‘declarative’ language, popular in Artificial Intelligence research.

You will learn more about it in the lecture course ‘Prolog for Artificial Intelligence’.

It is a logic programming language, and is said to be ‘relational’ rather than functional.

Its standard search strategy, backward chaining with unification and backtracking, is useful for a wide range of applications including databases, expert systems and theorem provers.

We will show how to implement a cut-down Prolog interpreter in ML.

Later, we will then use the same tools and techniques to build a simple theorem prover.

Prolog terms

Prolog code and data is represented using a uniform system of first order terms.

We will represent this using a recursive type similar to the one used for mathematical expressions.

```
datatype term =
    Var of string
  | Fn of string * (term list);
```

Note that we treat constants as nullary functions, i.e. functions that take an empty list of arguments.

Where we would formerly have used `Const s`, we will now use `Fn(s,[])`.

Note that we will treat functions of different arities (different numbers of arguments) as distinct, even if they have the same name.

Auxiliary functions (1)

Again we use some auxiliary functions:

```
fun itlist f [] b = b
      | itlist f (h::t) b =
          f h (itlist f t b);

fun assoc a ((x,y)::rest) =
  if a = x then y else assoc a rest;

fun exists p l = itlist
    (fn h => fn a => p(h) orelse a)
    l false;

fun fst(x,y) = x;

fun snd(x,y) = y;

val explode = map str o explode;
```

Auxiliary functions (2)

```
fun itlist2 f [] [] b = b
  | itlist2 f (h1::t1) (h2::t2) b =
      f h1 h2 (itlist2 f t1 t2 b);

load "Int";
val string_of_int = Int.toString;

fun filter p l = itlist
    (fn x => fn s =>
     if p x then x::s else s) l [];

fun partition p l =
  (filter p l,filter (fn x => not(p x)) l);
```

As well as this, we use many of the parser combinators etc. that were given in the last lecture. And we have a special exception for error messages:

```
exception error of string;
```

Lexical analysis (1)

In Prolog, identifiers that begin with an *upper case* letter or an underscore are treated as variables, while other alphanumeric identifiers, along with numerals, are treated as constants.

For example `X` and `Answer` are variables while `x` and `john` are constants.

We will lump all symbolic identifiers together as constants too, but we will distinguish the punctuation symbols: left and right brackets, commas and semicolons.

Non-punctuation symbols are collected together into the longest strings possible, so symbolic identifiers need not consist only of one character.

```
datatype token =
    Variable of string
  | Constant of string
  | Punct of string;
```

Lexical analysis (2)

```
val lex = let
 fun several p = many (some p)
 fun collect(h,t) =
   h^(itlist (fn s1 => fn s2 => s1^s2) t "")
 fun upper_alpha s =
   "A" <= s andalso s <= "Z" orelse s = "_"
 fun lower_alpha s = "a" <= s andalso s <= "z"
   orelse "0" <= s andalso s <= "9"
 fun punct s = s = "(" orelse s = ")" orelse s = "["
   orelse s = "]" orelse s = "," orelse s = "."
 fun space s = s = " " orelse s = "\n" orelse s = "\t"
 fun alnum s = upper_alpha s orelse lower_alpha s
 fun symbolic s = not (space s) andalso not (alnum s)
   andalso not (punct s)
 val rawvariable = some upper_alpha ++ several alnum
       >> (Variable o collect)
 val rawconstant = (some lower_alpha ++ several alnum
    || some symbolic ++ several symbolic)
       >> (Constant o collect)
 val rawpunct = some punct >>  Punct
 val token = (rawvariable || rawconstant || rawpunct)
             ++ several space >> fst
 val tokens = (several space ++ many token) >> snd
 val alltokens = (tokens ++ finished) >> fst
in fst o alltokens o explode end;
```

Parsing and printing

The printer is exactly the same as before.

The parser is almost the same as before; again we allow some operators like addition to be written infix. We also allow a special syntax for lists.

We regard `[x1,x2,...,xn]` as shorthand for

`cons(x1,cons(x2,...,cons(xn,nil)))`

where `cons` is the function corresponding to a list constructor, just like `::` in ML. The printer writes it as an infix dot. Also, `[H|T]` is allowed instead of `cons(H,T)`.

A Prolog *rule* is of one of these forms:

$$term.$$

$$term \quad \text{:-} \quad term, \ldots, term.$$

We have a parser for these, returning a `term * term list`.

Unification

Prolog uses a set of rules to solve a current *goal* by trying to match one of the rules against the goal.

A rule consisting of a single term can solve a goal immediately.

In the case of a rule $term$:- $term_1, \ldots, term_n$., if the goal matches $term$, then Prolog needs to solve each $term_i$ as a subgoal in order to finish the original goal.

However, goals and rules do not have to be exactly the same. Instead, Prolog assigns variables in both to make them match up. This process is called *unification*.

This means that we can end up proving a special case of the original goal, e.g. $P(f(X))$ instead of $P(Y)$.

Unification: examples and algorithm

To unify $f(g(X), Y)$ and $f(g(a), X)$ we need to set $X = a$ and $Y = a$. Then both terms are $f(g(a), a)$.

To unify $f(a, X, Y)$ and $f(X, a, Z)$ we need to set $X = a$ and $Y = Z$, and then both terms are $f(a, a, Z)$.

It is impossible to unify $f(X)$ and X.

There is a systematic procedure for finding a *most general* unifier.

Roughly, one descends the two terms recursively in parallel, and on finding a variable on either side, assigns it to whatever the term on the other side is.

One needs to check that the variable hasn't already been assigned to something else, and that it doesn't occur in the term being assigned to it (as in the last example above).

Unification: code

We have a set of existing instantiations, and we look each variable up to see if it is already assigned before proceeding.

```
fun unify (tm1 as Var(x)) tm2 insts =
      (let val tm1' = assoc x insts
        in unify tm1' tm2 insts
       end handle Not_found => augment (x,tm2) insts)
  | unify (tm1 as Fn(f1,args1)) (Var y) insts =
      (let val tm2' = assoc y insts
        in unify tm1 tm2' insts
       end handle Not_found => augment (y,tm1) insts)
  | unify (tm1 as Fn(f1,args1)) (Fn(f2,args2)) insts =
       if f1 = f2
       then itlist2 unify args1 args2 insts
       else raise (error "functions do not match");
```

We use the existing instantiations as an accumulator. Other functions to augment instantiation lists are:

```
fun occurs_in x (Var y) = (x = y)
  | occurs_in x (Fn(_,args)) =
      exists (occurs_in x) args;
```

Augmenting instantiations

```
fun subst insts (tm as Var y) =
      (assoc y insts handle Not_found => tm)
  | subst insts (Fn(f,args)) =
      Fn(f,map (subst insts) args);

local fun augment1 theta (x,s) =
  let val s' = subst theta s
   in if occurs_in x s andalso not(s = Var(x))
    then raise (error "Occurs check")
    else (x,s') end in
fun raw_augment p insts =
  p::(map (augment1 [p]) insts) end;

fun augment (v,t) insts =
  let val t' = subst insts t
   in case t' of
     Var(w) => if w <= v then
                 if w = v then insts
               else raw_augment (v,t') insts
               else raw_augment (w,Var(v)) insts
   | _ => if occurs_in v t'
          then raise (error "Occurs check")
          else raw_augment (v,t') insts
  end;
```

Backtracking

Prolog proceeds by depth-first search, but it may backtrack: even if a rule unifies successfully, if all the remaining goals cannot be solved under the resulting instantiations, then another initial rule is tried. Thus we consider the whole list of goals rather than one at a time:

```
fun first f [] = raise (error "No rules applicable")
  | first f (h::t) = f h handle error _ => first f t;

fun expand n rules insts goals =
  first (fn rule =>
    if goals = [] then insts else
    let val (conc,asms) =
          rename_rule (string_of_int n) rule
        val insts' = unify conc (hd goals) insts
        val (loc,glob) = partition
          (fn (v,_) => occurs_in v conc orelse
                   exists (occurs_in v) asms) insts'
        val goals' = (map (subst loc) asms) @
                       (tl goals)
     in expand (n + 1) rules glob goals'
     end) rules;
```

Other details

Note that we produce fresh variables for the rules each time:

```
fun rename s (Var v) = Var("~"^v^s)
  | rename s (Fn(f,args)) = Fn(f,map (rename s) args);
fun rename_rule s (conc,asms) =
  (rename s conc,map (rename s) asms);
```

Finally, we package everything up:

```
datatype outcome = No | Yes of (string * term) list;

fun prolog rules goal =
  let val insts = expand 0 rules [] [goal]
   in Yes(filter (fn (v,_) => occurs_in v goal)
                 insts)
  end handle error _ => No;
```

This says either that the goal cannot be solved, or that it can be solved with the given instantiations. Note that we only return one answer in this case, but this is easy to fix.

Prolog examples (1)

```
- val rules = parse_rules
   "male(albert).                        \
   \ male(edward).                        \
   \ female(alice).                       \
   \ female(victoria).                    \
   \ parents(edward,victoria,albert).     \
   \ parents(alice,victoria,albert).      \
   \ sister_of(X,Y) :-                    \
   \   female(X),                         \
   \   parents(X,M,F),                    \
   \   parents(Y,M,F).";
> val rules =
   [(`male(albert)`,[]), (`male(edward)`,[]),
    (`female(alice)`,[]), (`female(victoria)`,[]),
    (`parents(edward,victoria,albert)`,[]),
    (`parents(alice,victoria,albert)`,[]),
    (`sister_of(X,Y)`,
    [`female(X)`, `parents(X,M,F)`, `parents(Y,M,F)`]]
    : (term * term list) list
- prolog rules ("sister_of(alice,edward)");
> val it = Yes [] : outcome
- prolog rules (parse_term "sister_of(alice,X)");
> val it = Yes [("X",`edward`)] : outcome
- prolog rules (parse_term "sister_of(X,Y)");
> val it = Yes [("Y",`edward`), ("X",`alice`)] : outcome
```

Prolog examples (2)

```
- val r = parse_rules
   "append([],L,L).                            \
   \ append([H|T],L,[H|A]) :- append(T,L,A).";
- val r = [`append([],L,L)`, [];
    `append(H . T,L,H . A)`, [`append(T,L,A)`]]
    : (term * term list) list
- prolog r (parse_term "append([1,2],[3],[1,2,3])");
> val it = Yes [] : outcome
- prolog r (parse_term "append([1,2],[3,4],X)");
> val it = Yes ["X", `1 . (2 . (3 . (4 . [])))`]
           : outcome
- prolog r (parse_term "append([3,4],X,X)");
> val it = No;
- prolog r (parse_term "append([1,2],X,Y)");
> val it = Yes ["Y", `1 . (2 . X)`] : outcome
```

Prolog is less 'directional' than ML. However it has its limitations, e.g. the following will loop indefinitely:

```
- prolog r (parse_term "append(X,[3,4],X)");
```