Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 12

ML examples IV: Prolog and Theorem Proving

Topics covered:

- Prolog terms: parsing and printing
- Unification and Prolog backtracking
- Prolog examples
- A simple theorem prover
- Theorem proving 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.

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.

```
type 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.

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.

```
type token = Variable of string
           | Constant of string
           | Punctuation of string;;
```

Lexical analysis (2)

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

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.

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

We use the existing instantiations as an accumulator.

Augmenting instantiations

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

let rec subst insts = fun
   (Var y) -> (try assoc y insts with Not_found -> tm)
 | (Fn(f,args)) -> Fn(f,map (subst insts) args);;

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

let augment (v,t) insts =
  let t' = subst insts t in match t' with
    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;;
```

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:

```
let rec first f =
  fun [] -> raise (error "No rules applicable")
    | (h::t) -> try f h with error _ -> first f t;;

let rec expand n rules insts goals =
  first (fun rule ->
    if goals = [] then insts else
    let conc,asms =
      rename_rule (string_of_int n) rule in
    let insts' = unify conc (hd goals) insts in
    let local,global = partition
      (fun (v,_) -> occurs_in v conc or
              exists (occurs_in v) asms) insts' in
    let goals' = (map (subst local) asms) @
                 (tl goals) in
    expand (n + 1) rules global goals') rules;;
```

Other details

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

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

Finally, we package everything up:

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

let prolog rules goal =
  try let insts = expand 0 rules [] [goal] in
      Yes(filter (fun (v,_) -> occurs_in v goal)
                 insts)
  with 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)

```
#let 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).";;
rules : (term * term list) list =
 [`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)`]]
#prolog rules ("sister_of(alice,edward)");;
- : outcome = Yes []
#prolog rules (parse_term "sister_of(alice,X)");;
- : outcome = Yes ["X", `edward`]
#prolog rules (parse_term "sister_of(X,Y)");;
- : outcome = Yes ["Y", `edward`; "X", `alice`]
```

Prolog examples (2)

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

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)");;
```

Prolog-style theorem proving

With a few minor changes, Prolog-style search can be used for general theorem proving.

Examples of such provers include PTTP (Stickel, 1988) and leanTAP (Beckert and Possega 1994).

Unification is an efficient method for deciding how to specialize universally quantified variables (Prawitz, Robinson 1960).

For example, given the facts that $\forall X.\, p(X) \Rightarrow q(X)$ and $p(f(a))$, we can unify the two expressions involving p and thus discover that we need to set X to $f(a)$. By contrast, the very earliest theorem provers tried all possible terms!

Usually, depth-first search would go into an infinite loop, so we need to modify the Prolog strategy slightly. We will use *depth first iterative deepening*.

Manipulating formulas

We will simply use our terms to denote formulas, introducing new constants for the logical operators. Many of these are written infix.

Operator	Meaning
~(p)	not p
p & q	p and q
p | q	p or q
p --> q	p implies q
p <-> q	p if and only if q
forall(X,p)	for all X, p
exists(X,p)	there exists an X such that p

An alternative would be to introduce a separate type of formulas, but this would require separate parsing and printing support. We will avoid this, for the sake of simplicity.

Preprocessing formulas

It's convenient if the main part of the prover need not cope with implications and 'if and only if's. Therefore we first define a function that eliminates these in favour of the other connectives.

```
let rec proc tm =
  match tm with
    Fn("~",[t]) -> Fn("~",[proc t])
  | Fn("&",[t1; t2]) -> Fn("&",[proc t1; proc t2])
  | Fn("|",[t1; t2]) -> Fn("|",[proc t1; proc t2])
  | Fn("-->",[t1; t2]) ->
        proc (Fn("|",[Fn("~",[t1]); t2]))
  | Fn("<->",[t1; t2]) ->
        proc (Fn("&",[Fn("-->",[t1; t2]);
                      Fn("-->",[t2; t1])]))
  | Fn("forall",[x; t]) -> Fn("forall",[x; proc t])
  | Fn("exists",[x; t]) -> Fn("exists",[x; proc t])
  | t -> t;;
```

The next step is to push the negations down the formula, putting it into so-called 'negation normal form' (NNF).

NNF code

We define two mutually recursive functions that create NNF for a formula, and its negation.

```
let rec nnf_p tm =
  match tm with
    Fn("~",[t]) -> nnf_n t
  | Fn("&",[t1; t2]) -> Fn("&",[nnf_p t1; nnf_p t2])
  | Fn("|",[t1; t2]) -> Fn("|",[nnf_p t1; nnf_p t2])
  | Fn("forall",[x; t]) -> Fn("forall",[x; nnf_p t])
  | Fn("exists",[x; t]) -> Fn("exists",[x; nnf_p t])
  | t -> t

and nnf_n tm =
  match tm with
    Fn("~",[t]) -> nnf_p t
  | Fn("&",[t1; t2]) -> Fn("|",[nnf_n t1; nnf_n t2])
  | Fn("|",[t1; t2]) -> Fn("&",[nnf_n t1; nnf_n t2])
  | Fn("forall",[x; t]) -> Fn("exists",[x; nnf_n t])
  | Fn("exists",[x; t]) -> Fn("forall",[x; nnf_n t])
  | t -> Fn("~",[t]);;
```

We will convert the negation of the input formula into negation normal form, and the main prover will then try to derive a contradiction from it.

The main prover

At each stage, the prover has a current formula, a list of formulas yet to be considered, and a list of literals. It is trying to reach a contradiction.

If the current formula is p & q, then consider p and q separately, i.e. make p the current formula and add q to the formulas to be considered.

If the current formula is p | q, then try to get a contradiction from p and then from q.

If the current formula is forall(X,p), invent a new variable to replace X: the right value can be discovered later by unification.

If the current formula is exists(X,p), invent a new constant to replace X.

Otherwise, the formula must be a literal, so try to unify it with a contradictory literal.

If this fails, add it to the list of literals, and proceed with the next formula.

Continuations

We desire a similar backtracking strategy to Prolog: only if the current instantiations allow all remaining goals to be solved do we accept it.

We could use lists again, but instead we use *continuations*. A continuation is a function that is passed to another function and can be called from within it to ‘perform the rest of the computation’.

In our case, it takes a list of instantiations and tries to solve the remaining goals under these instantiations.

Thus, rather than explicitly trying to solve all remaining goals, we simply try calling the continuation.

Prover code

```
let rec prove fm unexp pl nl n cont i =
  if n < 0 then raise (error "No proof") else
  match fm with
    Fn("&",[p;q]) ->
        prove p (q::unexp) pl nl n cont i
  | Fn("|",[p;q]) ->
        prove p unexp pl nl n
        (prove q unexp pl nl n cont) i
  | Fn("forall",[Var x; p]) ->
        let v = mkvar() in
        prove (subst [x,Var v] p) (unexp@[fm])
              pl nl (n - 1) cont i
  | Fn("exists",[Var x; p]) ->
        let v = mkvar() in prove(subst [x,Fn(v,[])] p)
                                unexp pl nl (n - 1) cont i
  | Fn("~",[t]) ->
        (try first (fun t' -> let i' = unify t t' i in
                                   cont i') pl
         with error _ -> prove (hd unexp) (tl unexp)
                                    pl (t::nl) n cont i)
  | t ->
        (try first (fun t' -> let i' = unify t t' i in
                                   cont i') nl
         with error _ -> prove (hd unexp) (tl unexp)
                                    (t::pl) nl n cont i);;
```

Prover examples (1)

We set up the final prover as follows:

```
let prover =
  let rec prove_iter n t =
    try let insts = prove t [] [] [] n I [] in
        let globinsts = filter
          (fun (v,_) -> occurs_in v t) insts in
        n,globinsts
    with error _ -> prove_iter (n + 1) t in
  fun t -> prove_iter 0 (nnf_n(proc(parse_term t)));;
```

It tries to find the proof with the fewest universal instantiations. It returns the number required and any toplevel instantiations.

```
#let P1 = prover "p --> q <-> ~(q) --> ~(p)";;
P1 : int * (string * term) list = 0, []
#let P13 = prover "p | q & r <-> (p | q) & (p | r)";;
P13 : int * (string * term) list = 0, []
#let P16 = prover "(p --> q) | (q --> p)";;
P16 : int * (string * term) list = 0, []
#let P18 = prover "exists(Y,forall(X,p(Y)-->p(X)))";;
P18 : int * (string * term) list = 2, []
#let P19 = prover "exists(X,forall(Y,forall(Z,
                (p(Y)-->q(Z))-->p(X)-->q(X))))";;
P19 : int * (string * term) list = 6, []
```

Prover examples (2)

A bigger example is the following:

```
#let P55 = prover
  "lives(agatha) & lives(butler) & lives(charles) &
   (killed(agatha,agatha) | killed(butler,agatha) |
    killed(charles,agatha)) &
   (forall(X,forall(Y,
     killed(X,Y) --> hates(X,Y) & ~(richer(X,Y))))) &
   (forall(X,hates(agatha,X)
             --> ~(hates(charles,X)))) &
   (hates(agatha,agatha) & hates(agatha,charles)) &
   (forall(X,lives(X) & ~(richer(X,agatha))
             --> hates(butler,X))) &
   (forall(X,hates(agatha,X) --> hates(butler,X))) &
   (forall(X,~(hates(X,agatha)) | ~(hates(X,butler))
             | ~(hates(X,charles))))
   --> killed(agatha,agatha)";;
P55 : int * (string * term) list = 6, []
```

In fact, the prover can work out 'whodunit':

```
#let P55' = prover
  "......
   --> killed(X,agatha)";;
P55' : int * (string * term) list = 6, ["X", `agatha`]
```