Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 12

ML examples IV: Theorem Proving

Topics covered:

- A simple theorem prover
- Theorem proving examples
- Potential and limitations

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 leanT^AP (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.

```
fun proc tm =
  case tm of
    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.

```
fun nnf_p tm =
  case tm of
    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 =
 case tm of
    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

```
fun prove fm unexp pl nl n cont i =
  if n < 0 then raise (error "No proof") else
  case fm of
    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 val v = mkvar()
         in prove (subst [(x,Var v)] p) (unexp@[fm])
                  pl nl (n - 1) cont i end
  | Fn("exists",[Var x, p]) =>
        let val v = mkvar()
         in prove(subst [(x,Fn(v,[]))] p)
                  unexp pl nl (n - 1) cont i end
  | Fn("~",[t]) =>
        (first (fn t' => let val i' = unify t t' i
                         in cont i' end) pl
         handle error _ => prove (hd unexp) (tl unexp)
                           pl (t::nl) n cont i)
  | t => (first (fn t' => let val i' = unify t t' i
                           in cont i' end) nl
         handle error _ => prove (hd unexp) (tl unexp)
                             (t::pl) nl n cont i);
```

Prover examples (1)

We set up the final prover as follows:

```
val prover =
  let fun prove_iter n t =
       (let val insts = prove t [] [] [] n I []
            val  globinsts = filter
             (fn (v,_) => occurs_in v t) insts
        in (n,globinsts) end)
       handle error _ => prove_iter (n + 1) t
   in prove_iter 0 o nnf_n o proc o parse_term end;
```

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

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

Prover examples (2)

A bigger example is the following:

```
- val 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)";
> val P55 = (6, []) : int * (string * term) list
```

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

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

How powerful is the prover?

The above prover is reasonably simple, and is complete for first order logic. This means that if a term is logically valid, it can, in principle, prove it. You can try the prover for yourself:

```
- use "/home/jrh13/examples/prover.ml";
```

If a term isn't valid, it may fail, or may go into an infinite loop, e.g.

```
- prover "p --> q";
! Uncaught exception:
! Match
- prover "exists(X,p(X)) --> forall(X,p(X))";
! Uncaught exception:
! Match
- prover
   "(forall(X,p(f(X)) --> p(X)))      \
   \ --> forall(X,p(X))";
<loop>
```

Can we do better?

It's possible to make provers that are much more effective, using less naive search strategies. Nevertheless there are important theoretical and practical limitations on what can be done.

It follows from work of Church and Turing that there is no decision procedure for first order logic, only a semi-decision procedure like this one.

Moreover, from results of Gödel and Tarski, some areas of mathematics such as number theory with addition and multiplication can't be computably axiomatized in first order logic, so for general mathematics we don't even have a semidecision procedure.

Finally, even though most of present-day mathematics *can* be derived in principle from a simple set of axioms, provers like the above are far too inefficient in practice to prove many interesting theorems.

Problems with the prover

In practice it takes far too long for difficult theorems. So we'd like to have more interactive control over the prover, so we can guide it, e.g. help it to use lemmas.

Another difficulty is that the code might have bugs, causing the prover to 'prove' something that isn't true. If it's being used as a check on human reasoning, that's a bad idea.

It was in attempting to solve these and similar problems with existing provers that Milner and his co-workers developed 'Edinburgh LCF' and designed the original ML language.