A small LCF-style proof assistant for minimal propositional logic, in OCaml

Kelley van Evert, s4046854, IC

May 19, 2013

This document describes a small LCF-style proof assistant for minimal propositional logic, which I wrote for a course on proof assistants. It is written in OCaml (though without the use of the "O" in OCaml), and exemplifies the method of representing the logical state of a proof session as a theorem:

```
  type goalstate = goal list * theorem
```

... as opposed to representing it as a justification function, which reconstructs a proof for the original goal from the proofs for the various smaller goals that it was decomposed to by means of tactics:

```
  type goalstate = goal list * (theorem list -> theorem)
```

1 Kernel

The kernel, as a module, provides the `formula`, `theorem` and `goal` types, and the three basic inference rules to construct theorems. The `theorem` type is private, so that theorems cannot be constructed outside the module, and therefore only by means of the three basic inference rules. Thus objects of type `theorem` can be regarded as proofs, although they don't carry along this proof (hence the type constructor is named `Provable`).

```
module type KERNEL =
  sig
    type formula = Var of int | Imp of formula * formula
    type sequent = (formula list) * formula
    type theorem = private Provable of sequent
    type goal = Goal of sequent

    exception RuleException of string

    val assume : formula -> theorem
    val intro_rule : formula -> theorem -> theorem
    val elim_rule : theorem -> theorem -> theorem
  end;;
```

2 Tactics, goals and session state

A tactic is a function which breaks a goal into smaller pieces, but must also provide justification for this decomposition by providing a function that reconstructs a proof of the original goal from a set of proofs for the smaller goals.

```
type justification = theorem list -> theorem
type tactic = goal -> (goal list * justification)
```

Each logical inference rule has a corresponding tactic which applies the rule:

```
val assumption : tactic
val intro_tac : tactic
val elim_tac : formula -> tactic
```

And furthermore I provided two "tacticals", which take a tactic as argument, and return a new tactic that modified the behaviour of the given tactic slightly (in obvious ways).

```
val try_tac : tactic -> tactic
val repeat_tac : tactic -> tactic
```

As the user uses the proof assistant, conceptually working up the natural deduction tree starting from the initial goal, the session state is stored as an object with type `goalstate`. The easy way to represent this session, and thus to define this type, is to have it be a tuple of the list of openstanding goals, and a justification function. A completed proof would have no openstanding goals, and the justification function, when applied to the empty list, would generate the desired theorem. Note that this definition of the `goalstate` type corresponds exactly to the return type of a `tactic`.

A problem with this approach is that the justification functions of the various tactics used in a proof session are only used at the end of the session, when the proof has been completed. Therefore, it is hard to debug these functions, seen as they're only run once, and within a whole framework of others, at the same time. Mind that in a system like this, tactics are manageable things, that one can create "on the fly" as one sees fit; so that this really is a problem. One solution to this problem is for the system to simply "test" the function with various inputs at the moment it is supplied (this is what, for instance, the HOL Light system does).

Another solution is to represent the session state differently. The idea is to represent the logical state of the session by a theorem, which basically states that "the initial goal is provable by the currently openstanding goals", that is:

$$\ulcorner\Gamma_0 \vdash^? \phi_0\urcorner, \ulcorner\Gamma_1 \vdash^? \phi_1\urcorner, \ldots \vdash \psi$$

... in which ψ is the initial goal, and each $\ulcorner\Gamma_i \vdash^? \phi_i\urcorner$ is some encoding of the i-th openstanding goal. Thus we have the type definition:

```
  type goalstate = goal list * theorem
```

This approach requires some “logical trickery”, however, because of the fact that we cannot (mustn’t) circumpass the kernel in creating our theorems.

2.1 Encoding goals

If $g = \Gamma \vdash^? \phi$, then $\psi_g = \Gamma \Rightarrow \phi$ is the formula encoding of this goal (a nested implication containing as premises the various formulas in the environment of the goal). If gs is a list of goals, then we write Γ_{gs} for the list of encoded goals.

We also associate a theorem to each goal $g = \Gamma \vdash^? \phi$, by the function `goal_to_theorem`:

$$\texttt{goal\_to\_theorem}(\Gamma \vdash^? \phi) = \dfrac{}{\psi_g, \Gamma \vdash \phi}\,\texttt{g2t}$$

As an example, if $g = \Gamma \vdash^? \phi$ and $\Gamma = A, B, C$, then $\psi_g = \Gamma \Rightarrow \phi = A \Rightarrow B \Rightarrow C \Rightarrow \phi$, and the corresponding theorem $\texttt{goal\_to\_theorem(g)} = \psi_g, \Gamma \vdash \phi$ is obtained as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\psi_g, \Gamma \vdash \psi_g} \quad \dfrac{}{\psi_g, \Gamma \vdash A}}{\psi_g, \Gamma \vdash B \Rightarrow C \Rightarrow \phi} \quad \dfrac{}{\psi_g, \Gamma \vdash B}}{\psi_g, \Gamma \vdash C \Rightarrow \phi} \quad \dfrac{}{\psi_g, \Gamma \vdash C}}{\psi_g, \Gamma \vdash \phi}$$

The code for this function is as follows:

```
let goal_to_theorem (Goal (gamma, b)) : theorem =
  List.fold_left
    (fun th phi -> elim_rule th (assume phi))
    (assume (collapse_formula_list (b :: gamma)))
    gamma;;
```

2.2 Applying tactics

Now we are ready to define the function `by`, which takes a tactic and a goalstate, applies the tactic to the first openstanding goal in the goalstate, combines the results and returns a new goalstate.

```
  by : tactic -> goalstate -> goalstate
```

Let `tac` be the given tactic, and `s = (g :: gs1, th)` the given goalstate, in which $g = \Gamma \vdash^? \phi$ and `th` is the theorem $\psi_g, \Gamma_{gs1} \vdash \psi$. We run the tactic on the first openstanding goal, g, and get `(gs2, j) = tac g`, and let’s say that `gs2 = [h0, ... hN]`. The key part is how to construct the new state theorem, which should be $\Gamma_{gs2}, \Gamma_{gs1} \vdash \psi$, so that we can return the new goalstate `s' = (gs2 @ gs1,` $\Gamma_{gs2}, \Gamma_{gs1} \vdash \psi$`)`.

Essentially, we want to do a simple modus ponens on the formula encodings of the various goals involved, in which $\Gamma_{gs2} \vdash \psi_g$ should follow from our justification function `j`:

$$\dfrac{\dfrac{\dfrac{}{\psi_{\mathtt{g}}, \Gamma_{\mathtt{gs1}} \vdash \psi}}{\Gamma_{\mathtt{gs1}} \vdash \psi_{\mathtt{g}} \Rightarrow \psi} \qquad \Gamma_{\mathtt{gs2}} \vdash \psi_{\mathtt{g}}}{\Gamma_{\mathtt{gs2}}, \Gamma_{\mathtt{gs1}} \vdash \psi}$$

The tricky part lies in constructing this $\Gamma_{\mathtt{gs2}} \vdash \psi_{\mathtt{g}}$. Provided we have proofs for the goals of `gs1`, the justification function will allow us to construct:

$$\dfrac{\Gamma_{\mathtt{h0}} \vdash \phi_{\mathtt{h0}} \qquad \dots \qquad \Gamma_{\mathtt{hN}} \vdash \phi_{\mathtt{hN}}}{\Gamma \vdash \phi}\,\mathtt{j}$$

We don't have these proofs just yet, but instead apply `j` to the theorem encodings of the goals (note that $\Gamma_{\mathtt{gs2}} = \psi_{\mathtt{h0}}, \dots \psi_{\mathtt{hN}}$); after which we contract g's environment again to get the desired result (note that $\psi_{\mathtt{g}} = \Gamma \Rightarrow \phi$):

$$\dfrac{\dfrac{\dfrac{}{\psi_{\mathtt{h0}}, \Gamma_{\mathtt{h0}} \vdash \phi_{\mathtt{h0}}}\,\mathtt{g2t} \qquad \dots \qquad \dfrac{}{\psi_{\mathtt{hN}}, \Gamma_{\mathtt{hN}} \vdash \phi_{\mathtt{hN}}}\,\mathtt{g2t}}{\Gamma_{\mathtt{gs2}}, \Gamma \vdash \phi}\,\mathtt{j}}{\Gamma_{\mathtt{gs2}} \vdash \psi_{\mathtt{g}}}\text{(contracting g's environment)}$$

Plugging this last theorem into the simple modus ponens above, we get a full proof of the theorem $\Gamma_{\mathtt{gs2}}, \Gamma_{\mathtt{gs1}} \vdash \psi$, and we are done. The code for all this is:

```
let by (tac : tactic) (goals, th : goalstate) : goalstate =
  match goals, th with
  | g :: gs1, Provable (psi_g :: _, _) ->
    let (gs2, j) = tac g in
    let j_th = j (List.map goal_to_theorem gs2) in
    (* Contract g's environment: *)
    let j_th' = List.fold_right intro_rule
      (List.rev (diff psi_g (conclusion j_th))) j_th
    in
              (* Simple modus ponens on the various goals: *)
      gs2 @ gs1, elim_rule (intro_rule psi_g th) j_th'
  | _, _ -> raise (TacticException "There must be an open goal");;
```

3 Stateful proof environment

Finally, there is a simple stateful proof environment, in which we have various functions for manipulating the session state:

```
val p : unit -> unit    (* Prints the current goalstate *)
val g : formula -> unit (* Starts a new proof session for "|- formula" *)
val e : tactic -> unit  (* Applies the given tactic to the current goal *)
let b : unit -> unit    (* Go back in history, one step *)
```

```
val top_theorem : unit -> theorem (* Returns the built theorem, after the
                                     proof has been completed *)
```

4 Example proof session

```
g (formula ”A => (B => C) => (A => B) => C”);;
e (repeat_tac intro_tac);;
e (elim_tac (formula ”B”));;
e assumption;;
e (elim_tac (formula ”A”));;
e assumption;;
e assumption;;
print_theorem (top_theorem ());;
```