Second exercise for the course 'proof assistants'

The exercise is to implement a very small LCF-style prover for minimal propositional logic. Minimal propositional logic is the logic that only has *one* connective, implication, and that only has three inference rules:

$$\frac{}{\Gamma \cup \{A\} \vdash A} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

For an example, here is a very small proof in this logic:

$$\frac{\dfrac{}{a \vdash a}}{\vdash a \to a}$$

In this proof a is a free propositional variable.

Although these three rules are the customary natural deduction rules for minimal propositional logic, in order implement it in the style of HOL it is easier to replace them with the inference rules:

$$\frac{}{A \vdash A} \qquad \frac{\Gamma \vdash B}{\Gamma - \{A\} \vdash A \to B} \qquad \frac{\Gamma \vdash A \to B \quad \Delta \vdash A}{\Gamma \cup \Delta \vdash B}$$

These alternative rules allow one to deduce the same sequents as the original rules did.

The exercise is to write a short ML program that contains both a 'LCF kernel' for this logic, as well as a 'subgoal package' for it that allows one to interactively prove propositions by applying tactics to a goal. Please submit your program as one file, that for instance could be called '`prop.ml`'. You can interactively load this file in `ocaml` by issuing the command '`#use "prop.ml";;`' at the interpreter prompt.

The deadline for this exercise is **June 25**. As a half-way checkpoint, at **June 4** a version of the kernel of the prover has to be handed in.

*

Here is a step-by-step guide on how one might go about doing this exercise. It is not at all required to follow this in detail.

1. Implement a *logic kernel* that defines types

   ```
   form
   thm
   ```

   and functions

   ```
   assume     : form -> thm
   intro_rule : form -> thm -> thm
   elim_rule  : thm -> thm -> thm
   ```

that correspond to the inferences rules of the logic.

You are now at the halfway checkpoint of the exercise.

2. Implement a package for *goal-directed proof* that defines types

```
goal
goalstate
tactic
```

and a function

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

This '`by`' function applies the tactic to the first goal in the goalstate.

3. Implement *tactics* that correspond to the rules from the kernel:

```
assumption : tactic
intro_tac  : tactic
elim_tac   : form -> tactic
```

4. Implement interactive *commands* for doing goal-directed proofs:

```
g           : form -> goal
p           : unit -> goal
e           : tactic -> goal
b           : unit -> goal
```

These commands respectively set the current goal, return the current goal (= the first goal in the most recent goalstate in the proof history), apply a tactic to the current goal, and undo the last tactic that was applied. They use a global variable

```
history     : goalstate list ref
```

to keep the state of the proof, and raise an exception

```
QED
```

when there is no subgoal left to return. Make each command run `p()` at the end, so that each command will return the current goal.

5. Implement a function

```
top_thm     : unit -> thm
```

that returns the `thm` when an interactive proof is finished by running the justification function that has been constructed. This only needs to work if no goal is left.

To give an impression of the complexity of this exercise: my own solution along these lines consisted of 125 lines of ML code.

*

There are various things that you can do to make this exercise more work. They are not required to get full marks, but they might be fun working on.

– First of all, there are three different ways to implement tactics in an LCF-style:

(i) You can use the normal LCF style where one has

```
goal       = form list * form
goalstate  = goal list * (thm list -> thm)
tactic     = goal -> goalstate
```

This is the easiest approach.

(ii) You can follow HOL Light and instead use goals that look like

```
goal       = thm list * form
```

so where the assumptions in the goals are represented by `thm`s instead of by `form`s.

(iii) If you are really courageous you can try to follow the approach from ProofPower and Isabelle and use

```
goalstate  = goal list * thm
```

instead of `goalstate`s in the normal LCF style.
In message `<200601051723.25137.rda@lemma-one.com>` to the `hol-info` mailing list, Rob Arthan described this approach in the following way:

> *I can thoroughly recommend the way Kevin Blackburn implemented the subgoal package in ProofPower based on ideas of Roger Jones. The logical state of the proof is represented by a theorem whose conclusion represents the original goal and whose assumptions represents the outstanding subgoals (a sequent being essentially represented by a universally closed implication).*
>
> *Tactics have the traditional type* `(GOAL -> GOAL list * (THM list -> THM))`*. All useful tactics return justifications (the* `THM list -> THM` *bit) that are insensitive to the presence of additional assumptions in the theorems in the argument list. So with some very simple logical trickery, the subgoal package can arrange to apply the justification immediately and use it to replace the assumption in the goal-state theorem that represents the goal by new assumptions representing the new subgoals.*
>
> *This approach neatly solves some classic problems (the justifications are tested immediately and the theorem produced at the end is guaranteed to be identical with the original goal). It does theoretically restrict what the tactics can do (e.g., as regards type instantiation), but this is not a problem for any tactic I have ever seen.*

This third approach does not take too much extra code, but it requires some hard thinking about the 'logical trickery' that is needed to make this work. (I would really like it if someone would try to implement this variant of doing a subgoal package for the exercise.)

- For the exercise it is not required to implement parsers or pretty-printers. However, it would be nicer to be able to type

  ```
  form "(a -> b) -> c"
  ```

  (where the function `form` has type '`string -> form`') than to have to type something like

  ```
  Imp (Imp (Var "a", Var "b"), Var "c")
  ```

- For the exercise it is not required to turn the types `form` and `thm` into '`private`' types by putting them inside a module like it is done in HOL Light. However, doing this would clearly delineate the kernel, and would fully protect the logic against unsafe ML programming.
- An interesting modification would be to modify the type `thm` to have it contain a lambda-term that represents the proof as a *proof object*.
- One could add *labels* to the goals, and then implement a tactic that corresponds to Coq's '`apply`' tactic.
- Finally it might be interesting to implement a rule and a corresponding tactic that proves arbitrary propositions fully automatically, like HOL Light's '`TAUT`' or Coq's '`tauto`' tactic.