Designing a state transaction machine for Coq

Bruno Barras bruno.barras@inria.fr
Enrico Tassi enrico.tassi@inria.fr

In its earlier versions, the interaction between the user and Coq was through a simple read-eval-print loop. Upon completing a proof, the user would cut and paste proof scripts to a vernacular file. The second generation of interfaces, including Proof General and CoqIde, consists in linking Coq with a text editor. Buffers contain both declarations that have been checked and others that are planned to be submitted to Coq. The state of the prover is reflected by the regions that have been accepted, which are locked.

This lock raises the question of how such regions can be modified. Typically, the user has to unlock the region from the point where edits have to be done. Once the edits are done, he has to replay the previously unlocked region. This disturbs the editing flow in two respects: (1) the user has to wait that all the previously checked proofs are still valid, and (2) if one proof is broken, the user must fix the problem before he can resume working.

This is even more awkward when the change only consists in moving declarations to a more appropriate place. The penalty of waiting sometimes deter the user from reorganizing his development.

This work is part of a bigger project, Paral-ITP[1], that aims at providing a new form of interaction. Starting from of Makarius Wenzel's jEdit/scala interface with Isabelle [1], a new way of interacting with theorem provers should be introduced.

The main novelty for the user is the absence of a locked region. The prover is supposed to be sufficiently reactive to cope with the flow of edits issued by the user. This requires to have a more refined management of the state by distinguishing tasks (or transactions) that are required for the user can continue working (e.g. definitions) and those that can be delayed (e.g. opaque proofs).

Here we focus on the design of the state manager and the analysis of tasks that can be performed independently. We hope to get some early feedback from the user community on the design. What is described here is being implemented, still at the level of a prototype.

Design We should analyze the dependencies between the tasks to be performed, in order to reconstruct as quickly as possible the state corresponding to the point he is working on, after edits of another part of the theory. Tasks that are not directly needed for this purpose are postponed. We may consider various modalities under which these tasks are eventually executed: either in parallel, or on demand (lazily, this is what is currently being implemented).

For instance, the proof of a lemma is generally not relevant for the rest of the development. This means that re-checking the proof and proceeding to the rest of the development with the lemma assumed are independent tasks. It is common that type-checking definitions and lemma statements are not demanding huge amounts of resources, so we can expect to have a reasonable reactivity.

Errors occuring in the postponed tasks should also be reported to the user, in an asynchronous manner. But he is free to respond to them when he is ready.

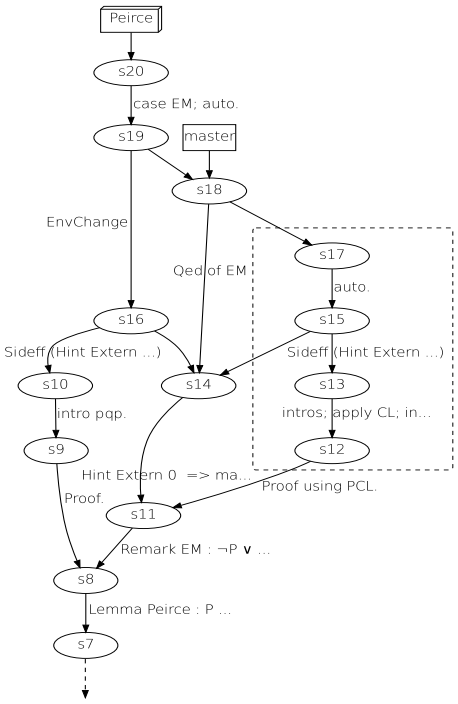
To identify parts of the script that can be executed in parallel we represent the parsed document with a DAG. Each branch correspond to independent sequences of tasks. All branches are eventually merged to form a state where all the tasks are executed.

The reactivity of the system is improved by dissociating the construction of the DAG from the linear vernacular file and the actual interpretation of the commands. Each edit requires rebuilding (at least partially) the DAG of states, but they may occur at a rate that does not allow the full computation of the final state.

[1] Paral-ITP: ANR-11-INSE-001 `http://paral-itp.lri.fr/`

Implementation To illustrate the state management, we consider the following proof script, involving the simultaneous construction of several proofs.

```
1  Require Import Unicode.Utf8.
   Definition F := False.
3  Section Ex.
   Variable P Q : Prop.
5  Hypothesis CL : ∀X Y, (X →¬Y →F) →(¬X ∨Y).
   Lemma Peirce : ((P →Q) →P) →P.
7  Proof.
   intro pqp.
9    Remark EM : ¬P ∨P.
     Proof using P CL.
11   apply CL; intros p np; auto.
     Hint Extern 0 =>
13     match goal with p : P, np : ¬P |- _ => case (np p) end.
     auto.
15   Qed.
     case EM; auto.
17 (* Qed.
   End Ex. *)
```



Nodes represent states of the prover and edges represent transactions, i.e. commands to transform one prover state into another. Since we want to decouple the parsing and execution phases, we need to be able to build such a data structure without actually executing the transactions. Commands need to be classified according to their effect on the state DAG.

This is in general not possible, since some commands affect the parsing of the lines that follow extending the grammar (like `Notation`) or changing the set of active grammar rules (like `Open Scope`). Fortunately this category of commands is limited and it is possible to put all of them in the script preamble. For example notations can be declared in two phases: the one actually extending the grammar (with `Reserve Notation`) and the one giving the interpretation. This alleviates the problem forcing the execution of the script preamble only.

Another category of commands is the one associated with the creation of a branch. A branch represent a computation that can be performed in parallel. Examples are `Lemma` (line 6) and `Remark` (line 9). The former transaction, leading to state `s8`, starts the branch on the left in the picture. This branch, that is not closed yet since line 17 is commented, is pointed too by the `Peirce` head (in vcs terminology) represented by a shaded square box. The `Remark` command opens another branch, that is closed by the `Qed` command on line 15. The nodes of this last branch are grouped into a dashed box, representing a computation that cannot only be performed in parallel, but also delayed indefinitely. Thanks to the `using P CL` annotation at line 10 the Coq system can correctly discharge the type of `EM` at the end of the section without knowing its proof term. Without this piece of information the only way to obtain `s18` would be to fully evaluate the branch for `EM`, thus compute `s17`, `s15`, `s13` and `s12`.

The semantics attributed by Coq to the `Hint` command at line 12 is particularly tricky. The hint that makes `auto` at line 14 succeed (while it failed at line 11 without the hint) is globally added to the system, and line 16 is able to close the proof thanks to that. Since our objective is to actually execute the branches for `EM` and `Peirce` without interleaving constraints we have to make both branches aware of the global state change due to the `Hint` command. Indeed this command is not only recorded in the master branch (see `s14`) but also in all the branches that are open at line 12 (see states `s16` and `s15`).

The last subtle point worth noting is that `Qed` at line 15 does not only merge the branch for `EM` back into master, but also globally affects the environment of the prover. Indeed the reference to the remark `EM` at line 16 is valid, but it is not so at line 9 for example. The DAG structure records this global environment change with state `s19`.

References

[1] M. Wenzel. *Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit.* In User Interfaces for Theorem Provers (UITP 2010), ENTCS, 2010.