Undoing and Managing a proof

Olivier Pons*

INRIA Sophia Antipolis - BP 93
06902 Sophia Antipolis - France
Olivier.Pons@inria.fr

Abstract. In this paper we work in the area of proof script management. We study the dependencies between commands in a script and how they can be exploited in the undoing process. We present some problems that we have met in our implementation of proof managing tools in the Ctcoq interface for the Coq System.

Introduction

Developing a theory or proving a non-trivial theorem is not a linear and monotonic process. It is sometimes necessary to change and refine the context and the specification of the problem, and the proof investigation is done by trial and error and by multiple attempts. Often this task is not convenient and not easy with proof assistants based on tactics.

Most systems maintain a historical dependency between steps. This type of dependency is not sufficient and logical dependency is more useful. We have experimented and implemented some tools to help the user to understand the structure of proofs and to manage proofs using this type of dependency.

The whole implementation has been done in Ctcoq, a graphical user-interface [BBC+96] for the Coq proof assistant [CCF+95] based on the Centaur system [INR94].

1 Proof Structure

Like many other proof systems (Lego[LP92,Pol94], Alf[Mag95]) the Coq proof assistant is based on a higher order type theory (for Coq this is the Inductive Calculus of Construction). Following the Curry-Howard isomorphism, a formula (or a specification) is represented by a type, and a proof of this formula (or the program that realizes it) by a λ-term of this type. Proving a formula consists in giving a term whose type is the formula.

For example, in Coq, we can express the associativity of addition with the following lemma. The notation `(n,m,p:nat)` must be read $\forall n, m, p.nat$ and `[ ]` is a notation for lambda abstraction.

* This work was supported by the CNET (contract number 1 96 A151 00)

```
Lemma plus_assoc_l : (n,m,p:nat)((plus n (plus m p))=(plus (plus n m) p)).

proof:
[n,m,p:nat]
 (nat_ind [n0:nat](plus n0 (plus m p))=(plus (plus n0 m) p)
   (refl_equal nat (plus m p))
   [n0:nat]
    [H:(plus n0 (plus m p))=(plus (plus n0 m) p)]
     (eq_ind_r nat (plus (plus n0 m) p)
       [n1:nat](S n1)=(S (plus (plus n0 m) p))
       (refl_equal nat (S (plus (plus n0 m) p))) (plus n0 (plus m p)) H)
   n)
```

1.1 The proof script

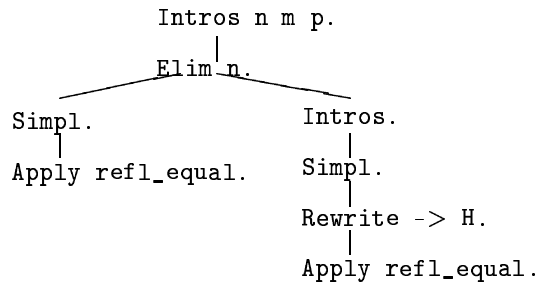
Constructing a proof term is a difficult task. In fact, the proof term is usually not handled directly. To build a proof term, we use a goal directed transition engine. This means that we state the goal to prove and we transform this formula, step by step, by applying commands that break it down in a number of subgoals until we reach a final state with no more goals. The commands used in this process are called **tactics**. The list of tactics, called the **proof script** can be edited directly by the user or generated by mechanism like proof-by-pointing or drag-and-drop[BKT94].

This script has a linear structure. But behind this, there is an arborescent structure where each node is a tactic and the children nodes are the tactics applied on the subgoals generated by the parent tactic. For example, the proof term given above corresponds to the script and the tree below:

```
Intros n m p.
Elim n.

Simpl.
Apply refl_equal.

Intros.
Simpl.
Rewrite -> H.
Apply refl_equal.
```



1.2 Navigating in a proof script

The first step towards understanding the proof is understanding its tree structure. We have implemented tools to navigate in this tree structure. With our tools, by selecting a tactic anywhere in the script, we can move up, down, and transversally according to the tree structure just by a key-strokes. For example in the script above selecting the tactic `intros` and striking `C-c C-U` we move up the selection on the tactic `Elim n`. The implementation of these facilities is described in section 2.

1.3 Undoing: Coherence of the remaining script

Generally, during goal-directed proofs, there are several goals that can be attacked indifferently. Processing a given goal is performed by combining the tactic with a numeric prefix referring to the goal's rank (when no number is provided the default value is 1). Pruning a proof branch may affect the position of goals and the numeric prefixes have to be updated.

This problem can be divided in two subtasks. First we want to be sure that the part of the script that is not undone is still valid (that is, if we replay it we will get the same result), secondly we want to be able to replay the undone part possibly after modification. Figure 1 show the undoing mechanisms.

Coherence of the preserved script The problem is to determine the lines where a numeric prefix change is needed. All the tactics in the script that are above the head of the branch to prune can stay unchanged. For the others, tactics that are "before" the branch head (with a lexicographic order on the path set) in the tactics tree can also stay unchanged. Tactics that are behind must have their numeric prefix decremented by $(n-1)$ where n is the number of open subgoals in the pruned branch at that point.

Here is the algorithm, where `T` is the undone branch head, `(number t)` is the numeric prefix number of tactic `t`, `(rank t)` is the position of `t` in the script and `(path t)` is the path of `t` in the proof tree. The order "<" on paths is simply a lexicographic order.

```
Update-preserved-part T
-----------------------
  shift := number-of-sub-goals T
  t        := t-initial
 WHILE t IN played-script  DO
     IF  rank (t) > rank (T)
       THEN  IF path (t) > path (T)
                 THEN IF  (path T) is-prefix-of? (path t)
                         THEN (shift := shift + (number-of-sub-goals t) -1)
                         ELSE (number t) <- (number t) - shift
                      END IF
       END IF
 (next t)
 END OF WHILE
```

Coherence of the undone part To produce a replayable script for the undone branch, we also need to update it. Here again, the update contains two phases. The first phase computes the rank of the goal attacked when pushing the undone script at the end of the preserved script. The second phase simply reorganizes the undone branch.

To compute the numeric prefix of the branch head, the algorithm first traverses the preserved part to determine the number of goals that will still be open when the preserved part is replayed. For each tactic whose path is smaller than

the one of the branch head in the lexicographic order, we compute the number of sub-goals whose path is smaller than the head path.

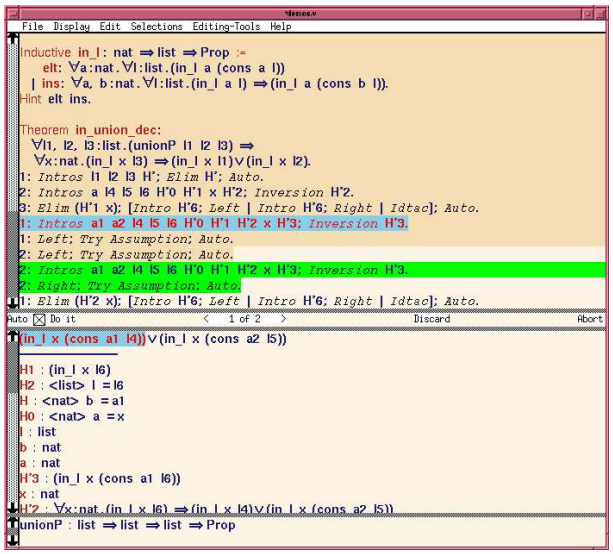
```
head-prefix T
-----------------
  result := 1
  t := t_initial
  WHILE t IN kept-script DO
     IF path (t) < path (T)
     THEN result := result - 1 +
       (number-of-subgoals-whose-path-is-smaller-than
           t path(T))
     END IF
  (next t)
  END OF WHILE
  (number T) <- result
```

The script corresponding to the undone branch is then sorted with an arbitrary sorting algorithm, using the lexicographic order on paths to compare two elements. The last step updates the numeric prefixes of tactics, taking into account the unsolved goals. This algorithm traverses the list of tactics and compares the relative positions of successive tactics. Only two cases may occur, because the script is already sorted.

1. tactic `t` follows `t'` and `t` attacks a subgoal generated by `t'`.
2. tactic `t` follows `t'` and there exists a path `p` such that `(path t') = p@i@p'` and `(path t) = p@j`, where `i` and `j` are two integer numbers such that `i < j`

In the first case, only the subgoals of `t'` that have a path lexicographically smaller than the one of `t` must be taken into account. In the second case, it is also necessary to take into account the subgoals of the closest common ancestor to `t'` and `t`. The algorithm works as follows:

```
update-numeric-prefixes-of-branch T
-----------------------------------
number := (number T)
t' = T
t = (next T)
WHILE t IN moved-script DO
   IF (path t) = (path t')@i
   THEN number := number + i - 1
        (number t) <- number
        t := (next t); t' := (next t')
   ELSE let p,p',i,j be such that
              (path t)=p@j
              (path t'=p@i@p'
        in
          number := number + (number-of-subgoals t') + j - i - 1
        t := (next t); t' := (next t')
END OF WHILE
```



The two darkest lines have been undone, updated and moved at the end of the played script they can be modified and replayed.The preserved part have been uptadted. The other selected line in the script is the one which have generated the selected sub-goal.

Fig. 1. "The Ctcoq environment after a *Undo-Redo*"

2 Notes on implementation

2.1 Primitive data structures

We use arborescent data structures to represent most kinds of data that occur in this work. Tactics, scripts, goals, logical formulas, and paths are all arborescent data structures. The tree processor we use provides doubly linked trees, so that getting the parent node of a term takes a constant time. The children nodes of a term are stored in a list, so that getting a child of rank n can take a time linear in n.

Nodes can carry annotations that could be used by the system but are neither usable nor even visible by the user. In our experiment, all executed tactics are annotated with a path annotation (which denotes the position in the proof tree), an integer value that indicates the number of created subgoals, and a theorem name that indicates the proof this tactic is attached to.

Because some operations can take a linear time, we use hashtables to maintain the relation between a path and the executed tactic that carries this path. A proof script can contain several goal directed proofs and there is one different hashtable for each of these proofs. The various hashtables are themselves kept in an extra hashtable, using the theorem names as keys. This global hashtable is itself an annotation of the whole proof script.

Interaction with Coq The proof tree structure that we maintain is a mirror of a similar structure maintained by the proof system, and each goal received from it is annotated by its path. We have also added to the Coq system a command that undoes the subproof starting on a sub-goal referred to by its path.

On the side, the user-interface maintains a list of open goal paths. When applying a tactic, we know from its numeric prefix on what sub-goal it applies. The Coq proof system replies by sending only the new sub-goals. The paths of these sub-goals are inserted in the path list and the old path is used to annotate the command that has just been executed, along with the number of new sub-goals and the theorem name.

2.2 Memory management

Hashtable have a memory cost that must be taken care of when undo steps reset the proof system to a state where many proofs are forgotten. In our experiment, the undo mechanism traps the `Reset` events and checks that all the hashtables corresponding to undone proofs are removed from the global hashtable (the one attached to the proof script root).

Another feature present in our experiment is the possibility to work on several proofs at the same time, using different proof windows each carrying its own proof script. This possibility to carry several proofs at the same time interferes very little with our undo mechanism. However, a multiple window interface must provide possibilities to open and close windows at will. When a window is closed, it is very important that the memory corresponding to the hashtables attached to the proofs performed in that window be recovered. Our choice to attach hashtable to the root of proof scripts makes this completely transparent: when a window is closed, the proof script it contains is freed, and so are the annotations it carries.

3 Advanced Problems and future work

3.1 Existential variables

Our undoing mechanism supposes that all branches of a proof tree are independent. This is not always true. Existential variables introduce constraints. If a tactic generates more than one sub-goal with existential variables, the instantiation of one of these variables in one of the sub-goals introduces a constraint on the other sub-goals for the continuation of the proof.The above example illustrate this problem:

```
Lemma pbmeta : (a,b,c,d:nat)a=b->b=c->a=d->a=c.
Intros;EApply trans_equal.        (* this leads to two sub-goals *)
  H : a=b
  H0 : b=c
  H1 : a=d
  ============================
   a=?312
```

```
sub-goal 2 is:
 ?312=c

2:EAuto.                          (* This forces ?312 to be equal to b *)
EAuto.                            (* This finishes the proof. *)
```

Thus, the valid script representing is the following sequence:

```
Intros;EApply trans_equal.
2:EAuto.
EAuto.
```

On the other hand, the following script fails, although it would be equivalent if branches were independent. This script is the one that is obtained when cutting the branch `2:EAuto.` and replaying it.

```
Intros;EApply trans_equal.
EAuto.
EAuto.
```

This problem has also been studied in the Alf proof editor [Mag95]. The result is a calculus of branch cutting on incomplete typed proof terms. However, the solution proposed in that work does not apply here. The branches manipulated in the Alf proof editor are explicit proof terms. When cutting one of these subterms, the other explicit branches are still valid in the remaining context. In our case, the manipulated branches are tactic trees. When the context changes because of the cut of another branch, a tactic may behave differently. The previous example is characteristic of this feature: the tactic `EAuto` performs the right thing on the first branch if the second is already solved and the wrong thing otherwise.

Preliminary studies of this problem indicate that the dependencies described by the proof tree structure have to be enriched with chronological dependencies when existential variables occur. Intuitively, a tactic `t` depends on another tactic `t'` when the path of `t'` is a prefix of `t` (this is the regular case) or when `t` and `t'` work on goals related by the presence of existential variables and `t'` occurred before `t` chronologically. Hopefully, one should be able to retain the regular strategy as long as no existential variable appears in goals.

Conclusion

The implemented tools are a first overview of the possible manipulations on a proof script. They do not need a lot of information from the proof system. All this has been implemented using the Ctcoq interface but it could have been done using another interface supplying the basic data (tactics, paths, number of sub-goals generated by each tactic, possibility to have annotations, hashtables)

Compared with the "Undo" mechanism present in the Coq system, the mechanism described here has the advantage that it does not limit to a fixed value the number of proof steps that can be undone. All the proof steps can be undone

even if several hundreds of steps have been done after. On the other hand, all the implemented tools suppose a complete independence of two proof steps done on two sub-goals whose paths are not comparable by a prefix order. This propriety is not ensured in the Coq system where the use of "existential variables" and their instantiations can lead to dependencies not represented in the proof tree. In practice, commands that introduce such existential variables can not be undone, lest by aborting the whole proof. Fortunately, these commands are not often used.

The tools presented here work only inside a single goal directed proof. The next step is to supply tools to manage the dependencies between several proofs or several definitions (actually, a finished and saved proof is a particular case of definition). We have been studying the internal representation of definition to extract the interesting iformation. We envision two levels of dependency computation, one inside a proof, and another between several proofs and definitions. A synthesis of these two levels could supply tools to split big proofs into several lemmas, each of them using only a minimal subset of hypotheses.

Acknoledgements

I would thank Yves Bertot and Laurence Rideau for their help and the anonymous referees whose comments pointed out vague and confusing part of the paper.

References

[BBC+96] J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac. *User Guide to the CtCoq Proof Environment.* INRIA, Feb 1996.

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by Pointing. In *International Symposium on Theoretical Aspects of Computer Science*, 1994.

[CCF+95] C. Cornes, J. Courant, J.C. Filliatre, G. Huet, P Manoury, C. Munoz, C. Murthy, C. Parent C. Paulin-Mohring, A. Saibi, and B. Werner. *The COQ Proof assistant, Reference Manual, Version 5.10.* INRIA, Le Chesnay Cedex, France, July 1995.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[INR94] INRIA. *Centaur 2.0*, 1994.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[Mag95] Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.