

Proof engineering

Olivier Pons^{1,2}

¹ INRIA Sophia Antipolis

² IIE, CNAM

pons@cnam.fr

Abstract. The main purpose of this article is to present a panorama of tools for formal proof analysis and management. The proposed tools are based upon several notions of dependency. Our principal objective is to facilitate the development and the maintenance of theories in proofs assistants in order to improve the productivity of such systems' users.

Today, programming environments, control version system, debugging and software engineering tools allowing analysis and programs management are usually and intensively used in programming. It is unanimously recognized that they improve programmers comfort and productivity. On the other side, proof assistants such as Coq[INR96], HOL[GM93], PVS[SOR93], Nuprl[CAB⁺86] or Lego[LEG] are increasingly used and deserve similar tools to facilitate their use. Although still marginal at the moment, the use of graphical proof interfaces makes it possible to increase the productivity of users and allow to work on proof which without interface would be difficult to achieve. An example of such proof is the one of Buchberger's algorithm that was carried out by Laurent Théry [Thé97] in the CtCoq[BBC⁺96] environment¹.

The main objective of this paper is to propose a framework to analyze and manage formal proofs produced by proof assistants. The construction of proof using those systems is a task relatively close to programming activity. Thus, problems encountered during the development of large proof are similar to those occurring in the development of large programs. Therefore we naturally took as a starting point the ideas and solutions developed for program analysis.

A practical use of the proposed tools should be based on a "well-designed" user interface. A partial implementation of these tools was carried out for the Coq system in the CtCoq environment.

In the first section of this document, we quickly present the framework of our work. We focus on the characteristics of proof system and graphical user interface that are relevant for our work. We will consider goal directed systems and show several ways to represent a proof in such systems. In the second section, we introduce the concept of dependences between the various parts of a proof. We describe various tools to understand and manage a given proof. The main functionalities we propose are an assistance in the proof understanding (visualizing

¹ CtCoq is a user interface for Coq based on the system Centaur[BCD⁺88] developed at INRIA Sophia-Antipolis to develop generic programming environments

and navigating the proof structure), an undo mechanism, some tools to contract and expand a proof, others to “lemmify” and factorize it. In this section, we also discuss problems related to the presence of existential variables.

Finally, the third section introduces the concept of dependence between objects of a formal theory and quickly presents some tools to manage and maintain a theory. The principal functionalities are: dependency graph visualization, theory slicing, code motion and theory reorganization, assistance in the propagation of modifications. We also discuss subjacent problems related to the different ways to compute the dependences and the context sensitivity of some tactics.

Our conclusion assess the experiment and draw up the line of a discussion on the first users reactions and therefore on the future improvements and perspectives.

1 Context

1.1 Proof representation

We consider systems in which the proof construction is goal directed. In such systems the user starts by giving the goal he wants to prove and then transforms it by means of *tactics*. Those tactics are basic rules of logical foundation, derived rules or automatic decisions procedures. Applying a tactic on a goal produces a list of sub-goals or returns an error message (one says that it fails). When the sub-goals list is empty, the goal is proven. If the list is not empty, in order to complete the proof, the user has to prove all the sub-goals by recursive application of tactics. At each step, the goals condition the applicable rules.

Proof tree: Thus the proof has naturally a tree structure. This representation is traditional in formalisms like sequents calculus or natural deduction in which it is called a derivation tree. In the context of proof assistant we call it a proof tree.

Proof script: On the other hand, generally, the only view that the user has of his proof is the list of the tactics he had written. This list is called a *proof script*. When a tactic application has generated several sub-goals the user can generally choose which one he wants to solve first. Thus, the proof script characterizes the way in which the proof is built. If this choice is not possible, the script just corresponds to a depth first left-right traversal of the proof tree, the two representations (script and tree) are then completely isomorph. Else, there are several scripts corresponding to the same proof tree. Depending on the system, there are several ways of indicating which sub-goal one wants to attack. We will retain the mechanism used by Coq which consists in ordering the remaining open sub-goals by a left-right traversal of the proof tree and to precede in the script each tactic by the index of the goal to which it is applied (the default index is 1).

Canonical representation: It is also possible to obtain a canonical representation of the proof by considering a proof in which tactics associated with each node are basic rules of subjacent logic. In systems based on a type theory like Coq, Lego or Nuprl, one can also represent the canonical proof by a λ -term. The canonical tree is simply the typing derivation of this term. According to the Curry-Howard isomorphism a statement is then a type and a proof is a term of this type.

Tactic operators: It is generally possible to combine tactics to get a new tactics. For such purpose, one uses tactic operators also called tacticals.

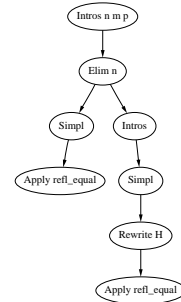
One can mainly finds, although this list is not exhaustive, the operators `Then`, `ThenL`, `Try` and `orelse`².

Example 1. Let us consider the goal below that states the left associativity of addition. ³ :

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

We can prove it with one of the two scripts below which correspond to the same proof tree :

<p>Intros n m p. Elim n. (* base case*) Simpl. Apply refl_equal. (* Induction*) Intros. Simpl. Rewrite -> H. Apply refl_equal.</p>	<p>Intros n m p. Elim n. (* we mix the the nodes *) 2:Intros. 1:Simpl. 2:Simpl. 2:Rewrite -> H. 1:Apply refl_equal. 1:Apply refl_equal.</p>
--	--



It is also possible to build a more compact script using tacticals. (Trivially the proof tree corresponding to the script below has only one node).

Intros;Elim n;[Simpl—Intros;Simpl;Rewrite-> H];Apply refl_equal.

² `Then T T1 T2 T3...Tn` : first apply `T` , then `T1` to each sub-goal produces by `T` then `T2` on all sub-goals produced by the application of `T1` *etc.* In Coq and in this paper this sequential composition operator is represented in infix notation by a “;”.
`ThenL T T1...Tn` : first apply `T` . If `T` does not produce `N` sub-goals there is a failure, else this operator applies in parallel each `Ti` to the corresponding sub-goal. The composition operator `ThenL` can be expressed by using `Then` and another operator which one will note `Parallel`. One will sometimes use the Coq notation `T ; [T1 | ... | Tn]` .

`Try T` : Try to apply `T` . If the application of `T` fails, this leaves the goal unchanged but does not produce an error.

`T1 0rse T2` : Try to apply `T1` and if this fails, apply `T2`. If the application of `T2` also fails, this produces an error.

³ it must be read : $\forall n, m, p : nat. n + (m + p) = (n + m) + p$

Nevertheless, all those proof scripts correspond to the same canonical representation. We give this λ -term as follows:

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)
```

where the notation $[x : nat]Y$ stands for $\lambda x : nat.Y$
and where `nat_ind` is the recursion theorem on integer :

nat_ind: $(P:(nat \rightarrow Prop)) (P\ 0) \rightarrow ((n:nat)(P\ n) \rightarrow (P\ (S\ n))) \rightarrow (n:nat)(P\ n)$

1.2 User interface

Currently, most of the users of proof systems such as Coq, Lego, or HOL, still work without genuine interface: they write their scripts with an editor and "copy-past" into the proof assistant top level.

Bertot, Théry and Kahn [TBK92] showed that the technology developed around the Centaur system to define programming environments could be used to provide graphical interfaces for interactive proof systems. Doing so, they came up new ideas such as "proof by selection" [BKT94], or the "drag-and-drop" mechanism [Ber97]. After various experiments on the systems HOL, Isabelle and Lego, this technology was fully implemented for the Coq system giving rise to the CtCoq (now PCoq) environment used for our work.

Some of the ideas developed in the CtCoq system were recently used by Aspinall, Goguen, Kleymann and Sequeira within a less structured framework. They used Emacs to give rise to a generic environment called ProofGeneral [AGKS99] which provides standards interfaces for Coq, Lego and Isabelle.

Other systems such as PVS, HOL or Isabelle also propose specific environments based on Emacs ⁴.

We now present some of the key aspects of graphical interfaces which were used during the development of our tools. For us, the fundamental point is the script management. The main problem is to keep coherent the contents of the editor and the state of the proof system: i. e., to ensure that, if in a new proof session, the user replays the script that has been saved from the editor, he will get the same state in the proof assistant.

In that purpose, the editor window is generally divided into several parts. the first part is a "read only" zone and contains the commands which were sent and accepted by the system. The second part is a zone that can be edited ⁵. It

⁴ For a quick panorama of the existing interfaces see [Pon99]

⁵ In fact CtCoq holds also other zones containing for example the commands sent but not yet accepted by the system.

contains commands which have not yet been sent to the proof assistant. When a command of the second zone is sent to the system and is accepted, it is transferred into the first zone, if it fails, the script is not modified and an error message is emitted in a dialogue window. Thus users have the guarantee that the script contained in the first zone corresponds exactly to the state of the system.

Another significant aspect is to make it possible to work with several windows, which allows to develop various proofs in parallel. At last, the structured environment, based on the handling of abstract syntax tree, allows to have a mechanism of incremental display (based on the language PPML) and facilitates the script instrumentation.

2 Proof management

2.1 Dependencies in a proof

The proof tree reveals the logical relations between the sub-goals (and thus between the tactics which are applied on those sub-goals and between the sub-proof corresponding to them). Indeed, if, as in example 1, the application of a tactic produces two sub-goals, they are “a-priori” independent (the validity of this assumption of independence is discussed in paragraph 2.4) and whatever the proofs the user provides for each sub-goal, they will make it possible to build a complete proof of the initial goal.

To model the dependence relation, we associate with each node of the proof tree its path in the tree. A path is simply a word on N^+ . We also introduce, on the paths, a partial order relation \prec_p . This relation is a traditional *prefix* order defined by:

$$c_1 \prec_p c_2 \Leftrightarrow \exists c \ c_2 = c_1.c$$

As for the script, it recalls the history of the proof. It introduces an historical dependence relation between tactics (thus between the sub-goals on which they apply). To model this relation, we can consider the script as a function from positive integers to the set of all the tactics. The inverse function, which associates to each tactic of the script its position, is called the rank function, we denote it rg . We also introduce, on the set of all the tactics considered, a total order $<_r$ defined by:

$$c_1 <_r c_2 \Leftrightarrow rg(c_1) < rg(c_2)$$

For the script to be valid, the rank function must check the following properties:

1. rg is bijective
2. $c \prec_p c' \Rightarrow rg(c) < rg(c')$

As explained before, the mechanisms of script management allows, to be sure that a saved script is coherent. Moreover, they provide a rudimentary undoing mechanism which preserves this coherence. We now introduce tools to improve proof and script comprehension. Then we will show how to optimize the undoing mechanism. The tools to be presented in the rest of this section, intensively uses our two notions of dependence even if that does not always explicitly appear in the intuitive presentation that we are going to develop.

2.2 Script and proof comprehension

Graphic visualization It is important to well understand the structure of the proof. For that, we will try to provide the user with a new representation of his proof by graphically visualizing the proof tree.

Effective trees drawing algorithms exist and their implementation presents little difficulties so that one can easily write a specialized program allowing to visualize proof trees. This is the approach followed, for example, by the PVS team using TclTk. Another approach consists in interfacing the environment and/or the proof system with a specialized software. We have adopted this approach by interfacing the daVinci[FW94,FW96] visualization system with the Coq system⁶.

The advantage of the distributed solution will fully appear in the section 3 where one reuses our interface to visualize the dependencies between objects of a theory. Those dependences can be modeled by directed acyclic graphs. The visualization of such graphs and in particular the edges crossings minimization is a very difficult problem. Therefore, to obtain acceptable results, the recourse to specialized systems like daVinci, Dotty or Graphlet is essential.

The experiment has shown the weakness of the graphic visualization of proof trees. Indeed on large proof, the tree grows very quickly in a vertical direction but also horizontally. The vertical long paths do not give important information on the structure of the proof. therefore, they can be grouped in one node either by a graphic artifice during the visualization (it is said that we focus on the points of choice) either physically by modifications on the script as presented in paragraph 2.5. But the horizontal growth reflects, as for it the various case and induction reasoning. Therefore, it is fundamental for the comprehension of the proof structure. Its alteration by a graphic artifice regrouping the nodes is of no interest.

In practice, on large proof, graphic visualization is thus usable only in connection with a partial use of the contraction tools presented at paragraph 2.5. These tools will make it possible to focus on the key points of the demonstration.

Navigating the script Another way to do (standard in CtCoq) is to instrument the script so as to be able to navigate it respecting the tree structure.

The principle of this navigation, which is, in a practical point of view, usable only in a graphic interface, is to select a tactic (using a selection mechanism) and then to apply a navigation command that moves the selection into the desired direction. On the second script of example 1 page 3 one can select the tactics `1:Simpl` with a mere click on the mouse and ask for its father. The result is to transfer the selection onto the tactics `Elim N`.

To implement this navigation, just annotate each node by its path in the proof tree. This information which should be invisible by the user, makes it possible to compute the path of the parent resulting from the displacement command. Then, we must be able to associate with this path the tactics corresponding to

⁶ A description of the prototype and others experiments are given in appendix of [Pon99]

it in the script. That can be done, either by traversing the script until one finds the tactics annotated by this path, or by maintaining an hashtable associating to each path the corresponding tactic in the script. As to each theorem corresponds a new proof tree, one needs an hashtable by theorem. These hashtables are stored by mean of a global table. The keys of which are the theorem's names; its values being the tables associated with each theorem. Let us note that this additional information spends memory which should be managed carefully⁷.

2.3 Undoing

An undoing mechanism is fundamental in any interactive tool: it ensures that errors can be done at lower cost since it is possible to reconsider a decision without restarting from the beginning. Nevertheless, we seek to minimize this cost. To do so, the size of the part of the script to be undone should be reduced.

As suggested by Vitter [Vit84], another key point is the possibility for the user to reconsider an undo command restoring whole or parts of what was undone. We call this an "Undo-Redo" mechanism. Within the framework of proof systems, undoing consists in removing the state produced by playing the tactics to undo. These tactics should also be removed from preserved script. In a well designed interface, the erased tactics should be put aside so as to be replayed if needed. Therefore an "undo command" can be undone as all other command sent to the proof system.

Historical undo. In all proof systems that we know, the undoing mechanism is historical. This is to say that the system manages a stack of states. Each time that a tactic is played, this generates a new state that is pushed on the stack. The user has a command allowing him to pop the last state allowing to undo the most recently played tactic.

The main consequence of this management per stack is that, to remove a tactic, it is necessary to remove all those which were played afterwards, even if they apply only to completely independent sub-goals (*i.e* incomparable for \prec_p).

Another drawback of the stack management is its high memory cost. This involves that this stack and thus the number of possible undo should be limited.


Logical undo. The mechanism we call logical undo allows to cure the disadvantages of the historical undo. The basic idea is to use the logical dependences from the proof tree. When the user does a logical undo on a given tactic, the only tactics that should be removed from the script are those that belong to the sub-tree of which the undo one is the root. Example 2 illustrates the differences between historical and logical undo.

⁷ The complete implementation is described in [Pon97].

Example 2.

Initial script

```

Intros n m p.
Elim n.
2:Intros.
Simpl.  undo here
2:Simpl.
2:Rewrite -> H.
1:Apply refl_equal.
1:Apply refl_equal.
-----

```

After an historical Undo :

```

Intros n m p.
Elim n.
2:Intros.
-----
Simpl.
2:Simpl.
2:Rewrite -> H.
1:Apply refl_equal.
1:Apply refl_equal.
-----

```

After a logical Undo :

```

Intros n m p.
Elim n.
2:Intros.
2:Simpl.
2:Rewrite -> H.
2:Apply refl_equal.
-----
1:Simpl.
1:Apply refl_equal.
-----

```

In the historical undo, all the tactics following the tactic to be undone are removed from the script. They are however preserved in a "writable zone" so that they can be replayed after being modified if needed.

In the logical undo, only the tactics depending on the `Simpl` tactic that has been selected are removed. The other tactics : `Simpl`, `Rewrite` and `Apply refl_equal` which tackle independents sub-goals (on another branch of the proof tree) are not affected.


This example helps to illustrate a first difficulty. In the case of an historical undo, when removing a tactic, we restored a state which has been previously reached and which is thus obviously coherent. In the case of a logical undo the script produced does not always correspond to a state which has already been reached. Therefore, to maintain the script coherence, we might have to modify the index of the tactics in the preserved part of the script (in our example, after the undo, the tactic `Apply refl_equal` applies on the second open sub-goal and thus its index should become 2).

The same problems arise for the erased part. If one wants to be able to replay it, it is sometimes necessary to modify the index in order to maintain the coherence as illustrated in example 3.

Example 3.

Script initial

```

Intros n m p.
Elim n.
2:Intros.
Simpl
2:Simpl.  undo here
2:Rewrite -> H.
1:Apply refl_equal.
1:Apply refl_equal.
-----

```

After a logical Undo:

```

Intros n m p.
Elim n.
2:Intros.
Simpl.
Apply refl_equal.
-----
1:Simpl.
1:Rewrite -> H.
1:Apply refl_equal.
-----

```

The logical undo mechanism can be divided into two phases. First of all, a reorganization of the script, during which some tactics are moved at the end of the "read only" zone and the indexes are modified to preserve the coherence.

Then, an historical undo is done on the reorganized script obtained in order to suppress the tactics which have been moved.

To carry out the indexes modifications by traversing the script, path annotations are not sufficient. It is necessary to have additional information on the number of sub-goals generated by each tactic. We can then compute the list of the paths of open sub-goals corresponding to each step. The index of the tactic played at a given step is simply the position of its path in the list of the open sub-goals of the preceding step.

In addition, we showed in [Pon99] that the information concerning the number of generated sub-goals is enough to rebuild the structure of the tree starting from script.

The logical undo that we have just presented works very well in a world in which the proof tree branches correspond to independent results; this is to say, in which the way the user proves sub-goal corresponding to a given branch does not influence the way the sub-goals of the other branches can be proved.

That is not always true in the systems which allow to handle existential variables. We now study the problems arising from the use of existential variables and from their interaction with the logical undo mechanism.

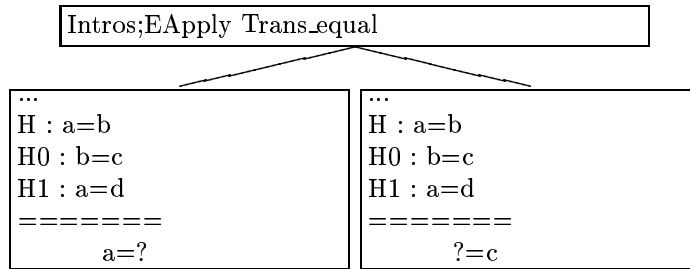
2.4 Existential variables

The use of existential variables. A classical use of existential variables is related to the introduction of existential quantifiers which is necessary to solve goals of the form $\exists x.P(x)$. To introduce the existential quantifier, we must give a witness of the property P. This is to say that we should know the solution before solving the problem. The use of existential variables makes it possible to delay the instantiation of the witness. For example, to solve $\exists x.x + 2 = 5$, without existential variable we instance x by 3 and we prove that $3+2 = 5$. Using existential variables, we introduce a variable X , and we try to prove $X + 2 = 5$. After reduction, we obtain $X = 3$ which allows us to instantiate X .

Example 4. Constraint introduced when sharing existential variables

```
Lemma pbmeta : (a,b,c,d:nat)a=b-> b=c-> a=d-> a=c.  
Intros;EApply trans_equal.
```

After the application of the theorem `trans_equal` the statement of which is :
($x, y, z : A$) $x=y \rightarrow y=z \rightarrow x=z$, we get two sub-goals. It results in the state as shown below:



We first solve the second one by the following command

```
2:EExact H0.
```

This introduces a new constraint `?=b`. Then we try to prove the remaining goal using `H1`

```
Try EExact H1.
```

The tactics `EExact H1` fails because the existential variable `?` is already constrained by `?=b` and thus cannot be unified with `d`. We conclude the proof by using the assumption `H`.

```
EExact H.
```

Consequences in the use of the logical undo: Let us continue with the complete script of example 4, and do a logical undo on the tactic corresponding to the second branch of the proof tree:

```
Intros;EApply Trans-equal.
2:EExact H0. ⚡ tactic to be undone
Try EExact H1.
EExact H.
```

The consequences of the application of this tactic are removed from the proof system (that does not suppress the constraints which have been introduced). On the interface side, the tactic is simply moved in the "writable zone".

```
Intros;EApply Trans-equal.
Try EExact H1.
EExact H.
```

```
-----
EExact H0.
```

The erased part can be replayed to finish the proof. The complete proof can be saved. However the obtained script is not valid. Indeed, it could not be replayed in a forthcoming session, because the introduced constraints will not be the same⁸.

⁸ There will be no constraint when `Try EExact H1` is applied. Therefore, it will not fail, and it will solve the first sub-goal introducing the new constraint `?=d`. Thus,

Solutions for a logical undo in the presence of existential variables:

We can mainly propose two solutions:

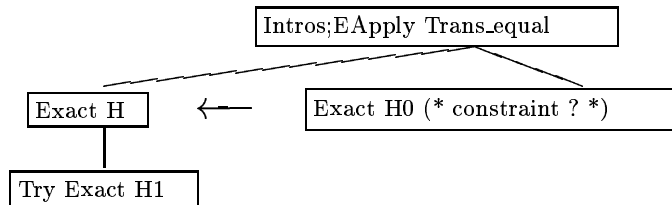
Enforce the instantiations. The idea is to add in the transformed script a command introducing the constraints initially generated by the erased tactics. Thus, if the script is replayed, the introduced constraints will remain the same. That makes it possible to dissociate the undo and the uninstantiation of existential variables.

Example 5.

```
Intros;EApply Trans_equal.  
Instantiate 1 b (* we have forced the instantiation *)  
Try EExact H1.  
EExact H.  
EExact H0.
```

Add a historical level of dependence. The idea is to add a new dependence relation between the tactics handling goals which share existential variables. If a tactic $t1$ has been played after a tactic $t2$, if they share existential variables and if $t2$ introduces a constraint on one of those common variables then $t1$ will depend on $t2$. The dependences are no longer represented by a tree but by a graph.

Example 6. If we consider the script of the preceding example, the tactics of the last two lines apply to goals containing a variable constrained by the tactics of the second line. Thus, we introduce a new relation of dependency as shown in the dependency graph below:



When a tactic is erased, all other tactics which depend on it (in the new relation) must be erased too.

Exact H will become obsolete because the goal that it uses to solve will no longer exist. Moreover, applying **EExact H0** to solve the second sub-goal will fail because this command is not compatible with the new constraint

This add-hoc solution presents a compromise between a rough historical undo and a pure logical undo. Moreover, it is possible to minimize the additional dependences introduced by the existential variables if we consider the tactics semantic as discussed in [Pon99].

Problems related to the constraints introduced by the existential appear in any tool handling proof trees. We have proposed add-hoc solutions to adapt the logical undo mechanism and guarantee its operationality within a framework with existential variables. Such solutions have not yet been studied for the tools presented in the next sections.

2.5 Contraction and expansion of proof:

The main object of the contraction operation is to obtain more compact and comprehensible scripts. It can also be partially used to push back the limits of the graphical visualization of the proof tree.

The spectrum of the applications of the expansion operation is much broader. We suggest debugging and errors localization, dead code elimination, script securization (for example, in Coq, by forcing the explicit naming of the assumptions in the local context).

2.6 Lemmification and factorization of proof:

The idea of the lemmification is to isolate inside a proof an intermediate result and to use it to build a new independent lemma which could be used in other proof without repeating the demonstration. Example 7 illustrates the lemmification in the Coq system.

Example 7. We initially proved the theorem `comPlus` which establishes the commutativity of the addition on natural integer.

Script initial	paths	sub-goal we want to isolate
Lemma <code>comPlus</code> :		
(p,q:nat)(plus p q)=(plus q p).		
Induction p.	[]	
Simpl.	[1]	
Induction q.	[1;1]	(q:nat)q=(plus q 0)
Trivial.	[1;1;1]	
Intros.	[1;1;2]	
Rewrite H.	[1;1;2;1]	
Trivial.	[1;1;2;1;1]	
Intros n H.	[2]	
Simpl.	[2;1]	
Intro q.	[2;1;1]	

Rewrite (H q).	[2;1;1;1]	
Elim q.	[2;1;1;1;1]	
(Simpl; Trivial).	[2;1;1;1;1;1]	(S (plus q n))=(plus q (S n))
Intros n0 H0.	[2;1;1;1;1;2]	
Simpl.	[2;1;1;1;1;2;1]	
(Rewrite H0; Trivial).	[2;1;1;1;1;2;1;1]	

Statements of the right-hand side column correspond to the result which one wants to isolate. We provide a functionality which, takes the path of the goal to isolate and the name of the theorem to be generated, then build the complete statement and its full proof script. Thus, here we can generate the two lemmas `plus_n_0` and `plus_n_Sm` respectively associated with the path [1;1] and [2;1;1;1;1]. Once the lemmas are generated, we can simplify the initial script.

# lemmify [1;1]	# lemmify [2;1;1;1;1]	(*the simplified script *)
"plus_n_0";;	"plus_n_Sm";;	Lemma comPlus :
Theorem plus_n_0 :	Theorem plus_n_Sm :	(p,q:nat)
(q:nat)q=(plus q 0).	(n,q:nat)	(plus p q)=(plus q p).
Intros .	(S (plus q n))=(plus q (S n)).	Induction p.
Induction q.	Intros n q .	Simpl.
Trivial.	Elim q.	Intros;Apply plus_n_0.
	(Simpl; Trivial).	Intros n H.
Intros.		Simpl.
Rewrite H.	Intros n H0.	Intro q.
Trivial.	Simpl.	Rewrite (H q).
Save.	(Rewrite H0; Trivial).	Intros;Apply plus_n_Sm.
	Save.	Save.

This example shows the call of this functionality from `caml` (subjaent to the `coq` `toplevel`). To use it, we have to give a path in the proof tree; such information is not a priori obvious for the user. This show the interest of a graphic interface in which a mere click is enough to select the path.

Idea of the implementation. The most intuitive way to do is:

1. Recover the local context associated with the sub-goal we want to isolate.
2. Abstract all the assumptions which it contains to get a new statement.
3. Rebuild the proof of this new statement. To do so, it is enough to reintroduce in the local context of the proof the assumptions we abstracted (command `Intro` of `Coq`) and then, to recover in the initial script the part which corresponds to the proof tree of the sub-goal to be insulate.

Nevertheless this method is far from being optimal. The principal disadvantage of this approach is that the local context often contains many assumptions which are not necessary to prove the isolated sub-goal. When abstracting them we obtain a too constrained statement. We can optimize the statement creation

by abstracting only the assumptions that are useful for the proof. This method which makes it possible to produce more general statements raises two problems. First of all, it is necessary to determine if an assumption is useful for the proof. To do so, we analyze the canonical representation of the proof. The intersection of the set of all free identifiers that occur in this representation with the set of all the identifiers of the hypothesis local base gives the minimal set of assumptions to be abstracted.

The second problem is related to the naming and the referencing of the assumptions of the hypothesis base. Indeed, when an assumption is introduced into the local context, the identifier which is associated to it can be explicitly provided by the user (for example `Intros H1 H2` in Coq) or implicitly by the system. In the second case the names chosen by the system depend on the local context. However when we do not abstract all the assumptions, we modify this context. Thus, if there are other implicit introductions in the script which we have recovered, the names of assumptions that will be introduced may be modified. Therefore, the possible references to these assumptions may be erroneous. To avoid this kind of problem it is necessary to be able to explicitly name all the introduced assumptions. That could be done a posteriori, by using a securization tool like the one mentioned in the preceding section.

The lemmification introduces new theorems into the global context. That modifies the relations of dependences between the objects of the total context. We are going to study those dependencies.

3 Theory management

In this this section we consider the notion of dependence between the objects (theorems and definitions) of a theory. Once this concept is defined, we can associate with any theory a dependency graph. This graph can be used as a basis to define many tools handling theories.

3.1 Dependence between the objects of a theory:

We say that a theorem A depends on B if its proof uses B or if its statement refers to this theorem. We could think that to compute which results depend on a theorem, it is enough to recover the list of the identifiers which appear in its statement and proof script. Example 8 shows that this is not sufficient. It also reveals that it is necessary to have recourse to the canonical representation.

Example 8. The left associativity of the list concatenation

$$\forall l, m, n : list. l@(m@n) = (l@m)@n$$

The proof script is:

Lemma `ass_app` : (l,m,n : list) (app l (app m n)) = (app (app l m) n) .

Proof.

Intros.

Apply `sym_equal`.

Auto.

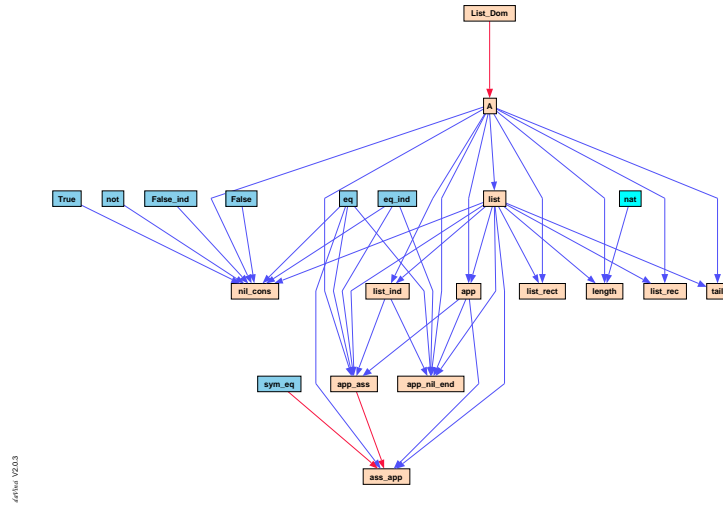
Save.

and the canonical representation is:

```
ass_app =
[|,m,n: list]
( sym_equal list ( app ( app l m ) n ) ( app l ( app m n ) ) ( app_ass l m n )
  : (l,m,n:list) ( app l ( app m n ) ) = ( app ( app l m ) n )
```

Thus, by computing the dependences of `ass_app` on the script we obtain only `list`, `app` and `sym_equal` whereas on the canonical representation, we get also `app_ass` which characterizes the right associativity and which has been introduced by the automatic decision procedure `Auto`.

By carrying out this computation on all the objects of the environment (during their introduction or a posteriori) we can build a dependency graph as the one of the List theory given below..



Theory slicing. Given a theorem and a theory, what we call theory slicing consists in cleaning the theory, keeping only the results which are useful to prove the theorem. This is more or less analogous to program slicing in data flow analysis. Let us see how it works.

Let T be a theory and t a theorem. We compute the transitive closure of the dependency sub-graph resulting from t . We recover the list of the source files in which these identifiers appear. Then, we traverse those files to remove all

the objects which are not associated with one of the identifiers of the transitive closure. To do so, we write using using caml-lex small program which analyzes the source files and removes all these objects.

Nevertheless we must be careful. In example 8, the information in the script is inevitably a subset of the information computed in the canonical form of the proof. In fact, this is not always true. Indeed, in Coq, for example, the proof terms are stored in normal form and some identifiers (appearing in the proof of a sub-goal introduced by a cut) can disappear after the normalization and nevertheless be necessary to replay the proof script. If the script uses automatic decision procedures, some identifiers may appear neither in the script nor in the normalized term but they might be necessary to replay the script. Examples are given in [Pon99]. We avoid this kind of problems by computing the dependences before normalization.

Code motion and theory reorganization: For future reuses and the long-term maintenance of proof is significant, to ensure that theorems of a same field are really stored in the same area.

But, a situation that frequently leads to results being in a wrong place is when developing a new theory using another previously developed theory, a user discovers that he misses a theorem in this last theory. (For example, when developing the polynomial theory using the list theory, he realizes that he misses a property of the lists). Most of the time, he then proves the needed result in the context in which he is working and also save it in the theory he is developing (in our example the theory of the polynomials). That results in duplicating code and work because other users will not know that this result was proven since it is not in its “usual place”. Thus, it is significant to allow the reorganization of the theory. But placing a theorem in its “usual” spot is not always obvious. It is first necessary to know that the result to be moved does not use results from the current theory. To check that, we will use the dependency graph to search which theorems are necessary to prove the theorem to be moved. Those theorems have to be moved too.

Help in the propagation of the modifications We understand by modification, some axiomatic changes to satisfy new specifications; some definition or function modifications, to optimize the representation; or in the systems such Coq or Nuprl which propose a program extraction mechanism, some proof modifications to optimize the extracted code. . .

In [Pon99] we propose an algorithm to help the user to minimize the number of needed modifications. At each step, it proposes a choice of theorem to be modified.

Moreover, this incremental method to choose the theorems to be modified, allows optimizations based on the nature of the dependences. Indeed, in mathematics only the sequence of theorems which have been proved is significant but not the way in which they have been proved (principle of “proof-irrelevance”). To keep with this usual mathematical practice system like Coq allows to consider the objects of the environment as opaque or transparent. This allows

to define the concepts of opaque and dependences dependence. The notion of opaque dependences make it possible to stop the propagation of modifications each time that the proof of a theorem is modified but its statement is not.

Context sensibility Most of our tools produce modifications of the global context which can modify the behavior of the automatic decision procedure. A weakening of the global context, as produced by a theory slicing, is a priori of no risk. The context enrichments which can result from a code motion or from modifications are more dangerous. Suppose for example that analyzing the dependences, we know that a theorem T depends on a set of results \mathcal{R} . If a theorem T_1 , not belonging to \mathcal{R} , is modified that should not invalidate the proof of T . However if the script of proof contains automatic decision procedure, the replay may fails. Indeed, before the modification the theorem T_1 could not be used by the decision procedure (else it would appear it in \mathcal{R}). But after the modifications this procedure can use it to solves some sub-goals, and invalidate the remained script. A priori, if a decision procedure solve more things, one can suppose that the script in the new context should be a sub-set of the precedent. In this case, the tools of section 2 help to find and remove the obsolete tactics.

However if a script uses automatic decision procedure jointly with not monotonous tacticals, such as `Or else`⁹, the set of the sub-goals produced by script when it is played in the new context is not alway a sub-set of what it was in the old context. It can even be completely different, preventing any reuse. A solution is suggested in [PBR98,Pon99], is to avoid this kind of problems. It consists to artificially reduce the set of the results usable by the automatic decision procedures and to limit them to the results that where present in the old context.

4 Conclusion

The starting point of this work is the concept of dependence. The dependences are basically represented by a tree in the case of proofs, and by a direct acyclic graph in the case of theories. The main idea is to use them to propose a set of proof engineering tools. The design of the proposed tools may seem simple but difficulties appear quickly: existential variables, problems of naming and referencing the assumptions in the local context, context sensitivity of some tactics *etc.*

This research task aims at practical considerations, it is thus fundamental to study the impact of it on a great number of users. The general impression that we want to convey is that the proposed tools are practicable and really helping in the development and maintenance of interactive proof. The tools working on the proofs are quickly necessary even during the first utilization of a proof assistant. The beginner will need to be guided during the construction of his proof and will quickly need to undo and redo some parts of his work. Thus, those tools are probably more instinctive.

⁹ examples of problems involved in the use of the `Or else` is in [PBR98,Pon99]

On the contrary, the tools working on theories are more designed to expert users already able to build large developments. Moreover, they definitely require a larger investment from the user. He may use our functionality to clean and to reorganize his work even if he has not used our interface to do his development. Such a skilled user represents a more narrowed population and it is consequently more difficult to analyze the impact of these tools.

This distinction is found in our results, the tools working on proof were developed first and have been used, for example, in the mathematical proof developments or in the compilers proof done by the Lemme team at INRIA Sophia-Antipolis.

Surprisingly enough, new needs appear through diverted uses of the existing tools (for example the use of logical undo to reorganize a script). Today, as regards the more recent tool working on theories, that are not yet implemented in their totality, we have no feed back. An exception is the visualization tools which are already used to illustrate mathematical developments.

To allow a real evaluation of the proposed tools, it will be necessary to put them at the disposal of a broader number of users. To do so, in addition to the need to develop and integrate our prototypes, it is necessary to study with a particular care the ergonomic problems.

Another line of reflection is the study of the influence of the script language characteristics on the development of the proposed tools. The simplicity with which such tools can be developed may even be used to evaluate the language itself.

At last, we want to be generic, even if the prototype implementation were carried out for the Coq system. It seems that most of the proposed tools can be adapted to other systems. Nevertheless, the influence of the system characteristics, such as the use of existential variables, the existence of proof term, the way to refer to local hypothesis, *etc.* will have to be studied more in details. For example, an adaptation of the logical undo in a language, such as Isabelle, based on higher order unification seems an interesting challenge.

References

- [AGKS99] David Aspinall, Healfdene Goguen, Thomas Kleymann, and Dilip Sequeira. *Proof General 2.1*. LFCS, University of Edinburgh., August 1999.
- [BBC⁺96] Janet Bertot, Yves Bertot, Yann Coscoy, Healfdene Goguen, and Francis Montagnac. *User Guide to the CtCoq Proof Environment*. INRIA, Feb 1996.
- [BCD⁺88] Patrick Borrás, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the system. In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).
- [Ber97] Yves Bertot. Direct manipulation of algebraic formulae in interactive proof systems. In *Electronic proceedings for the conference UITP'97*, Sophia Antipolis, September 1997.
- [BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by Pointing. In *International Symposium on Theoretical Aspects of Computer Science*, 1994.

- [CAB⁺86] Robert Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harber, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.
- [FW94] Michael Frhlich and Mattias Werner. The graph visualization system daVinci - a user interface for applications. Technical Report 5/94, Department of Computer Science; University of Bremen, September 1994.
- [FW96] Michael Frhlich and Mattias Werner. daVinci V2.0 Online Documentation, 1996. http://www.informatik.uni-bremen.de/~davinci/doc_V2.0.
- [GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.
- [INR96] INRIA. *The Coq Proof Assistant Reference Manual*, December 1996. Version 6.1.
- [LEG] The LEGO World Wide Web page. url <http://www.dcs.ed.ac.uk/home/lego>.
- [PBR98] Olivier Pons, Yves Bertot, and Laurence Rideau. Notions of dependency in proof assistants. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1998 "*, Sophia-Antipolis, France, 1998.
- [Pon97] Olivier Pons. Undoing and managing a proof. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1997 "*, Sophia-Antipolis, France, 1997.
- [Pon99] Olivier Pons. *Conception et réalisation d'outils d'aide au développement de grosse théories dans les systèmes de preuve interactifs*. Thèse de Doctorat, Conservatoire National des Arts et Métiers, 1999.
- [SOR93] Natarajan Shankar, Sam Owre, and John M. Rushby. A tutorial on specification and verification using PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1993. (Beta Release).
- [TBK92] Laurent Théry, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5), 1992. Proceedings of the 5th Symposium on Software Development Environments.
- [Thé97] Laurent Théry. Proving and computing: a certified version of the Buchberger's algorithm. Technical Report 3275, INRIA, Oct 1997.
- [Vit84] Jeffrey Scott Vitter. US&R: a new framework for redoing. *IEEE SOFTWARE*, 1(4):39–52, October 1984.