Notions of dependency in proof assistants

Olivier Pons, Yves Bertot and Laurence Rideau

INRIA Sophia Antipolis - BP 93
06902 Sophia Antipolis - France
{Olivier.Pons, Yves.Bertot, Laurence Rideau}@sophia.inria.fr

Abstract. This article describes uses of dependencies in tools to maintain big proofs in interactive proof assistants. The aim of this article is to describe the notion of dependency in proof assistants, how it can be used to design tools to maintain large proofs and how those tools could be integrated in proof environment.

1 Introduction

When developping a large proof or a large theory in an interactive proof system, it is often necessary to modify previous results and axioms or to move theorems from one context to another. To maintain the consistency of the mathematical development after such an operation it is necessary to take care of the results that depend on the modified objects.

Notions of dependency are common to a large variety of proof systems. Our objective is to use them to develop general tools to help users in the task of maintaining their large proof developments.

We have concentrated on two activities, which we believe to be the most frequent ones when maintaining large proof developments, and which could benefit a lot from automatic support.

Changing axioms and basic definitions. During a proof or a theory development, user often have to modify some of the basic facts. For instance, if the objective is to re-use a proof in a more general context, one may need to weaken the assumptions on which this proof relies. It is then useful to know which part of the development may be reached by these modifications and must be checked again. We want to produce tools to guide the user in this task with some interactive help.

Code motion. A development is a set of definitions and theorems but it is generally organized so that related parts are grouped together.. Ensuring that related theorems are all stored in the same area is important for the long term maintainability of the proofs. Unfortunately, users are sometimes reluctant to organize correctly their mathematical results, as they are intimidated by the task of restoring the consistency of their proof development.

in: *User Interfaces for Theorem Provers (UITP)*,
Nijmegen, The Netherlands, July, 1998.

A situation that frequently leads to mathematical results being misplaced is that users discover that a theorem is missing only when they are using a theory. Users then tend to prove the missing theorem in their own context, and store it there. This often leads to duplicated efforts, as other users will not know that the theorem has been proved if it is not stored in its natural place. However, putting a theorem in its natural place requires an effort, as adding a theorem in a well-used theory may have consequences in very large contexts, because of automatic proof procedures whose behavior depends on the theorems present in the context.

In the rest of this paper, there are five sections. Section describes the dependencies that we use in our tools and the data on which they are computed. Section 3 describes the most basic tool that uses these dependencies: a visualizer. Section 4 describes how these dependencies are used to support the work implied by modifications of axioms or lemmas. Section 5 describes the issues involved when moving lemmas around. Section 6 brings a conclusion to this work.

2 Computing dependencies

Computing dependencies is a well known issue for people intersted in the maintenance of large software systems. The usual approach is to consider files as the basic units (for instance a compiler processes one file at a time), and tools like `make`[Fel79] use descriptions of the dependencies between files to process these files in order, while tools like `makedepend`[Wal84] produce these descriptions by analyzing the C code they contain.

But we need a finer grain and this is what we study in the rest of this article.

Some provers maintain a concept of proof objects (in the form of λ-terms as in `Coq[?], Lego[LP92], Alf [MN94]` ..., in the form of derivation as in `Jape, EUODHILOS...`). The relations between these different proof objects, which we will call logical relations of dependency, make it possible to understand which objects are necessary to the construction of a given object.

2.1 Use of dependency on objects

Pure object versus object used to build it: Some provers (`Coq,Lego,Hol [GM93] ...`) provide concepts of tactics and proof scripts. A tactic is just a command which transforms an initial goal into zero or several sub-goals remaining to prove. The proof is finished when there are no more sub-goals. A sequence of tactics constitutes a proof script.

In some systems (Coq, Lego) scripts can be used in the construction of the proof object which is the only thing stored in the system.

But in fact it is the proof script that we want to The proof script is the real source code that was initially given by the user. Because it may use automatic proof search procedures, this proof script may be more concise and abstract than the resulting proof object. It may therefore be easier to maintain. That will be mainly done by using the logical dependencies. To compute those dependencies we must use the most informative structure. This is generally the proof object when it exist. In fact proof script usually do not contain all the information which is hidden by complicated and automatic tactics.

The computation of the dependencies is done by going throw the object and finding all the object used in it.

Example: Consider the following `Coq` proof script:

```
Lemma ass_app : (l,m,n : list)(app l (app m n))=(app (app l m) n).
Intros.
Apply sym_equal.
Auto.
```

This script leads the system to construct the following proof object:

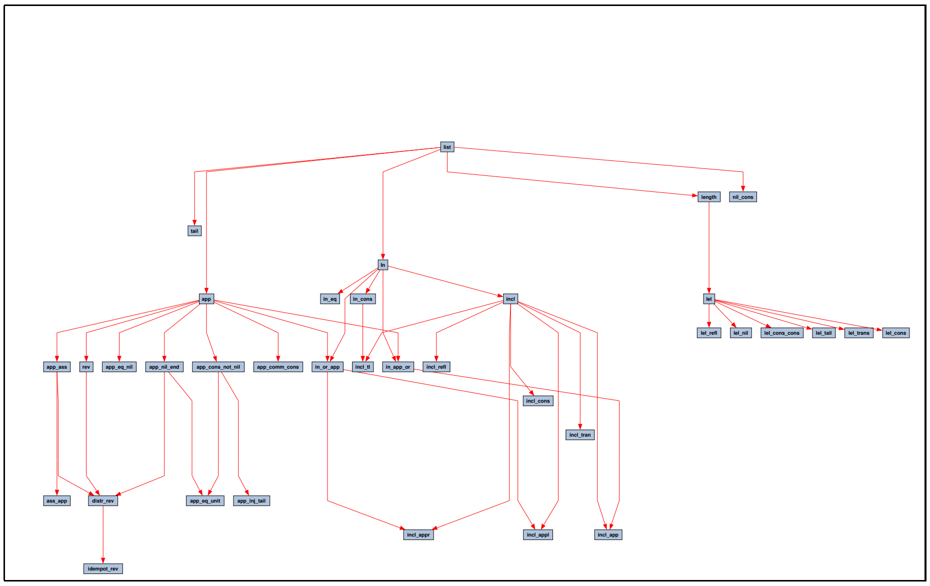
```
ass_app =
[l,m,n:list]
(sym_equal list (app (app l m) n) (app l (app m n)) (app_ass l m n))
```

Going through this term we can say that it depends on `sym_equal, list, app, app_ass`. And even if they do not appear in the proof script those objects are necessary for the construction of the proof.

Performing this kind of analysis for all the objects of the development makes it possible to construct a complete dependency graph.

3 Visualizing and navigating dependency links

When we have a dependency graph, before any change in the development it is important to allow the user to understand and check the structure of his development. We can provide a global visualization of dependencies by compiling our graph data structure toward a formalism (like [Him97a]) understood by a grapher [Him97b,Frö97,MF96]. The following figure gives as an example the dependency graph between the theorems and definitions of the `List` theory provided in the standard library of Coq.



For large developments, this display mechanism becomes quickly too obscure and it will be good to provide tools to navigate in this graph. When selecting an object and with a single key stroke, we want to highlight all the objects on which it depends, the objects that directly depend on it, the objects that transitively depend on it, etc.

4 To change an axiomatization

Now that we have tools to understand the structure of a development we discuss the design of a tool to help the user restore the consistency of their development after changes of axioms or lemmas.

4.1 Initial operation of the user

When the user needs to modify established results or axioms, he can require the help of the dependency tool to evaluate the amount of theorems that will need to be adapted to the new context. The tool simply computes the set of all the objects that can be reached through the dependency links from the modified objects. The tool can even be used *before* actually committing the modification, to evaluate the amount of extra work it will imply.

4.2 Working method

The dependency tool answers by proposing a list of theorems to be modified. It can propose extra information to ease the choice of the user (visualizing the statement of theorems, counting the number of descendants, ...). The user chooses and modifies a theorem taken from this list. The dependency tool then re-computes its graph to take into account the new modified object and proposes a new selection of theorems that must be redone, or informs that the end of the modifications propagation process has been reached.

4.3 Opaque and transparent dependencies

For a given modification, the sub-graph of the dependency graph that contains all the objects that depend transitively on the modified objects is often a gross over-estimation of the amount of proofs that need to be re-done. Theorems depend on each other only through their statements and not their proof. We call that an opaque dependency. As a consequence, the need to propagate modifications will stop as soon as one can adapt the proof of a theorem without changing its statement. For instance, if theorem `T` depends on theorem `U` that depends on `V`, a change in the statement of `V` will require a change in the proof of
tt U, but not in `T` if `U`'s statement is left unchanged.

In systems based on typed theory, an opaque dependency is a dependency on an object's type. On the other hand a dependency on the term is called a transparent dependency. Transparent dependencies correspond to objects whose type give too little information on their actual value (for instance, `plus`, `mult`, and `power` have the same type `nat->nat->nat`).

Instead of analyzing separately each dependency, we consider transparent and opaque objects: all the dependencies on opaque objects are opaque. It is harder to determine the status of a dependency on a transparent object, but a safe approximation is to consider such dependencies as transparent. Practically, the problem now is to know which objects are opaque. Usually theorems are opaque, while definitions are transparent. This can usually be determined with a good knowledge of the proof system behavior. The dependency tool simply considers that an opaque node has not been modified if its statement does not change after the modification.

By analogy, separate compilation uses interfaces to describe modules. These interfaces play the same role as theorem statements and procedures declared but not described in the interfaces are opaque. In the `C` programming language for instance, procedures are usually opaque while macros are transparent.

4.4 Necessary data structures

We maintain a graph of dependencies. The algorithm of propagation marks the nodes of the graph (so that we constantly know their status). The graph is doubly linked so that each node can immediately know the status of its parents.

4.5 The end problem

Theoretically, the propagation of the modifications should stop when we have gone through all the nodes reachable from the modified nodes.

Nevertheless the presence of automatic decision procedures that can be seen as a “black boxes” complicates the matter. A modification of the code or the context in which these procedures execute can change their behavior and thus invalidate the proof script. Thus, we cannot be sure to be able to replay the code after that all the modifications provided by the calculation of the logical dependencies have been done. We come back later to this issue, as it is strongly related to that of code motion.

4.6 Remanent information

Recovering a proof development is a longlasting activity, that may span over several days. It is necessary to allow for interruptions and store the current state of the task.

The information that needs to be kept consists in three parts:

1. The whole consistent script before the first modification,
2. the script that contains all the objects modified so far,
3. the dependency graph, annotated to indicate the object being modified at the moment of interruption.

4.7 Automatic support for adapting individual objects

When updating an individual theorem, it is useful to replay the proof commands that were used to prove this theorem in the old context.

There are two possibles cases:

1. The replay succeeds. In this case, there is not much more to do. If the theorem's statement is unchanged and the theorem is opaque then the theorems that depend only on the current one do not even need to be studied. Their proof can also be replayed automatically.
2. The replay fails. In this case the script, and maybe the theorem's statement, need to be updated. A simultaneous replay of the previous script in the old context and the new context, in two different sessions, will help reuse a large part of this script. The local dependencies as described in [Pon97] can also be used for this purpose.

5 Code motion

We consider the reordering of a set of theorems by moving proof script fragments from one place to the other. We only want to consider moves that respect the structure imposed by the dependency graph. If two theorem `T` and `U` are unrelated by the dependency relation, then `U` can be moved in front of `T` or behind. Theoretically, this move should not have any consequence on the validity of the proof script. In the presence of automatic procedures whose behavior depends on the context, this will not be true.

5.1 Failure due to displacement

Let us consider a proof script that contains the proofs of two theorems `T` and `U` in that order, and so that the proof of `U` does not depend on the proof of `T`. If `U` is moved in front of `T` and the proof of `T` uses automatic procedures, the behavior of these automatic procedures may be changed by the presence of `U` in a way that adds a dependency on `U` for the proof of `T`. For instance, the proof of `T` may contain a subgoal that is solved easily when `U` is in the context and not otherwise. In this case, the initial proof of `T` contains commands to solve this subgoal the hard way, while this subgoal simply vanishes when executing the same proof script in the new context. The now useless commands interfere with the rest of the proof and make the proof of `T` fail.

It seems that the problem is only that automatic procedures become more powerful when one adds theorems in their context. In this case, it should be possible to automatically clean the proof script for `T`, by removing the now useless proof commands. To do this, the tool must compare the runs of the proof script in the two contexts. When a goal appears in the old proof and is missing in the new one, the corresponding commands from the proof scripts can simply be discarded.

This solution is only an approximation that relies on the fact that the subgoals for the proof of `T` in the new context are all subgoals that were already present in the proof of `T` in the old context. The next section shows that this property is not always present.

A counter example in Coq In this example, we use a few tactics from the Coq system. The autmatic tactic `Auto` uses all the theorems declared with a `Hint` command. The tactic `Apply thm` where `thm` has the statement $\mathtt{A}_1$ `->` `...->` $\mathtt{A}_k$ `-> B`, matches the current goal with `B` and returns the k sub-goals corresponding to $\mathtt{A}_1, \ldots, \mathtt{A}_k$. The tactic `Split` breaks systematically goals corresponding to inductive inductions with one constructor (conjunctions fall in this category) and fails on the other kinds of goals. It corresponds to using `Apply` with this constructor. Tactics can be composed with a semi-column, where $\mathtt{tactic}_1$`;`$\mathtt{tactic}_2$ applies $\mathtt{tactic}_2$ to all the subgoals generated by $\mathtt{tactic}_1$. Tactics can also be combined using an `Orelse` combinator with an obvious meaning.

We first construct a context by defining some theorems and definitions:

```
Inductive B: Prop -> Prop -> Prop -> Prop :=
   B_intro: (a, b, c:Prop) a -> b -> c ->(B a b c).
Parameters H1, H2, H3, H4:Prop.
Parameters th1:H1; th3:H3.
Hint th1.
Parameter th:(B H1 H2 H3 /\ H3).
```

Then, in this context, we state a specially crafted goal and apply a compound tactic.

```
Goal ((H4 /\ H4) /\ (H4 /\ H4)) /\ (B H1 H2 H3 /\ H3).
```

```
Split; (Split; Auto; Split) Orelse Apply th.
```

This command leaves four subgoals with the same statement: `H4`. As a quick explanation, we can see that the first `Split` generates two subgoals. On the first one the first branch of the `Orelse` compound succeeds and produces four subgoals of statement `H4`. On the second one, the second split produces three new subgoals: `H1`, `H2` and `H3 /\ H3`. The tactic `Auto` solves the first one and does nothing for the other two. Then the `Split` command fails on the subgoal of statement `H2`. The whole first branch of the `Orelse` compound fails, but the second branch succeeds.

Now let us consider that we move theorems that prove `H2` and `H4` before this proof, for instance with the following commands:

```
Parameters th2:H2; th4: H4.
Hint th2 th4.
```

The compound tactic produces completely different subgoals. The first `Split` still produces two subgoals, but the first branch of the `Orelse` compound does not fail anymore on these subgoals, thanks to the `Auto` command. So the whole command produces two subgoals with the same statement: `H3`. We see here, that `Auto` has become more powerful due to the motion of theorems `th2` and `th4` and that the subgoals generated by the compound tactic are not a subset of the subgoals that existed before. These goals will not be solved by script fragments coming from the previous script.

This is due to the fact that the `Orelse` combinator is not monotonous: if `Auto` solves more goals in the new context, we are not ensured that `Auto Orelse tac` will solve more goals. However, `Orelse` is seldom used and an automatic approach like the one given in the previous section will give good results in many cases.

6 Conclusion

In this paper we address two important tasks for computer-verified proof maintenance. The data we want to maintain is the proof scripts, that is, the sequences of commands that can be sent to a proof system to make it verify the proof.

The first task we consider concerns the modification of logical statements for established facts or axioms. When users need to perform this kind of modification, they have to propagate the modification in the proof development and this can be very time-consuming. The second task concerns code re-organization. This activity is important to ensure the readability and the long-term usability of the proof development.

We show that a notion of dependency is central in these issues. We propose to construct tools around a dependency graph. The first tool makes it possible to visualize dependencies and to navigate the graph. This helps the user understand the structure of the proof development. The information this tool can bring is useful before taking the decision to modify or move an artefact. A second tool helps the user when he has modified the statement of a lemma or an axiom. It uses the dependency graph to indicate to the user the theorems and proofs that need updating. While this tool does not directly help the user making the right modification, it helps propagating all its consequences to other objects. A third tool supports the activity of code motion, where dependencies can be used to indicate the set of theorems that must be moved together.

This work shows that there are two notions of dependencies. The dependency graph that we study uses *direct* dependencies, where a theorem depends on another only if the proof of the former refers to the latter. However, since we consider proof script maintenance, we are also faced with indirect dependencies, where theorems may influence the behavior of proof search procedures, so that the move of a theorem may invalidate proofs that were initially unrelated.

We believe that this case is rare, and we think that a more thorough study of the abstract properties of tactics and tactic combinators is needed to support or invalidate this claim. In particular, we have given an example that traces the failure of re-use to a non-monotonicity of the `Orelse` combinator.

References

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. *The Coq Proof Assistant User's Guide.* INRIA, May 1993. Version 5.8.

[Fel79] Stuart I. Feldman. Make—a program for maintaining computer programs. *spe*, 9(4):255–65, April 1979.

[Frö97] M. Fröhlich. *Incremental Graphlayout in the Visualization System daVinci (in german language).* PhD thesis, Department of Computer Science; University of Bremen, November 1997.

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

[Him97a] M. Himsolt. Gml: A portable graph file format. Technical report, Universität Passau, 1997.

[Him97b] M. Himsolt. Graphlet manuals: Gml, graphscript, c++ interface. Technical report, Universität Passau, 1997.

[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.

[MF96] M. Werner M. Fröhlich. davinci v2.0 online documentation, 1996. Available at http://www.informatik.uni-bremen.de/~davinci/doc_V2.0.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

[Pon97] Olivier Pons. Undoing and managing a proof. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1997"*, Sophia-Antipolis, France, 1997. Available at http://www.inria.fr/croap/events/uitp97-papers.html.

[Wal84] Kim Walden. Automatic generation of make dependencies. *spe*, 14(6):575–585, June 1984.