

Dependency graphs for interactive theorem provers

Yves Bertot, Olivier Pons, and Loïc Pottier

October 2000

Abstract

We propose tools to visualize large proof developments as graphs of theorems and definitions where edges denote the dependency between two theorems. In particular, we study means to limit the size of graphs. Experiments have been done with the Coq theorem prover [DFH⁺93] and the GraphViz [EGKN] and daVinci [FW98] graph visualization suites.

1 Introduction

High quality software relies more and more on formal methods, where mechanical proofs are used to ascertain that every possible behavior exhibited by the software respects some set of constraints. According to this point of view, proof development is a special case of software development. It should benefit from the tools already common in other branches of software engineering. A common feature in software development tools is the use of graphical notations to display the relations between the various entities that occur in a complex design. This work is about using graph displaying tools to display the dependencies between various entities and theorems in a mechanically checked proof development.

In usual program development tools, graph displaying tools have been proposed to observe the control-flow or data-flow graphs associated with programs. The graphs can be recovered from the programs through a mechanical analysis of the programs. The same kind of systematic analysis can be performed on formal proof developments, where the programs are replaced with the documents passed to the theorem prover as a guidance. There actually are several kinds of theorem provers. Some attempt to provide strong automation through complex proof search algorithms. Other behave more like proof checkers, letting the human user decide the big steps and providing automatic support only for the details.

We have worked with a proof checker, Coq, that is based on type theory. Type theory provides a framework to represent formal proofs in a style that is very close to strongly typed programs: actually the formal language also contains a programming language and Coq is especially suited to develop certified programs. For graph visualization, the important feature of Coq is that proofs

are explicit formal objects, expressed in a clearly defined language. It is then possible to analyze these objects to extract dependency information. We shall see in this document that the analysis that is needed to compute dependencies between theorems is extremely simple.

Experiments with toy proofs showed the interest of displaying graphs, but as we moved on to larger proof developments, we discovered that the graphs occurring in formal proof development exhibit a high complexity that often makes them impractical. Intuitively, the relations between the various data-structures in a piece of software are all explicitly expressed in a formal development, so that vertices appear to be more connected than for simple control-flow graphs or data-flow graphs of traditional software, where implicit relations between components are often left untold.

To make the graphs of practical interest, it is necessary to reduce the number of components. A simple strategy to reduce the number of edges is to remove all the edges that are redundant because they are in the transitive closure of other sets of edges (the transitive edges). We also studied two techniques to reduce the number of vertices in the graphs, either by simply removing vertices and all the edges that come and go from these nodes, or by grouping several vertices together, transferring the edges to the new group vertex. For these two techniques, we study how much work could be done automatically and we show what kind of interactive support can be used.

Although all our experiments were done with the Coq proof system and the graph displaying tools `dotty` [EGKN, Kou94] and `daVinci` [FW98], very little of what we describe really depends specifically on these programs. Actually, the one important feature is that Coq provides a way to recover the list of all theorem proofs in the order they were performed and to analyze these proofs to understand what other theorems are used in these proofs.

The first section of this paper is this introduction, the second section describes how we compute the dependencies for one given theorem through an analysis of its proof. The third section describes the approach we propose to remove edges. The fourth section describes strategies to improve readability by grouping nodes together. The fifth section describes strategies to remove nodes. The sixth section describes ideas about graphical display of the graphs that are obtained through our tools and their interactive manipulation. The seventh section gives a brief review of related work. The eighth section gives a conclusion and enumerates the points that deserve a more thorough treatment in further work. Only the second section is really specific to type-theory based proof systems like Coq.

The tool described in this paper is available freely on the Internet as a tool for Coq users, at the universal resource location <http://www-sop.inria.fr/lemme/coq-graphs>.

2 Dependency computation

Theorems are named proofs, in the same manner that procedures or subroutines are named pieces of code. This analogy is exploited even further in proof systems based on type theory, where proof checking is done in the same manner as program type-checking. In such proof systems, proof objects are built as one proceeds through a formal description of the proof and the object is then verified by a simple analysis tool. Only when the verification succeeds is the theorem actually accepted and stored in a database, the context, where it can be used for further reference in subsequent proofs. One key feature we exploit is that the proof is stored in the context with the theorem name and statement.

The context does not only contain theorems, but also definitions for any kind of mathematical objects, data-structures, functions, predicates, . . . All these objects are referred to in the same manner, and references may occur both in the statement of theorems and in their proofs.

Type-theory proof systems are based on a strong analogy between types and formulae. For instance, the formula $A \Rightarrow B$ can be viewed as the type of a function, which produces a proof of B (an object of type B), when given a proof of A . Building a proof then corresponds to building a term in some λ -calculus, whose type is the logical formula to be proved. Tautologies are the types of simple λ -terms, for instance $A \Rightarrow A$ is the type of the function $\lambda x.x$. Pushing the notion of types far enough, it is possible to understand that \forall quantification occurs when typing *dependently typed functions*, functions whose returned type actually depends on the input value. Having said this, it is possible to encode most of the usual logical constructs in this formalism.

A common notion of λ -calculi is that of *free* and *bound* variables. A bound variable is a variable introduced when one describes a function or a for-all quantification and a free variable is a variable provided by the context. For instance, in the sentence

the function, which to x associates $x + 1$ is monotonous

the variable x is bound, while the names *monotonous* and $+$ are free: in a formal development they would need to be defined elsewhere¹. In this sentence the role played by x is well-known while we understand that the concepts by the other names need to be taken from some context. This example shows that it is enough to compute the set of free variables in a term to know which other notions it depends on.

Let us see on an example proof how this works. We take the proof that addition is commutative, given in figure 1. This proof is done by induction on the natural numbers (theorem *nat_ind*), where one must prove the property for 0 and S . We use some properties of equality equal terms can be substituted one for the other (theorem *eq_ind*), and a function applied to equal elements returns equal elements (theorem *f_equal*). We also use two theorems on addition, the first states that adding a number with 0 returns this number (theorem

¹the other name in the sentence, *function* is a builtin part of the language.

```

plus_sym:
λn, m : nat.
  nat_ind(λx. x + m = m + x,
    plus_n_0(m),
    λy : nat. λh : y + m = m + y.
      eq_ind(nat, S(plus(m, y)), λz. S(plus(y, m)) = z,
        f_equal(nat, nat, S, plus(y, m), plus(m, y), H),
        plus(m, S(y)), plus_n_Sm(m, y)), n)

```

Figure 1: Proving that *plus* is commutative in the calculus of constructions

plus_n_O) and the second one asserts that adding a number with the successor of another returns the successor of the sum (theorem *plus_n_Sm*):

```

plus_n_O: ∀n : nat. n = plus(n, 0)
plus_n_Sm: ∀n, m : nat. S(plus(n, m)) = plus(n, S(m))

```

So here the list of free variables is *nat*, *eq*, *plus*, *nat_ind*, 0, *S*, *eq_ind*, *f_equal*, *plus_n_O*, and *plus_n_Sm*.

By nature, the graph is acyclic, because all definitions are placed in the context in strict order and no object can depend on another object that is not defined before it. Still, special attention must be paid to the case of mutually defined recursive types and functions. In the case of Coq, the encoding of mutually recursive definitions brings everything down to one single definition at a time. More work would be needed if we intended to adapt our tools to a proof system where mutually defined objects could introduced cycles in the dependency graphs.

3 Systematic edge removal

The basic graphs obtained when we systematically apply the dependency computation on all theorems are huge and completely unpractical for human usage. Of course, graph display tools often provide means to handle large graphs. For instance *daVinci* provides a *Survey* capability and *dotty* provides zooming facilities, but it is also meaningful to reduce the logical complexity of graphs, for instance when one wants to print them and use them as illustrations in technical documents.

One simple approach to reducing the size of the graphs being observed without loosing too much information about actual dependency between objects is to remove transitive edges. In our experiments, removing transitive edges has proved very useful, this transformation alone making it possible to replace overly intricate figures with readable ones as in figure 2.

In our first implementation, we do this using a simple but expensive algorithm that first produces the transitive closure of the graph and then removes edges between nodes *a* and *b* when there exists a *c* such that there are two edges from *a* to *c* and from *c* to *b* in the transitive closure. This algorithm for edge

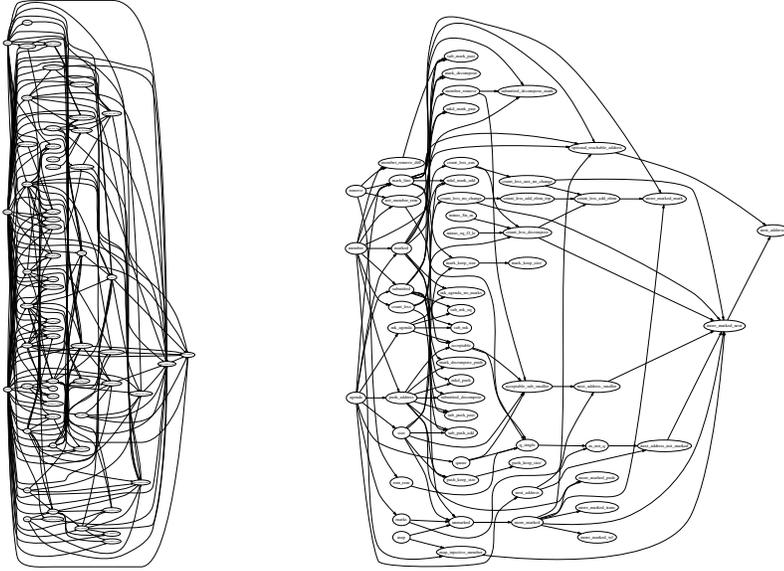


Figure 2: Removing edges. *Both figures represent the same graph, but transitive edges have been removed on the right. This figure was produced using `dotty`*

removal works only if there is no cycle in the graph. If we apply this algorithm to a graph containing cycles, then every edge participating in a cycle is removed. This gives the misleading impression that the vertices concerned by these edges are more isolated than they really are.

As we already discussed, we have made sure that our initial graph computation produces acyclic graphs. We are also going to see how to make sure that this property is also maintained by other transformations presented in this paper. As a corollary, there is no reason why graph removal should be applied first.

4 Grouping strategies

Reducing the number of vertices in the dependency graph is a more subtle question than reducing the number edges. When a node is removed there is a lot of information pertaining to this node that is likely to disappear. Still, it is sensible to group together vertices that correspond to closely related concepts.

4.1 Basic grouping operation

When grouping two vertices a and b together, the two vertices are replaced with a new one c . What should happen with the edges entering into and coming from a and b ?

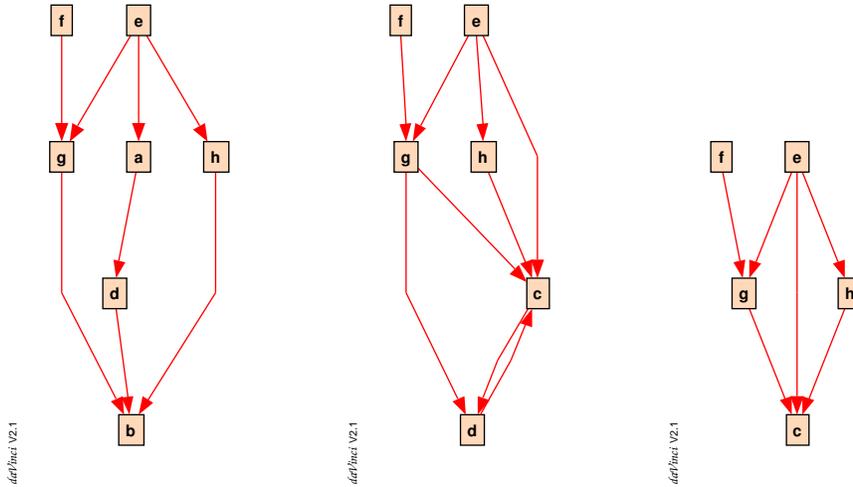


Figure 3: Grouping vertices. Vertices a and b are grouped in a new node c . The vertex d is also collapsed into the group. New transitive edges (one, between e and c) should be removed. These illustrations were obtained using `daVinci`.

If no dependency information is to be lost, then it is necessary to express that every vertex that depended on them should now depend on the new vertex. The same goes for the vertices on which a and b depend.

Now, if there is a vertex d that depended on a and on which b depended, the new vertex c is in a cycle, where c depends on d and d depends on c . To avoid the complication, it is better to also group d with c and all vertices that (transitively) depend on a and on which b depends should be treated in the same manner. An example of this operation is given in figure 3.

We provide tools that make it possible to group more than two nodes together, but so far we only implemented this by successively grouping pairs of nodes. To detect the vertices that are intermediary between two other vertices (a and b in our example), one solution is to compute the transitive closure of the graph and check all the vertices that are transitively reachable from a and see if they in turn reach b . Many optimizations could be envisioned if we decide to consider group operations for large groups without decomposing these as successive pairwise grouping, but we have not studied these optimizations yet.

4.2 Lexical grouping

Closely related concepts are sometimes named alike. It is especially the case for theorems that are created by automatic tools. An example of such theorems are the induction theorems created for inductive definitions in the Coq system. The names of these theorems are obtained by adding specific suffixes to the name

of the inductive type being defined. In the tool we propose, these theorems are systematically grouped with the inductive type they refer to.

4.3 User-directed grouping

For other needs, we rely on the user to specify the theorems and definitions that need to be grouped together. We provide the possibility for the user to describe the group information in files that can be processed by the grouping algorithm. Each entry in the file represent a grouping operation by giving the name of the new vertex and the names of the vertices that are grouped into it. For the grouping operation used as an example in the previous section, the grouping directive has the following form:

```
[c:a b]
```

Many grouping operations consist in grouping one vertex into another, so that the target name actually is the name of one of the grouped vertices. For this operation, we allow the user to omit the target vertex in the list of grouped vertices, so that the following directive is perfectly licit.

```
[c:b]
```

This directive has the same meaning as the following one.

```
[c:c b]
```

5 Pruning strategies

It is interesting to prune nodes for two purposes. The first purpose is to remove theorems and definitions that are considered part of the common knowledge, for instance the basic definitions of usual logical constructs, their basic properties, the formalization of natural numbers, the theorems of arithmetics, etc... In a scientific paper, the description of a large formal proof development can be decomposed into several sections, each using a different graph as illustration, but all the graphs may be different views of the same dependency graph obtained by removing different subsets of vertices.

Our tool makes it possible for the user to specify which of the theorems can be removed from the picture. Specifying vertices to remove is simply done by adding new kinds of directives in our group specification files. the following directive instructs our tool to remove vertices `a`, `b`, and `c`.

```
-- a b c --
```

Pruning nodes and grouping nodes are distinct operations. When pruning nodes, the edges entering into them and leaving from them are simply removed from the graph. In this sense, pruning nodes with entering *and* leaving edges will alter the dependency information in the graph also for nodes that remain in the graph, and the pruning functionality can easily be abused.

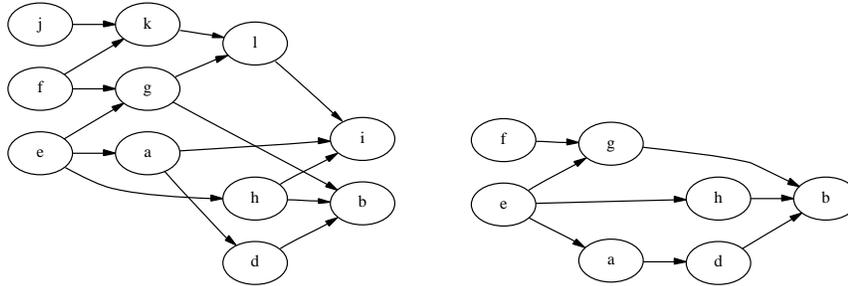


Figure 4: Pruning useless theorems. *In this example, the theorem i has been considered useless, all theorems that only contribute to it, that is, j, k, and l, have also been removed.*

5.1 Pruning unused theorems

Displaying the graph associated to a proof development is a good way to highlight the parts of this proof development that are actually unused. These are the theorems that have no outgoing edges. Now, there always are unused theorems, and some of these theorems are the mathematical results one wanted to establish in the first place. But often, large development also contain lemmas that were intended for use in failed proof attempts and that have not been removed from the theory files.

Unused theorems often do not come alone: once they have been removed, the theorems that only contributed to removed theorems become unused, and it has proved useful to repeat the process recursively, until there are no theorems that contributed only to the mentioned ones (transitively). Figure 4 gives an example of graph simplification produced in this manner.

5.2 Implementing frontiers

Another approach to systematically removing theorems is to prune all theorems that are defined before a given *frontier* theorem, with the intuitive justification that the pruned theorems are part of the common knowledge and need not be shown in the resulting graph. To know what notion of time to use, we simply use the proof system's context as a reference: viewing this context as a list with the most recent theorems appearing first, we simply need to restrict our analysis of the context to all theorems up to the frontier theorem.

It is also possible to have an intermediary frontier, that is used only to prune the edges arriving to nodes beyond this intermediary frontier. Thus theorems beyond the intermediary frontier are mentioned if they contribute to theorems newer than the intermediary frontier but no dependency computation is performed for these theorems. Theorems between the frontier and the intermediary frontier are simply not mentioned.

5.3 Logical reset

Having the context of all past theorems organized as a list of definitions makes it easy to implement an undoing mechanism, where all theorems that were introduced in the context after a given theorem may be simply forgotten. This operation is called **Reset** in the Coq command language.

Now that we have computed the dependencies between theorems it is possible to implement a less brutal undoing mechanism, where only the theorems that depend on a given theorem are forgotten. To ensure the consistency of the context, however, we must work on an unedited graph, where no vertex was pruned (except maybe vertices that were pruned using the frontier strategy). We have implemented this capability in Coq and experimented with it, calling a *logical reset*.

Now, if logical reset is to be made available to users, an extra tool should also be provided to handle scripts, that is, the documents fed into the proof system to perform the proofs. In normal situations, the script is just another representation of the context: all the text occurring before the current position corresponds to the theorems that have been proved. When one resets to a previous theorem, it is only necessary to move the current point to that theorem. When performing a logical reset operation, it is no longer a matter of simply moving the current point up the script, because some theorems occurring after the undone theorem may still be valid in the new context. We have also implemented a prototype tool that analyses the script text and produces a new script that corresponds faithfully to the new state of the context after the logical reset.

6 Graph displaying

Once computed, the graphs can be displayed using an off-the-shelf graph displaying tool. We have experimented with two tools, *daVinci* and *dotty*, a tool in the **GraphViz** family (all illustrations up to this page have been produced using *dotty*). These graph displaying tools most often also provide the possibility to manipulate the graph. This capability comes in handy to help the user prepare the grouping and pruning directive files.

6.1 Interactive manipulation

When manipulating graphs that have been produced through a mechanical analysis of formal documents, the operations that appear most frequently are different from the case where one interactively constructs a graph from scratch. It is worth the effort of customizing the graph displaying tool's user-interface to make the frequent operations easier to perform.

The first frequent operation is the grouping operation. Here again, we can decompose this operation into elementary steps that are pairwise groupings, but this approach has the drawback of requesting too much "bandwidth": when grouping n nodes into another, one will need to express n times which node is the grouping target and also possibly n times that the operation is a grouping

operation. We have experimented with a possibility to set the target node aside, providing a special selection or creation mechanism. The operation that needs to be repeated n times is reduced to simply selecting the node to group with the target and then expressing that one wants to group this node with the target.

The graph displaying tool `dotty` has a very limited user-interface, with specific menus for vertices that are popped when clicking on a vertex with the right button of the mouse. To implement our approach, it is only necessary to add two options, one for setting the target and one for actually grouping, to this menu. Still, having to slide the mouse to select the right option in a pop-up menu has proved quite tedious.

We have provided the same kind of behavior in `daVinci`, but instead of having to indicate the operation to perform at each step, we have implemented a *grouping* mode: when entering grouping mode, one expresses which node is going to be the target, then every node that is selected until exiting the mode will be immediately grouped with the target node. This behavior has two main drawbacks. First it is based on a mode, which always imposes a burden on users, as they need to remember what operations are possible in each mode. Second, it commits the choices too quickly: if the user selected a vertex by error, this selection operation is immediately interpreted as a grouping operation, which is not a trivial operation. This eager behavior is acceptable only if undoing the step is also made very easy.

If the graph displaying tool makes it easy to select several vertices, it is more clever to directly provide the grouping operation as a menu operation that is triggered after the users have selected all the nodes that they want to group together. Most interactive tools use the convention that the `Shift` key on the keyboard can be used to express that the object selected with the left button of the mouse needs to be added to a group selection, or removed if it was already selected. This approach is much better, since the steps that are easily done are also easily undone, and the complex operation of looking for an option in a menu also corresponds to an operation that is more complex to undo. We have been able to implement this behavior in our experiments with `daVinci` but not with `dotty`, whose user-interface capabilities are cruder.

In interactive systems, it is also necessary to think in terms of undoing. Commands that are easy to perform should be easy to undo. In our case, setting the target is an easily undone operation: actually, setting the target does not modify the graph and if a target has been wrongly set, it is easy to correct the operation by just setting another target. On the other hand, if two vertices are grouped together, it requires some computation to undo the grouping operation: intermediary vertices between the target and the selected vertex have to be recovered and edges have to be restored between all these nodes. In our current experiments, we have not yet come up with satisfactory undoing capabilities.

6.2 Using graphs in hypertext documents

Graphs can also be displayed in hypertext pages, making the vertices active so that when users point at them an extra view is opened that makes it possible to observe the text associated to this vertex. This capability is provided for instance in the **GraphViz** tool suite: the `dot` program makes it possible to generate both a bitmap file (in `gif` format) and a map that will be used in `html` navigators to decide which part of the bitmap are active and which hypertext link they point to. We have developed a simple tool to update this map so that the links are really links to anchors in the `html` files generated by the `coq2html` tool provided with Coq.

To provide the same capabilities using `daVinci`, we use the postscript generator of `daVinci` and perform a lexical analysis on this postscript file to recover the position information for each vertex. It is then possible to generate a bitmap file from the postscript using a postscript to bitmap converter (for instance the command `convert` available on most Unix systems). Our tools also constructs the map that is used as in the previous paragraph to describe the parts of the bitmap that are active and the corresponding hypertext links.

If a vertex is a group vertex, there may be two possibilities for the hypertext link this vertex carries. Either this group vertex has the same name as one of the theorems the group contains and it relevant to have the hypertext link point to that theorem, or this group vertex has a name that is not the name of a theorem and it is better to make the hypertext link point to a fragment of text where all the hidden names appear, where each name actually is an hypertext link to its proof.

7 Related work

There is a very active community on the topic of graph visualization and graph drawing, for instance there is an annual conference cycle entitled *Graph Drawing*. Our restricted study of graph displaying suites also hides the variety of graph displaying tools that are becoming available, mostly thanks to Internet publicity.

Graph visualization tools have also been implemented for a variety of proof systems Coq [PBR98, PP98], Lego [LP92], PVS [SOR93], HOL [GM93, SB94, Sym95], Isabelle [OEC97], Kiv [Rei92], Omega[HKK⁺94, SHB⁺99]. Generally, interaction with system makes it possible to visualize proof trees, theory dependencies, project . . . However, few proof systems make it possible to know exact dependencies between theorems, because the computation of these dependencies often requires an explicit representation of proofs and such a representation is often not produced, as it is considered too costly. Our reliance on type theory based theorem avoids this problem, but similar work could also be done with the explicit proof representation package developed for HOL by Wong [Won95].



Figure 5: A real life example

8 Conclusion

The work presented in this paper is meant to fulfill practical needs that arose in the development of large scale proofs, that is, proof developments representing more than a few man months of effort. In such proof developments, there are a few hundreds of proof units such as theorems, lemmas, or definitions. Automatic support to build documents describing the structure of these developments is welcome. For instance figure 5 is taken from a formal proof we performed to certify a byte-code verifier.

From a theoretical point of view, most of the capabilities described in this paper are well-known operations on abstract graphs. We have rather tried to isolate those capabilities that did correspond to needs that we had experienced in our proof developments. In this sense, this paper is an enumeration of the graph-based tools that we think are most useful and simple ways to implement these tools.

Still, having introduced graph displaying in our proof development environment, we have also been able to think of new functionalities that could be provided by the proof system, like the *logical reset* capability.

Being both developers and users of this collection of tools, we have a biased perception on the actual improvement these tools bring in the proof development process. To achieve a better comprehension on the usability of graph-related capabilities we hope to collect to feedback from a wider community of users. We disseminate these tools on the Internet, making them available at the address:

`www-sop.inria.fr/lemme/graphs`

Acknowledgements Olivier Pons is supported by the Portuguese Science Foundation (Fundação para a Ciência e a Tecnologia) under the Fellowship PRAXIS-XXI/BPD/22108/99.

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.
- [EGKN] John Ellson, Emden Gansner, Eleftherios Koutsofios, and Stephen North. Graphviz. Available on the internet at URL <http://www.research.att.com/sw/tools/graphviz>.
- [FW98] Michael Fröhlich and Mattias Werner. daVinci V2.1 Online Documentation, July 1998. accessible at http://www.informatik.uni-bremen.de/~davinci/doc_V2.1.

- [GM93] Michael J.C. Gordon and Thomas.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [HKK⁺94] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg Siekmann. Ω -MKRP: A proof development environment. In Alan Bundy, editor, *Automated Deduction — CADE-12*, Proceedings of the 12th International Conference on Automated Deduction, pages 788–792, Nancy, France, 1994. Springer-Verlag, Berlin, Germany. LNAI 814.
- [Kou94] Eleftherios Koutsoufios. Editing graphs with *dotty*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1994. This report, and the program, is included in the **graphviz** package, available for non-commercial use at <http://www.research.att.com/~sw/tools/graphviz/>.
- [LP92] Zhaohui Luo and Robert Pollack. LEGO proof development system: User’s manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King’s Buildings, Edinburgh EH9 3JZ, May 1992. Updated version. Available by anonymous ftp with LEGO distribution.
- [OEC97] Maris A. Ozols, Katherine A. Eastaughffe, and Antony Cant. Xisabelle: A system description. In *Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 400–403. Springer-Verlag, July 1997.
- [PBR98] Olivier Pons, Yves Bertot, and Laurence Rideau. Notions of dependency in proof assistants. In *User Interfaces for Theorem Provers 1998*, Eindhoven University of Technology, 1998.
- [PP98] Loic Pottier and Olivier Pons. dependtohtml: Creating hypertext graphical representation of directed graphs, 1998. Available on the internet at URL <http://www.inria.fr/croap/personnel/Loic.Pottier/Dependtohtml/README.html>.
- [Rei92] Wolfgang Reif. The KIV system: Systematic construction of verified software. In D. Kapur, editor, *Automated Deduction: CADE-11 - Proc. of the 11th International Conference on Automated Deduction*, pages 753–757. Springer, Berlin, Heidelberg, 1992.
- [SB94] Tom Schubert and John Biggs. A tree-based, graphical interface for large proof development. In Tom Melham and Juanito Camilleri, editors, *Supplementary Proceedings of the 1994 HOL*, 1994. available at URL <http://www.dcs.gla.ac.uk/hug94/sproc.html>.
- [SHB⁺99] Jörg Siekmann, Stephan M. Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase,

Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet, and Volker Sorge. *LCUI: Lovely Ω User Interface. Formal Aspects of Computing*, 11(3):326–342, 1999.

- [SOR93] Natarajan Shankar, Sam Owre, and John M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. A new edition for PVS Version 2 is expected in 1998.
- [Sym95] Donald Syme. A new interface for hol - ideas, issues, and implementation. In *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 324–339. Springer-Verlag, September 1995.
- [Won95] Wai Wong. Recording and checking HOL proofs. In Phillip J. Windley E. Thomas Shubert and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 353–368. Springer-Verlag, 1995.