Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 75–86
www.elsevier.com/locate/entcs

Tool Support for Proof Engineering

Anne Mulhern[1,2] Charles Fischer[3] Ben Liblit[4]

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI USA

Abstract

Modern integrated development environments (IDEs) provide programmers with a variety of sophisticated tools for program visualization and manipulation. These tools assist the programmer in understanding legacy code and making coordinated changes across large parts of a program. Similar tools incorporated into an integrated proof environment (IPE) would assist proof developers in understanding and manipulating the increasingly larger proofs that are being developed. In this paper we propose some tools and techniques developed for software engineering that we believe would be equally applicable in proof engineering.

Keywords: IDE, IPE, proof visualization, program visualization, refactoring, program extraction, Coq, proof dependencies, proof transformations, proof strategies, proof framework, proof reuse, proof explanation

1 Introduction

Modern integrated development environments (IDEs) provide programmers with a variety of sophisticated tools for program understanding and manipulation. In addition to such basics as syntax highlighting and project building, these tools commonly offer refactorings and program visualization components. Many of the techniques developed for IDEs can be transferred directly to the world of UITPs. Others can be modified to exploit the special nature of theorem provers.

The idea of transferring IDE techniques to theorem provers is not new [2,7,21,36]. However, there have been significant advances in IDEs in the last decade. Many of these advances have been motivated by the needs of developers who must maintain and extend large bodies of existing code. The increasing complexity of real world programs means that even an experienced programmer will struggle to understand

[1] This work has been supported in part by funding from a Graduate Women in Science Ruth Dickie Grant-in-aid award.

[2] Email: mulhern@cs.wisc.edu

[3] Email: fischer@cs.wisc.edu

[4] Email: liblit@cs.wisc.edu

doi:10.1016/j.entcs.2006.09.023

the relationships between different software components. When extending or fixing existing code the programmer may spend hours or days merely figuring out what other parts of the program these changes may affect. Moreover, the changes the programmer must make may be scattered across several program components. For this reason, numerous software management tools have been developed to assist in visualizing program properties. Others allow a programmer to navigate a project easily and to make automatic changes across multiple files.

As automated theorem proving matures, the proportion of old proofs to new as well as their size will continue to grow. Tools to visualize, understand, and automatically change these proofs will become vital. Integrated proof environments (IPEs) [5] should incorporate these tools in the same manner as IDEs.

In the following sections we discuss several techniques useful in software development that can be extended to theorem proving. These techniques are navigation by derivation, multiple views, automatic refactorings, and proof visualization in the large.

2 Navigation by Derivation

Formal proofs, even relatively simple ones, are necessarily very large. For example, a formalization of the Sudoku puzzle and an accompanying solution procedure in Coq [35] required approximately 5000 lines. A formal proof of the four color theorem [12, 41] took about 60,000 lines and a few years to develop. Sophisticated automated proof assistants have been developed to assist in the construction of such proofs using *tactics*. These tactics may be manually selected by the user or automatically chosen by the proof assistant. The structure of a proof object generated by these tactics may be difficult for a user to predict even when the user has selected the tactic. When a tactic is selected automatically the structure may be further obscured. The proof objects themselves may be far too large to be easily read. For example, the Sudoku development mentioned above contains a proof that the permutation relation on two lists is invertible. That is, where a pair of lists are permutations of each other, and the head elements of the lists are equal, the tails of the two lists must also be permutations of each other. About ten lines of tactics are required to complete the proof of the theorem, but at roughly 750 lines the generated proof is two orders of magnitude larger. Nonetheless, there are many occasions on which it becomes necessary to study such proofs. A tactic implemented in a proof assistant may not be working as expected; it may be necessary to inspect proof objects themselves in order to debug the tactic. A user may be developing a proof specifically to exploit a proof assistant's extraction mechanism and may need to inspect the proofs to understand why the extracted code is inefficient or, in some cases, non-existent [8]. It may be necessary to rediscover what auxiliary theorems were used to prove a given theorem; such auxiliary theorems may be selected without the user's intervention by a proof assistant with support for automation.

[5] The authors would like to thank one of the anonymous reviewers for acquainting them with this term.

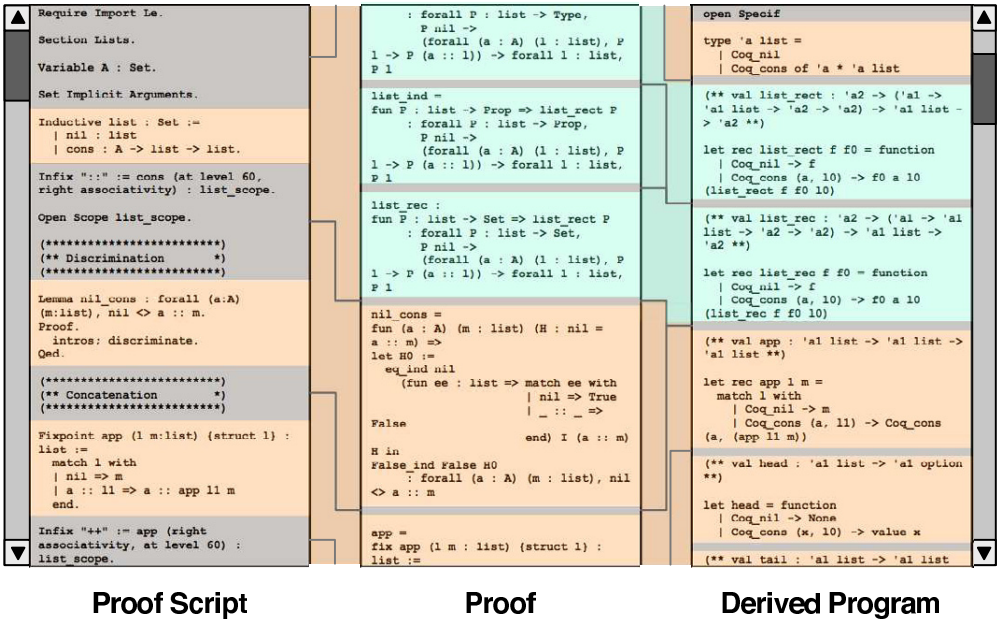


Fig. 1. Overall structure of a three panel proof navigation tool. The proof is taken from the Coq List library, one of the standard libraries in the Coq distribution. The scroll bars on the left and right allow the user to navigate the proof script and the derived program respectively.

Most programmers are familiar with the Unix *diff* utility which identifies the textual differences between two files. A number of visual tools exploit an underlying diff tool. For example, the Eclipse Compare view allows the user to compare up to three files. The tool automatically aligns the differences between the files and matches corresponding parts using visual cues. This technique, using visual cues to identify associated entities, can be extended to other domains. For example, a proof developer will often have two perspectives on a given proof. The first perspective consists of the definitions and theorems along with their corresponding tactics. The second perspective consists of the same definitions and theorems, this time associated with their proofs. There is a correspondence between the tactics and the terms of the proof. This correspondence differs from that arising in file comparison. In one way it is more straightforward since the proof has a formal relationship to the tactics whereas in a file comparison the relationship between the files must be discovered by an heuristic. However, the correspondence is also more complex. One tactic may correspond to multiple terms in a proof. Hence, an interactive tool which allows the user to select a tactic or group of tactics and responds by highlighting the associated terms in a proof would be a valuable aid to proof understanding.

A number of theorem provers, e.g., PX [13], Minlog [22], Isabelle/HOL [23], NuPRL [24] and Coq [34], exploit the Curry-Howard isomorphism [10,40] to offer a *program extraction* facility [19,20,27]. A program extraction facility automatically generates programs from proofs. In the extraction process the logical parts of a proof are deleted and the computational parts are translated into the source code of the

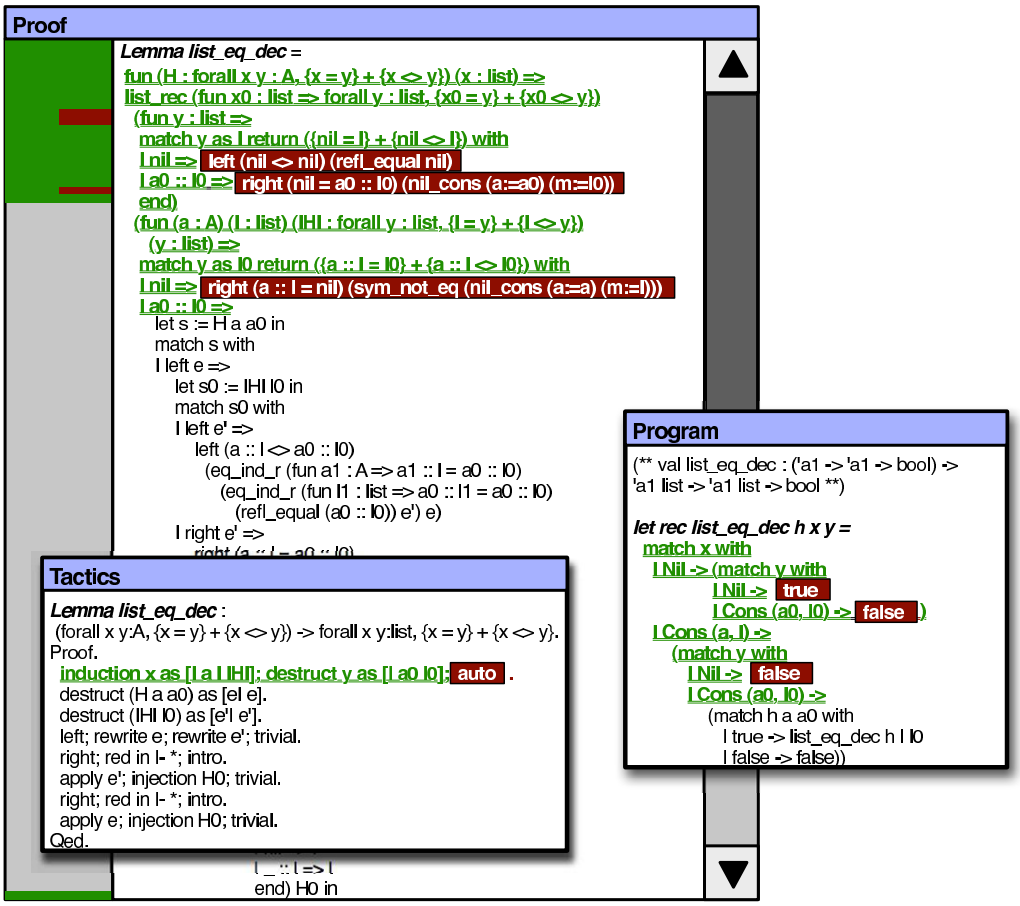


Fig. 2. Proof of the decidability of equality on lists. The *Tactics* pane on the left displays the proof tactics while the *Program* pane on the right displays the extracted program. The *Proof* pane displays the proof proper.

target language. Programs extracted from the proofs of their desired properties are known as *certified* programs. As long as the extraction facility and proof checker are themselves correct, a certified program is guaranteed to be a correct implementation of its specification, i.e., the proof from which it is extracted. Generally, the extracted programs are several orders of magnitude smaller than their associated proofs and much easier to understand. In the case of theorem provers with an extraction mechanism a three way association would be appropriate and useful. Figure 1 shows the overall structure of such a navigation tool.

Each component is associated with its corresponding component in the adjacent panel. Examples of proof script components are definitions or theorems with tactics, examples of proof components are definitions or proofs, examples of components in an extracted program are definitions of types or functions. Corresponding components are automatically aligned as the user focuses on different areas in the proof script or extracted program. Light gray is used for portions of the proof script that are not incorporated into the proof such as directives to the proof engine or comments. Narrow gray bars are also used to separate proof and program compo-

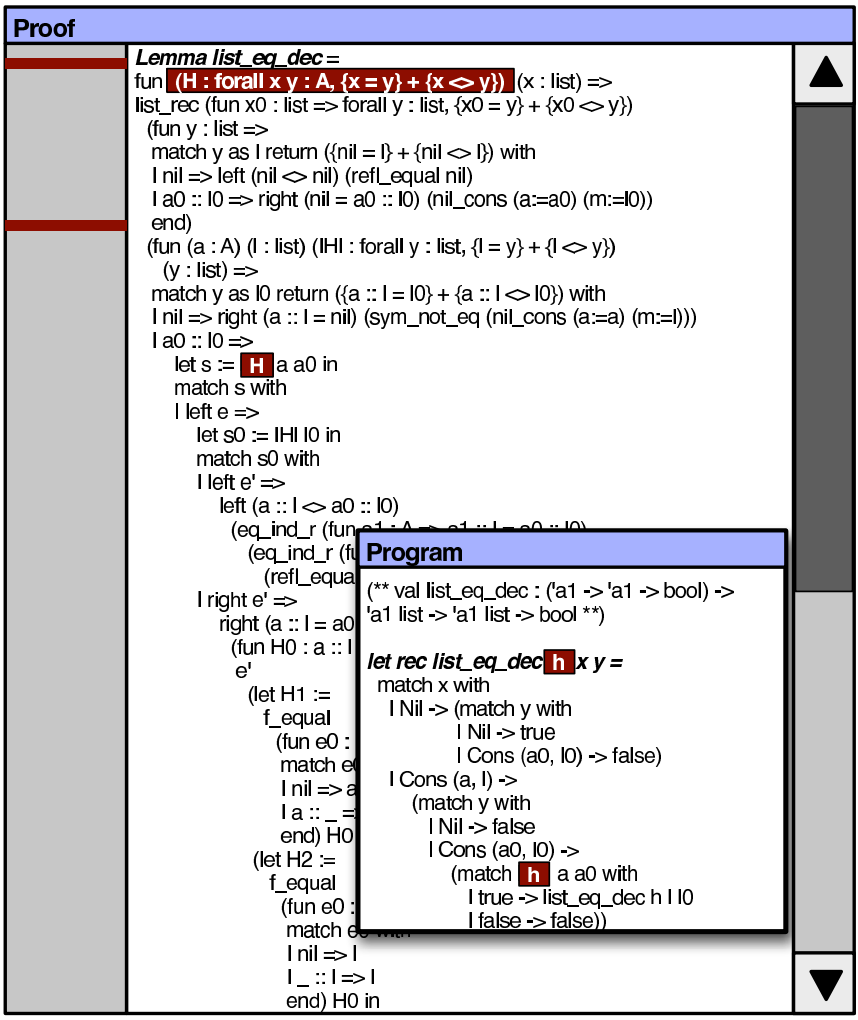


Fig. 3. Proof of the decidability of equality on lists. The user has highlighted the `h` parameter in the `list_eq_dec` function. Uses of the `h` parameter in the function and the corresponding `H` parameter of the proof are highlighted.

nents. Pale blue indicates that a component has been generated indirectly from a component in the proof script. In this example, some induction principles for the list type have been automatically generated. Some components of the proof do not have corresponding components in the extracted program. In this case the adjacent separators are merged in the program pane.

The tool in Figure 1 is useful for high-level inspection. The user may also want to examine individual proof entities in more detail. Figure 2 shows a proof and its associated tactics and program. In the *Tactics* pane on the left the `auto` tactic has been selected. Preceding tactics are green and subsequent tactics are left in black. The proof terms generated by the highlighted tactic are themselves highlighted and proof terms generated by the preceding tactics are in green. The bar on the left of the *Proof* pane summarizes the entire proof. Note that there is a green line at the bottom of the bar indicating that the last few lines of the proof are generated

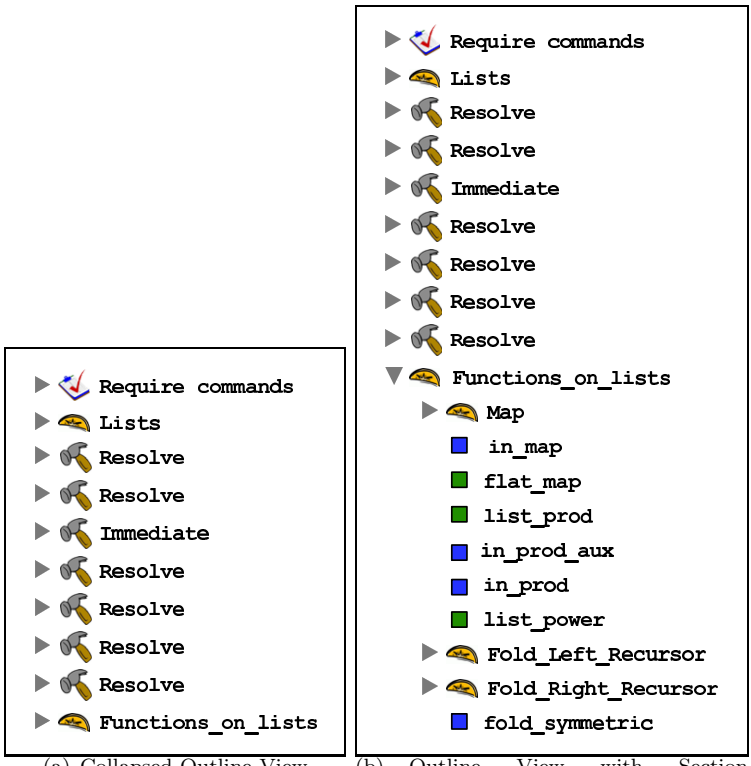
```
Require Import Le.

Section Lists.

Variable A : Set.

Set Implicit Arguments.

Inductive list : Set :=
  | nil : list
  | cons : A -> list -> list.

Infix "::" := cons (at level 60, right associativity) : list_scope.

Open Scope list_scope.

(*************************)
(** Discrimination       *)
(*************************)

Lemma nil_cons: forall (a:A) (m:list), nil <> a :: m.
Proof.
  intros; discriminate.
Qed.
```

Fig. 4. View of a proof script showing syntax highlighting. The highlighting scheme is adapted from that in the CoqIDE.

by the tactics preceding `auto`. The *Program* pane on the right shows the extracted program. The corresponding terms in the generated program are highlighted.

In the preceding example, elements in the proof were selected via the proof script. It is also possible to select these elements via the extracted program or to select elements in the program via the proof. Figure 3 shows the same proof as before. In this example, however, the user has selected an element in the *Program* pane, specifically `h`, the formal argument of the `list_eq_dec` function. Uses of `h` in `list_eq_dec` and corresponding elements in the proof are highlighted. The summary bar in the *Proof* pane indicates that there are no matches other than those visible in the text. This confirms our intuition about the proof. `h` is a function which decides whether two list elements are equal. Its corresponding proof, `H`, is a proof of the decidability of equality on list elements. `h` is applied to the head element of each list to determine whether the two are equal and in the case where the elements are equal is passed as an argument in the recursive call (otherwise `list_eq_dec` returns false). In the corresponding inductive proof we would expect that `H` is also used just once, as an hypothesis in the proof that lists are equal if their heads and their tails are equal, and we see that this is the case.

When a program is compiled with debugging enabled the compiler encodes extra information for the debugger's use in the generated object files. In particular, it stores debugger "symbol tables" [33] which are mappings between the source code and the generated object code. Using this information a symbolic debugger can execute a machine instruction and yet display to the user the corresponding source code. We envision a similar approach for a theorem prover. As the prover executes tactics to generate a proof it can store a mapping between the tactics and the generated proof object, making it available to a program navigation tool such as that described above. We have observed that the correspondence between the tactics and the proof object may be complex; but compilers and debuggers are able



(a) Collapsed Outline View (b) Outline View with Section `Functions_on_lists` expanded

Fig. 5. View of a proof script outline.

to generate and navigate the equally intricate mappings between source code and highly optimized machine code.

3 Common Conveniences

3.1 Multiple Views

Syntax highlighting, which is ubiquitous in IDEs, is available in some form in a number of proof assistants [29,34]. Figure 4 shows a Coq proof script. The various sorts of keywords are distinguished by the use of different colors, and this helps us to understand the basic structure of the small portion of the program we are looking at. When we zoom out, the syntax coloring becomes virtually useless. But this problem can be addressed by techniques already in use in a number of IDEs. For example, the Eclipse [9] Java Perspective provides an *Outline* view which allows the user to see the basic structure of an individual file at a glance. The *Outline* view is used for navigation as well. Figure 5 shows a suggested outline for the proof script of Figure 4. Another idea that could be extended directly to proof assistants is the technique of collapsing and expanding parts of a source file. Often a programmer wishes to elide certain parts of a source file that are irrelevant, so that the rest of the file becomes easier to understand. In a similar fashion a proof developer may wish

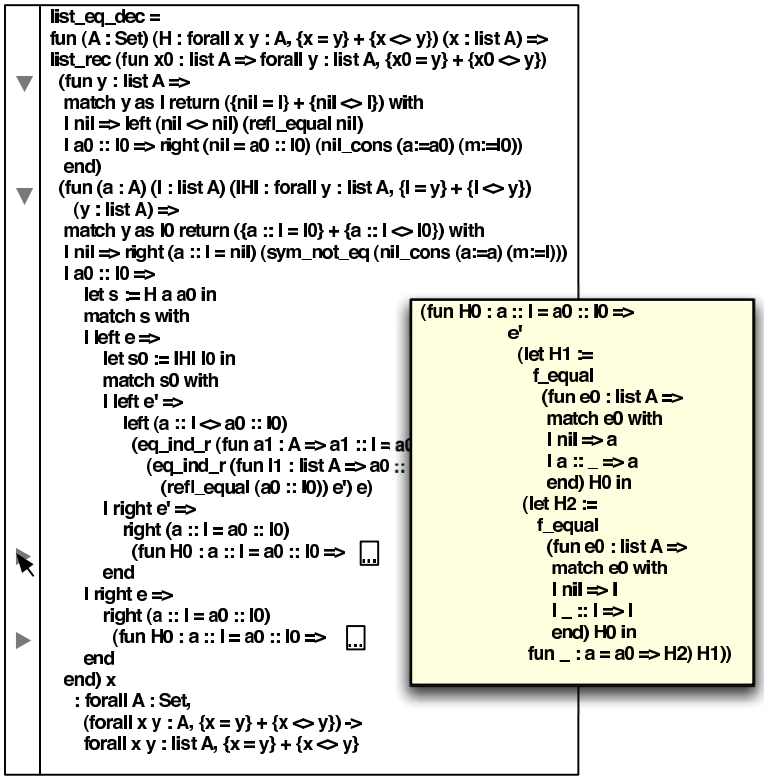


Fig. 6. A proof of the decidability of equality on lists with two functions collapsed. The collapsed function is inspected by allowing the cursor to hover over the arrow; pressing the arrow causes the function to be expanded.

to elide portions of a proof script, of a proof, or of its associated program. Figure 6 shows the proof of the decidability of equality on lists with two of the functions in the proof collapsed. The first collapsed function is a proof that equality of the heads of the lists is irrelevant under the hypothesis that the tails are unequal (in which case it is clear that the lists are unequal). The second function is a similar proof, with heads and tails reversed. Such subproofs, although required to complete a formal proof, and in some cases constituting a significant proportion of the whole proof, are generally uninteresting to the human reader.

3.2 Automatic Refactoring

A refactoring is a way of restructuring a program so that the overall organization of the program is improved but the behavior is unchanged [25]. Where large parts of a proof have been developed separately, refactoring may be necessary to make common the underlying assumptions of the different components [12]. Refactorings may also facilitate proof reuse [16]. While modern IDEs offer extensive support for automatic refactorings [30, 37, 38] UITPs offer very little. IDEs offer support for renaming of functions and variables; UITPs should offer a similar facility for renaming lemmas. IDEs offer facilities for restructuring programs; for example, a local variable may be converted to a field in a Java class definition. In the same

way, UITPs should offer facilities for restructuring existing proof scripts; in Coq, for example, a user might wish to encapsulate a group of proof entities within a module. In the Eclipse Java IDE, a developer can generalize the type of a field, lifting the field to its supertype [38] and changing all uses of the field appropriately. Similarly, UITPs should offer refactoring support for abstracting over definitions and hypotheses [28]. Many other refactorings are likely to be dependent on the logic and organization of the individual proof assistant.

Additionally we propose a requirement for transformations similar to the "best effort" standard used by IDE developers. When a developer changes the signature of a method an IDE may "do its best" by changing the signature of all overriding and overridden methods appropriately. However, if the signature is changed by the addition of a formal parameter, it will generally be impossible to automatically determine the actual parameter to be passed at the invocation site. After the transformation the resulting type mismatch will induce compiler errors in the program. However, the IDE has eased the programmer's task by automatically performing a task that the programmer would otherwise need to perform manually. The programmer can complete the transformation by identifying the call sites that must be changed, determining the actual parameter to be passed at each call site, and updating the code correctly. Generally, the compiler itself will assist the programmer in identifying the call sites which must be updated through specific error messages.

UITP developers may feel that an automatic transformation that makes a correct proof incorrect is simply unacceptable. We argue that if the transformation gets the proof developer "closer" to the correct proof that he actually desires such a "best effort" transformation is still of value and worth incorporating in a UITP. A developer may realize only after substantial work has been done on a proof that some component must be changed. For example, it may turn out to be the case that a list must have not only the familiar properties of lists but also the extra property that its elements are sorted for a proof to be completed. One method of expressing this additional property in Coq is through the use of dependent types [3]. If the developer changes the type of the list to include a proof that it is sorted then any previously developed theorems that include this list must also have their type changed. It is relatively easy to implements such a straightforward transformation. It may even be possible for a refactoring tool to modify the tactic scripts for certain proofs that do not rely on the sorted property so that the proof can be reconstructed entirely. But perhaps the developer must now construct additional lemmas to prove that the sorted property is preserved by some transformations defined in the proof. The proof cannot be completed without this additional manual work on the part of the developer. Still, a refactoring tool that automated the straightforward steps and left the developer to perform the more difficult steps that cannot easily be automated would be desirable.

4 Proof Visualization in the Large

Program visualization is a well established field. Techniques to represent programs visually are used in teaching [5, 15] and in the professional world [39] and new techniques are continually developed [18, 26, 31, 32]. These techniques incorporate both static visualization [18, 5, 39, 26, 32] and animations [15]. Often they use a complicated visual vocabulary to communicate relationships among many entities in a program.

An important insight of Ball and Eick [1] is that a less complicated visual vocabulary can also convey useful information. They show how a coloring scheme can be used to convey to the programmer the overall "shape" of an application. They use color to encode unary properties of individual lines such as the number of times a line has been changed. Such coloring can allow a programmer to see at a glance some overall property of the program. For example, parts of the system that are predominantly red are edited frequently and most likely contain bugs. Parts that are blue are edited less frequently and are likely to be relatively bug free. This approach can be extended to textual units of larger granularity such as procedures or files and has been used in applications such as fault localization [17].

Techniques for *proof* visualization are less common. Proof animations [14] exist for restricted domains such as graph properties [11]. Static visualization techniques are used to describe the relationships among proof entities [4,6]. We argue that the insights of Ball and Eick can be applied to proof visualization as well as program visualization. They can be applied in a straightforward way to encode such properties as revision information which are really identical between proofs and programs. Other properties are more specific to UITPs. In a proof assistant with an automatic component theorems may be applied without a user specifically requesting them. A coloring scheme that encoded the relative frequency with which different theorems were used could be used to visualize "hot spots" in much the same way a coloring scheme that encodes software profiling information is used.

5 Conclusion

We have described a number of ways in which techniques developed to assist programmers in maintaining and extending large programs can be of use to proof developers who must maintain and extend large proofs. Many software projects involve a considerable number of people working over several years. As the discipline of automated theorem proving matures proofs of similar size and complexity, which are now considered extraordinary [41], will grow more common. Program extraction is gaining acceptance as a technique for developing programs which must be correct. As these trends continue, the tools we have described will become more and more valuable to proof developers.

Moreover, we feel that the theoretical difficulties of developing the tools that we have described are negligible. For example, the navigation tool described in Section 2 requires an underlying encoding which records the correspondence between the

proof script, its associated proof, and the derived program. It is clear that this data is available. The relationship between the entities in a proof script and its corresponding proof must be calculated by the proof engine that develops the proof. Similarly, the relationship of the terms in a proof to the corresponding terms in the extracted program must be calculated by the program extraction mechanism. The difficulty does not lie in establishing these relationships but rather in recording them and displaying them in a useful manner.

On the other hand, work in this area may yield significant theoretical insights. The refactorings described in Section 3.2 are all quite straightforward; just a bit more sophisticated than textual replacement. Some program refactorings are much more ambitious. For instance, Tip et al. [37] describe a refactoring from Java programs that do not exploit a polymorphic type system to ones that do. More ambitious refactorings for theorem provers could very well yield unexpected insights.

Acknowledgements

We would like to thank Dr. Kenneth Kunen for his advice and encouragement.

References

[1] Ball, T. and S. G. Eick, *Software visualization in the large*, IEEE Computer **29** (1996), pp. 33–43.

[2] Bertot, Y., *The CtCoq system: design and architecture*, Formal Asp. Comput. **11** (1999), pp. 225–243.

[3] Bertot, Y. and P. Casteran, "Interactive Theorem Proving and Program Development : Coq'Art: The Calculus of Inductive Constructions," Texts in Theoretical Computer Science **XXV**, Springer, 2004.

[4] Bertot, Y., O. Pons and L. Pottier, *Dependency Graphs for Interactive Theorem Provers*, Technical Report RR-4052, INRIA (2000).

[5] *BlueJ — Teaching Java — Learning Java*, http://www.bluej.org/.

[6] Boite, O., *Proof reuse with extended inductive types*, in: K. Slind, A. Bunker and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science **3223** (2004), pp. 50–65.

[7] Caplan, J. E. and M. T. Harandi, *A logical framework for software proof reuse*, in: *SSR '95: Proceedings of the 1995 Symposium on Software reusability* (1995), pp. 106–113.

[8] Cruz-Filipe, L. and P. Letouzey, *A large-scale experiment in executing extracted programs*, in: *12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus'2005*, 2005, To appear.

[9] *Eclipse.org home*, http://www.eclipse.org/.

[10] Girard, J.-Y., P. Taylor and Y. Lafont, "Proofs and Types," Cambridge University Press, 1989.

[11] Gloor, P. A., D. B. Johnson, F. Makedon and P. Metaxas, *A Visualization System for Correctness Proofs of Graph Algorithms*, Technical Report PCS-TR92-180, Dartmouth College, Computer Science, Hanover, NH (1992).

[12] Gonthier, G., *A computer-checked proof of the Four Color Theorem*.

[13] Hayashi, S. and H. Nakano, "PX: A Computational Logic," MIT Press, Cambridge, MA, USA, 1988.

[14] Hayashi, S., R. Sumitomo and K. Shii, *Towards the animation of proofs—testing proofs by examples*, Theor. Comput. Sci. **272** (2002), pp. 177–195.

[15] *Jeliot :: Home*, http://www.cs.joensuu.fi/~jeliot/.

[16] Johnsen, E. B. and C. Lüth, *Theorem reuse by proof term transformation.*, in: K. Slind, A. Bunker and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science **3223** (2004), pp. 152–167.

[17] Jones, J. A., M. J. Harrold and J. Stasko, *Visualization of test information to assist fault localization*, in: *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (2002), pp. 467–477.

[18] Jones, J. A., A. Orso and M. J. Harrold, *Gammatella: visualizing program-execution data for deployed software*, Information Visualization **3** (2004), pp. 173–188.

[19] Letouzey, P., *A New Extraction for Coq*, in: H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, Lecture Notes in Computer Science **2646** (2003).

[20] Letouzey, P., "Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq," Ph.D. thesis, Université Paris-Sud (2004).

[21] Luo, Z., *Developing reuse technology in proof engineering*, in: *Proceedings of AISB95, Workshop on Automated Reasoning: bridging the gap between theory and practice,*, Sheffield, U.K., 1995.

[22] *Minlog system*, http://www.minlog-system.de/.

[23] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic," Number 2283 in Lecture Notes in Computer Science, Springer, 2002.

[24] *PRL Automated Reasoning Project at Cornell*, http://www.cs.cornell.edu/Info/Projects/NuPrl/.

[25] Opdyke, W. F., "Refactoring object-oriented frameworks," Ph.D. thesis, Champaign, IL, USA (1992).

[26] Panas, T., R. Lincke and W. Löwe, *The VizzAnalyzer handbook*, Technical report, Växjö University (2005).

[27] Paulin-Mohring, C. and B. Werner, *Synthesis of ML programs in the system Coq*, J. Symb. Comput. **15** (1993), pp. 607–640.

[28] Pons, O., *Proof generalization and proof reuse.*

[29] *Proof General*, http://proofgeneral.inf.ed.ac.uk/.

[30] Rajesh, J. and D. Janakiram, *Jiad: a tool to infer design patterns in refactoring*, in: *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming* (2004), pp. 227–237.

[31] Rodrigues, N., *Haskell slicing*, http://labdotnet.di.uminho.pt/HaSlicing/HaSlicing.aspx.

[32] Rodrigues, N. F. and L. S. Barbosa, *Component identification through program slicing*, in: *Proceedings of the Second International Workshop on Formal Aspects of Component Software*, 2005.

[33] Rosenberg, J. B., "How Debuggers Work: Algorithms, Data Structures, and Architecture," John Wiley & Sons, Inc., New York, NY, USA, 1996.

[34] The Coq Development Team, "The Coq Proof Assistant Reference Manual," (2004).

[35] Théry, L., *Sudoku in Coq* (2006).

[36] Théry, L., Y. Bertot and G. Kahn, *Real theorem provers deserve real user-interfaces*, in: *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments* (1992), pp. 120–129.

[37] Tip, F., R. Fuhrer, J. Dolby and A. Kieżun, *Refactoring techniques for migrating applications to generic Java container classes*, IBM Research Report RC 23238, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA (2004).

[38] Tip, F., A. Kieżun and D. Bäumer, *Refactoring for generalization using type constraints*, in: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, Anaheim, CA, USA, 2003, pp. 13–26.

[39] *Object Management Group — UML*, http://www.uml.org/.

[40] Wadler, P., *Proofs are programs* (2000).

[41] Wiedijk, F., *The seventeen provers of the world* (2005).