DISSERTATION PROPOSAL

BENJAMIN BERMAN

1. Introduction

A long time ago my cello teacher pointed out that the way to play a difficult passage of music is not simply to grit one's teeth and keep practicing but to figure out how to make playing those notes easy. The major goal of the proposed dissertation is to reapply the same idea in the context of interactive theorem proving with the Coq proof assistant[5]: I intend to show ways to make the difficult task of using Coq easier by improving the user interface. As well as solving serious usability problems for an important and powerful tool for creating machine-checked proofs, many of the techniques I am developing and testing are widely applicable to other forms of coding.

The research involved in this proposed dissertation, described more fully below, breaks down into two related main parts. Part one is the development of "CoqEdit", a new theorem proving environment for Coq, based on the jEdit text editor. CoqEdit will mimic the main features of the existing environments for Coq, but will have the important property of being easily extended using Java. Part two is the development and testing of several such extensions.

There are several points that I hope will become clear as I describe the research for the proposed dissertation below. First is that the research will make a significant positive contribution to society. While Coq is a powerful tool, and is already being used for important work, its power has come at the cost of complexity, which makes the tool difficult to learn and use. Finding and implementing better ways to deal with this complexity in the user interface of the tool can allow more users to perform a greater number, and a greater variety, of tasks. At a more general level, the research contributes to a small but growing literature on user interfaces for proof assistants. The work this literature represents can be viewed as an extension of work on proof assistants, which in turn can be viewed as an extension of work on symbolic logic: symbolic logic aims to make working with statements easier, proof assistants aim to make working with symbolic logic easier, and user interfaces for proof assistants aim to make working with proof assistants easier. These all are part of the (positive, I hope we can assume) academic effort to improve argumentative clarity and factual certainty.

In addition, generalized somewhat differently, the research will contribute to our notions of how user interfaces can help people write code with a computer. The complexities of the tool in fact help make it suitable for such research, since a) they are partly the result of the variety of features of the tool and tasks for which the tool may be used (each of which provides an opportunity for design) and b) the difficulties caused by the complexity may make the effects of good user interface design more apparent. Furthermore, although Coq has properties that make it very appealing for developing programs (in particular, programs that are free of bugs), it also pushes at the boundaries of languages that programmers may consider practical for the time-constrained software development of the "real world". However, if, as in the proposed research, we design user interfaces that address the specific problems associated with using a language, perhaps making the user interface as integral to using the language as its syntax, these boundaries may shift outward. This means that not only are we improving the usability of languages in which people already are coding, we are also expanding the range of languages in which coding is actually possible.

The second point that I hope will become clear in this proposal is that this research will be an intellectual contribution, i.e. that the project requires some hard original thinking. User interface development is sometimes "just" a matter of selecting some buttons and other widgets, laying them out in a window, and connecting them to code from the back end. While this sort of work can actually be somewhat challenging to do right (just one of the hurdles is that testing is difficult to automate), the project goes well beyond this by identifying specific problems, inventing novel solutions, and testing these solutions in studies with human subjects.

The third and final point is that this work is actually doable. Some of it has already been accomplished and the results will be described below. The remaining work I also describe below, in enough detail, I hope, to make it seem reasonably straightforward.

In the remainder of this proposal I will first give a description of Coq, including its significance, a description of current user interfaces, some examples of theorem proving using the tool, and some usability problems that I find particularly striking. I will continue with a description of a survey, and its results, on user interfaces for Coq that was sent to subscribers to the Coq-Club mailing list. Then, in the heart of this proposal, I will describe jEdit, CoqEdit, three experimental extensions to CoqEdit, and several associated user studies. I will conclude with an overview of related work and a timeline for completing the remaining work.

2. Coq and the Need for Improved User Interfaces

2.1. **Basic Theorem Proving in Coq.**

Basic theorem proving in Coq can be thought of as the process of creating a "proof tree" of inferences. The user first enters the lemma (or theorem) he or she wishes to prove. The system responds by printing out the lemma again, generally in essentially the same form; this response is the root "goal" of the tree. The user then enters a "tactic"–a short command like "apply more_general_lemma"–into the system, and the system will respond by producing either an error message (to indicate that the tactic may not be applied to the goal) or by replacing the goal with zero or more new child goals, one of which will be "in focus" as the "current" goal. Proving all of these new child goals will prove the parent goal (if zero new child goals were produced, the goal is proved immediately). Proving the current goal may be done using the same technique used with its parent, i.e. entering a tactic to replace the goal with a (possibly empty) set of child goals to prove, and which goal is the current goal changes automatically as goals are introduced and eliminated. The original lemma is proved if tactics have successfully been used to create a finite tree of descendants–i.e. when there are no more goals to prove.

One example useful in making this more clear can be found in Huet, Kahn, and Paulin-Mohring's Coq tutorial [17]. Assume, for now, that we are just using Coq's read-eval-print loop, "`coqtop`". Consider the lemma

$$(A \to B \to C) \to (A \to B) \to A \to C \tag{1}$$

where of course A, B, and C are propositional variables and "$\to$" means "implies" and is right-associative[1] (I discuss later how improved user interfaces may assist users, particularly novice users, in dealing with operator associativities and precedences). Assuming, also, that we have opened up a new "section" where we have told Coq that A, B, and C are propositional variables, when we enter this lemma at the prompt, Coq responds by printing out

```
A : Prop
B : Prop
C : Prop
_____________________________________
(A -> B -> C) -> (A -> B) -> A -> C
```

For the purposes of this example, I will write such "sequents" using the standard turnstile ($\vdash$) notation. The response then becomes:

$$A : \mathit{Prop}, B : \mathit{Prop}, C : \mathit{Prop} \vdash (A \to B \to C) \to (A \to B) \to A \to C \tag{2}$$

In general, the statements to the left of the $\vdash$, separated by commas, give the "context" in which the provability of the statement to the right of the turnstile is to be considered.[2] Another way to think about the sequent is that the statements to the left of the turnstile entail the statement to the right (or, at least, that is what we would like to prove). In this example sequent's context, the colon indicates type, so for instance "$A : \mathit{Prop}$" just means "A is a variable of type Prop" or, equivalently, "A is a proposition."

Note that this is an extremely simple example; one could actually use Coq's `auto` tactic to prove it automatically. Theorems and lemmas in Coq typically involve many types and operators besides propositions and implications. Other standard introductory examples involve natural numbers and lists, along with their associated operators and the other usual operators for propositions (e.g. negation). In fact, Coq allows users to define their own types and add axioms regarding those types, allowing users to model and reason about, for instance, the possible effects of statements in a programming language, or more general mathematics like points and lines in geometry.

[1] So this lemma is equivalent to $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

[2] Another list of statements, Coq's "environment," is implicitly to the left of the turnstile. The distinction between environment and context is that statements in the context are considered true only locally, i.e. only for either the current section of lemmas being proved or for the particular goal being proved. Generally, the environment is large, and contains many irrelevant statements, so displaying it is considered impractical.

The sequence of tactics, "`intro H`", "`intros H' HA`", "`apply H`", "`exact HA`", "`apply H'`", and finally "`exact HA`" can be used to prove the sequent above (`H`, `H'`, and `HA` are arguments given to `intro`, `intros`, `apply`, and `exact`). The first tactic, "`intro H`", operates on (2), moving the left side of the outermost implication (to the right of the turnstile) into the context (i.e. the left side of the turnstile). The new subgoal, replacing (2), is

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C \vdash (A \to B) \to A \to C \tag{3}$$

The colon in the statement "$H : A \to B \to C$" is generally interpreted differently, by the user, than the colons in "$A : Prop, B : Prop, C : Prop$". Instead of stating that "*H is of type $A \to B \to C$*", the user should likely interpret the statement as "*H is proof of $A \to B \to C$*". However, for theoretical reasons, namely the Curry-Howard correspondence between proofs and the formulas they prove, on the one hand, and terms[3] and types they inhabit, on the other, Coq is allowed to ignore this distinction and interpret the colon uniformly. If fact, in proving a theorem, a Coq user actually constructs a program with a type corresponding to the theorem. This apparent overloading of the colon operator may be a source of confusion for novice users and while there are no plans in this proposal for directly mitigating the confusion, extensions to the proposed user interface might do so by marking which colons should be interpreted in which ways. Furthermore, by clarifying other aspects of the system, we hope to free up more of novice users' time and energy for understanding this and other important aspects of theorem proving with Coq that we may not be able to address.

Tactics allow users to reason "backwards"–if the user proves the new sequents(s), then the user has proved the old sequent. In other words, the user is trying to figure out what could explain the current goal, instead of trying to figure out what the current goal entails.[4] Above, the (successful) use of the "`intro`" tactic allows the user to state that **if** in a context where A, B, and C are propositions *and* $A \to B \to C$ it is the case that $(A \to B) \to A \to C$, **then** in a context containing only that A, B, and C are propositions it is the case that $(A \to B \to C) \to (A \to B) \to A \to C$. The fact that the tactic produced no error allows the user to be much more certain of the truth of this statement than he would if he just checked it by hand.[5]

[3] "Terms," such as the "A" in "$A : Prop$", are roughly the same as terminating programs or subcomponents thereof; they may be evaluated to some final value. Note also that the types of terms are terms themselves and have their own types (not all terms are types, however). A type of the form $\Phi \to \Theta$, where both Φ and Θ are types that may or may not also contain →symbols, is the type of a function from terms of type Φ to terms of type Θ. For instance, a term of type $nat \to nat$ would be a function from natural numbers to natural numbers.

[4] Another possible point of confusion for novice users: when the differences between the old and new sequents are in the contexts, rather than the succedents (i.e. on the left of the turnstiles rather than on the right) we may say we are doing forward reasoning, even though we are still adding new goals further away from our root goal. This may make most sense when one views a sequent as a partially completed Fitch-style proof–this forward reasoning is at the level of statements within sequents, rather than at the level of sequents.

[5] In general some uncertainty remains when using computer programs to check proofs. One danger is the possibility of mistranslating back and forth between the user's natural language and the computer program's language–this might happen, for instance, if a novice user were to assume an operator is left-associative

The tactic "`intros H' HA`" is equivalent to two intro tactics, "`intro H'`" followed by "`intro HA`", so it replaces (3) with

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C, H' : A \to B, HA : A \vdash C \tag{4}$$

Next, the tactic "`apply H`" replaces (4) with *two* new subgoals:

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C, H' : A \to B, HA : A \vdash A \tag{5}$$

and

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C, H' : A \to B, HA : A \vdash B \tag{6}$$

This successful use of "`apply H`" says that the proof H, that $A \to (B \to C)$, (parentheses added just for clarity) can be used to prove C, but, in order to do so, the user must prove both A and B. Note that, in contrast with use of the `intro` tactic, after using the `apply` tactic the contexts has not changed. Also note that the first of these two becomes the current goal.

The next tactic, "`exact HA`," eliminates (5), not replacing it with any new goal (if there is already proof of `A`, in this case `HA` in the context, then there is nothing left to do; "`apply HA`" would have the same effect), and focus moves automatically to (6). The tactic "`apply H'`" replaces (6) with a new goal, but this new goal is identical to (5) (we can use $A \to B$ to prove B if we can prove A), and so "`exact HA`" can be used again to to eliminate it. Since there are no more goals, the proof is complete.

2.2. **Coq's Significance.**

The example above is intended to give some sense of what interactive theorem proving with Coq is all about, and the complexities that novice users face, but it barely scratches the surface of Coq's full power and complexity. It also does little to suggest Coq's significance. Most of the applications accounting for this importance can be divided into those relating (more directly) to computer science and those relating to mathematics.[6]

On the computer science side, Coq has an important place in research on ensuring that computer software and hardware is free of bugs. Given the increasing use of computers in areas where bugs (including security vulnerabilities) can have serious negative consequences (aviation, banking, heath care, etc.), such research is becoming increasingly important. Given, also, that exhaustive testing of the systems involved in these areas is generally infeasible, researchers have recognized the need to actually prove the correctness of these systems (i.e. that the systems conform to their specifications). While fully-automatic SAT solvers (for propositional satisfiability) and SMT (satisfiability modulo theory) solvers are

when it is actually right-associative. Another related danger, perhaps even more serious, is the possibility of stating the wrong theorem, or set of theorems. For instance, a user might prove that some function, f, never returns zero, but that user might then forget to prove that some other function, g, also never returns zero. An important role of theorem prover user interfaces is to mitigate these dangers by providing clear feedback and by making additional checks easier (e.g. quickly checking that $f(-1) = 1$, $f(0) = 1$, and $f(1) = 3$ might help the user realize that the property of f that he actually wants to prove is that its return value is positive, not just nonzero).

[6] See the categorization of user contributions on the Coq website: http://coq.inria.fr/pylons/pylons/contribs/bycat/v8.4

being used to implement advanced static analysis techniques with promising results (e.g. [16, 12]) and can determine the satisfiability of large numbers of large formulas, keeping humans involved in the theorem proving process allows the search for a proof to be tailored to the particular theorem at hand, and therefore allows a wider range, in a sense, of theorems to be proved. Furthermore, contrary to what might have been suggested by the step-by-step detail of the example above, many subproblems can be solved automatically by Coq and other interactive theorem provers, and work is being done to send subproblems of interactive theorem provers to automatic tools [9] in order to combine the best of both worlds. Notable computer science-related achievements, some in industrial contexts, for Coq and other interactive theorem provers include verification of the seL4 microkernel [18] in Isabelle[2], the CompCert verified compiler[20] for Clight (a large subset of the C programming language) in Coq, Java Card EAL7 certification[13] using Coq, and, at higher levels of abstraction, verification of the type safety of a semantics for Standard ML [19] using Twelf[6] and use of the CertiCrypt framework [1] built on top of of Coq to verify cryptographic protocols (e.g. [7]).[7]

On the mathematics side, Coq is being used to formalize and check proofs of a variety of mathematical sub-disciplines, as demonstrated by user contributions listed on the Coq website. Perhaps Coq's most notable success story is its use in proving the Four Color Theorem [15]. Other interactive theorem provers are also having success in general mathematics. For instance, Matita [3], which is closely related to Coq, was used in a proof of Lebesgue's dominated convergence theorem [11]. There are in fact efforts to create libraries of formalized, machine-checked mathematics, the largest of which is the Mizar Mathematical Library [14]. ITPs are also a potential competitor for computer algebra systems (e.g. Mathematica) with the major advantage that they allow transparency in the reasoning process, a significant factor limiting computer algebra use in mathematics research according to [10].

The potential for transparency also helps make interactive theorem provers, like Coq, a potentially useful tool in mathematics, logic, and computer science education. Rather than simply giving students the answers to homework problems, interactive theorem provers might be used to to check students' work, find the precise location of errors and correct misconceptions early. Interest in adapting theorem provers for educational purposes can be seen in many references listed later in this document; Benjamin Pierce et al.'s *Software Foundations*[23], a textbook, written mostly as comments in files containing Coq code and which includes exercises having solutions that may be checked by Coq, serves as an example of how the tool can be effectively used in education. More general interest in educational systems that check student work can be seen in logic tutorial systems such as "P-Logic Tutor" [22], "Logic Tutor" [21], "Fitch" (software accompanying the textbook *Language, Proof, and Logic* [8]), and "ProofMood"[4].

Why Coq over other ITPs? ...

[7] An earlier version of this paragraph, from which come most of the included references, was written by Dr. Aaron Stump for an unpublished research proposal. Many of the references from the next paragraph also come from this proposal.

3. Conclusion

I hope to have made several points in this proposal. First, that this is important work, both because the Coq interactive theorem prover is an important tool that could benefit significantly from improved user interfaces and because many of the ideas generalize to other forms of coding. Second, that as an intellectual challenge this work is non-trivial, not only because of the normal programming problems that must be overcome but because designing good user interfaces for complicated systems, which includes the identification of tractable problems and the testing of potential solutions, is non-trivial. Finally, that, despite this non-trivial nature, the work can be accomplished.

References

[1] CertiCrypt: Computer-Aided Cryptographic Proofs in Coq. `http://certicrypt.gforge.inria.fr/`.

[2] Isabelle. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`.

[3] Matita. `http://matita.cs.unibo.it/`.

[4] ProofMood. `http://www.proofmood.com/`.

[5] The Coq Proof Assistant. `http://coq.inria.fr`.

[6] Twelf. `http://twelf.org/wiki/Main_Page`.

[7] Gilles Barthe, Daniel Hedin, Santiago Zanella Béguelin, Benjamin Grégoire, and Sylvain Heraud. A machine-checked formalization of sigma-protocols. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 246–260. IEEE, 2010.

[8] Jon Barwise, John Etchemendy, Gerard Allwein, Dave Barker-Plummer, and Albert Liu. *Language, proof and logic*. CSLI publications, 2000.

[9] Sascha Böhme and Tobias Nipkow. Sledgehammer: judgement day. In *Automated Reasoning*, pages 107–121. Springer, 2010.

[10] Andrea Bunt, Michael Terry, and Edward Lank. Friend or foe?: examining cas use in mathematics research. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 229–238. ACM, 2009.

[11] Claudio Sacerdoti Coen and Enrico Tassi. A constructive and formal proof of lebesgues dominated convergence theorem in the interactive theorem prover matita. *Journal of Formalized Reasoning*, 1:51–89, 2008.

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *Static Analysis*, pages 236–252. Springer, 2011.

[13] NV Gemalto. Gemalto achieves major breakthrough in security technology with javacard highest level of certification. *Press release at http://www.gemalto.com/php/pr_view.php?id=239*.

[14] Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[15] Georges Gonthier. A computer-checked proof of the four colour theorem. *preprint*, 2005.

[16] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *ACM SIGPLAN Notices*, volume 43, pages 281–292. ACM, 2008.

[17] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant a tutorial. *Rapport Technique*, 178, 1997.

[18] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[19] Daniel K Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In *ACM SIGPLAN Notices*, volume 42, pages 173–184. ACM, 2007.

[20] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[21] Leanna Lesta and Kalina Yacef. An intelligent teaching assistant system for logic. In *Intelligent Tutoring Systems*, pages 421–431. Springer, 2002.

[22] Stacy Lukins, Alan Levicki, and Jennifer Burg. A tutorial program for propositional logic with human/computer interactive learning. In *ACM SIGCSE Bulletin*, volume 34, pages 381–385. ACM, 2002.

[23] Benjamin C Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjoberg, and Brent Yorgey. Software foundations. *Course notes, online at http://www. cis. upenn. edu/˜ bcpierce/sf*, 2010.