Dissertation Proposal:
User Interfaces for the Coq Proof Assistant

Benjamin Berman

July 14, 2013

Contents

Chapter 1

Introduction

The general principle behind the dissertation that I propose in this document is that in order to accomplish difficult tasks, one generally needs to make these tasks easy–to move boulders, try a lever. The significance of this principle was demonstrated to me when, 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 also figure out how playing those notes could, for instance, be made less physically awkward by changing the position of one's elbow. In general, when it comes to "virtuosic" tasks–tasks that require large amounts of skill–it is easy to ignore this "making things easy" principle and focus on putting more time and effort into practicing or studying, even though following the principle is often a requirement for success. The major goal of the proposed dissertation is to apply the principle in the context of the virtuosic tasks that are involved in interactive theorem proving with the *Coq* proof assistant[10][1]: 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.

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 also 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[10][2]: 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

[1] "Proof assistant" and "interactive theorem prover" are synonymous

[2] "Proof assistant" and "interactive theorem prover" are synonymous

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.

Chapter 2

Coq and the Need for Improved User Interfaces

2.0.1 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 [56]. Assume, for now, that we are just using Coq's read-eval-print loop, "`coqtop`". Consider the lemma

$$(A \to B \to C) \to (A \to B) \to A \to C \tag{2.1}$$

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,

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

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:

$$A : Prop, B : Prop, C : Prop \vdash (A \to B \to C) \to (A \to B) \to A \to C \tag{2.2}$$

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 "$A : Prop$" 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's *Gallina* language allows users to declare or define variables, functions, types, constructors for types, axioms, etc., 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.

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 (2.2), 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 (2.2), is

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C \vdash (A \to B) \to A \to C \tag{2.3}$$

[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 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 $\to$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 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

The tactic "`intros H' HA`" is equivalent to two intro tactics, "`intro H'`" followed by "`intro HA`", so it replaces (2.3) with

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C, H' : A \to B, HA : A \vdash C \tag{2.4}$$

Next, the tactic "`apply H`" replaces (2.4) with *two* new subgoals:

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C, H' : A \to B, HA : A \vdash A \tag{2.5}$$

and

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C, H' : A \to B, HA : A \vdash B \tag{2.6}$$

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 (2.5), 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 (2.6). The tactic "`apply H'`" replaces (2.6) with a new goal, but this new goal is identical to (2.5) (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.

2.0.2 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

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. [50, 39]) 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 [29] 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 [60] in Isabelle[4], the CompCert verified compiler[68] for Clight (a large subset of the C programming language) in Coq, Java Card EAL7 certification[45] using Coq, and, at higher levels of abstraction, verification of the type safety of a semantics for Standard ML [66] using Twelf[11] and use of the CertiCrypt framework [2] built on top of of Coq to verify cryptographic protocols (e.g. [20]).[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 [47]. Other interactive theorem provers are also having success in general mathematics. For instance, Matita [6], which is closely related to Coq, was used in a proof of Lebesgue's dominated convergence theorem [36]. There are in fact efforts to create libraries of formalized, machine-checked mathematics, the largest of which is the Mizar Mathematical Library [46]. 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 [32].

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*[83], 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" [70], "Logic Tutor" [69], "Fitch" (software accompanying the textbook *Language, Proof, and Logic* [21]), and "ProofMood"[8].

[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.

The part of the case for Coq's significance that is presented above is more a case for interactive theorem provers in general than Coq in particular; after reading it one may wonder, why try to improve Coq usability instead of usability for some other proof assistant? The answer is that it is already one of the most powerful and successful such tools. Adam Chlipala, in the introduction to his book *Certified Programming with Dependent Types* [35], presents a list of major advantages over other proof assistants in use: its use of a higher-order language with dependent types, the fact that it produces proofs that can be checked by a small program (i.e. it satisfies the "de Bruijn criterion"), its proof automation language, and its support for "proof by reflection." [8] As evidence of its resulting success, note that Coq was awarded the 2013 ACM SIGPLAN Programming Languages Software Award [90].

2.0.3 Current User Interfaces and Problems They Present to Novice Users

The example presented earlier can be used to illustrate some more of the common challenges for users. For novice users, one of the biggest challenges is to discover exactly what Coq's tactics do when applied to various arguments and goals. Only four tactics were used in the example, but many more are standard (the Coq Reference Manual[77] lists almost 200 in its tactics index), and Coq allows new tactics to be defined. Other challenges, for both novice and expert users, will be discussed below, but the lack of support for users trying to understand tactic effects is, by itself, probably sufficient justification for the development of new user interfaces.

The two major user interfaces for Coq are currently *Proof General*[7, 16] and *CoqIDE* (which is available from the Coq website[10], and is bundled with Coq). Interacting with Coq using one is quite similar to interacting with Coq using the other, the main difference being that Proof General is actually an Emacs mode (and so has the advantages and disadvantages of the peculiarities of the Emacs text editor, e.g. numerous shortcuts and arguably a steep learning curve).

[8] Basically, this is proof by providing a procedure to get a proof. Coq allows one to prove that these procedures produce correct proofs.

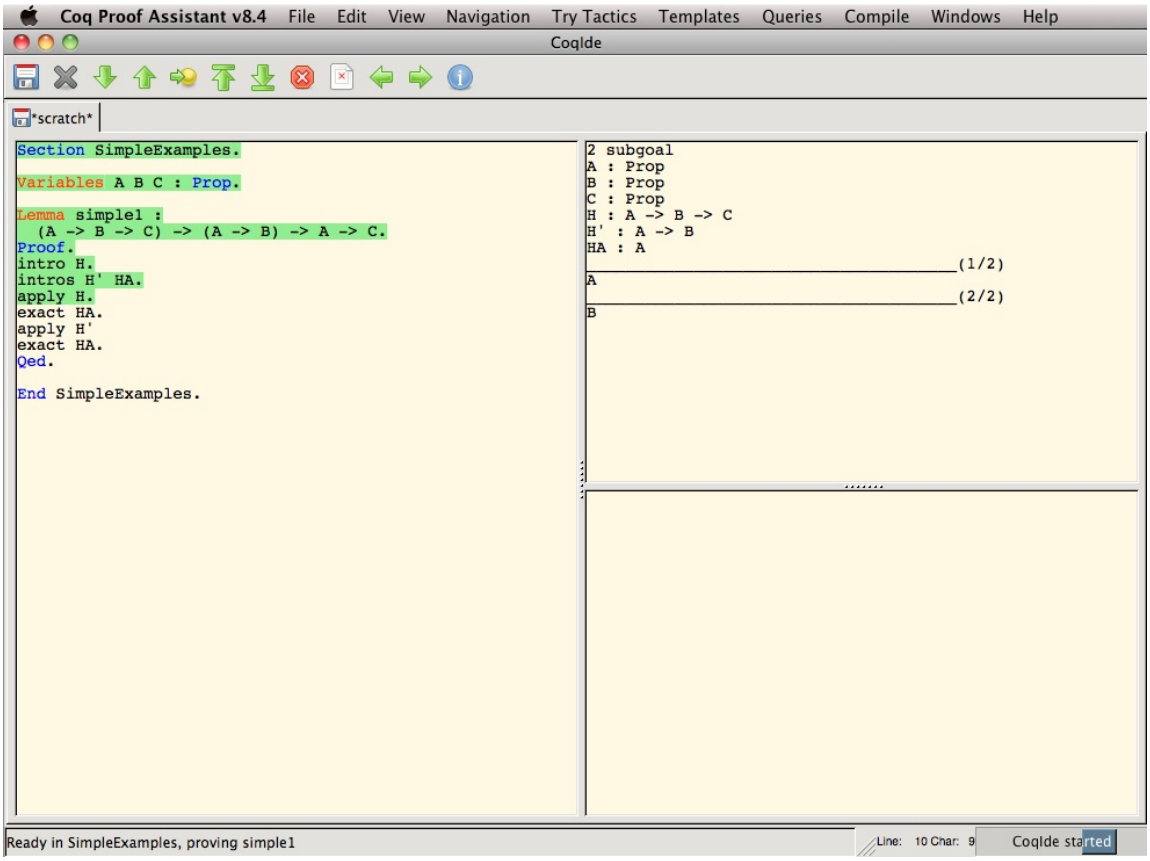

Figure 2.1: CoqIde, displaying the result of entering the tactic "`apply H`" in the top-right panel within the proof of `(A -> B -> C) -> (A -> B) -> A -> C`.

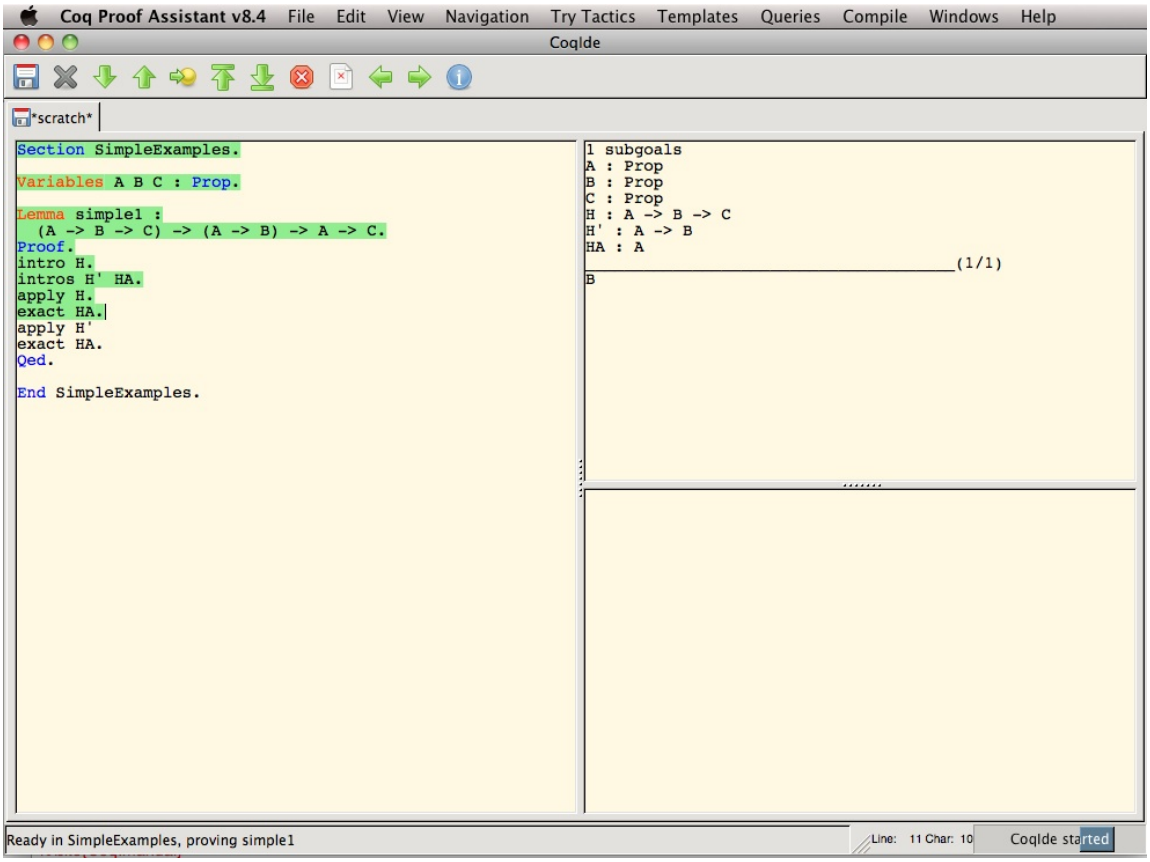

Figure 2.2: CoqIde after moving the end of the evaluated portion of the script forward by one sentence from the state shown in Figure 2.1.

Figure 2.1 and Figure 2.2 show the CoqIde user interface as it appears while entering the proof from the the earlier example, that $(A \to B \to C) \to (A \to B) \to A \to C$, into Coq.[9] The larger panel on the left shows a script that will, in general, contain definitions, theorems, and the sequences of tactics used to create proofs of these theorems.[10] A portion

[9]Note that "`simple1`", in "`Lemma simple1 : (A -> B -> C) -> (A -> B) -> A -> C.`", is the identifier we are binding to the *proof* of the lemma, and not to the lemma itself. Without recognizing this, the fact that the keyword "`Lemma`" could have been replaced by the keyword "`Definition`" may be yet another source of confusion since it suggests that Coq thinks lemmas and definitions are basically the same thing! As one might expect, we could also bind an identifier to the lemma itself. If we were to bind the identifier "`SimpleLemma`" to this lemma, we would most likely use the `Definition` keyword in combination with "`:=`", and write
"`Definition (SimpleLemma : Prop) := (A -> B -> C) -> (A -> B) -> A -> C.`"

[10]Here the declaration of A, B, and C, and the definition of the proof of the lemma are within a section

of this script, starting at the beginning, may be highlighted in green to show that it has been successfully processed by Coq. Another portion of the script, following this green highlighting or starting at the beginning if there is no green highlighting, may be highlighted in blue to show where the "sentences" of the script are either being evaluated or have been queued for evaluation (sentences in the script are separated by periods followed by whitespace, as in English). Whenever Coq is not already processing a sentence, and there are queued sentences, the first sentence in the queue is automatically dequeued and sent to Coq, i.e. if there is a first blue-highlighted sentence, Coq is trying to evaluate it.

If a sentence is successfully processed, its highlighting changes to green and the output resulting from the successful processing is printed in one of the two panels on the right side of the window. Assuming the system is in "proof mode" (e.g. after processing the sentence "`Lemma simple1`..." in Figure 2.1 and Figure 2.2, but before evaluating "`Qed.`"), the top panel displays the current goal, including its context, followed by just the consequents of any remaining goals. The bottom panel is used to display various messages, e.g. error message and acknowledgements of successful definitions. Otherwise, if processing of a sentence results in an error, all sentences queued for processing are removed from the queue, the blue highlighting representing that queue is removed, and the font of the offending part of the offending sentence is changed to bold, underlined red. In general, processing a sentence is not guaranteed to produce a result of any kind (error or otherwise) in any specified amount of time (some sentences are semi-decision procedures), so CoqIde allows users to interrupt the processing of a sentence. This also has the effect of removing all sentences from the processing queue and removing all blue highlighting. Frequently, however, processing is fast enough that the blue highlighting never even becomes actually visible.

Users can extend the highlighted region both forward, to evaluate unhighlighted sentences, and backwards, to undo the effects of evaluation. Users can instruct the system to extend the highlighting forward by one sentence, to retract it back by one sentence (though in the latest version of Coq, this sometimes will actually move the highlighting back several sentences), to extend or retract it to the cursor, to remove all of it (i.e. retract it to the start of the script), and to extend it to the end of the script. These instructions can be entered into the system using toolbar buttons, drop-down menu items, or keyboard shortcuts.

Figure 2.1 and Figure 2.2 illustrate some of these points. In Figure 2.1, in the top right, we see the the result of evaluating "`apply H`". In Figure 2.2, we see the highlighting extended and the result of "`exact HA` in the top right–the elimination of the first of the two subgoals in Figure 2.1 and the change in focus to the second.

This interface is problematic for novices trying to learn the effects of tactics. Unfortunately, because the particular example being discussed is so simple, the severity of the

---

that has been named "`SimpleExamples`." This sets the scope A, B, and C to just the section. Outside of the section, reference to simple1 is allowed. However, simple1 is changed to a proof that $\forall(A : Prop), (\forall(B : Prop), (\forall(C : Prop), ((A \to B \to C) \to (A \to B) \to A \to C)))$, generally written `forall A B C : Prop, (A -> B -> C) -> (A -> B) -> A -> C`.

problem may not be immediately apparent. In Figure 2.1, the two goals resulting from using the tactic "apply H" (the statement at the end of the highlighted region) appear to be displayed in the top right panel. In fact, as mentioned earlier, only the first goal is fully displayed–the context for the second is not. (In this case, the contexts for both are identical, but this is not always what happens). To see the context of the second goal, probably the easiest, or at least most natural, thing for the user to do is highlight forward through all the tactics used to prove the first goal (just one tactic here, but potentially many in general). It is up to the user to determine how far to highlight (or un-highlight, if looking at an earlier sibling goal) by keeping track of the list of goals in the top-right panel.

In addition to the problem of moving through the script to fully see siblings, note that no distinction is made between sibling and non-sibling goals in the list presented. For instance, instead of using "`exact HA`" to transition to Figure 2.2, the user could have used a tactic that produced two new goals. The list of goals would then contain three goals, but only the first two would be siblings. The user interface leaves it up to the user, however, to determine this by keeping track of the number of goals.[11]

Proof general does introduce a few features not present in CoqIde. For instance, instead of making the highlighted region un-editable ("locking" it), typing in the highlighted region retracts the highlighting back to the end of the sentence immediately before the cursor. Unfortunately, these features are not really aimed at showing the effects of tactics. A third user interface, *Proof Web*[9] does make a serious attempt. ProofWeb, for the most part, is a web-based version of CoqIde. However, it has a major improvement, shown in the bottom right of Figure 2.3: a visualization of the partially completed proof tree.

ProofWeb's display of the tree follows the convention where inferences are drawn with a horizontal line separating horizontally listed premises, above, from the conclusion below, and where each horizontal line is labeled with the name of the corresponding inference rule (or, in the case of Coq, by the corresponding tactic name) to the right of the line. These inferences can be chained together so that the root of the proof tree is drawn at the bottom and the leaves are drawn at the top. As an example, the portion of the proof tree constructed by ProofWeb that corresponds to "`apply H`" is shown in Figure 2.4 (the ellipses indicate that the child nodes are still unproved), and Figure 2.5 fully displays the partially completed tree. The user is able to much more directly see the goal to which `apply H` is applied and the goals this application produces.

[11] This sort of debugging gets even harder when one introduces proof automation features that allow combinations of basic tactic use attempts.

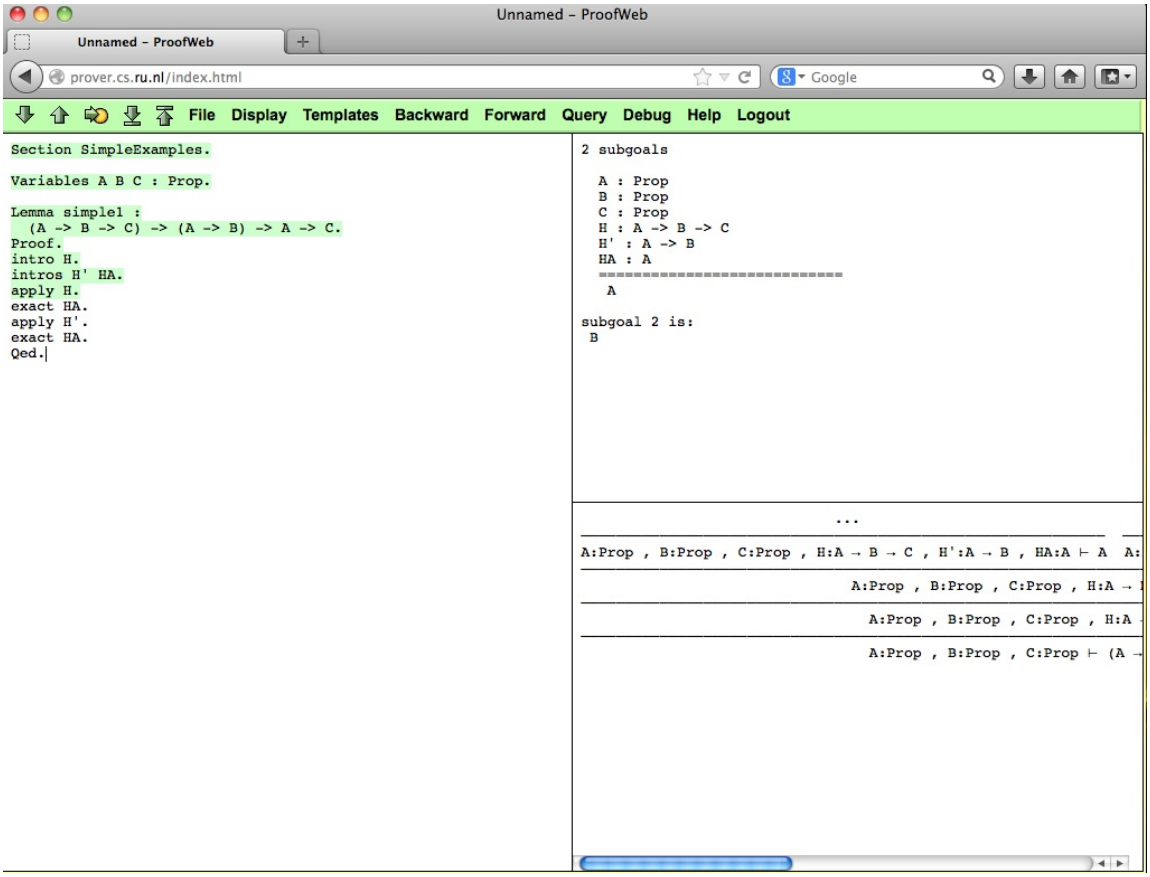

Figure 2.3: ProofWeb, with a partial proof tree displayed in the bottom right.

Unfortunately, as one can probably already tell, this sort of visualization does not scale particularly well.[12] Contexts may may have dozens of items, many of which may be much longer than "`H : A -> B -> C`, and the number of nodes in the proof tree may also be very large. As a result, to see the effect of a tactic the user may have to pan around the window; this is especially likely if one is looking at a tactic used near the root of the tree, since the width of the tree at its leaves forces apart nodes near the root. Even if the user does not need to pan (ProofWeb has a feature that allows the tree to be displayed in a separate window, which can sometimes make panning unneccessary), the distances at which nodes with sibling and parent-child relationships must sometimes be placed may make it difficult for the user to compare such sequents and to determine if a direct relationship in fact exists (e.g. determine if two sequents that printed next to one

[12]Later in this document I provide what I hope is a clearly better alternative.

another are siblings or "cousins"). The latter task is possibly especially difficult using this visualization since it involves checking for gaps in co-linear line segments and the human brain tends to connect such lines.[13] The proof tree visualization, especially if there is a need to pan, is not particularly helpful in showing the location of the current goal (users may have to search the leaves of the tree to find the leftmost ellipsis).

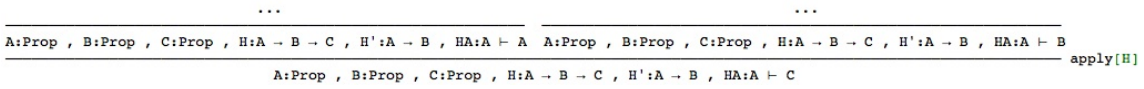

Figure 2.4: The portion of ProofWeb's tree visualization corresponding to the tactic "`apply H`".

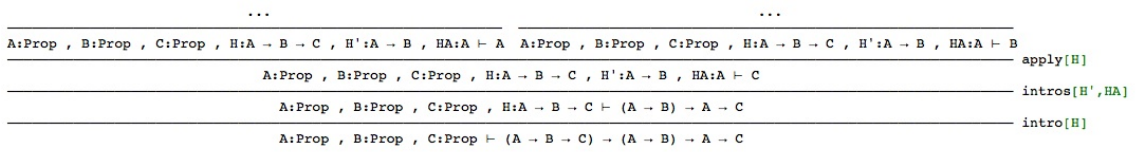

Figure 2.5: The partially completed tree from Figure 2.3, fully displayed.

The problems with current user interfaces that are discussed above are with respect to the scenario in which a novice user is inspecting an existing proof in order to determine the effects of various tactics under various conditions. Many other scenarios with overlapping and related challenges also exist. Such challenges include, but are not limited to,

- locating particular items in a context (or in the larger environment)
- finding similar nodes in a proof tree
- deciding which tactic to apply
- keeping track of different proof attempts
- optimizing a proof or organizing a set of definitions, theorems and proofs for human understanding
- doing all these things efficiently.

[13] This is the Gestalt law of "good continuation; see, for instance, [44].

2.0.4 Coq User Interface Survey

Section 2.0.3 described some usability problems that I, as a novice Coq user, noticed. In this section, I describe an online survey (and its results) that Professor Juan Pablo Hourcade, Professor Aaron Stump, and I, in December 2011, invited subscribers to the Coq-Club mailing list to fill out. This survey asked Coq users for their opinions and experiences regarding existing Coq user interfaces, and for their ideas regarding new interfaces. Our motivation was both to validate our own ideas about new Coq user interfaces and to generate new ones. We received 48 responses, including many detailed responses to the essay questions in the survey.

The survey consisted of 19 questions, of which 13 were multiple choice and the rest short answer or essay. The questions can be divided into three groups: 7 questions asking for background information on the respondent and how the respondent uses Coq, 9 questions asking for various ratings of the interface respondents use, and 3 open-ended questions directly related to the development of new user interfaces. To these open-ended questions, we received many lengthy and thoughtful responses. [14]

The responses to the first group of questions showed a full range of (self-reported) Coq expertise levels, although a majority of responses indicated a high degree of expertise (on a scale going from 1=novice to 5=expert, 2 respondents rated themselves at level 1, 12 at level 2, 9 at level 3, 17 at level 4, and 8 at level 5). 9 respondents indicated they had been using Coq for less than 1 year, 27 for 1-5 years, 7 for 5-10 years, and 5 for more than 10 years. Users of Proof General outnumbered users of CoqIDE 31 to 16. 24 respondents indicated using Coq for programming language or program verification research, 10 indicated using Coq for formalization of mathematics, and 8 indicated teaching. (There were 47 responses total to this free-response question.)

In the second group, to the question "How satisfied are you with the interface you typically use?", respondents gave a slightly positive average response (4.6 on a 1 to 7 point scale). This was somewhat surprising to us at first, but it may have been an artifact of how the the question was asked. For one thing, we did not present any sort of alternative interface, and current interfaces are, in fact, a significant improvement over the basic command prompt. A second factor may be that many respondents have become accustomed to their current interface and may have viewed the question as asking how willing they would be to learn to use a new interface. More than 25% of respondents, however, did indicate some level of dissatisfaction. Furthermore, answers to four questions revealed difficult tasks for users. These questions asked users how difficult it is, using the interface they typically use, to

- understand the relationships between subgoals,
- switch back and forth between potential proofs of a subgoal,

[14] A more detailed survey report can be found at http://www.cs.uiowa.edu/ baberman/coquisuvey.html.

- compare similar subgoals, and
- tell what options for proving a subgoal are available.

On a scale where 1="Very Difficult" and 7="Very Easy", the mean values for the answers to these questions were 2.74, 3.46, 2.35, and 2.57, respectively. Responses to a fifth question, How difficult is it for you to (mentally) parse Coq syntax?, produced a mean value of 5.02 on the same scale, with only 4 responses indicating Difficult or Somewhat difficult. Again, this may have been an artifact of the way the question was asked–how difficult, compared to what?

In the third group, we received almost 4,500 words (total) in response to the questions

- "What information would you like to have more readily available when working with Coq?",
- "What do you think are the hardest parts of learning interactive theorem proving with Coq?", and
- "What advice/requests/ideas do you have for creating better Coq user interfaces?"

Because of this volume, I categorized the responses to each question.

For the first question, the first category was "library documentation." Respondents noted that Coq's "`SearchAbout`" command is a little hard to use, that they would like more simple examples of using Coq commands, that theorem names are not very readable, and that they would like integration of documentation, a la the Eclipse IDE's javadoc support.[15] The second category was "available tactics"/"relevant lemmas, relevant definitions": respondents wanted the names of previously proved statements they could apply and, additionally, whether a tactic could be used to automatically prove either the the current goal or its negation. The third category was information on terms, e.g. the type of a term, the value to which it reduces, or other implicit information (such information can already be made available by using commands like "`Check`" and "`Print`"). The fourth category was proof structure, including information on the relationships both between goals within a proof and between theorems and definitions. Miscellaneous responses included similarities between terms, differences between terms and expected terms, and tactic debugging with custom breakpoints.

For the question "What do you think are the hardest parts of learning interactive theorem proving with Coq?", the first category of response was type theory– that learning the type theory behind Coq is one of the hardest parts. The second category had to do with lack of good tutorials. The third category was that there are numerous poorly documented commands (again, the need for simple examples was mentioned). Finally, the

[15]For the reader not familiar with Integrated Development Environments, which are essentially text editors with features specialized to programming in various languages. Eclipse[3] is one of the more popular IDEs for Java programming.

fourth category was proof readability, e.g. lack of support for mathematical notation and proof script organization.

For the question "What advice/requests/ideas do you have for creating better Coq user interfaces?", the first category was programming IDE features (e.g. auto-indentation, safe and correct renaming of identifiers, refactoring of tactics and groups of tactics, and background automation). The second category was proof structure–representing proof structure by for instance grouping sibling goals, and allowing more flexibility to the order in which one works on goals. The third category was syntax, which included having better ways to indicate where one wants to rewrite part of a term or where one wants to unfold a definition and automatic naming of hypotheses. Some miscellaneous suggestions were to make more use of the mouse, avoid unnecessary re-execution of potentially long-running commands, and to have different editing and presentation tools.

Even given the responses from self-described novice Coq users, the group of respondents is still heavily biased towards acceptance of arcane, complicated software. The responses summarized above demonstrate that, even by this group, room for improvement is seen.

Chapter 3

Proposed and Completed Research

3.1 Overview

In this section, I describe CoqEdit, three extensions to CoqEdit, and plans for evaluating two of these extensions with human participants. As an interlude in the descriptions of the extensions, I include a presentation of the development and testing "Keyboard-Card Menus" as these are a component of one of the planned extensions.

3.2 CoqEdit

CoqEdit is a new user interface of Coq, that I, along with Harley Eades and under the supervision of Professors Juan Pablo Hourcade and Aaron Stump, have been developing. CoqEdit is a plugin to the *jEdit*[5] text editor, a free and open source editor editor written in Java.[1] For the most part, interaction with CoqEdit, at least in its unextended form, imitates the style seen with Proof General and CoqIde.

Figure 3.1 shows an initial version of the plugin. This initial version is functional in that one can communicate with the `coqtop` command prompt by moving the highlighting forwards and backwards (using submenu items not seen in figure 6, or the items' shortcuts) to send messages to `coqtop`, which can then respond by changing the text printed in the two panels on the right. It also allows allows `coqtop` to be interrupted when processing long-running commands, and includes the XML files telling jEdit how to do syntax highlighting for Coq's .v ("Vernacular") files. Note that these two panels actually sit within a jEdit "dockable window", which may be hidden (i.e. collapsed so that only its labeling tab is showing), undocked (i.e. made to float as a normal window), or docked in some other

[1] Another jEdit plugin, *Isabelle/jEdit*, has been developed to provide support for the Isabelle[4] interactive theorem prover and is bundled with Isabelle itself; see, for instance, [98]. Note that it has a rather different, and arguably more advanced, "asynchronous" style of interaction. While this style of interaction may one day be available to Coq users, for now we are concentrating on other issues.

position (e.g. below the area containing text, instead of to its right).

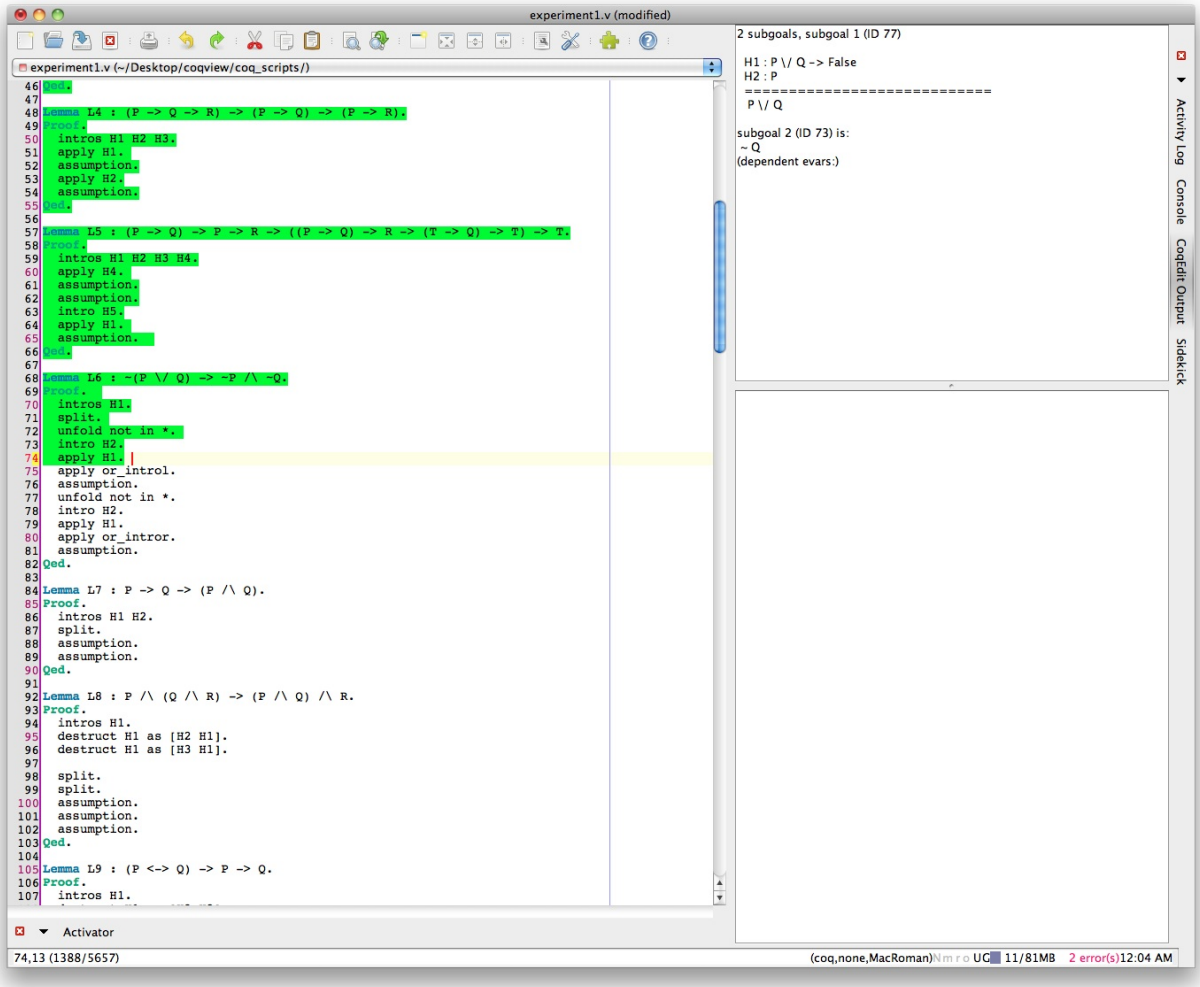

Figure 3.1: An initial version of CoqEdit.

Although it does not contain some of the language-specific features of IDEs like Eclipse, jEdit does contain several noteworthy ones, besides its support for plugins. These include syntax highlighting and auto-indentation for numerous languages, word completion, abbreviations, folding and many others (see the jEdit homepage[5], and the features page to which it links). The fact that jEdit runs on a Java Virtual Machine allows it to be used with Mac OS X, OS/2, Unix, VMS and Windows, and should also make extending

CoqEdit support to multiple platforms relatively easy (I am currently developing it just under Mac OS X, but I expect it to immediately work just as well on Linux). Particularly important for the development of plugins is its BeanShell scripting language for writing macros, essentially an interpreted version of Java that makes experimentation with the jEdit API relatively easy, especially for Java programmers. Plugins are, for the most part, actually collections of BeanShell scripts that generally invoke compiled Java code.

One of the features of jEdit's plugin system is that one can specify dependencies between plugins. If plugin A is specified as depending on plugin B, then activating[2] A in the plugin manager automatically activates B. This allows code from B to be invoked from A. This makes it possible to write extensions to CoqEdit as jEdit plugins that allow features to be added incrementally and in various combinations. In a nutshell, we can write plugins that plug into our CoqEdit plugin.

Currently, I am working on a major revision of the initial version shown in Figure 3.1. Although this revision will include bug fixes and implement a few remaining unimplemented features, its main purpose is to improve the software architecture in order to make future CoqEdit extensions easier. *This foundation for future user interface research will be one of the major contributions involved in this dissertation.* It will make taking advantage of the extensive Java libraries, particularly those for 2D graphics, and of the Java development community's expertise, much more straightforward.

This new, still quite basic, user interface will however improve upon Proof General and CoqIde in a couple of different ways. First, it will cache the output of sentence evaluation and allow a window into this cache using movable dark green highlighting of a sentence within the evaluated region (Proof General caches this output, but requires users to hover over the text with the mouse to make a tooltip with it appear). Second, it will allow both multiple instances of `coqtop` to run simultaneously *and* multiple text areas to show multiple areas within a buffer that is being evaluated.

3.3 CoqEdit Extensions

3.3.1 "Proof Previews"

Our first experimental extension, "**Proof Previews**", is shown in Figure 3.2. When the user presses CTRL-SPACEBAR while the system is in proof mode, a popup window containing a list of possible tactic/argument combinations (i.e. ones that do not immediately produce errors or take especially long to run) appears just below the end of the evaluated section. The user can use the up and down arrow keys change the selected item in the list and the ENTER (or "return") key to insert the selected item into the script. In addition, the output that would result from evaluating the selected item is displayed in the bottom output panel. The general goal of this plugin is to allow users, novice users in particular,

[2] "Loading" a plugin does not necessarily "activate" it.

to quickly explore their available options.

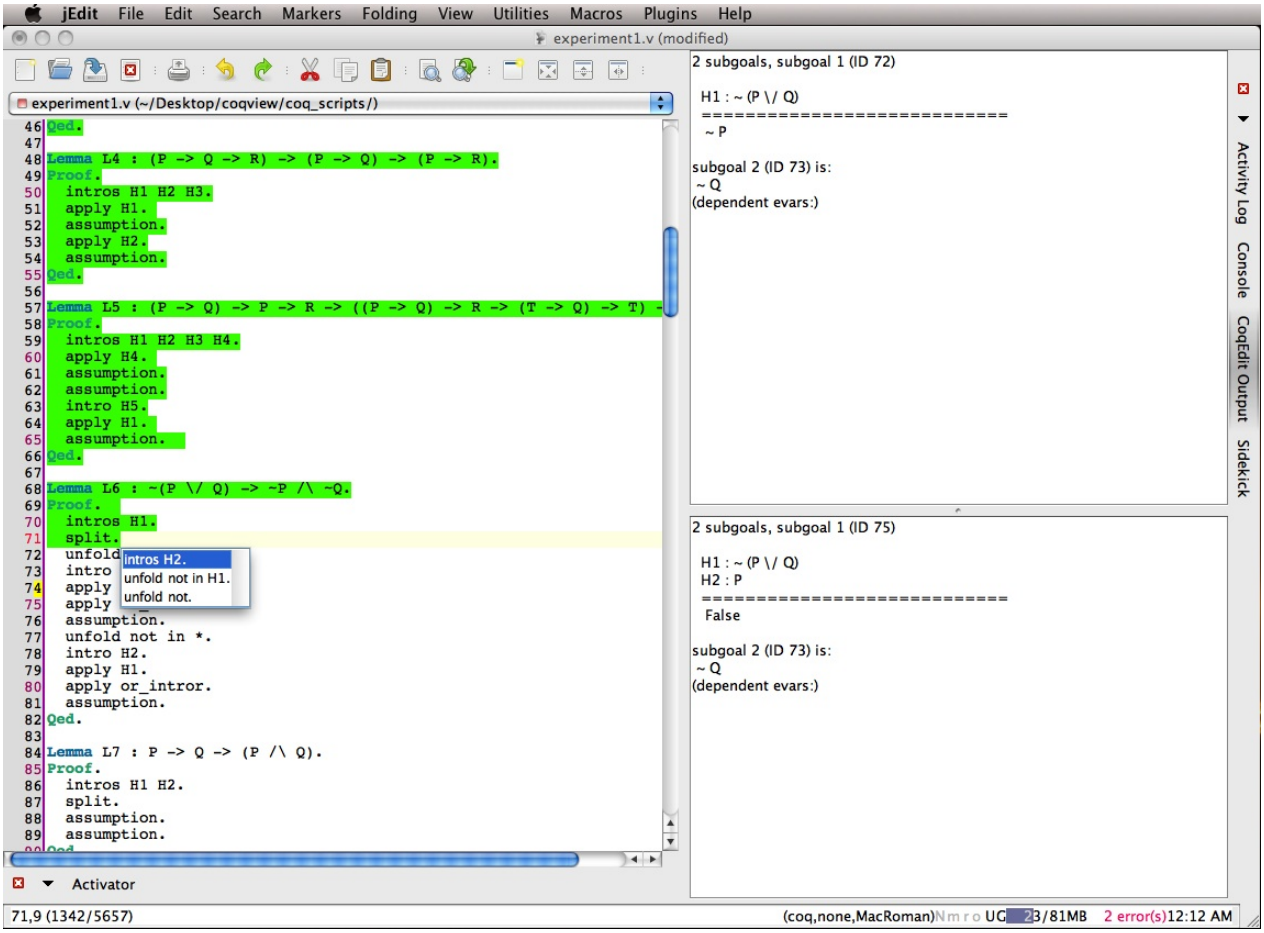

Figure 3.2: CoqEdit with the Proof Previews extension. The result of the highlighted tactic is displayed in the bottom output panel.

The tactic list generation capabilities for this plugin are quite limited at the moment: only tactics used in propositional logic exercises and examples may appear in the list. While this may still be useful for educational purposes, our implementation is mainly to be used for usability testing and to demonstrate the utility of the general idea. A more fully developed version of this plugin would be possible, but would likely require some complicated indexing of Coq's standard library.

3.3.2 "Proof Transitions"

Our second experimental extension, "**Proof Transitions**", evolved from two separate ideas. The first is **proof tree visualization**. Originally, we proposed visualizing Coq's proof trees as in Figure 3.3, where

- the red half circles represent goals,
- the blue half circles, placed on top of the red half circles to form complete circles, represent tactics successfully used with the goals corresponding to these red half circles,
- child goals are arranged *above* parent nodes (as in Figure 2.5),
- thin red lines connect parent nodes to unproved child branches,
- thick black lines connect parent nodes to proved child nodes, and
- the yellow arrow represents the current node.

(Figure 3.3 visualizes the example from section 2.0.1).

In addition to helping users in seeing which child goals are associated with which parent goals and tactics, and potentially in traversing/adding to f the trees' nodes in arbitrary orderings, this sort of visualization could be enhanced further to provide valuable information. For instance, Figure 3.4 shows nodes being highlighted based on similarities in the tactics [3]. The information associated with these nodes could be compared side-by-side using an additional output window whose location within the proof tree could be represented by an additional arrow, as in Figure 3.5.

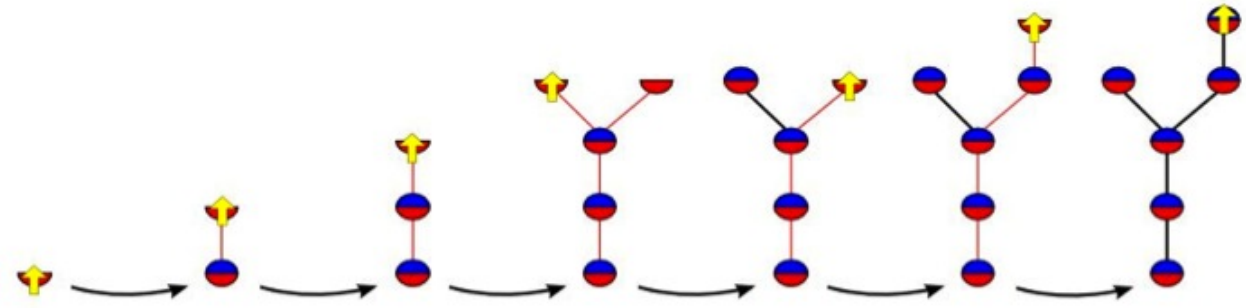

Figure 3.3: A series of partial proof tree visualizations of the example of section 2.0.1, in an earlier style.

[3] ...which might be particularly helpful in avoiding loops proofs

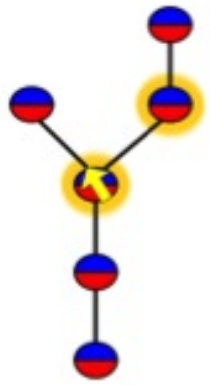

Figure 3.4: Similar nodes being highlighted while the node under the yellow arrow is inspected.

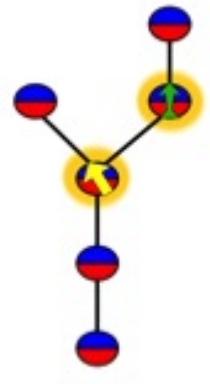

Figure 3.5: The proof tree visualization of Figure 3.4, with an additional arrow to represent the location of another output window within the proof tree.

Proof tree visualizations could also be enhanced to support management of different versions of proof tree branches, as in Figure 3.6, and to support proof presentation, as in Figure 3.7 which shows how a branch deemed uninteresting might be removed from the visualization. These examples suggest the existence of a wide range of possible user interface improvements that may not even have been thought of yet, and that are part of the rationale for making CoqEdit more easily extendable.

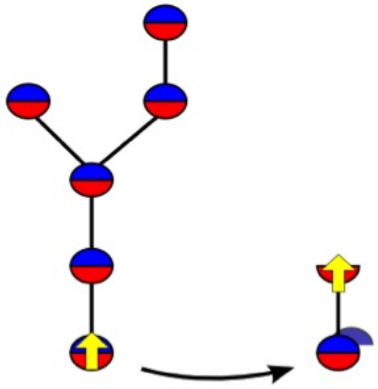

Figure 3.6: A user might decide to try an alternate proof of a branch (or the entirety) of the tree, or the entire tree. A saved copy of the original branch might be represented by a "shadowing" blue half-circle, as seen on the right-hand side of this figure.

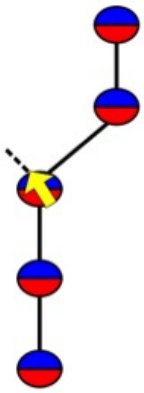

Figure 3.7: The visualization of Figure 3.4, with a branch hidden. This sort of hiding might be useful in making the structure of the remaining visible branches more clear.

The second idea from which the Proof Transitions extension evolved was "**Inference Rule Highlighting**". The idea here is to use highlighting the to show the pattern, or inference rule, being instantiated by a tactic application to a goal. In Figure 3.8, we see an instance of the general rule seen in Figure 3.9. The blue box contains a tactics while the red box below contains the old goal and the red box above contains the new goal (in general, one might see several red boxes above). The goal is for users to be able to clearly see the effects of tactics on particular goals, i.e. which bits of text get moved where, and for novice users to gain an understanding of how the tactic works more generally.

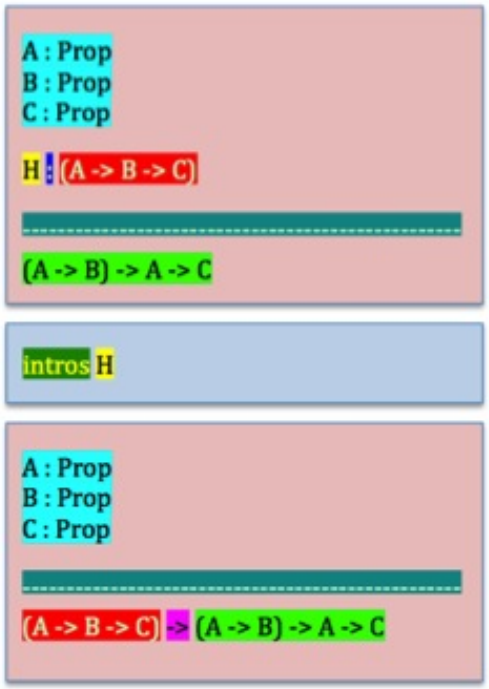

Figure 3.8: Inference Rule Highlighting example: child, parent, and tactic, with highlighting showing where the tactic has moved and added text.

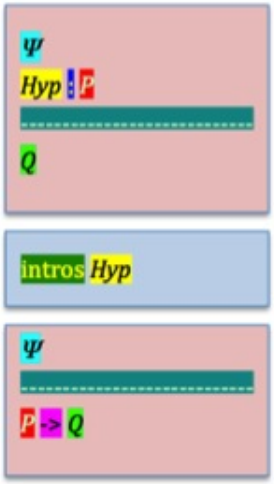

Figure 3.9: The general inference rule on which Figure 3.8 is based.

This particular way debugging tactics may work for many of the basic tactics, but it does not scale very well to more complicated tactics, since, for one thing, one eventually runs out of easily-distinguishable colors and, furthermore, many tactics do not correspond to simple general rules like that seen in Figure 3.9 (e.g. the `auto` tactic that tries to automatically prove a goal). Proof Transitions will provide an alternative which, in an initial form, can be seen demonstrated in Figure 3.10, Figure 3.11, Figure 3.12, and Figure 3.13. Figure 3.10 shows the initial goal. Figure 3.11 shows how immediately after the tactic `intros H1 H2` has been processed by the system, a blue box, containing the tactic, is placed immediately on top of the old goal and the new goal, in another red box, is placed further up with a line connecting its red box and the blue box. As an option, as seen in Figure 3.12, highlighting and underlining can be added to the boxes' text according to the following rules:

- red highlighting in the bottom red box to show text that is deleted altogether,
- green highlighting in the top red box to show text that is entirely new,
- yellow highlighting in any box to show text that is moved (or copied, in the case of identifiers given as the tactic arguments),
- red underlining in the bottom red box to show text that is either moved or deleted altogether, and
- green underlining in the top red box to show text that is new (either entirely new or moved/copied into the box)

In addition, the user can have lines drawn to connect the moved/copied portions of text, as seen in Figure 3.13. Note that the lines are actually drawn one-by-one in an animation, which makes it easier to see what is connected to what than might appear to be the case just looking at Figure 3.13. As an alternative to drawing lines, the yellow-highlighted text might be made to float up to the new positions. Note also that this extends the sorts of visualizations one commonly sees of the *UNIX* `diff` command[4]: instead of just indicating what text has been added, what removed, and what changed, we also show what has been moved. Consequently, it would seem likely that variations of the idea might be applied in numerous other coding environments.

[4] `diff` is used to find changes between versions of text files

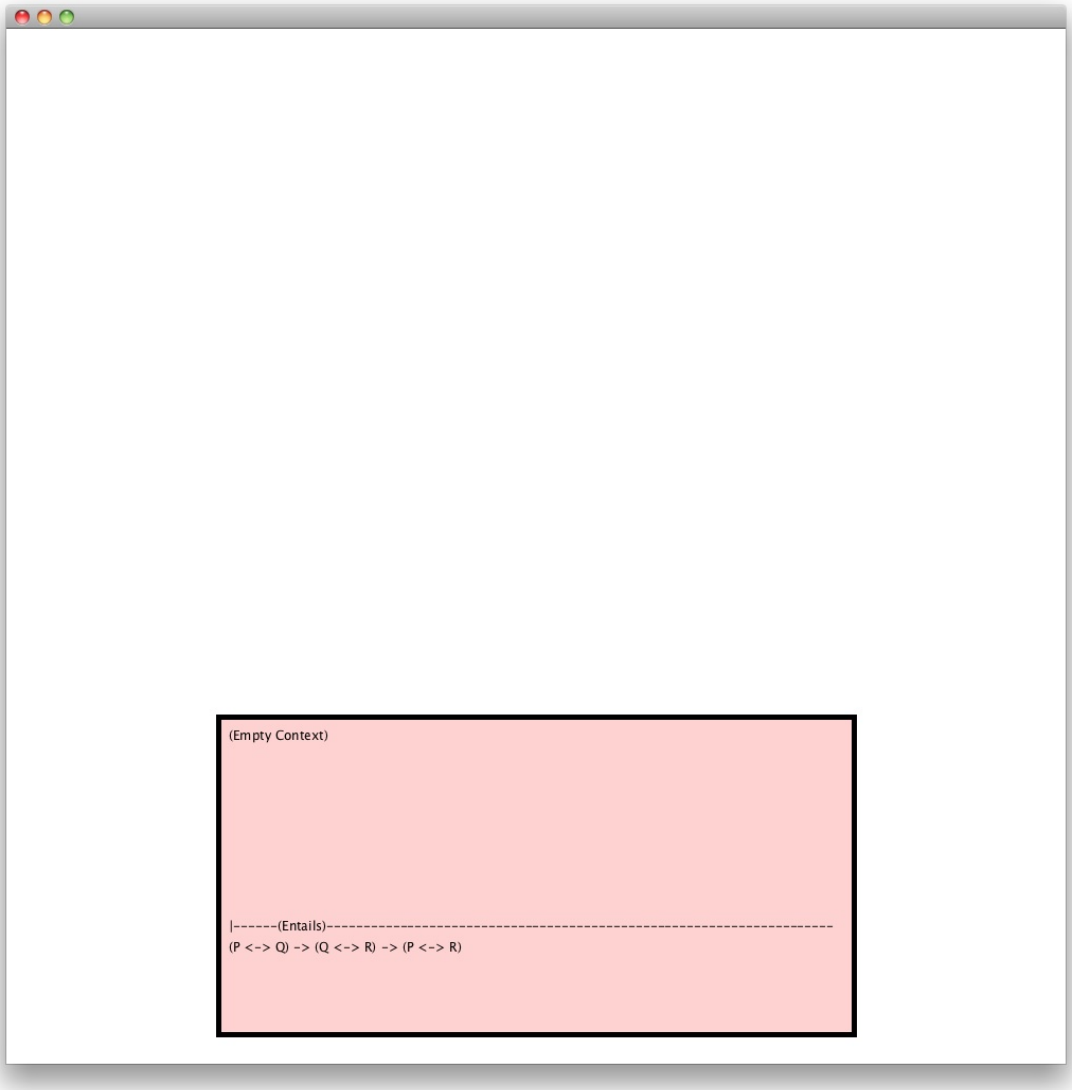

Figure 3.10: Proof transitions: a goal, pre-transition

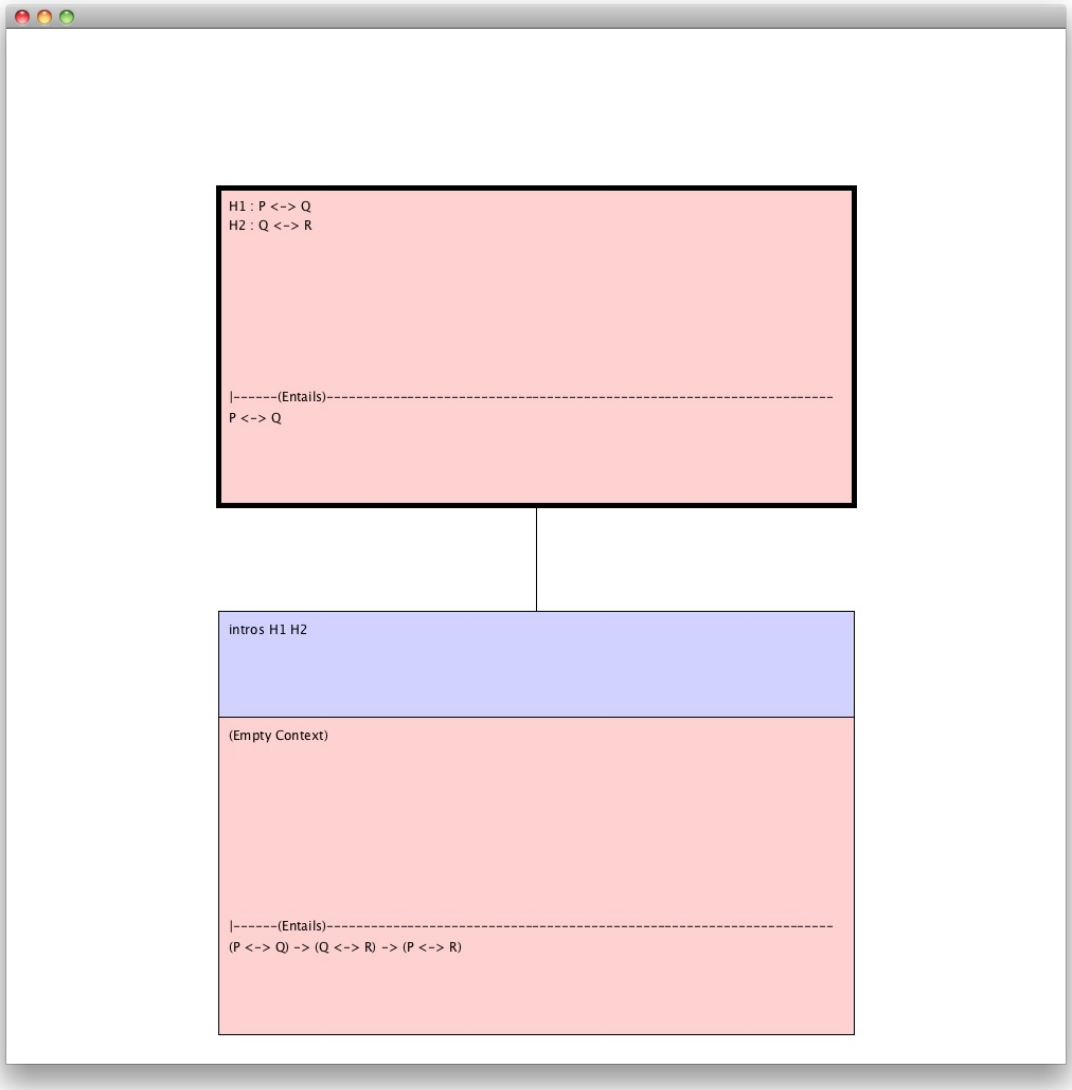

Figure 3.11: After a tactic has been applied to the goal in Figure 3.10.

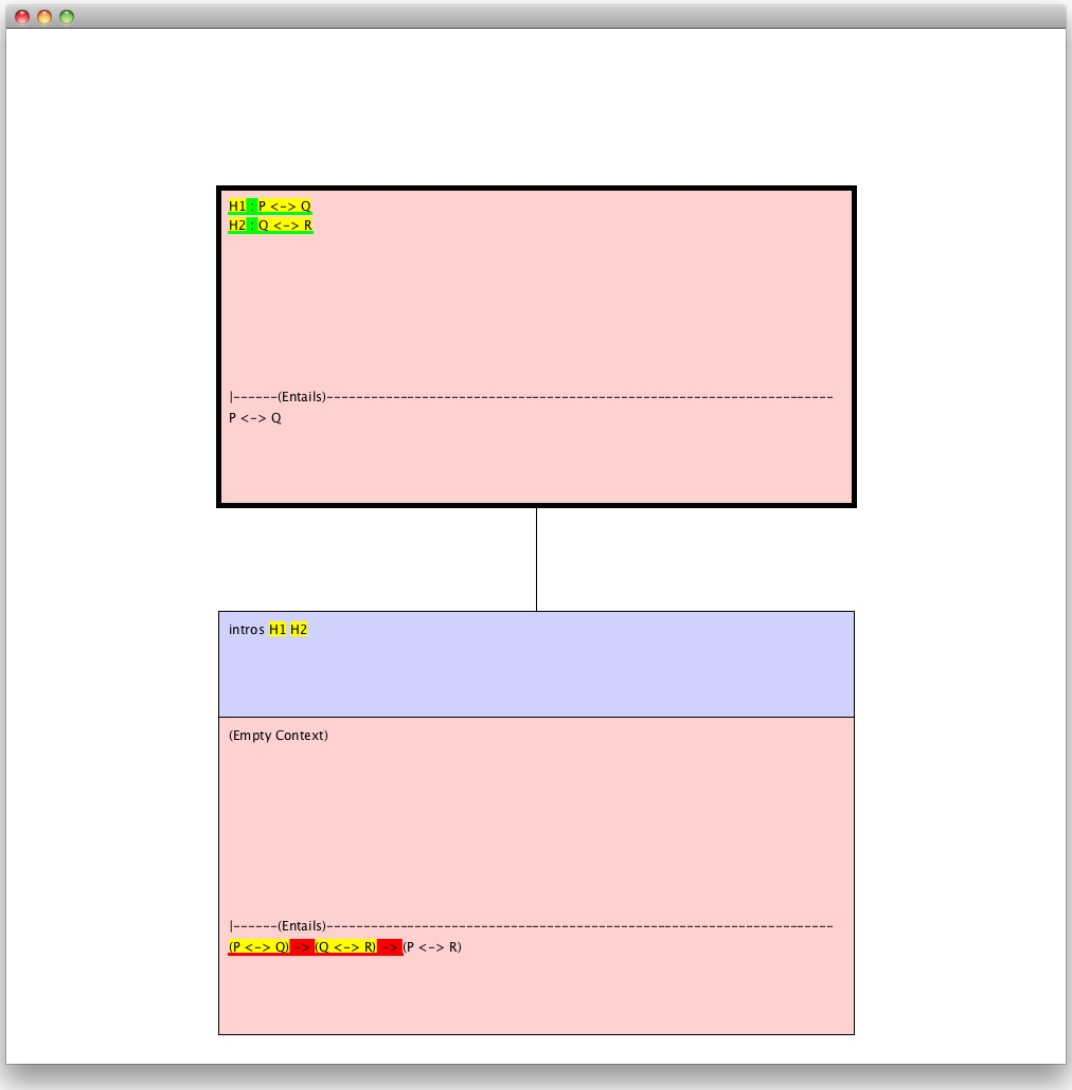

Figure 3.12: Figure 3.11, with highlighting to show changed text.

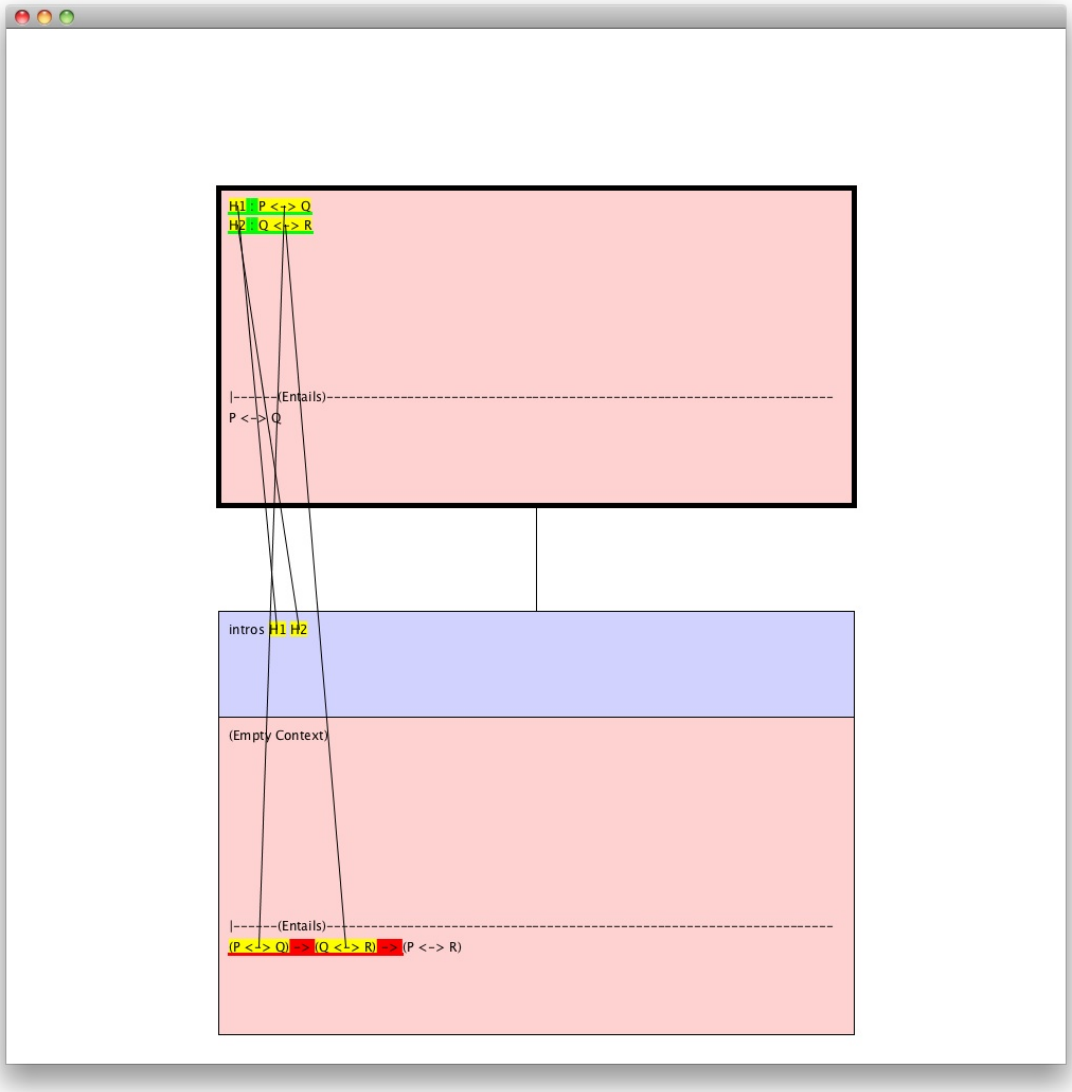

Figure 3.13: After a tactic has been applied to the goal in Figure 3.10, stage 3. Note that the lines would be added one at a time.

Because development of this plugin is being done using the *Piccolo* library for zoomable user interfaces, it will be relatively easy to also provide support for proof tree visualization. Figure 3.14 shows what zooming out from Figure 3.11 might look like, assuming the two boxes were the root and its first child and another two lines were leaving the tactic.

Figure 3.15 shows the proof tree of Figure 3.14 with the second child node of the root node set as the current node; note how one can automatically adjust the layout of the tree based on the requirement that the current node be centered above its parent (if the current node is not the root, of course) and without a lot of change in the layout of the branches.

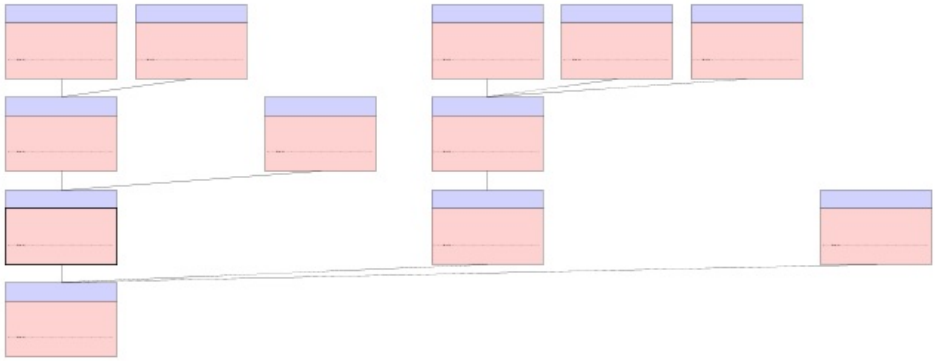

Figure 3.14: The boxes of Figure 3.10, Figure 3.11, Figure 3.12, and Figure 3.13, as part of a proof tree visualization.

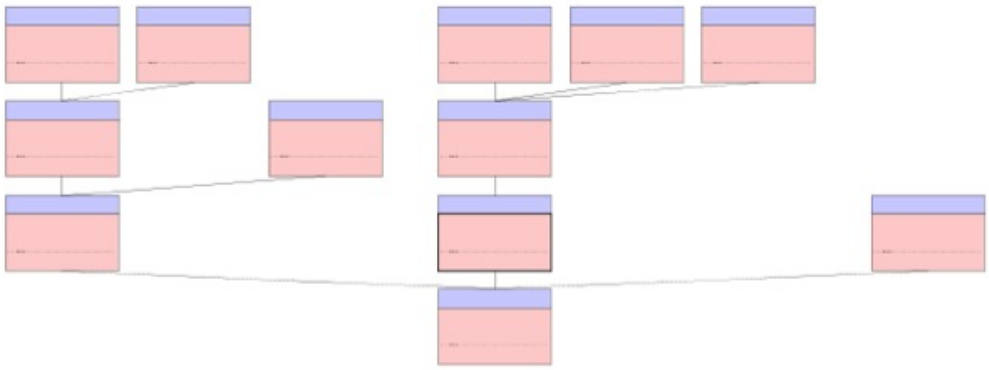

Figure 3.15: The tree of Figure 3.14, laid out with the root node's second child, instead of its first, set as the current node

Please note that this design of this extension is likely to change significantly in the future, and that for the purposes of this dissertation I am likely only to produce a "Wizard of Oz" version for user testing purposes. More specifically, I will be producing a pair of plugins that simulate an actual Proof Transitions extension acting on a particular pair of proof scripts. For a real version of the plugin, it would probably make sense to have more support in the `coqtop` back end for these features beforehand, particularly with regards to determining which portions of goals are moved. One purpose for developing these plugins is to demonstrate that such support ultimately would be useful.

3.3.3 Interlude: Keyboard-Card Menus

In this section, I describe work on "Keyboard-Card Menus", which I hope to incorporate into a third extension to CoqEdit. The contents of this section is to be presented at the Interaccin 2013 conference and published in its proceedings under the title *Keyboard-Card Menus: Faster Learning of Many Fast Commands.* It was written by myself and my advisor, Professor Juan Pablo Hourcade, at the University of Iowa.

Abstract

We present a study comparing "keyboard-card" menus' presentation of non-traditional shortcuts with a presentation of these same shortcuts that uses dropdown menus. Keyboard-

card menus are a new type of menu system in which potentially hundreds of menu items are arranged in sets of keyboard patterns that are designed to be navigated using only a computer keyboard's character keys, for fast access. In selecting items from these menus, novice users physically rehearse the same actions that an expert would use. The data from our study shows that keyboard-card menus have significant advantages over dropdown menus in making the transition to expert use faster.

Introduction

Keyboard-card menus are a new type of menu we have developed which present menu items in a computer-keyboard shaped arrangements. They are designed to be navigated using the keyboard, and in doing so users end up learning shortcuts. Our testing shows significant advantages over a presentation of these same shortcuts that uses dropdown menus. Our ultimate goal in working on keyboard-card menus is to develop a system that avoids making tradeoffs between three properties: how easy the system is to learn, how efficiently experts are able to select menu items, and how many menu items are easily accessible.

While widely-used interaction techniques often support two of these three properties, they do so at the expense of the third, which can be considerably limiting for certain tasks and people. Consider an example we had in mind while developing the menu: undergraduate college students writing and manipulating mathematics. Many of these students do not have the time to learn an unfamiliar, non-WYSIWYG (what-you-see-is-what-you-get) system like *LaTeX*, but also need an efficient system since they are asked to write many equations in homework assignments (especially if they are asked to "show their work"), *and* they need to be able to access many mathematical symbols. Handwriting recognition systems might be much easier to learn, but even if the system's recognition accuracy is 100%, in our opinion handwriting speed is a low bar for efficiency (14 to 15 year old students' handwriting, when copying, has been measured as averaging only 118 characters per minute or, at 5 characters per word, 24 words per minute [48]). WYSIWYG systems like *Mathematica* and Microsoft *Word*'s equation editor allow use of either the mouse or keyboard shortcuts to select symbols from menus and toolbars. While this might be seen as supporting both novice and expert users, since keyboard shortcuts can provide speed advantages once learned, it has been shown that, in general, many users never make the transition to using shortcuts, even in heavily used applications like *Word* [65].

Writing and manipulating mathematics is not the only application where the users would need to be quickly trained to efficiently select from a large number of menu items. Novice writers of other sorts of code could benefit as well; for instance, novice HTML coders must spend considerable time learning which tags to select from the large number available. We suspect that there are many other sorts of activities, particularly in data entry, classification, and retrieval, that could be made more practical by supporting a better balance between efficiency and learnability in applications where items must be selected from a large pool.

In the next section, we describe how keyboard-card menus present menu items, how they are navigated, and some related work. We go on to describe our study, which compared keyboard-card menus with dropdown menus, and discuss its results, which show significant advantages for keyboard-card menus. Keyboard navigation for keyboard-card menus (and keyboard navigation for the dropdown menus in our study) is unusual in that users are required to press and hold character keys (i.e. to use the character keys themselves as modifier keys); we call these interactions "rolled-chords," or "rolled-chord shortcuts". We therefore include a discussion of the advantages and disadvantages of this approach relative to other ways the keyboard might be used in navigating the menus.

Keyboard-Card Menu Design: Keyboard-Card Menus

Inspired in part by work on keyboard-based menus for wearable computers, seen in [73], we have developed keyboard-card menus which lay out menu items in keyboard patterns that *show* users what keys to press, instead of simply listing menu items with shortcut key names; in principle, at least, users do not even need to know the name of the key they want to press. (An example keyboard-card menu–one used in the study described below–is shown in the lower half of Figure 3.16, and in Figure 3.17).

More specifically, the menu system consists of colored "keyboard-cards", each of which serves as a submenu (except for one card that serves as a root menu). Each keyboard-card has printed on it a set of squares arranged in a keyboard pattern. Printed on top of some (potentially all) of these squares are the menu items that can be selected by pressing the corresponding keys. To make the affordances of the menu system more obvious to users, squares with menu items are also a lighter color than the rest of the card and each square corresponding to a submenu has an arrow in the bottom right corner. As a secondary mechanism for associating the menu item with the key, and to make it even more obvious to users that they should use the keyboard rather than the mouse, each square with a menu item also has the corresponding key's character printed in white in a large font, behind the menu item text.

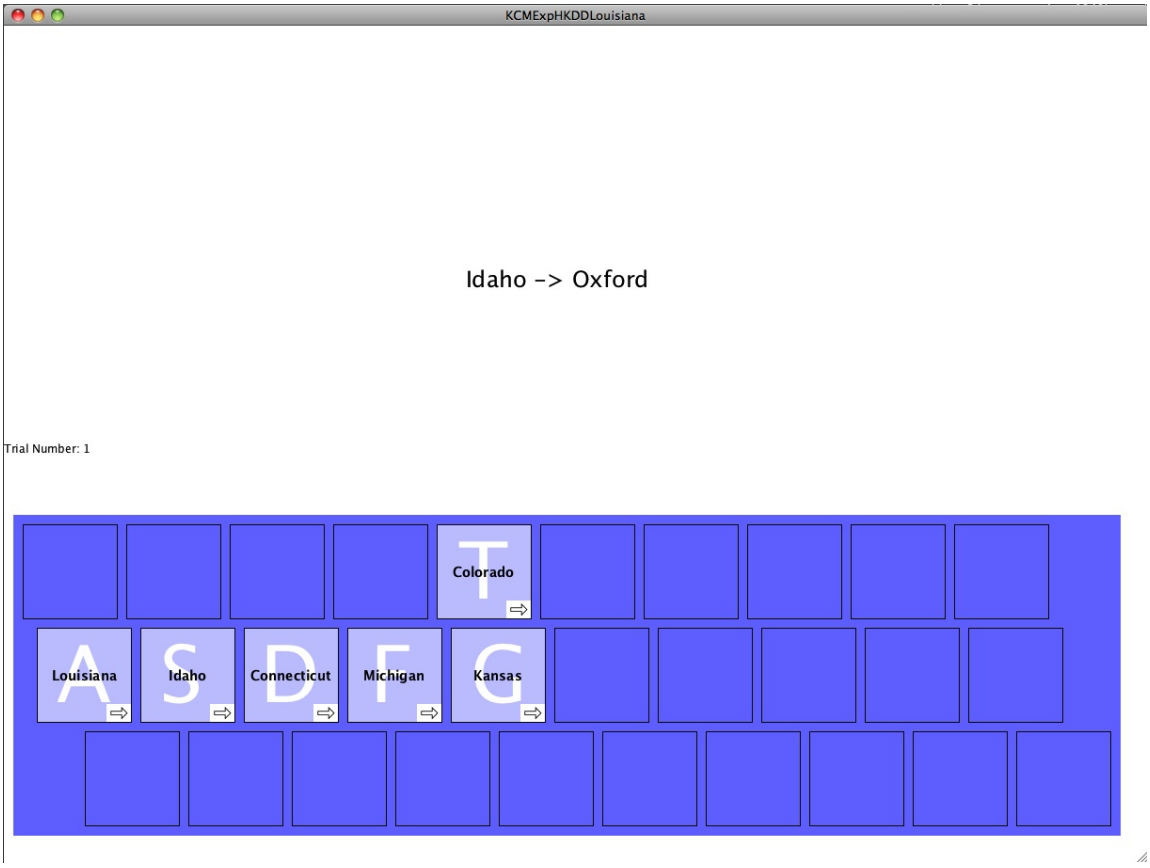

Figure 3.16: One of the two keyboard-card menu root cards used in the study, with a prompt for the participant. A small arrow in the corner of each key indicates that there is a submenu associated with that key. Participants in the study did not have to rely on this design detail.

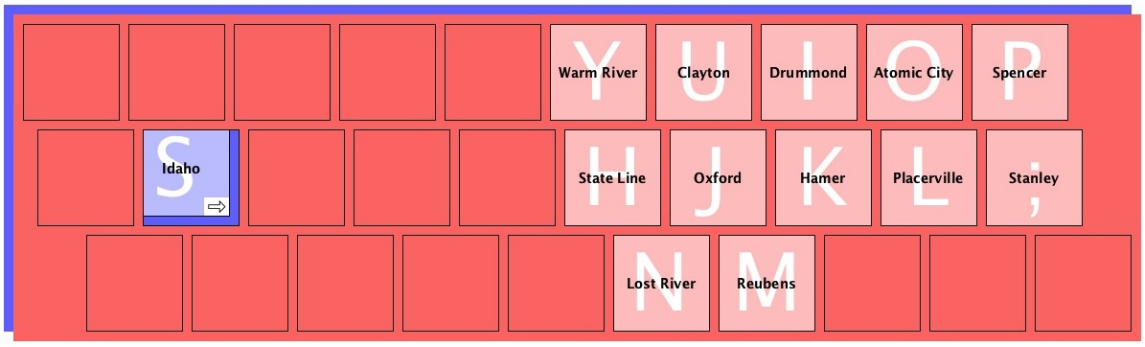

Figure 3.17: The keyboard-card menu from Figure 3.16 with the S key pressed. Note how a hole has been "punched out" of the card.

Initially, when no keys are pressed, only the root card, with top-level menu items, is visible (as in the lower half of Figure 3.16). Whenever a key corresponding to a submenu (rather than a leaf in the hierarchy) is pressed–*and held down*–that submenu's keyboard-card is placed on top of the existing card(s), slightly offset to show the number of keyboard-cards below it (as in Figure 3.17, which would be displayed while the S key is held down). Releasing that key hides that keyboard-card again. (Although the issue was ignored in our study, since no more than two cards, including the root card, were "stacked" at a time, releasing a key to hide a keyboard-card would most likely also just hide any cards on top of it, even if the keys associated with those cards were still pressed). Holding down a key corresponding to a leaf in the menu hierarchy, in addition to performing whatever action is assigned, thickens the outline of the corresponding square on the fully-visible top card.

Because menu item selection is performed entirely from the keyboard, without the user having to move his hands, it can be very efficient. In addition, simply by navigating the menu hierarchy, users end up practicing, and thereby learning, shortcuts needed to access menu items, and may actually stop needing the menu altogether. This idea of "physical rehearsal"–teaching and reinforcing the physical act of using a shortcut during the process of navigating a menu hierarchy–can be found in work on gestures and "marking menus" by Kurtenbach and Buxton [62, 64].

Keyboard-Card Menu Design: Rolled-Chord Shortcuts

The term "rolled-chord" comes from western classical music and refers to playing several notes together but initiating them at different times. Here we use the term to refer to pressing multiple computer keys at the same time, but initiating the presses in a particular order. However, while this term could be used to describe shortcuts involving special modifier keys (e.g. Ctrl-X, Alt-F4, Ctrl-Shift-S, etc.) we use the term to describe shortcuts

where the modifier keys may in fact be letter keys. (In applications where users need to type normal text, we assume that a separate mode would exist, as with the text editor *vi*). For instance, one might press *and hold* the F key, then press *and hold* the D key, and, finally, press the J key to select some particular menu item. In this paper, we consider only rolled-chord shortcuts using the letter keys plus the semicolon, comma, period, and forward slash keys on a QWERTY keyboard, since avoiding the use of special modifier keys has the advantage of reducing hand movement and therefore potentially increasing speed.

Rolled-chords are to be contrasted with "standard" chording in which the exact order of key presses does not matter because it is assumed that key presses are initiated simultaneously. Standard chording may be used to achieve high speeds of data entry–on specialized keyboards, text entry speeds of up to 300 words per minute are possible [88]. While one would expect standard chording to be faster because no time is needed to ensure a particular ordering of key presses, ordering does provide more potential shortcuts and, in particular, more shortcuts that may be accessed without moving one's fingers off the home row of keys. Furthermore, for sequences of shortcuts where the initial keys are identical, we can allow users to hold down those initial keys while pressing and releasing the final keys; for instance, over the course of entering the sequence of shortcuts "F-J, F-K, F-J", the F key does not need to be released.

Rolled-chord shortcuts may also be contrasted with the use of short sequences of typed characters (e.g. ": q ENTER" is used to quit in *vi*). While such sequences remove limitations on the numbers of shortcuts that can be provided, rolled-chords, even using only two-key sequences, still provide hundreds of possible shortcuts (we suspect that having more than three, and possibly more than two, keys per shortcut would be impractical for many users) and, if absolutely needed, sequences of rolled-chords could still be treated as shortcuts.

Rolled-chords also have several important advantages over simple key sequences. First, in sequences like "F-J, F-K, F-J" we have reduced the number of keystrokes needed, as explained above. Second, rolled-chord shortcuts enforce the rule, when assigning shortcuts to commands, that the keys pressed in a shortcut never use the same finger twice. We suspect that following this rule increases speed; it also, as pointed out in [84], removes the possibility of accidentally entering a repeated letter sequence like "F-F" because of an operating system's key press repeater. Third, when mistakes are made in selecting an initial key of a shortcut, no additional keystroke (e.g. "Backspace" or "Delete" key press) is needed to undo the mistake–the user simply releases the key. This feature is especially important not only because it makes mistakes more preventable, but because it means that *when the available shortcuts are displayed in a navigable hierarchy, as with keyboard-card menus, users can "peek" into submenus without having to press an additional key to back out again.* We expect that this will help novice users considerably in exploring hierarchies and in searching for items that have non-obvious locations within the menu hierarchy.

Related Work

Encouraging shortcut use within graphical user interfaces is an active area of research. A theoretical explanation of the problem of low levels of shortcut usage can be found in *Paradox of the Active User* [33] which notes that people are biased towards using software to solve their immediate problems, rather than towards exploring what the software can do, and also generally prefer to use known solutions rather than new ones. The severity of the problem has been empirically verified in [65], and in trying to solve it researchers have gone so far as suggesting that the option to click on a dropdown menu item should be disabled when a keyboard shortcut is available [49].

Our general approach–graphically supporting users in entering commands from the keyboard–has been taken before in a several alternate ways. "GEKA" [53] populates a list of command names and shortcuts, refining the list as users type, allowing a command line-like experience with a graphical user interface. "HotKeyCoach" [61] uses a transparent popup window to inform and remind users of available (traditional) shortcut alternatives in a non-disruptive way as they work with an application. "Blur" [86] combines features seen in "GEKA" and "HotKeyCoach", notifying users when a command could be invoked by pressing escape and then typing "hot commands" (e.g. "align left") into a transparent popup window; it also provides lists of command recommendations. "ExposeHK" shows users available (traditional) keyboard shortcuts when a user presses a modifier key; it replaces toolbar icons with printed shortcuts, expands all dropdown menus at once (with shortcuts printed next to the menu item name), and uses tooltips to show shortcuts for icons in a Microsoft *Office 2007*-style ribbon [74].

The line of work by Kurtenbach et al. on "marking menus" (which extend "pie menus" by allowing expert users to select items without looking at the menu) is in many ways parallel to ours, a major difference simply being the intended set of applications: they assume a context in which users need some sort of pointing device most of the time (e.g. technical drawing), while we assume a context where most of the time users want to have both hands on the keyboard (e.g. text editing). In particular, the design of keyboard-card menus follows three principles used in the design of marking menus [64]:

- *Self-revelation*: interactively telling users what commands are available and what these commands do
- *Guidance*: showing users how to invoke commands during the process of self-revelation
- *Rehearsal*: guiding novice users through the same physical motions that an expert would use

A more recent alternative to marking menus, but still following these design principles, is "OctoPocus" [24]. Kurtenbach et al. also address the issue of providing large numbers of quickly accessible menu items in work on the "HotBox" [63].

Work that most closely relates to ours may be explorations of interaction with multi-touch surfaces, since these provide a new opportunity for computer chording. Relevant work includes "Multi-finger Chorded Toolglass" [75], "Multi-Touch Menu" [17], "Finger-count shortcuts" [18], "Multi-touch Marking Menus" [67], and, perhaps most significantly, "Arpege" [23]. An advantage of using a multitouch surface is that thumb mobility may be exploited, as seen in [17] and [75]. However, physical keyboards appear to be faster to type with than virtual keyboards on a multi-touch surface [95], and using them in presenting chords does not create occlusion problems (involving the hands) which, as highlighted in [23], would likely make the presentation rolled chords much more complicated.

Study Description: Study Motivation

Keyboard-card menus are one of many possible ways to present rolled-chord shortcuts. Using dropdown menus with menu items marked with letters, as in Figure 3.18, as a baseline for comparison makes sense because of their advantages over keyboard-card menus and other types of menus, including that

- most existing applications already use dropdown menus, making the incorporation of rolled-chord shortcuts into the design of these applications somewhat more straightforward for software developers
- users may feel more comfortable with them, since they are almost certainly more familiar with them
- they do not hide submenu siblings (e.g. "Massachusetts - F" and "Utah -T" are still visible in Figure 3.18)
- they are more compact (at least for relatively small hierarchies)

Reducing the advantage of compactness, screen resolutions have improved dramatically over the past decade [97] and secondary monitors are often available. Some advantages of keyboard-card menus are that

- since the size and shape of keyboard-card menus is relatively constant, the content being manipulated could never possibly be covered up by the menu
- since dropdown menus typically have only a single row of items at the root level, they may allow for broader overall menu hierarchies, which have advantages over deep hierarchies [81]
- the keyboard pattern makes using of the keyboard rather than the mouse the obvious choice

The last point may be the most important, though it is unclear how many users would decide to use the mouse rather than the keyboard if presented with labeled dropdown menus. Finally, there is our *experiment hypothesis*:

- new users are able to learn menu items' shortcuts faster

By this we mean both that the number of selections needed to memorize a shortcut is low and that the time to enter a shortcut that has not been memorized is low.

There are several reasons to think that keyboard-card menus might help users learn shortcuts faster. First, although the size of each menu item's text is the same in both the keyboard-card and dropdown menu presentations, the character used to access the menu item is bigger, and therefore may be easier to read. Second, the greater amount of space between menu items may also make misreading the character associated with a menu item more difficult. Third, placing menu items in a two-dimensional array might help users exploit approximate information about locations because there are more characteristics that can be remembered. For instance, given set of menu items, if these menu items are displayed in a one dimensional list, one might be able to remember that menu item "A" is above menu item "B", but, if the items are displayed in a two-dimensional array, one might be able to remember that "A" is above "B" *and* one might be able to remember that "A" is to the left of "B". Fourth, with keyboard-card menus there is an alternate, and possibly faster, way to associate the key with the menu item, since the system could be used without looking at the character printed under the menu item.

Study Description: Study Design

To see if the keyboard-card menu improves rolled-chord shortcut learnability relative to a dropdown menu presentation (i.e. relative to dropdown menus labeled as in Figure 3.18 that one navigates by pressing and holding keys), we performed a 20 participant (8 male) within-subjects comparison with adult self-described touch-typists as our participants, based loosely on [12] and [49]. Participants were recruited through a mass email and word of mouth at a large state university in the United States. After a 1 minute typing test (using the "Space Cowboys" test on typingtest.com), each participant performed up to 720 selection tasks ("trials") over the course of at most 50 minutes, using one of the two menu systems, followed by half of a brief questionnaire. The trials were then repeated, over the course of a second period of at most 50 minutes, using the second menu system with a second hierarchy of menu items, followed by the second half of the questionnaire. Before each 50-minute period, participants performed 10 warm-up trials. Participants were told that they could ask questions or take breaks at any point that would not count against the 50 minutes for each menu system, and participants were asked, at the 25-minute mark, if they would like to take a break. Participants were also told that two thirds of the compensation would be prorated based on how many of the 1440 trials they completed.

In each trial, each participant was first prompted by text on the screen to press SPACE-BAR, which started an invisible timer for the trial. They were then presented with text of the form "U.S. state -¿ small town", e.g. "Delaware -¿ Little Creek" (see Figure 3.16); while the states were presumably recognizable by participants, the towns were unlikely

to be recognized since they were selected from lists of the smallest towns in those states. The trial ended when the town was correctly selected from the submenu labeled with the state's name, at which point the elapsed time was recorded and the participant was again prompted to press SPACEBAR to start the next trial. The number of incorrect selections of town names during each trial was also recorded (usually this was zero).

Two non-overlapping hierarchies of states and towns were used. Each consisted of 6 states and 72 towns, 12 from each state. The states were assigned the A, S, D, F, G, and T keys and the towns were assigned the Y, U, I, O, P, H, J, K, L, semicolon, N, and M keys (as in Figure 3.16, Figure 3.17, and Figure 3.18). From each hierarchy, 14 towns, 2 or 3 from each state, were randomly selected for use in trials. The 720 trials for each menu system were divided into 12 blocks of 60 trials, and the numbers of occurrences of each town in each block were 12, 12, 6, 6, 4, 4, 3, 3, 2, 2, 2, 2, 1, and 1, with the order varying randomly from block to block. This organization of hierarchies and trials, and the Zipfian distribution used ("which has been shown to represent command use frequency in real applications"), was used in [49].

While the same 14 towns for each hierarchy were used across participants, the numbers of occurrences were paired, randomly, with different towns for different participants (the pairing did not change across blocks for a given participant). For instance, participants A and B would both see "Delaware -¿ Little Creek", but participant A might see it 6 times in every block while participant B might only see it 2 times in every block.

The numbers of participants were balanced between the four possibilities determined by whether the keyboard-card or dropdown menu was seen first and by which set of states and towns was used with which menu type.

The software for the experiment was run on a 2.8 GHz Intel Core 2 Duo MacBook Pro with Mac OS X Snow Leopard using Java 1.6. However, participants viewed the software on an 18 inch, 1280 x 1024 pixel separate monitor (Dell Model No. 1800FP) and used a separate keyboard (Dell Model No. L100). The experiment was performed in a faculty member's office on a university's campus.

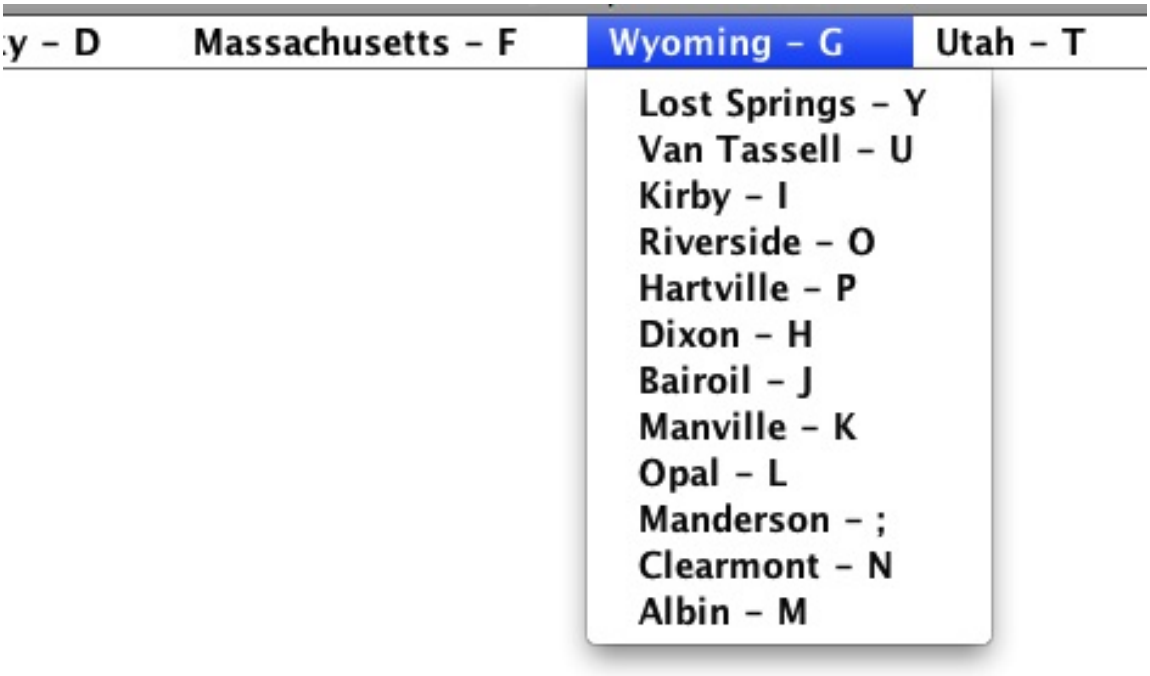

Figure 3.18: Part of a dropdown menu, navigable by pressing and holding keys rather than with the mouse, used in the study; note the letters on each item. Also note that the text is the same font and size as in the keyboard-card menus.

Study Results

The average selection times for the two different menu systems can be seen in Figure 3.19. Note that while outlier trials were removed (a trial was considered an outlier if its time was greater than 3*(Q3-Q1)+Q3, where Q1 and Q3 were the first and third quartile values; 0.55% of trials were considered outliers), average times for participants who did not complete all 1440 trials were left in. Averaged trial completion times for each block are only complete for all participants and menu types in blocks 1-8. One participant was only able to complete trials in blocks 1-8 for both menu types. Two other participants were only able to complete trials in blocks 1-10 using the dropdown menus, but were able to complete trials in all blocks using the keyboard-card menus. The remaining 17 participants were able to complete trials in all blocks.

To analyze this data we first fit quadratic equations to the completion time data for each participant (with trial number as the independent variable and completion time as the dependent variable), leaving out times for trials where any participant was missing data, the result of which was complete data for 323 trials over blocks 1-8. Paired t-tests

on the coefficients of the fitted quadratics showed statistically significant (p = 0.0458) differences between the average second-order coefficients for keyboard-card and dropdown menus (i.e. there were statistically significant differences in the "curviness" of the data for the keyboard-card and dropdown menus). In addition, we performed a repeated measures ANOVA on the average completion times for blocks 1-7 (where all participants completed all trials), which showed statistically significant differences between the two menu types (p = .044) as well as between the blocks (p ¡ .001).

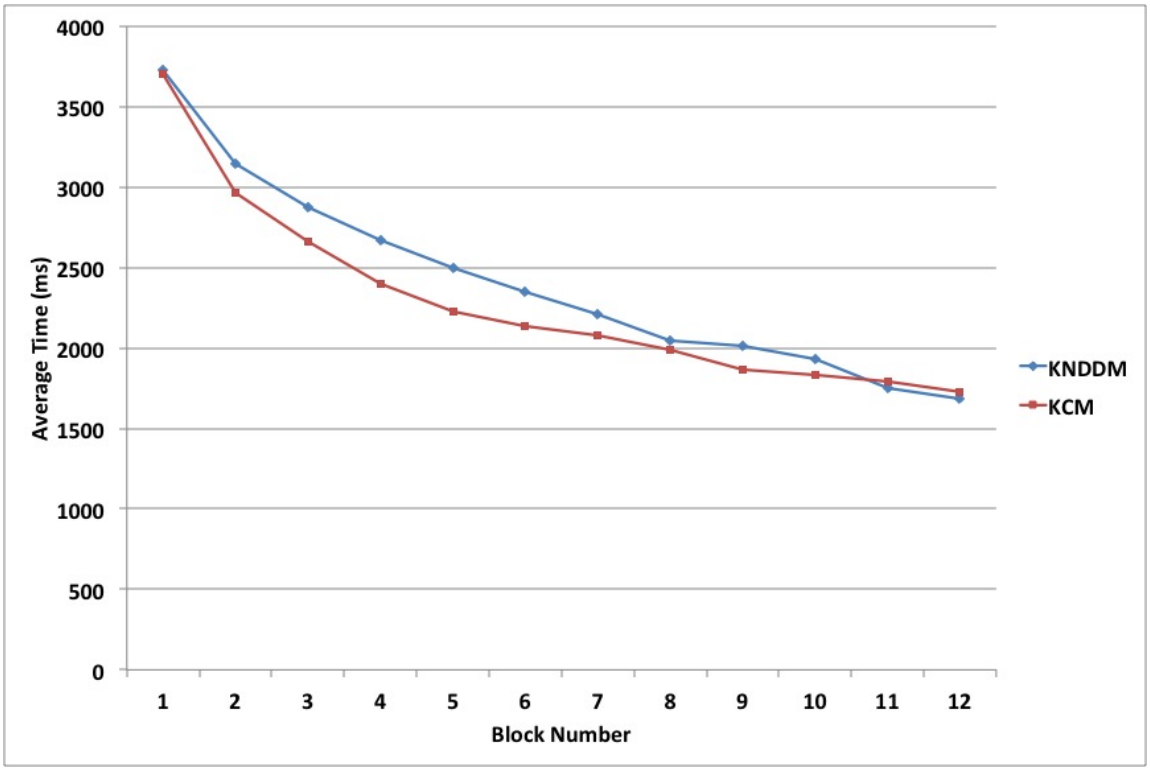

Figure 3.19: Average trial completion times, in milliseconds, for the keyboard-navigated dropdown menus and keyboard-card menus in the study. Note that data is only complete for blocks 1-8

Average error rates for each trial and menu type are shown Figure 3.20. Overall error rates across blocks 1-7 for the dropdown menus and keyboard-card menus were 10.4% and 8.4% respectively, where errors were defined as the incorrect selection of a leaf in the menu hierarchy and the error rate for a block was calculated as $e/(e+t)$ where e = the number of errors in the block's non-outlier trials and t = the number of non-outlier trials in the block. This difference was not statistically significant. However, t-tests on the data for the 5th, 9th

and 10th block differences did show statistical significance (p-values were .032, .008, and .049 respectively; note that 9th and 10th blocks exclude data for one participant). Though a repeated measures ANOVA on the error rates for blocks 1-7 also failed to show statistical significance, fitting quadratics to the error rates for each participant at each block where data was available, and then performing t-tests on the coefficients *nearly* showed statistical significance at the 5% level for the quadratic and linear coefficients (p-values were .0544 and .0624 respectively)

Several interesting correlations can also be seen in the overall (i.e. across blocks) data for each participant. Error rates for dropdown menus were correlated (R =0.718) with the error rates for keyboard-card menus and overall average times for dropdown menus were correlated (R =0.911) with overall average times for keyboard-card menus. However, error rates and average times were negatively correlated for both dropdown menus and keyboard-card menus (R = -0.510 and -0.479, respectively), so it appears that some participants chose to concentrate more on being accurate, at the expense of speed, and some chose to do the opposite.

In addition, the difference between the error rates for dropdown and keyboard-card menus was positively correlated with the error rate for dropdown menus (see Figure 3.21) and the difference between the average times for keyboard-card and dropdown menus was positively correlated with the average times for dropdown menus (see Figure 3.22). In other words, participants whose error rates were higher made even more mistakes when using the dropdown menus, and participants whose average times were slow were even slower when using dropdown menus.

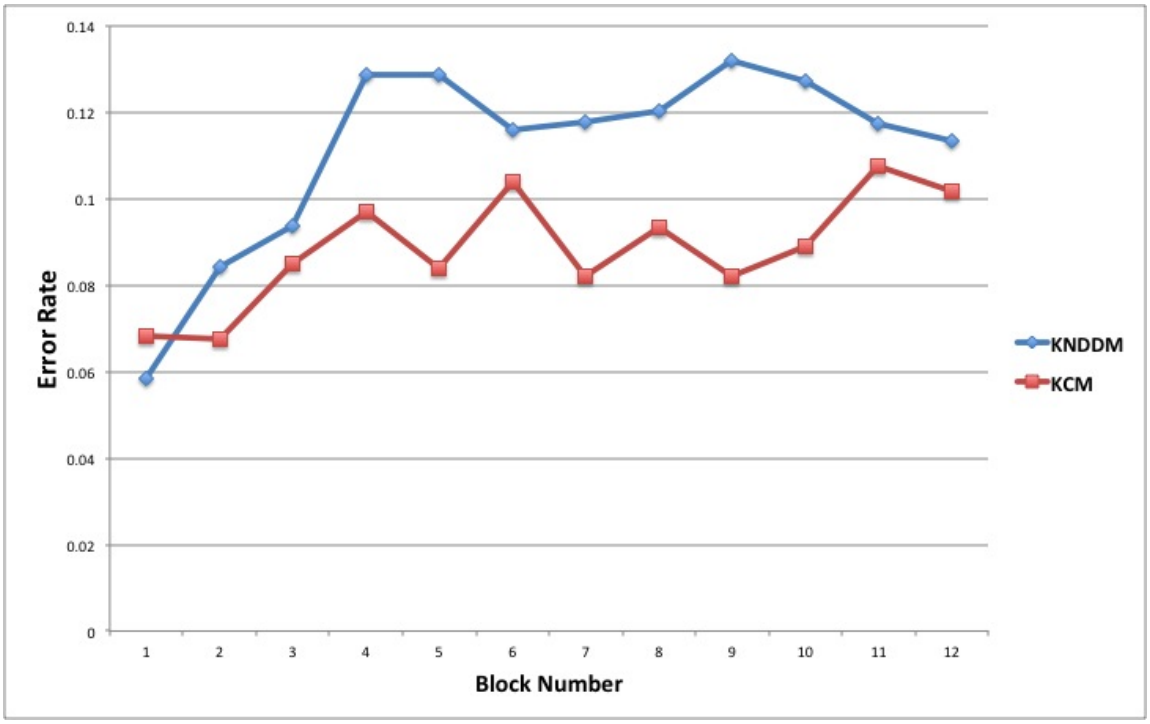

Figure 3.20: Error rates for the keyboard-navigated dropdown menus and keyboard-card menus in the study. Note that data is only complete for blocks 1-8.

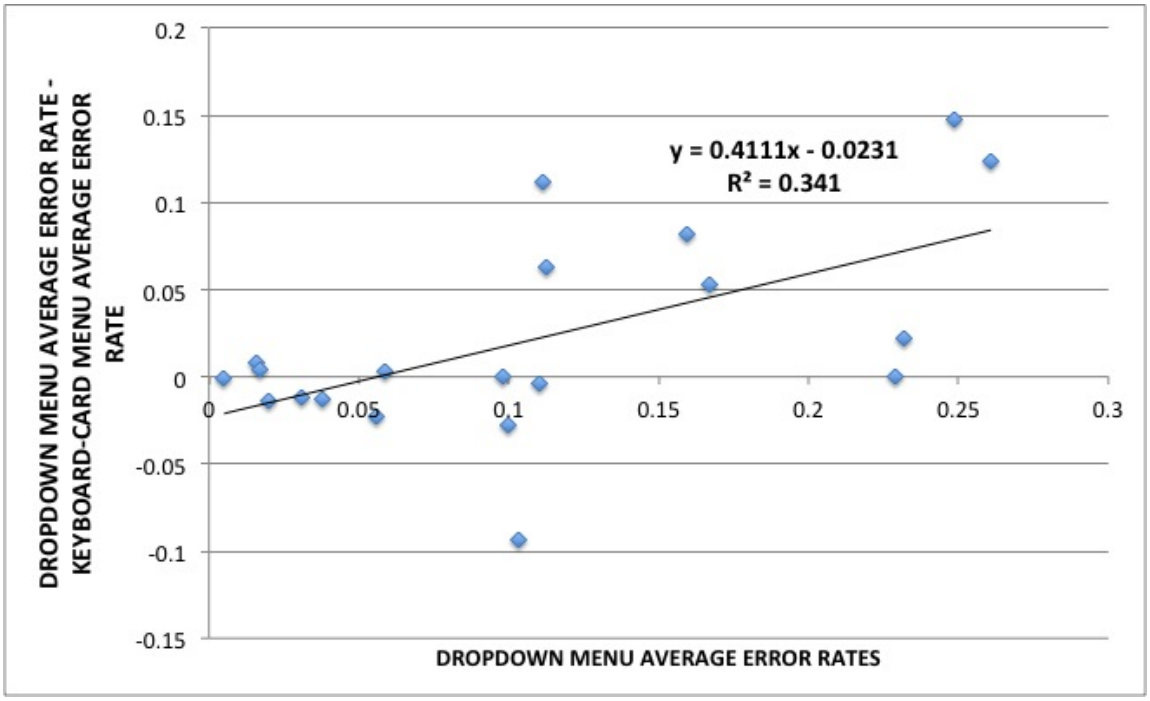

Figure 3.21: The differences between the overall average error rates for dropdown menus and keyboard-card menus, for each participant, plotted against the error rate for dropdown menus.

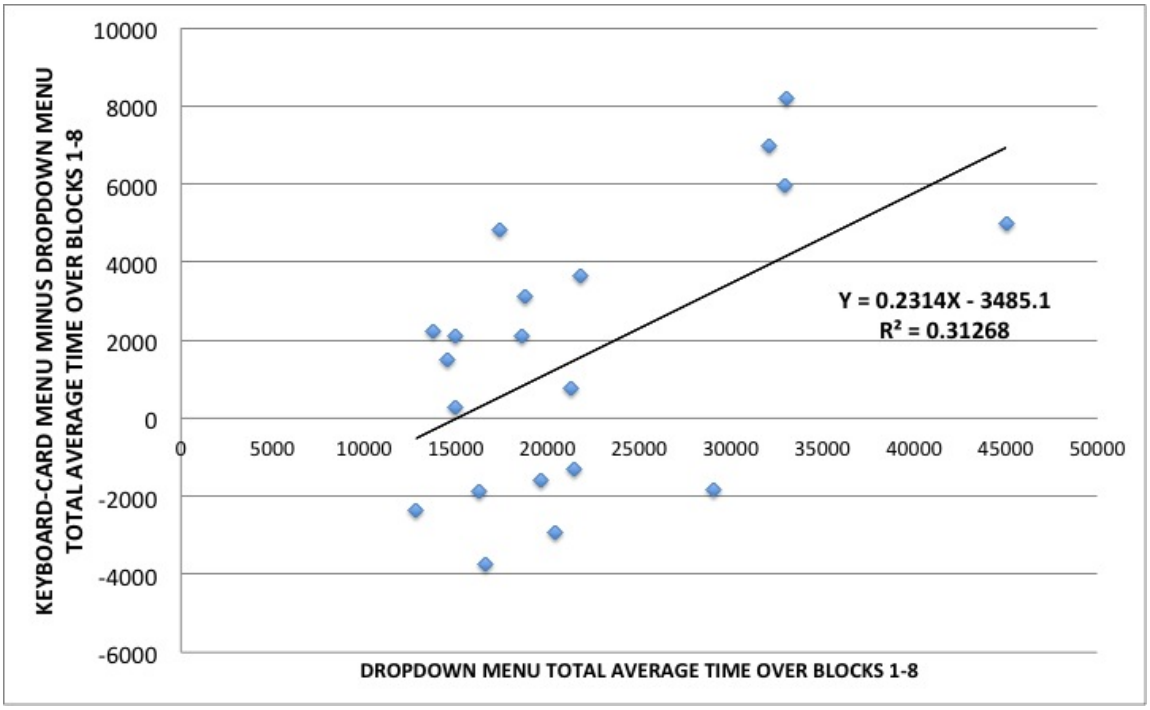

Figure 3.22: The differences between the overall average times for dropdown menus and keyboard-card menus, for each participant, plotted against the time for dropdown menus.

Results from the questionnaire can be seen in the table, where

- Q1="How quickly do you feel you were able to become familiar with the menu item locations over the course of performing the trials?"
- Q2="At the beginning of the set of trials for this menu system, how quickly do you feel you were able to select menu items?", and
- Q3="At the end of the set of trials for this menu system, how quickly do you feel you were able to select menu items?"

Ratings were given on a scale from 0="Very Slowly" to 9="Very Quickly". Though the results were favorable for keyboard-card menus, t-tests did not reveal statistically significant differences between the average ratings (p-values given in the table). These results were not correlated with typing speed.

	KCM Ave.	KCM Std. Dev.	DDM Ave.	DDM Std. Dev.	P-values
Q1	6.0	1.5	4.8	2.4	.165
Q2	3.0	1.8	2.2	1.6	.119
Q3	7.3	1.3	6.7	2.1	.525

Table 3.1: Questionnaire Results.

Discussion

The data supports the hypothesis that the keyboard-card menu's presentation makes rolled-chord shortcuts easier to learn than does the dropdown menu's presentation. We further hypothesize, first, that the eventual convergence of the two curves in Figure 3.19 means that by the 12th block users memorized the 14 shortcuts (and so were not using either presentation), and second, that the initial divergence of the two curves indicates that the keyboard-card menu's presentation mostly helps users in finding and recalling shortcuts they have seen but not fully memorized.

Although in Figure 3.19 the separation between the curves appears to be small, it may be exaggerated considerably in more realistic applications by two factors. First, the period in which users have seen menu items but have not yet memorized their locations (i.e. the middle section of Figure 3.19) may be extended over a greater period of time (e.g. hours or days, rather than tens of minutes) affecting the number of menu selections needed for memorization. Second, the effect of errors on the average selection times would likely be much greater, since only the time needed to make an error, recognize it, and make a correct selection is included in the Figure 3.19 data; in realistic applications, the time to undo an error would also have to be included.

The negative correlation seen between error rates and average times shows users making a tradeoff between speed and accuracy. However, the data seen in Figure 3.21 and Figure 3.22 suggests that use of a keyboard-card menu instead of a dropdown menu can help both users who prefer speed to accuracy and users who prefer accuracy to speed.

Many issues surrounding keyboard-card menus remain unexplored. For instance, some participants disliked the use of bright colors. Would refining the system improve performance? Another color-related refinement would be to use different colored/textured cards for each submenu (in the study, all sub-cards in the second level were red). Other important questions include: Does the menu actually encourage first time users of a system to use the shortcuts? What happens when icons instead of, or in addition to, labels are used? What is the best way to map a hierarchy to the menu? What happens when three finger shortcuts are used? Can keyboard-card menus' presentation be effective with other sorts of shortcuts?

Conclusion

While selecting from a large set of menu items in an efficient manner is certainly possible, expecting users to learn how to do so using traditional shortcuts displayed in dropdown menus is often unrealistic. In order to open up new realms of computer applications to a wider range of users (college students typing of mathematics at the computer, for instance), we are rethinking shortcuts and, in particular, how to present them. We have developed keyboard-card menus, and have tested them against dropdown menus as a way of presenting how to enter shortcuts, using rolled-chords as our shortcut choice in the experiment. Our data shows that keyboard-card menus, taking advantage of increasing amounts of available screen space with their keyboard-shaped presentation, can be more effective than a presentation based on dropdown menus as a method for presenting large numbers of shortcuts in an easy-to-learn way.

Acknowledgements

We would like to thank our participants for their time and effort. This research was supported by NSF award CCF-1250306.

3.3.4 "Propositional Logic Syntax Shortcuts"

Like Proof Transitions, the "**Proposition Logic Syntax Shortcuts**" CoqEdit extension combines two ideas. The first idea is keyboard-card menus, described above. One reason I would like to develop Propositional Logic Syntax Shortcuts is to see if keyboard-card menus can be successfully integrated into a real application.

The second idea incorporated into Propositional Logic Syntax Shortcuts is "**Syntax Tree Highlighting**". Syntax Tree Highlighting is text highlighting of parse-able text document (e.g. program) where

- only text of a node in the concrete syntax tree of the document
- the children of the node are shown as distinct sections of highlighting
- something, e.g. a line under the highlighting, shows which child nodes are leaf nodes
- the highlighting of one of the non-leaf children, if there are any, indicates that it is "selected"
- the user can navigate the syntax tree by moving the indication of selection from non-leaf node to non-leaf node and by moving the highlighting to the selected non-leaf child node or the parent of the node with highlighting.

Syntax Tree Highlighting can be useful in understanding the syntax of text (which, of course, is generally a prerequisite for understanding its semantics), and may be particularly

helpful in resolving ambiguities that result when the author of the text did not include parentheses. Ambiguities of this sort can occur in in many languages, which suggests that Syntax Tree Highlighting may be widely useful. A good example is a math problem that, according to an article in *Slate* magazine[51], has been circulating on *Facebook*. The problem is to compute

$$6 \div 2(1 + 2) \tag{3.1}$$

The fact that there even is a *Slate* story on this problem indicates that it really is ambiguous, despite attempts, described in the story, at establishing a standard algorithm. Computer scientists and programmers encounter such ambiguities frequently when reading and writing programs, especially in languages new to them. The situation with Coq is especially bad, in part because Coq allows users to define their own "notations" and to give them precedence levels and directions of associativity (e.g. "`+`" and "`/
`" are notations).

Even when there is no real ambiguity that needs to be resolved, Syntax Tree Highlighting could make code more readable. For instance, given fully parenthesized text, finding matching pairs of parentheses can be difficult, and finding all of the "next level" parentheses within a given pair is even harder. "Pretty printing" techniques, involving indentation, can also help make code more readable, but keeping indentation levels straight can still be difficult, if the text file in question is long, and there is only so far over one can indent without having to scroll over. Furthermore, in many cases coders must deal with code written by other coders or generated by a computer (e.g. the terms generated in Coq by sequences of tactics), and in these cases there are generally no guarantees about parenthesization or pretty printing.

Syntax Tree Highlighting may also be useful in selecting important sections of text (compare moving its highlighting with selection in the usual character-by-character or line-by-line way) to manipulate (e.g. rewrite, cut, copy, paste over) and "inspect" these sections. In the case of Coq, "inspecting" these sections could mean type checking or evaluating them.

Figure 3.23 gives an example of what Syntax Tree Highlighting might look like in Coq. In (a) through (d), we see the highlighting moving through the nodes of the syntax tree of the term, going first to the right-most child node and then to the left-most. Along the way, a few things are made clear:

- the scope of "`forall x :  G,`" extends all the way to the end of the term.
- "`=`" binds more tightly than "`/\`"
- "`X`" is actually an infix operator (probably declared earlier in the file), not an argument to "`x`"

In (e), we see (d)'s selected child node replaced with a new term, "`a`", demonstrating Syntax Tree Highlighting could be used in manipulating text. As noted earlier, this highlighting

could also be used to inspect nodes so, for instance, in (a) the user might see "`Prop`" displayed in some other window to indicate that "`forall x : G, x X inv x = e /\ inv x X x = e`" has type `Prop`.

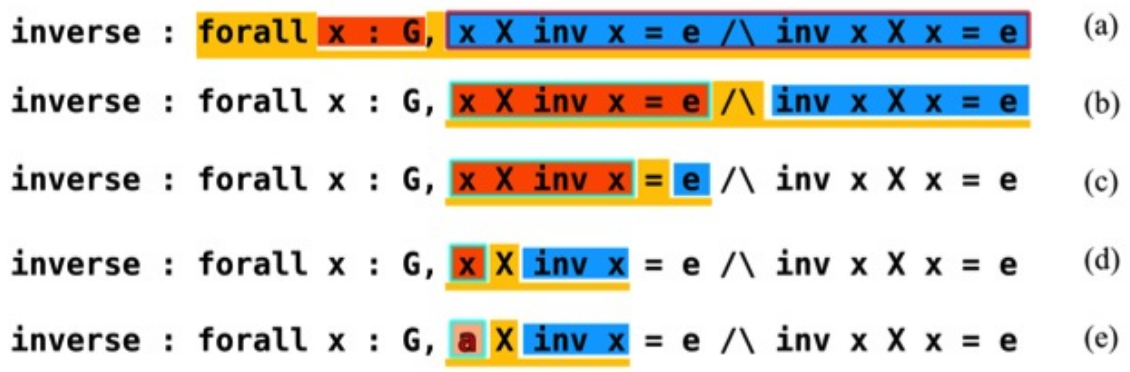

Figure 3.23: Example of Syntax Tree Highlighting in Coq.

Propositional Logic Syntax Shortcuts involves, on the one hand, some scaling back of my full ambitions for Syntax Tree Highlighting in Coq, and, on the other hand, an implementation of a generalizable form of coding and perhaps a more effective use of keyboard-card menus than I had originally envisioned. For this plugin, I will be considering only a subset of Coq's language that may be used in doing "Fitch-style" proofs in classical propositional logic, as presented in the popular textbook *Logic in Computer Science: Modelling and Reasoning about Systems* by Michael Huth and Mark Ryan [57]. Proofs in this style do not involve turnstiles or colons. Instead, they are lists of formulas where each formula is either a premise or a hypothesis, or follows, according to an inference rule from a given set of such rules, from zero or more formulas at earlier positions in the list. In [57] the formulas in these lists are generally written no more than one to a line, each with a short justification (e.g. the name of an inference rule and the line numbers formulas used), and with (possibly nested) boxes surrounding formulas to show where it is to be assumed that a hypothesis, at the top of the box, is valid (i.e. to show the scope of the hypothesis). Although Coq is not really designed for writing proofs in this style, in Figure 3.24 I show how to imitate it within a Coq .v file. Note that "Fact" is a synonym for "Lemma" or "Theorem", that the proof/hypothesis names l1, l2, l3, stand for line numbers, and that the simple proofs are applications of axioms or theorems that correspond to natural deduction inference rules given in [57].

```
104 Parameter phi : Prop.
105
106
107 Section _1.
108   Hypothesis l1 : ~(phi \/ ~phi).
109   Section _1_1.
110     Hypothesis l2 : phi.
111     Fact l3 : phi \/ ~phi. Proof. apply (or_intro_l l2). Qed.
112     Fact l4 : False. Proof. apply (not_elim l3 l1). Qed.
113   End _1_1.
114   Fact l5 : ~phi. Proof. apply (not_intro l4). Qed.
115   Fact l6 : phi \/ ~phi. Proof. apply (or_intro_r l5). Qed.
116   Fact l7 : False. Proof. apply (not_elim l6 l1). Qed.
117 End _1.
118 Fact l8 : ~~(phi \/ ~phi). Proof. apply (not_intro l7). Qed.
119 Fact l9 : phi \/ ~phi. Proof. apply (not_not_elim l8). Qed.
```

Figure 3.24: Syntax Tree Highlighting for Propositional Logic Syntax Shortcuts

```
104 Parameter phi : Prop.
105
106
107 Section _1.
108   Hypothesis l1 : ~(phi \/ ~phi).
109   Section _1_1.
110     Hypothesis l2 : phi.
111     Fact l3 : phi \/ ~phi. Proof. apply (or_intro_l l2). Qed.
112     Fact l4 : False. Proof. apply (not_elim l3 l1). Qed.
113   End _1_1.
114   Fact l5 : ~phi. Proof. apply (not_intro l4). Qed.
115   Fact l6 : phi \/ ~phi. Proof. apply (or_intro_r l5). Qed.
116   Fact l7 : False. Proof. apply (not_elim l6 l1). Qed.
117 End _1.
118 Fact l8 : ~~(phi \/ ~phi). Proof. apply (not_intro l7). Qed.
119 Fact l9 : phi \/ ~phi. Proof. apply (not_not_elim l8). Qed.
```

Figure 3.25: Syntax Tree Highlighting for Propositional Logic Syntax Shortcuts, with highlighting moved from Figure 3.24

```
103
104 Parameter phi : Prop.
105
106
107 Section _1.
108   Hypothesis l1 : ~(phi \/ ~phi).
109   Section _1_1.
110     Hypothesis l2 : phi.
111     Fact l3 : ?. Proof. apply (or_intro_l l2). Qed.
112     Fact l4 : False. Proof. apply (not_elim l3 l1). Qed.
113   End _1_1.
114   Fact l5 : ~phi. Proof. apply (not_intro l4). Qed.
115   Fact l6 : phi \/ ~phi. Proof. apply (or_intro_r l5). Qed.
116   Fact l7 : False. Proof. apply (not_elim l6 l1). Qed.
117 End _1.
118 Fact l8 : ~~(phi \/ ~phi). Proof. apply (not_intro l7). Qed.
119 Fact l9 : phi \/ ~phi. Proof. apply (not_not_elim l8). Qed.
120
```

Figure 3.26: Syntax Tree Highlighting for Propositional Logic Syntax Shortcuts, with highlighting moved from Figure 3.24

3.3.5 Extension Testing

3.4 Timeline for Research

Chapter 4

Related Work

The need for effective theorem prover user interfaces has been recognized for over 20 years (e.g. [92]) and a series of conferences has even been organized to address this need specifically.[1] Many interesting and helpful ideas have been proposed, both for Coq and for related systems. These ideas vary widely and include integration of Coq, and other theorem provers, with dynamic geometry software [80, 85], easier-to-read declarative languages for proof scripts [37], using wikis for creating proof repositories [38], and automating the process of using libraries [15].

One approach taken to improve theorem prover user interfaces has been "Proof by Pointing"[26], an algorithm to build a proof tree by pointing to portions of a goal. This was implemented in the *CtCoq* interface [25], again, later, in the Java-based Pcoq interface [14] and also in the *Jape* system [31]. Note that some schemes for writing and displaying proofs, notably Fitch style proofs, to not display hypotheses with their conclusion, which may lead to ambiguity when using Proof by Pointing [30].

An example of a recent project is *Panoptes* [43], which allows visualization of proofs produced by the IMPS Interactive Mathematical Proof System. The system has numerous features allowing users to zoom in on parts of the graph, collapse nodes, rearrange the positions of nodes, label and highlight nodes, and inspect details associated with nodes using pop-up windows. while these features allow the user to manipulate the presentation of a proof, they do not allow the user to manipulate the proof itself–to change the proof, one must use an Emacs-based environment.

Another significant project is an interactive visualizer for the *ACL2* theorem prover[1], described in [19]. This tool allows for visualization at three different levels. It does so first at the level of relationships between theorems and their proofs (the directed acyclic graph describing which lemmas from which libraries are used to prove a given theorem). Next, it shows a proof tree, similar to the visualization given in section 3.3.2, where a node represents a statement that is being proved using the node's children. Color coding is used

[1] See http://www.informatik.uni-bremen.de/uitp/ for more information

here to indicate the action taken by the prover at a node, and the tree is represented using three-dimensional "cone trees." The contents of the individual nodes, as text, can also be displayed alongside this tree. At the third level, this text's syntax tree can be visualized (as lines connecting points again, though in a more usual two-dimensional arrangement, and still in contrast with the Syntax Tree Highlighting discussed above). Selecting text at this level will highlight the corresponding portion of the tree visualization. The system also supports textual pattern matching, where the degree of matching is indicated by the color of the text and the tree visualization. Both at this level and at the proof structure level (i.e. the second level) the system is capable of zooming and panning, and, at the proof structure level, rotating is also possible.

A few other significant projects aiming to improve the presentation of machine-checked proofs include *Proviola*, which allows users to move through a proof script displayed on a web page and view the results that Coq would produce [91]; an "Interactive Derivation Viewer" for visualizing derivations written in the TPTP language [93]; the LOUI interface for the OMEGA proof assistant, featuring graphical visualization, term browsing, and natural language proof presentation [89]; and the Tecton system, featuring tree visualization combined with hypertext navigation of nodes [59]. Theoretical and methodological work has also been done and can be seen in [41, 27, 72, 96, 46], and [71]. Work aimed at making ITPs more suitable for novices and educational settings includes [94, 78, 80, 28, 40, 52, 87] and [82].

In addition to previous work specifically on theorem prover user interfaces, it is important to consider more general human-computer interaction research and research on software development tools. Starting with the latter, one can consider the data of [79] showing that the Eclipse IDE's "rename," "move," "extract", and "inline" refactoring commands are in fact used by many programmers and are frequently invoked via key bindings. The data also shows that a large percentage of *all* commands executed by developers using the IDE extensively were invoked via key bindings, and one of the top commands was "content assist", which suggests possible text to insert (similar to Proof Previews, though it does not show the effects of evaluating this text).

In addition to IDEs, we may consider software visualization tools: [22] report on a survey of users of software visualization tools such as *daVinci* (now called *uDraw(Graph)*), *GraphViz*, *Grasp* (now *jGrasp*, and *Tom Sawyer Software*–more than 40 different tools altogether. Some of the "functional aspects" of the software visualization tools that were considered most useful, and that might also have useful analogs in theorem prover user interfaces, were "search tools for graphical and/or textual elements", "hierarchical representations" and "navigation across hierarchies", "use of colors", and "easy access, from the symbol list, to the corresponding source code." Among functional aspects considered least useful were "3D representations and layouts, and virtual reality techniques" and "animation effects", though the later was considered "quite useful" when the software was implemented in a declarative language. Software visualization tools were considered beneficial in increasing productivity and managing complexity, and were considered particularly

important in the "software comprehension process."

A survey and taxonomy of software visualization is presented in [34]. Similar to [19], it divides tools into three groups according to their use at three different levels of abstraction: line, class, and architecture. They also differentiate between tools used for visualizing code at a particular time and those that allow for visualization of the evolution of software. Many of the techniques used by these tools could be applied to visualization for interactive theorem provers. *Seesoft* [42], for instance, miniaturizes lines of code with lines of color-coded pixels, and *sv3D* [76] extends this idea using three-dimensional arrays of blocks where information about lines of code is encoded in the height and color of the blocks. Other techniques potentially useful for interactive theorem prover user interfaces include using animation and color gradients to show the direction of relationships between software components [13, 54] and using texture and 3D object primitives called "geons" to encode additional information [55, 58].

Chapter 5

Conclusion and Acknowledgements

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.

Much of the work presented has been supported by NSF award CCF-1250306. I also give special thanks to my advisor, Professor Juan Pablo Hourcade, Professor Aaron Stump, and Harley Eades, for working on this project with me, and to my committee.

Chapter 6

Bibliography

[1] ACL2. http://www.cs.utexas.edu/users/moore/acl2/.

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

[3] Eclipse. http://www.eclipse.org/.

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

[5] jEdit. http://www.jedit.org/.

[6] Matita. http://matita.cs.unibo.it/.

[7] Proof General. http://proofgeneral.inf.ed.ac.uk/.

[8] ProofMood. http://www.proofmood.com/.

[9] ProofWeb. http://prover.cs.ru.nl/.

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

[11] Twelf. http://twelf.org/wiki/Main_Page.

[12] D. Ahlström, A. Cockburn, C. Gutwin, and P. Irani. Why it's quick to be square: modeling new and existing hierarchical menu designs. In *Proc. CHI*, pages 1371–1380. ACM Press, 2010.

[13] Sazzadul Alam and Philippe Dugerdil. Evospaces visualization tool: Exploring software architecture in 3d. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 269–270. IEEE, 2007.

[14] Ahmed Amerkad, Yves Bertot, Loïc Pottier, Laurence Rideau, et al. Mathematics and proof presentation in pcoq. 2001.

[15] Andrea Asperti and Claudio Sacerdoti Coen. Some considerations on the usability of interactive provers. In *Intelligent Computer Mathematics*, pages 147–156. Springer, 2010.

[16] David Aspinall. Proof general: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 38–43. Springer, 2000.

[17] Gilles Bailly, Alexandre Demeure, Eric Lecolinet, and Laurence Nigay. Multitouch menu (mtm). In *Proceedings of the 20th International Conference of the Association Francophone d'Interaction Homme-Machine*, IHM '08, pages 165–168, 2008.

[18] Gilles Bailly, Eric Lecolinet, and Yves Guiard. Finger-count & radial-stroke shortcuts: 2 techniques for augmenting linear menus on multi-touch surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 591–594, New York, NY, USA, 2010. ACM.

[19] Chandrajit Bajaj, Shashank Khandelwal, J Moore, and Vinay Siddavanahalli. *Interactive symbolic visualization of semi-automatic theorem proving.* Computer Science Department, University of Texas at Austin, 2003.

[20] 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.

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

[22] Sarita Bassil and Rudolf K Keller. Software visualization tools: Survey and analysis. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 7–17. IEEE, 2001.

[23] O Bau, E Ghomi, and W Mackay. Arpege: Design and learning of multi-finger chord gestures. *Submitted to ACM TOCHI*, 2010.

[24] Olivier Bau and Wendy E. Mackay. Octopocus: a dynamic guide for learning gesture-based command sets. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, UIST '08, pages 37–46, New York, NY, USA, 2008. ACM.

[25] Janet Bertot and Yves Bertot. Ctcoq: A system presentation. In *Algebraic Methodology and Software Technology*, pages 600–603. Springer, 1996.

[26] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, pages 141–160. Springer, 1994.

[27] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(2):161–194, 1998.

[28] William Billingsley and Peter Robinson. Student proof exercises using mathstiles and isabelle/hol in an intelligent book. *Journal of Automated Reasoning*, 39(2):181–218, 2007.

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

[30] Richard Bornat and Bernard Sufrin. Displaying sequent-calculus proofs in natural-deduction style: experience with the jape proof calculator. In *International Workshop on Proof Transformation and Presentation, Dagstuhl*. Citeseer, 1997.

[31] Richard Bornat and Bernard Sufrin. Animating formal proof at the surface: the jape proof calculator. *The Computer Journal*, 42(3):177–192, 1999.

[32] 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.

[33] John M Carroll and Mary Beth Rosson. Paradox of the active user. In *Interfacing thought: cognitive aspects of human-computer interaction*, pages 80–111. MIT Press, 1987.

[34] Pierre Caserta and Olivier Zendra. Visualization of the static aspects of software: a survey. *Visualization and Computer Graphics, IEEE Transactions on*, 17(7):913–933, 2011.

[35] Adam Chlipala. Certified programming with dependent types, 2011.

[36] 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.

[37] Pierre Corbineau. A declarative language for the coq proof assistant. In *Types for Proofs and Programs*, pages 69–84. Springer, 2008.

[38] Pierre Corbineau and Cezary Kaliszyk. Cooperative repositories for formal proofs. In *Towards Mechanized Mathematical Assistants*, pages 221–234. Springer, 2007.

[39] 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.

[40] Peter C Dillinger, Panagiotis Manolios, Daron Vroon, and J Strother Moore. Acl2s:the acl2 sedan. *Electronic Notes in Theoretical Computer Science*, 174(2):3–18, 2007.

[41] K Eastaughffe. Support for interactive theorem proving: Some design principles and their application. In *Workshop on User Interfaces for Theorem Provers*, pages 96–103, 1998.

[42] SC Eick, Joseph L Steffen, and Eric E Sumner Jr. Seesoft-a tool for visualizing line oriented software statistics. *Software Engineering, IEEE Transactions on*, 18(11):957–968, 1992.

[43] William M Farmer and Orlin G Grigorov. Panoptes: An exploration tool for formal proofs. *Electronic Notes in Theoretical Computer Science*, 226:39–48, 2009.

[44] David J Field, Anthony Hayes, and Robert F Hess. Contour integration by the human visual system: Evidence for a local association field. *Vision research*, 33(2):173–193, 1993.

[45] 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*.

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

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

[48] Steve Graham, Virginia Berninger, Naomi Weintraub, and William Schafer. Development of handwriting speed and legibility in grades 1–9. *The Journal of Educational Research*, 92(1):42–52, 1998.

[49] T. Grossman, P. Dragicevic, and R. Balakrishnan. Strategies for accelerating on-line learning of hotkeys. In *Proc. CHI*, pages 1591–1600. ACM Press, 2007.

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

[51] Tara Haelle. What is the answer to that stupid math problem on facebook? `http://www.slate.com/articles/health_and_science/science/2013/03/facebook_math_problem_why_pemdas_doesn_t_always_give_a_clear_answer.html`, 2013.

[52] Maxim Hendriks, Cezary Kaliszyk, Femke van Raamsdonk, and Freek Wiedijk. Teaching logic using a state-of-the-art proof assistant. *Acta Didactica Napocensia*, 3(2):35–48, 2010.

[53] J. Hendy, K. Booth, and J McGrenere. Graphically enhanced keyboard accelerators for guis. In *Proc. GI '10*, pages 3–10. Canadian Information Processing Society, 2010.

[54] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):741–748, 2006.

[55] Danny Holten, Roel Vliegen, and Jarke J Van Wijk. Visual realism for the visualization of software metrics. In *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on*, pages 1–6. IEEE, 2005.

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

[57] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[58] Pourang Irani, Maureen Tingley, and Colin Ware. Using perceptual syntax to enhance semantic content in diagrams. *Computer Graphics and Applications, IEEE*, 21(5):76–84, 2001.

[59] Deepak Kapur, Xumin Nie, and David R. Musser. An overview of the tecton proof system. *Theoretical Computer Science*, 133(2):307–339, 1994.

[60] 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.

[61] Brian Krisler and Richard Alterman. Training towards mastery: overcoming the active user paradox. In *Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges*, NordiCHI '08, pages 239–248, New York, NY, USA, 2008. ACM.

[62] G. Kurtenbach and W. Buxton. User learning and performance with marking menus. In *Proc. CHI*, pages 258–264. ACM Press, 1994.

[63] Gordon Kurtenbach, George W. Fitzmaurice, Russell N. Owen, and Thomas Baudel. The hotbox: efficient access to a large number of menu-items. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '99, pages 231–237, New York, NY, USA, 1999. ACM.

[64] Gordon Paul Kurtenbach. *The design and evaluation of marking menus*. PhD thesis, University of Toronto, 1993.

[65] D. Lane, H. Napier, C. Peres, and A. Sandor. The hidden costs of graphical user interfaces: The failure to make the transition from menus and icon toolbars to keyboard shortcuts. *International Journal of Human-Computer Interaction*, 18:133–144, 2005.

[66] 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.

[67] G. Julian Lepinski, Tovi Grossman, and George Fitzmaurice. The design and evaluation of multitouch marking menus. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2233–2242, New York, NY, USA, 2010. ACM.

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

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

[70] 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.

[71] Christoph Lüth. User interfaces for theorem provers: Necessary nuisance or unexplored potential? *Electronic Communications of the EASST*, 23, 2009.

[72] Christoph Lüth and Burkhart Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167–189, 1999.

[73] K. Lyons, N. J. Patel, and T. Starner. Keymenu: A keyboard based hierarchical menu. In *Proc. ISWC '03*, pages 240–241. IEEE Computer Society, 2003.

[74] Sylvain Malacria, Gilles Bailly, Joel Harrison, Andy Cockburn, and Carl Gutwin. Promoting hotkey use through rehearsal with exposehk. 2013.

[75] Shahzad Malik. *An exploration of multi-finger interaction on multi-touch surfaces*. PhD thesis, University of Toronto, 2007.

[76] Andrian Marcus, Louis Feng, and Jonathan I Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff. ACM, 2003.

[77] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2013. Version 8.4pl2.

[78] Andreas Meier, Erica Melis, and Martin Pollet. Adaptable mixed-initiative proof planning for educational interaction. *Electronic Notes in Theoretical Computer Science*, 103:105–120, 2004.

[79] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.

[80] Julien Narboux. A graphical user interface for formal proofs in geometry. *Journal of Automated Reasoning*, 39(2):161–180, 2007.

[81] K. L. Norman. *The Psychology of Menu Selection: Designing Cognitive Control at the Human/Computer Interface*. Ablex Publishing Corporation, Norwood, NJ, 1991.

[82] Benjamin Pierce. Proof Assistant as Teaching Assistant: A View from the Trenches. `http://www.cis.upenn.edu/~bcpierce/papers/LambdaTA-ITP.pdf`, 2010. Keynote address at the International Conference on Interactive Theorem Proving (ITP).

[83] 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.

[84] Kevin Purdy. Make chrome less distracting with vimium (and these settings). `http://lifehacker.com/5925220/make-chrome-less-distracting-with-vimium-and-these-settings/`, 2012.

[85] Pedro Quaresma and Predrag Janičić. Geothmsa web system for euclidean constructive geometry. *Electronic Notes in Theoretical Computer Science*, 174(2):35–48, 2007.

[86] Joey Scarr, Andy Cockburn, Carl Gutwin, and Philip Quinn. Dips and ceilings: understanding and supporting transitions to expertise in user interfaces. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, pages 2741–2750. ACM, 2011.

[87] Wolfgang Schreiner. The risc proofnavigator: a proving assistant for program verification in the classroom. *Formal Aspects of Computing*, 21(3):277–291, 2009.

[88] B. Shneiderman and C. Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Boston, fifth edition, 2010.

[89] J Siekmann, S Hess, C Benzmüller, L Cheikhrouhou, A Fiedler, H Horacek, M Kohlhase, K Konrad, A Meier, E Melis, et al. Loui: L ovely ω mega u ser i nterface. *Formal Aspects of Computing*, 11:326–342, 1999.

[90] ACM SIGPLAN. Programming languages software award. *Announcement at http://www.sigplan.org/Awards/Software/Main*, 2013.

[91] Carst Tankink, Herman Geuvers, and James McKinna. Narrating formal proof (work in progress). *Electronic Notes in Theoretical Computer Science*, 285:71–83, 2012.

[92] Laurent Théry, Yves Bertot, and Gilles Kahn. Real theorem provers deserve real user-interfaces. In *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, SDE 5, pages 120–129, New York, NY, USA, 1992. ACM.

[93] Steven Trac, Yury Puzis, and Geoff Sutcliffe. An interactive derivation viewer. *Electronic Notes in Theoretical Computer Science*, 174(2):109–123, 2007.

[94] Dimitra Tsovaltzi and Armin Fiedler. An approach to facilitating reflection in a mathematics tutoring system. In *Proceedings of AIED Workshop on Learner Modelling for Reflection*, pages 278–287, 2003.

[95] Paul D. Varcholik, Joseph J. LaViola Jr., and Charles E. Hughes. Establishing a baseline for text entry for a multi-touch virtual keyboard. *International Journal of Human-Computer Studies*, 70(10):657 – 672, 2012. ¡ce:title¿Special issue on Developing, Evaluating and Deploying Multi-touch Systems¡/ce:title¿.

[96] Norbert Völker. Thoughts on requirements and design issues of user interfaces for proof assistants. *Electronic Notes in Theoretical Computer Science*, 103:139–159, 2004.

[97] W3Schools. Browser Display Statistics. `http://www.w3schools.com/browsers/browsers_display.asp`, 2013.

[98] Makarius Wenzel. Asynchronous proof processing with isabelle/scala and isabelle/jedit. *Electronic Notes in Theoretical Computer Science*, 285:101–114, 2012.