DEVELOPMENT AND USER TESTING OF NEW USER INTERFACES FOR MATHEMATICS AND PROGRAMMING TOOLS

by

Benjamin Berman

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

September 2014

Thesis Supervisor: Associate Professor Juan Pablo Hourcade

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Benjamin Berman

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the September 2014 graduation.

Thesis Committee: _____
Juan Pablo Hourcade, Thesis Supervisor

_____
Member Two

_____
Member Three

_____
Member Four

_____
Member Five

TABLE OF CONTENTS

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

The general principle behind this dissertation is that in order to accomplish difficult tasks, one generally needs to make these tasks easy–for moving boulders you might want some leverage. 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 this dissertation is to apply the principle in the context of the virtuosic tasks that are involved in interactive theorem proving: I intend to show ways to make the difficult task of using interactive theorem provers easier by improving their user interfaces. As well as solving serious usability problems for important and powerful tools for creating machine-checked proofs, many of the techniques I am developing and testing are widely applicable to other forms of coding.

My work has focused on the *Coq* proof assistant[11] [1]. In chapter 2, this is justified by arguing that Coq is a particularly important and powerful, but complex

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

and hard-to-use, interactive theorem prover. More specifically, in chapter 2 I first give a description of Coq, explain the significance of interactive theorem proving in general and Coq in particular, give an example of theorem proving using the tool, describe current user interfaces, and describe some usability problems that I find particularly striking. I continue with a description of a survey (including its results) on user interfaces for Coq that was sent to subscribers to the Coq-Club mailing list. Chapter 5 gives further motivation by summarizing related work.

The core research involved in this dissertation is described primarily in chapter 3 and chapter 4. Chapter 3 describes "CoqEdit", a new theorem proving environment for Coq, based on the jEdit text editor. CoqEdit mimics the main features of existing environments for Coq, but has the important property of being relatively easy to extend using Java. Chapter 3 continues with a description of prototypes of two potential extensions to CoqEdit. The ideas for these prototypes were based in part on responses from the survey and several more in-depth interviews with expert users (graduate students) and in part on my own experience and intuition and the experience and intuition of my collaborators, Aaron Stump, Harley Eades, and Juan Pablo Hourcade. The chapter concludes with a description of a user study examining how these two extensions help, and potentially hinder, novice Coq users.

Chapter 4 describes a third prototype. Although this prototype has not been tested with users, it presents a new, *general* scheme for manipulating text that I hope may be built upon and widely applied. The scheme involves the combination of two relatively novel ideas: "Keyboard-Card Menus" and "Syntax Tree Highlighting". I

describe both these two ideas and how they may be combined to help, for instance, introductory logic students using a particular subset of Coq.

There are several points that I hope will become clear in this dissertation. First is how the research makes a *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 of dealing with this complexity through 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.[2] 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 contributes 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

[2]The research draws from and contributes to two generally separate sub-disciplines of computer science, namely programming languages theory and human-computer interaction. I have Professors Juan Pablo Hourcade and Aaron Stump to thank for facilitating, and being involved with, unusual interdisciplinary work.

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 dissertation is that this research is 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 with human subjects.

The third and final point to be made clear in this dissertation is simply that developing and testing new user interfaces for mathematics and programming tools is a rich area of research. Further important and interesting questions can be both

raised and answered.

CHAPTER 2

INTERACTIVE THEOREM PROVING, COQ, AND THE NEED FOR IMPROVED USER INTERFACES

2.1 Basic Theorem Proving in Coq

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

One example useful in making this more clear can be found in Huet, Kahn, and Paulin-Mohring's Coq tutorial [61]. 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 A, B, and C are propositional variables and "$\to$" means "implies" and is right-associative[1] (I discuss later how improved user interfaces may assist users, particularly novice users, in dealing with operator associativities and precedences). Assuming, also, that we have opened up a new "section" where we have told Coq that A, B, and C are propositional variables, when we enter this lemma at the prompt, Coq responds by printing out

```
A : Prop

B : Prop

C : Prop

------------------------------------

(A -> B -> C) -> (A -> B) -> A -> C
```

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

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

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

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

be considered.[2] Another way to think about the sequent is that the statements to the left of the turnstile, taken together, 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

[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 not considered worth the trouble.

new subgoal, replacing (2.2), therefore is

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

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

Tactics allow users to reason "backwards"–if the user proves the new se-

[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, in Coq, 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 term 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.

quents(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** given a context containing that A, B, and C are propositions *and* $A \to B \to C$ it is necessarily the case that $(A \to B) \to A \to C$, **then** given 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 about the truth of this statement than he would if he just checked it by hand.[5]

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

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C, H' : A \to B, HA : A \vdash C \qquad (2.4)$$

[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 value is positive, not just nonzero).

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

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C, H' : A \to B, HA : A \vdash A \qquad (2.5)$$

and

$$A : Prop, B : Prop, C : Prop, H : A \to B \to C, H' : A \to B, HA : A \vdash B \qquad (2.6)$$

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 in the child goals are the same as in the parent. Also note that the first of these two becomes the current goal.

The next tactic, "`exact HA`," eliminates (2.5) without replacing it with any new goal (which makes sense, since 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.

2.2 The Significance of Interactive Theorem Proving in General and Coq in Particular

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 and that the specifications themselves have reasonable properties). While fully-automatic SAT solvers (for propositional satisfiability) and SMT (satisfiability modulo theory) solvers are being used to implement advanced static analysis techniques with promising results (e.g. [56, 44]) 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

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

theorem provers to automatic tools [34] 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 [66] in Isabelle[5], the CompCert verified compiler[75] for Clight (a large subset of the C programming language) in Coq, Java Card EAL7 certification[51] using Coq, and, at higher levels of abstraction, verification of the type safety of a semantics for Standard ML [73] using Twelf[12] and use of the CertiCrypt framework [2] built on top of of Coq to verify cryptographic protocols (e.g. [22]).[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 [53]. Other interactive theorem provers are also having success in general mathematics. For instance, Matita [7], which is closely related to Coq, was used in a proof of Lebesgue's dominated convergence theorem [41]. There are in fact efforts to create libraries of formalized, machine-checked mathematics, the largest of which is the Mizar Mathematical Library [52]. 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 [37].

The potential for transparency also helps make interactive theorem provers,

[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.

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*[91], 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* [77], *Logic Tutor* [76], *Fitch* (software accompanying the textbook *Language, Proof, and Logic* [23]), and *ProofMood*[9].

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* [40], 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 further evidence of its resulting success, note that Coq was awarded both the 2013 ACM SIGPLAN Programming Languages Software Award [100] and the 2013 ACM Software System Award [13].

2.3 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[84] 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*[8, 18] and *CoqIDE* (which is available from the Coq website[11], and is bundled with Coq). Interacting with Coq using one of these interfaces 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).

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 $(A \to B \to C) \to (A \to$

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

$B) \rightarrow A \rightarrow C$) 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 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, so 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

---

[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 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 $\forall(A : Prop), (\forall(B : Prop), (\forall(C : Prop), ((A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C)))$, generally written `forall A B C : Prop, (A -> B -> C) -> (A -> B) -> A -> C`.

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 messages 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 is never actually visible to the user.

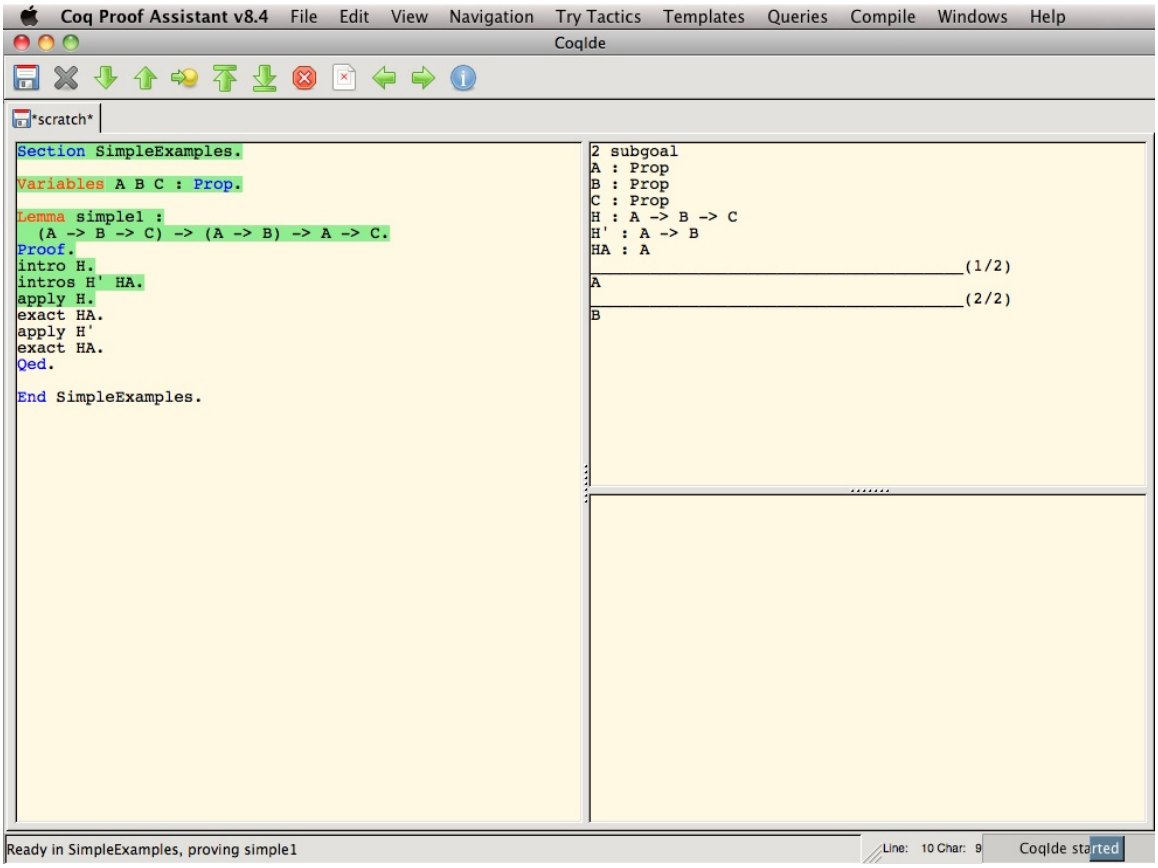

Figure 2.1. CoqIde, displaying the result of entering the tactic "`apply H`" in the top-right panel within the proof of `(A -> B -> C) -> (A -> B) -> A -> C`.

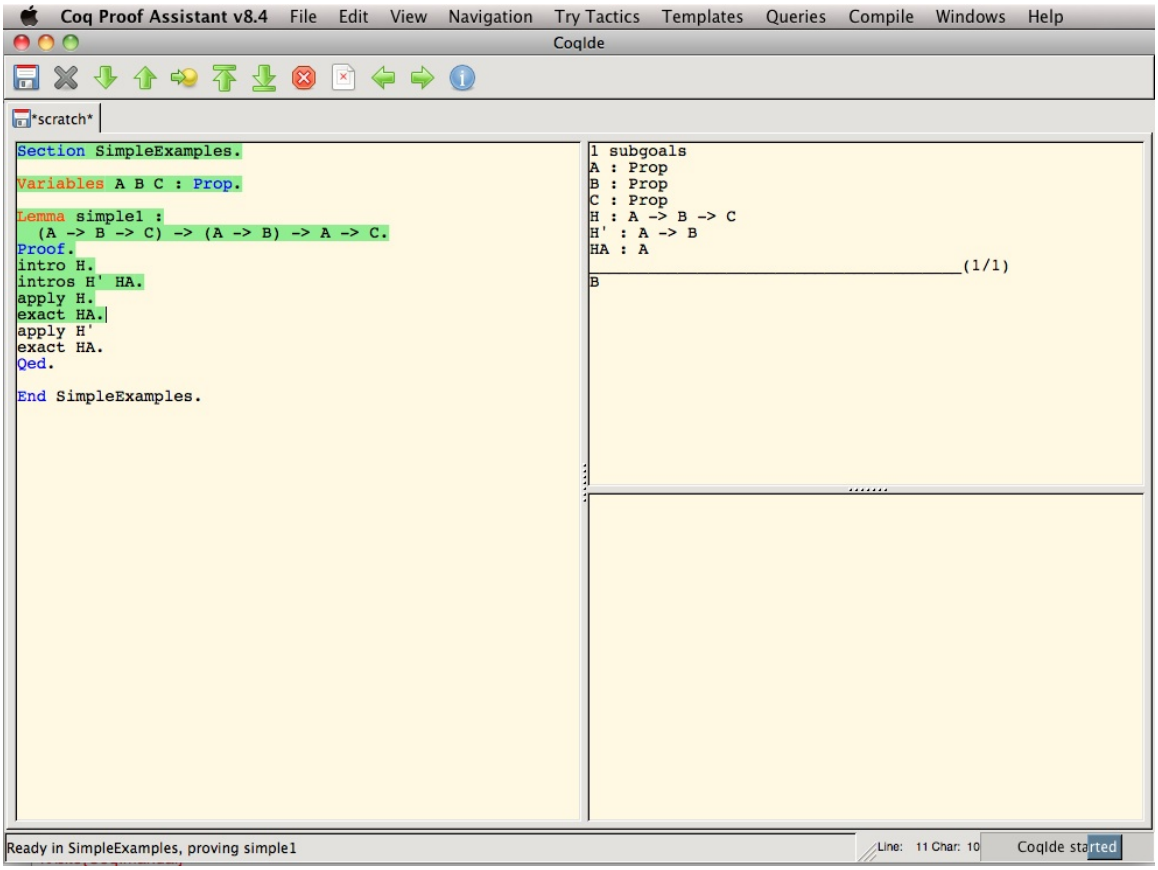

Figure 2.2. CoqIde after moving the end of the evaluated portion of the script forward by one sentence from the state shown in Figure 2.1.

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, namely 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 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 fact that users must move through the script to fully see siblings, 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 that is immediately before the cursor. Unfortunately, these features are not really aimed at showing the effects of tactics. A third user interface, *ProofWeb*[10] 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

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

is able to much more directly see the goal to which `apply H` is applied and the goals this application produces.

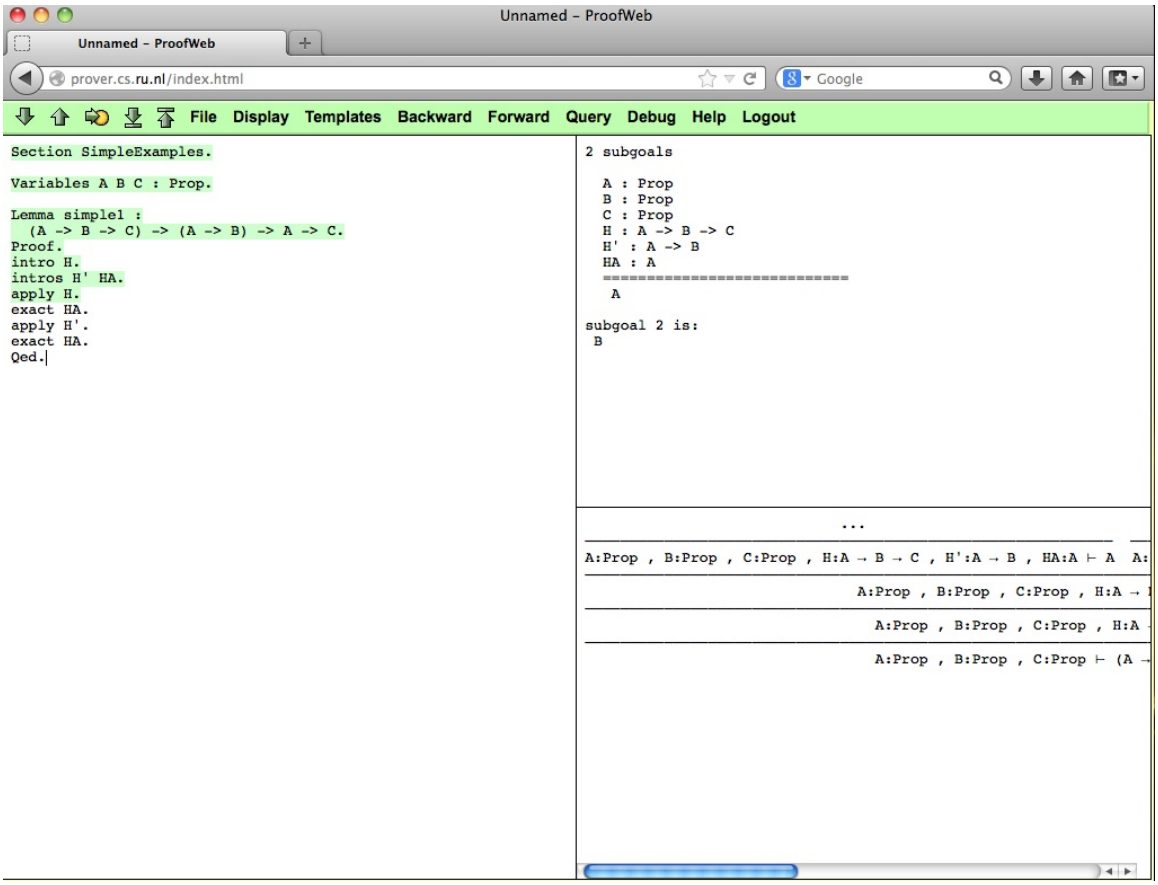

Figure 2.3. ProofWeb, with a partial proof tree displayed in the bottom right.

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, the user may have to pan around the window

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

to see the effect of a tactic; 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 are printed next to one 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 also 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).

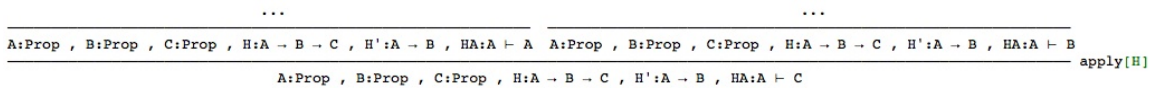

Figure 2.4. The portion of ProofWeb's tree visualization corresponding to the tactic "`apply H`".

[13]This is the Gestalt law of "good continuation; see, for instance, [50].

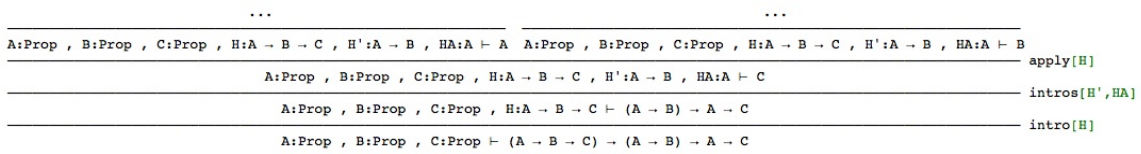

Figure 2.5. The partially completed tree from Figure 2.3, fully displayed.

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.

2.4 Coq User Interface Survey

Section 2.3 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

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

of mathematics, and 8 indicated teaching.

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,
- 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 (e.g. to what the task might or might not in comparison be difficult was left unspecified).

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 (very roughly) 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

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

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 was with lack of good tutorials. The third category was that there are numerous poorly documented commands (the need for simple examples was mentioned again). Finally, the 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

COQEDIT, PROOF PREVIEWS, AND PROOF TRANSITIONS

3.1 Software Description

3.1.1 Basic CoqEdit

The basic version of CoqEdit is a jEdit plugin providing a new user interface to Coq that is intended to imitate CoqIDE (see Figure 2.1), Proof General, and ProofWeb (see Figure 2.3), described in section 2.3. A significant difference between it and these previous user interfaces, however, is the use of two different shades of green, as seen in Figure 3.1: all sentences highlighted with (some sort of) green are ones that have been successfully evaluated and cached, but the sentence highlighted with dark green has the result of its evaluation displayed in one of the two sub-windows on the right. The user can move the dark green sentence around in this cached area. This is in contrast to user interfaces like CoqIDE where there is only one shade of green, the evaluation result displayed on the right is always that of the last green sentence, and there is no caching of output. Using CoqIDE, to check the result of an sentence evaluated earlier one must actually undo the evaluation of subsequent sentences (Proof General, like CoqEdit, caches evaluated sentence output, but has users hover over the evaluated sentence to see its result). CoqIDE gets away with this because, frequently, re-evaluating a sentence is instantaneous. However, this is not always the case, especially when using tactics that automatically search for proofs for goals (in general these are actually semi-decidable).

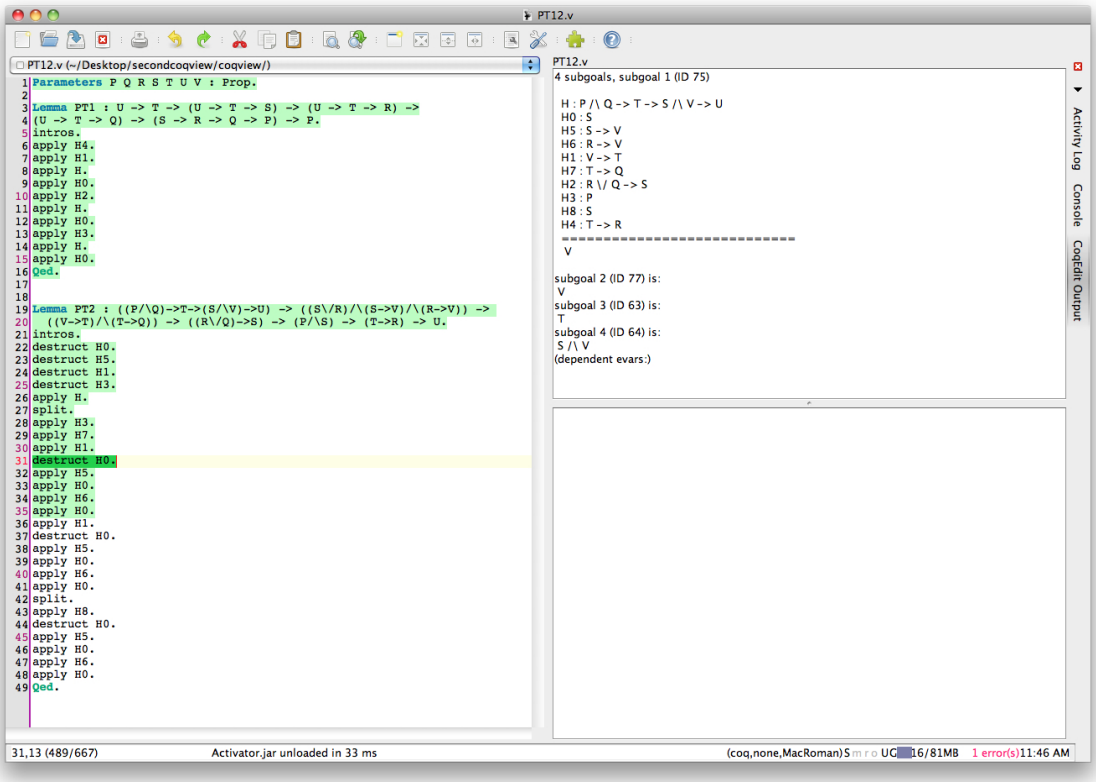

Figure 3.1. The basic CoqEdit user interface

To move the dark green highlighting forwards or backwards, one can select drop-down menu items (not shown in Figure 3.1, accessible through the "CoqEdit" submenu added to jEdit's top-level "Plugins" menu. However, it is expected that generally users will use shortcuts which can be added and customized through the jEdit options menu (e.g. Ctrl-N for "Forward one sentence" and Ctrl-P for "Back one sentence"). There are 5 other menu items, which I hope are fairly self-explanatory:

- "Show CoqEdit Output Panel,"
- "Go to cursor,"

- "Go to start,"
- "Go to end," and
- "Interrupt evaluation"

There are a few details regarding the user interface which may be unclear from the labels. "Forward one sentence" moves the dark green highlighting forward within the light green section *and*, when the dark green sentence is at the end of the green section, extends a purple section forward as seen in Figure 3.2. This purple section shows the sentence currently being evaluated and the subsequent sentences queued for processing.

Frequently, evaluation occurs quickly enough that the purple section is invisible, but, as mentioned earlier, long-running and non-terminating commands may be encountered. For these cases, the "Interrupt evaluation" menu item is provided which, in addition to interrupting any current sentence processing, removes all sentences from the queue of sentences to process and removes all purple highlighting.

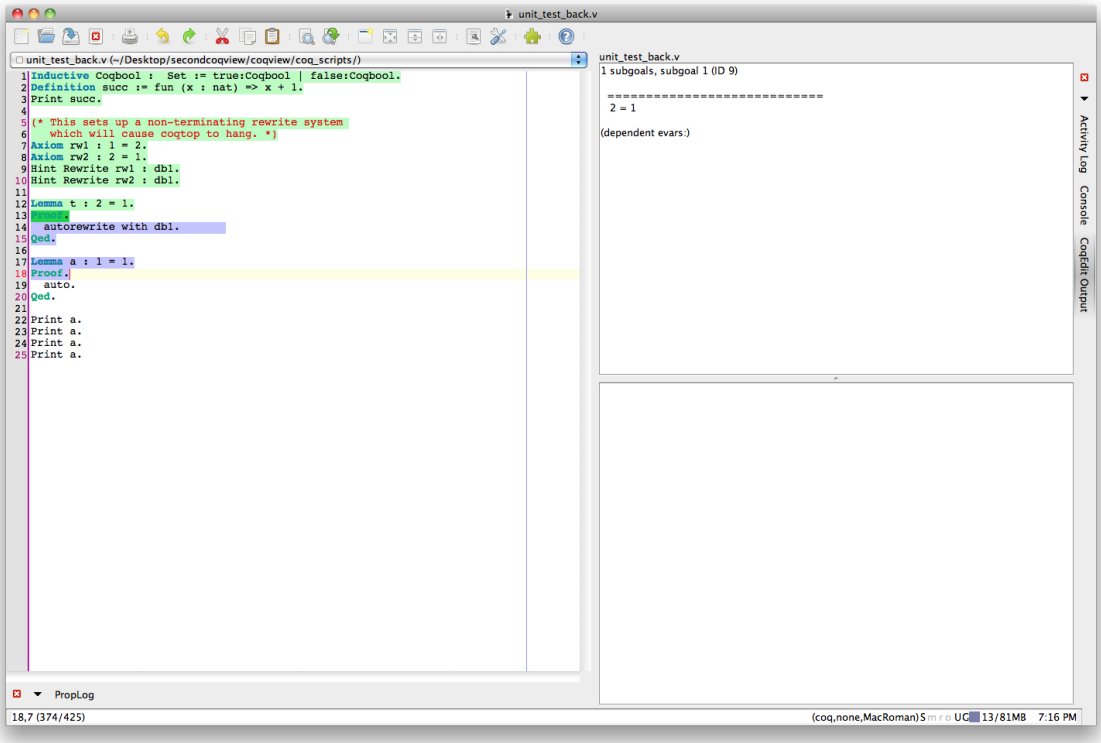

Figure 3.2. The basic CoqEdit user interface, when the first sentence highlighted with purple is being evaluated and the subsequent purple-highlighted sentences are queued for processing. Thanks go to Harley Eades for providing this example of a non-terminating tactic.

The "Interrupt evaluation" menu item is not strictly necessary; an equivalent result could be achieved by inserting a character into the first purple sentence (and then deleting that character). In general, typing into a highlighted region, green, purple, or red (used to indicate errors–see Figure 3.1.1), undoes the evaluation/partial processing/queuing of the sentence into (from) which characters were inserted (deleted), and from all subsequent sentences. To reflect this, the highlighting from

these sentences is removed.[1]

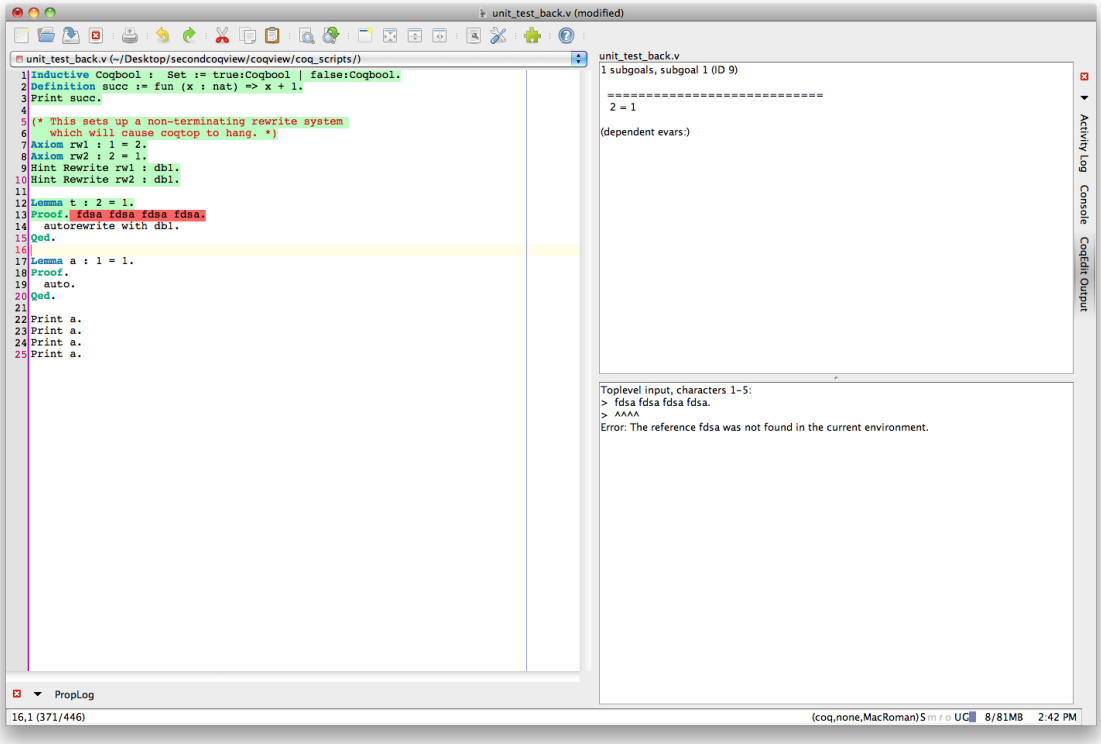

Figure 3.3. The basic CoqEdit user interface, with an error-producing sentence highlighted in red.

To try to evaluate all sentences in the buffer, the user can use the "Go to end" menu item. This of course may not always be successful since non-terminating and error sentences may be encountered, as in Figure 3.2 and . Note that when an error is processed, processing stops, the sentences-to-process queue is emptied, and

[1]Typing into the sequence of tactics used to prove an earlier theorem or lemma actually causes the highlighting to be moved to just before the start of the lemma; this reflects the behavior of the underlying command line tool.

the highlighting is updated to reflect this. Users are thus informed of incorrect proof attempts.

"Go to end" is effectively the same as repeated invoking "Forward one sentence" until there are no more sentences that can be queued for evaluation. Similarly, "Go to start" is effectively the same as invoking "Back one sentence" repeatedly until the dark green highlighting is at the first sentence of the buffer, and "Go to cursor" attempts to move the dark green highlighting forwards or backwards to the sentence containing the cursor by invoking either "Forward one sentence" or "Back one sentence" repeatedly (when the first purple sentence is evaluated, the dark green highlighting will move to it).

The navigational commands have the side effect of moving the cursor either to the end of the purple section (if moving forward) or to the end of the dark green sentence. Moving the cursor in turn has the side effect of automatically scrolling the window so as to keep the cursor (and the most recently changed highlighting) visible. This turns out to be highly convenient, allowing the user to avoid a large amount of manual scrolling.

As a jEdit plugin, CoqEdit lives in an ecosystem of other plugins which may have specified dependencies on one another.[2] One can view jEdit plugins that depend on CoqEdit as plugins to the plugin. The next subsections describe two ideas for such plugins that were developed, and the next section describes how these ideas were

[2]If plugin A is specified (in its properties file) as depending on plugin B, then loading plugin A will automatically load plugin B (if it can be found among the downloaded plugins and is not already loaded) and unloading plugin B will (with a warning popup window) unload plugin A.

tested with users in an experiment. As mentioned in the introduction, these ideas were developed somewhat informally and were based on survey responses, several more in-depth interviews with graduate students who use Coq, my own experience and intuition and the experience and intuition of Aaron Stump, Harley Eades, and Juan Pablo Hourcade, my collaborators on this project.

3.1.2 Proof Transitions

"Proof Transitions" is a prototype for a proof tree visualization plugin for CoqEdit (or a similar plugin-based user interface for Coq or Coq-like proof assistants). As a jEdit/CoqEdit plugin it would function in a manner similar to that of the proof tree visualization of ProofWeb (see Figure 2.3, Figure 2.4, and Figure 2.5): as sentences of the buffer are evaluated and highlighted in green, nodes are added to the proof tree visualization. However, the actual system, in its current prototype state at least, exists as a limited set of visualizations of complete proofs generated by.[3] A fully developed plugin is beyond the scope of the research presented here–the goal of this research is to provide ideas for and insight into future work on such visualizations.[4]

---

[3]More specifically, I reverse-engineered parts of Coq's tactic system and pretty printing system so that the fairly complex visualizations for several large propositional logic proofs could be generated relatively easily from an initial goal's syntax tree and a tree of tactics used to prove the initial goal.

[4]User interface development, particularly for proof assistants it seems, presents a chicken-and-egg problem: back-end developers would like to avoid providing access to information that front-end developers don't use, but front-end developers have a hard time saying if they need access to that information until they test interfaces that use that that access.

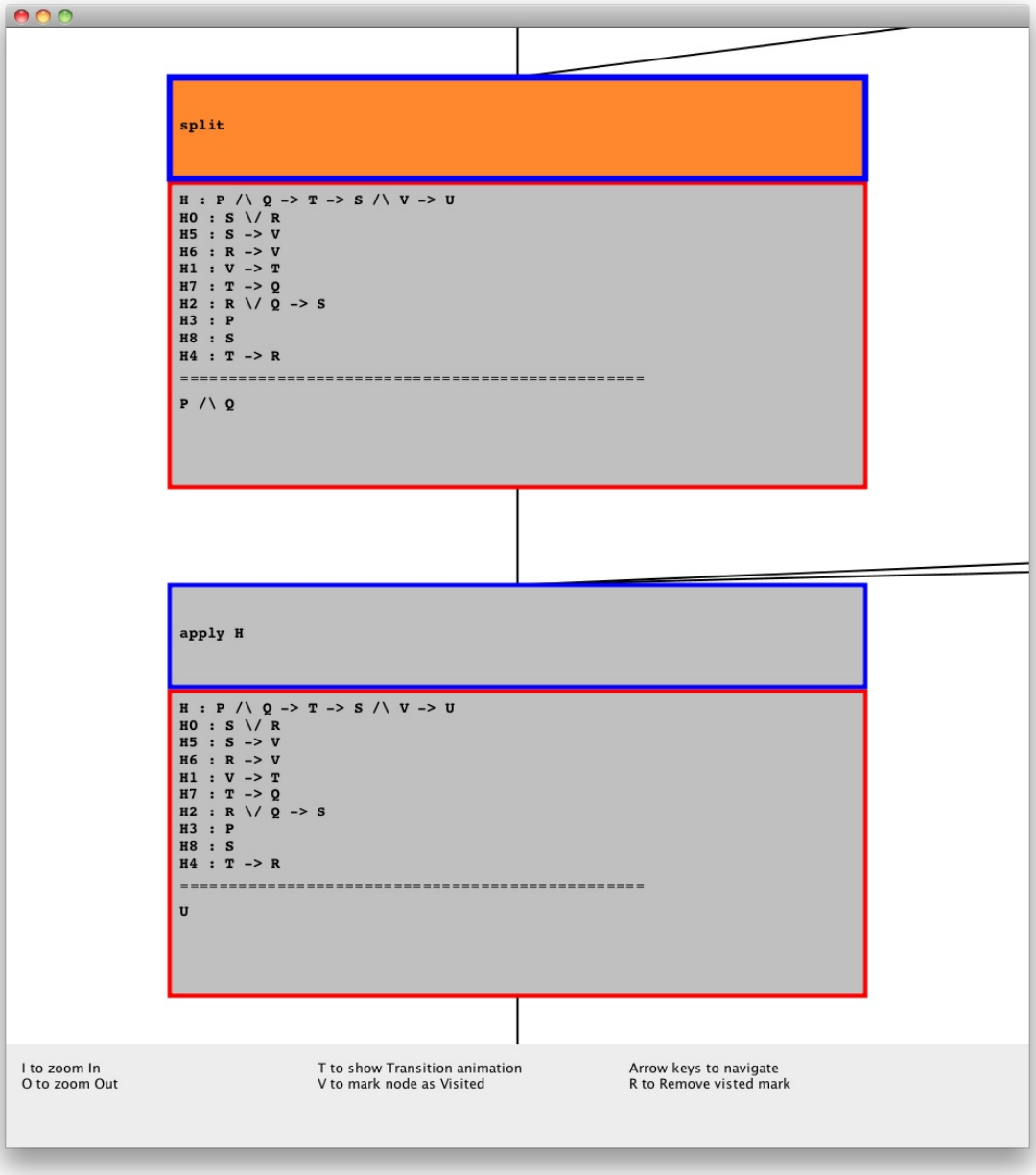

Figure 3.4

Proof Transitions provides a zoomed-in and a zoomed-out level for viewing proofs. Figure 3.4 shows the zoomed-in level. Each node contains two parts: the bottom box, outlined in red, contains the text of the goal, while the smaller top box,

outlined in blue, contains the tactic used on that goal.

At the zoomed-in level two nodes are always visible: the "current" node, which has its top box colored orange, and the current node's parent node, or, if the current node is the root node, the root node and its left-most child. As one can see in Figure 3.4, the visualization follows the convention where parent nodes are displayed below their children.

Figure 3.5 shows how the proof tree is displayed when the user zooms out from the current node in Figure 3.4. At the bottom of the window in both Figure 3.4 and Figure 3.5 is a "cheat sheet" for the key presses used to manipulate the visualization. From this, the user can learn or be reminded that the I and O keys are used to zoom in and out respectively. Since there are only two zoom levels, pressing I while zoomed in, or O while zoomed out, does nothing.

Figure 3.6, Figure 3.7, and Figure 3.8 show what happens when one uses the arrow keys to move the current node to the sibling directly to right, twice, and then to the left-most child of the third of the three siblings (right arrow key, right arrow key, up arrow key). Repeatedly using the left and right arrow keys will cycle through the current set of siblings.

As shown, the visualization moves nodes so as to keep the current node above its parent (so that both the current node and its parent are visible when one zooms in). To try to keep the shape of the tree relatively stable (i.e. to help users know which nodes are the same in Figure 3.5, Figure 3.6, Figure 3.7, **??**, for instance), the subtrees under the siblings, "uncles", "great uncles", etc., are each moved as a

unit with each subtree getting its own allotment of space horizontally, *and*, except for the parent of the current node, each parent node is kept centered above its left-most child (moving to the parent node from a child node other than the left-most requires shifting the subtree under the parent to the right to maintain this property).

To make it even more clear that one is looking at the same set of nodes when one moves to a sibling, child, or parent node, the sliding around of nodes is animated over a fraction of a second. Figure 3.13 shows a snapshot of what moving to a sibling looks like when zoomed in. Note that this animation does not limit the speed at which one can move from node to node (pressing, say, the right arrow key, while an animation is playing moves the nodes to where they would be at the end of that animation and then starts the new animation).

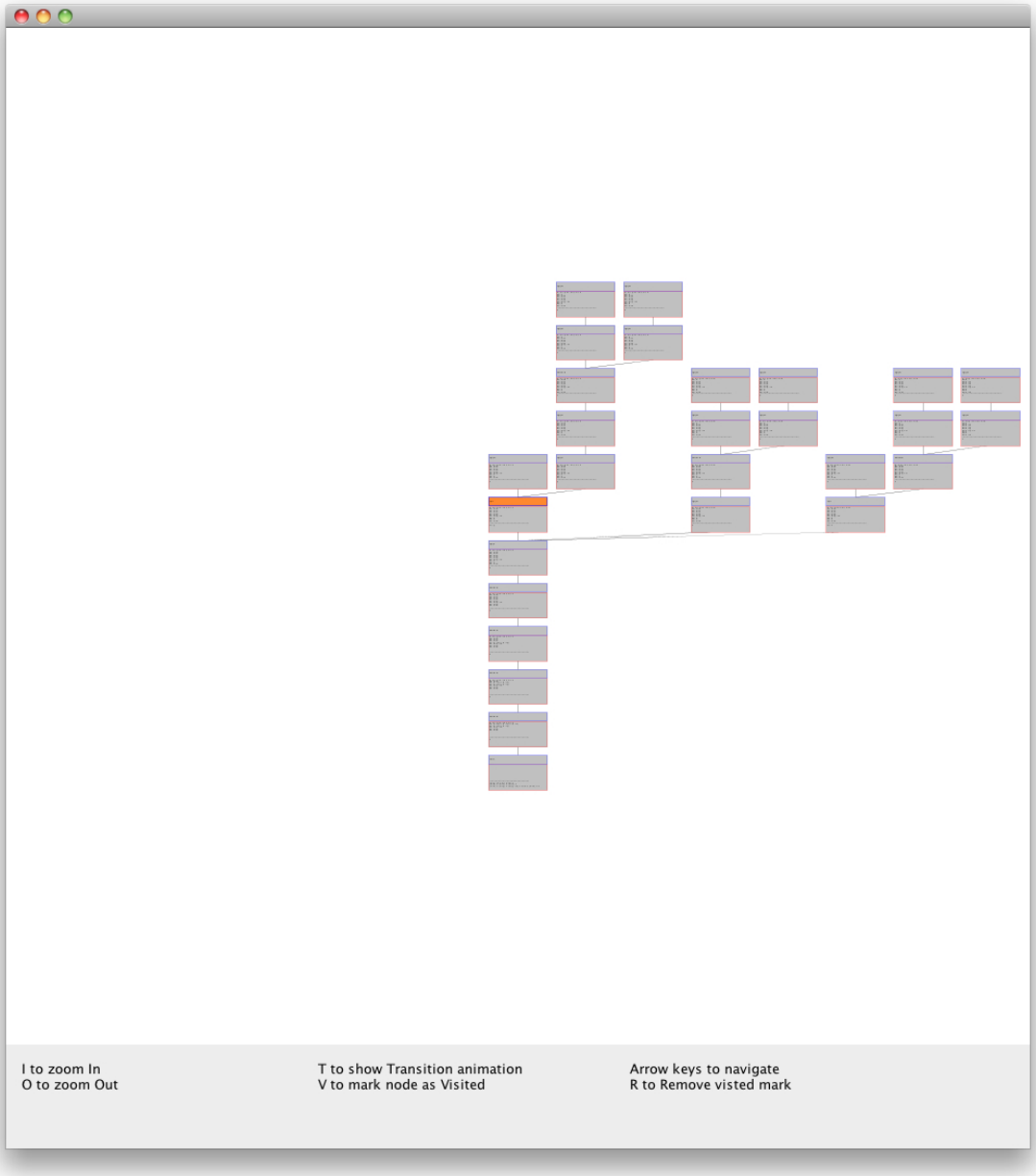
I to zoom In
O to zoom Out
T to show Transition animation
V to mark node as Visited
Arrow keys to navigate
R to Remove visted mark

Figure 3.5

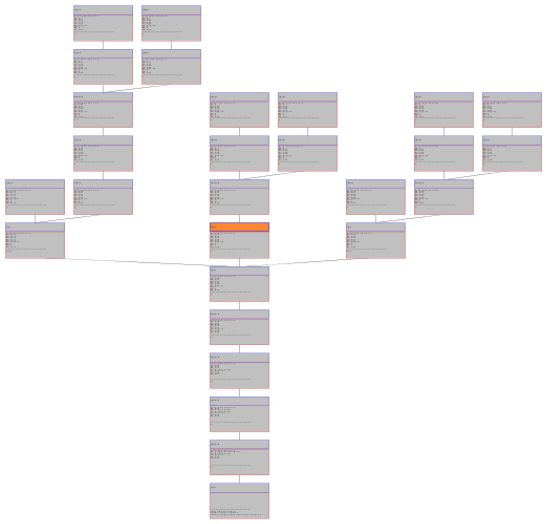

Figure 3.6

Figure 3.7

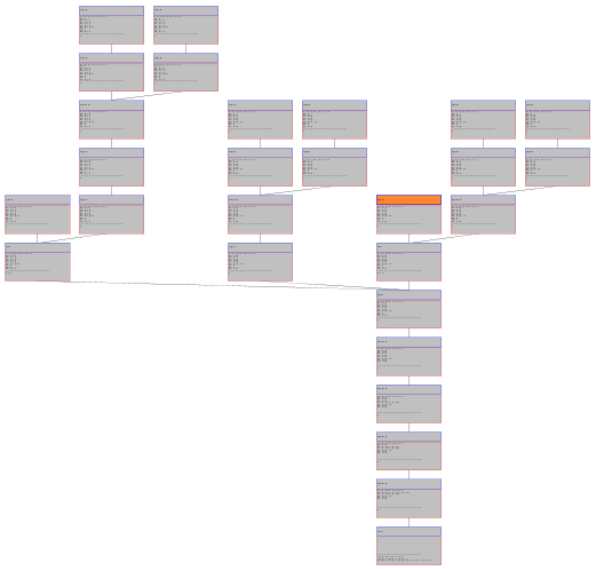

Figure 3.8

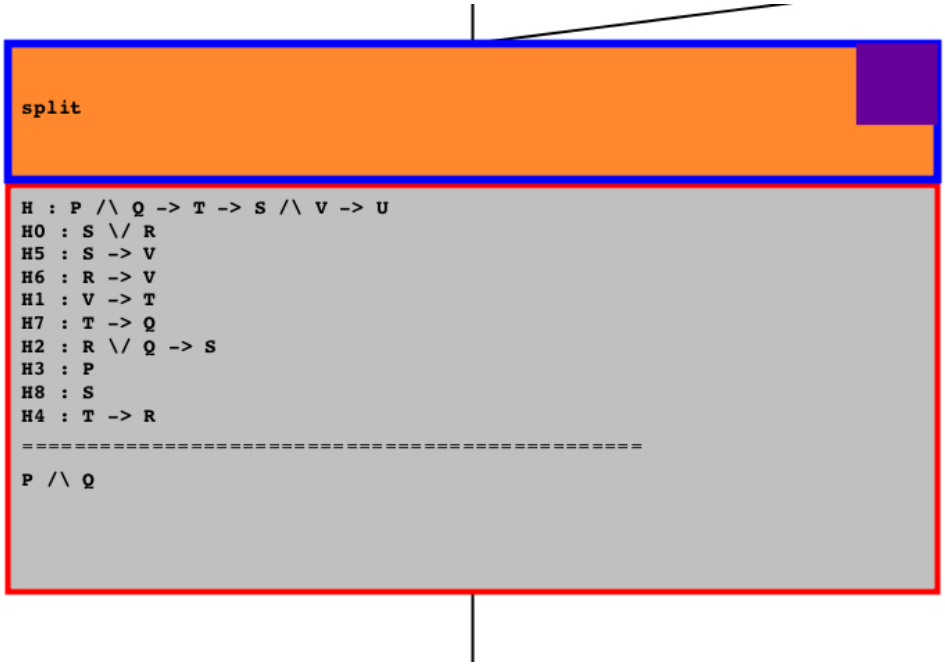

Figure 3.9

One problem that became evident in initial "pilot" testing was that it was hard for users to keep track of which nodes they had inspected. Users both visited nodes to which they had already been, thinking that they had not been there before, and they missed branches, thinking that explored them when in fact they had not. To combat this problem, users are given the ability to add a small purple square to the upper right corner of a node to mark it as visited. This is shown in Figure 3.9; these purple squares are also visible when zoomed out. Adding a purple square to the current node is accomplished by pressing the V key and removing a purple square (that might accidentally have been added) is a accomplished by pressing the R key. These marks can be added and removed both while zoomed out and while zoomed in.

In Figure 3.10 through Figure 3.17 we see the other major feature of Proof Transitions—transition highlighting and animation. Pressing the T key has the following effects:

- If the user is not already zoomed in, the visualization zooms in as if the I key were pressed.
- The view shifts upwards so that the current node (which, recall, has the orange tactic box) is in the bottom half of the window and its left-most child is in the top half; see Figure 3.10 and Figure 3.11.
- In the bottom node, text that disappears completely in every child node is highlighted in red. In Figure 3.10 this happens to be a conjunction symbol

("/\") whereas in Figure 3.11 through **??** (which all show the same parent and set of children) it is a disjunction symbol ("\/"). This highlighted difference is actually the reason one child node is created by the "destruct" tactic seen in Figure 3.10 in contrast to the the two nodes created by the same destruct tactic (albeit with a different argument) in Figure 3.11 through Figure 3.17.

- At the same time that the red highlighting is added to the parent node, green highlighting is added to each child node to mark text that is new in the child node (i.e. does not correspond in some way with text in the parent node). Note that while there is green highlighting in Figure 3.10, since "H7 : " does not appear anywhere in the bottom node, there is no green highlighting in any of the child nodes in Figure 3.11 through Figure 3.17 since each child is entirely the result of removing and rearranging text from the parent node.

- Blinking yellow highlighting is used to show where the tactic is finding or producing matching text. For the sake of brevity, this is only shown in Figure 3.11 through Figure 3.17 and not in Figure 3.10. Blinking occurs at a rate of two blinks per second and each matching pair gets five blinks. Pressing T again, or moving to a different node, clears the red and green highlighting and stops the animation.

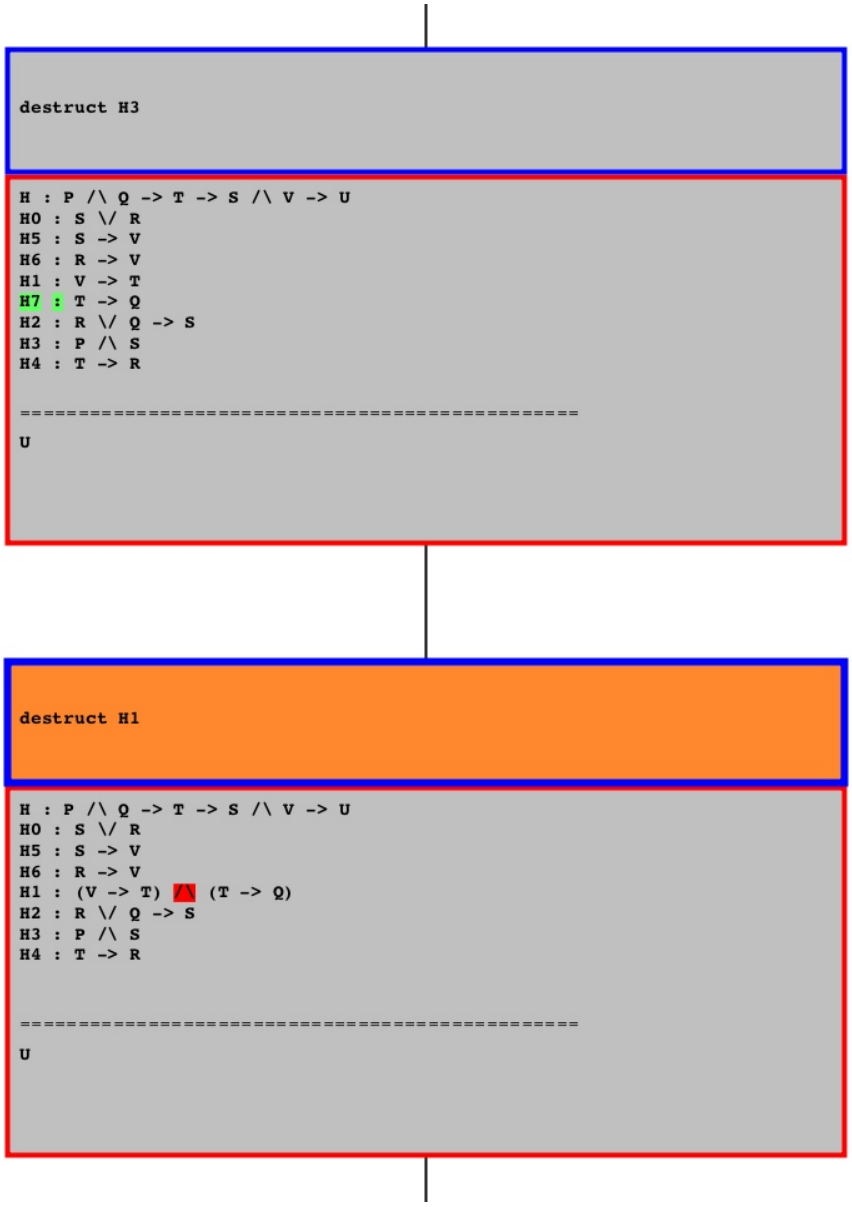

Figure 3.10. The red highlighting marks text in the parent that does not occur in the (or, more generally, any) child node. The green highlighting marks text occuring in the (each) child node that does not match text in the parent. Note that the gray background, while perhaps somewhat unpleasing aesthetically, was selected so that all the highlighting colors would be visible.

Figure 3.11 through Figure 3.17 document how the text-matching works in detail. In Figure 3.11, the animation ignores the child nodes altogether and uses

the blinking yellow text to point out that the "H0" argument to the destruct tactic matches the H0 variable in the goal's context. In Figure 3.12, the blinking points out that the S on the left side of the disjunction in hypothesis H0 [5] in the parent matches[6] the S of hypothesis H0 in the first child. Similarly, in Figure 3.14 the blinking points out that the Rs match. Figure 3.13 is included to make clear that a fraction of a second is used to animate the movement to the second child node (and back to the first) that occurs between Figure 3.12 and Figure 3.14. Figure 3.15 through Figure 3.17 simply show the portions of the contexts that are unchanged between parent and children and that the goals also are unchanged (for the sake of brevity, the blinking highlighting used to show matching consequents between the parent and right node are not displayed as a figure in this document).

---

[5]As noted in the previous chapter, to be (more) pedantic we should say the S on the left side of the formula proved by the hypothesized proof labeled by H0

[6]By "matches" I really mean "matches due to the way the destruct tactic works" and not "matches, perhaps by coincidence". The animation does not aim to point out, for instance, that H0 and H8 in the first child's context are hypothesized proofs of the same formula.

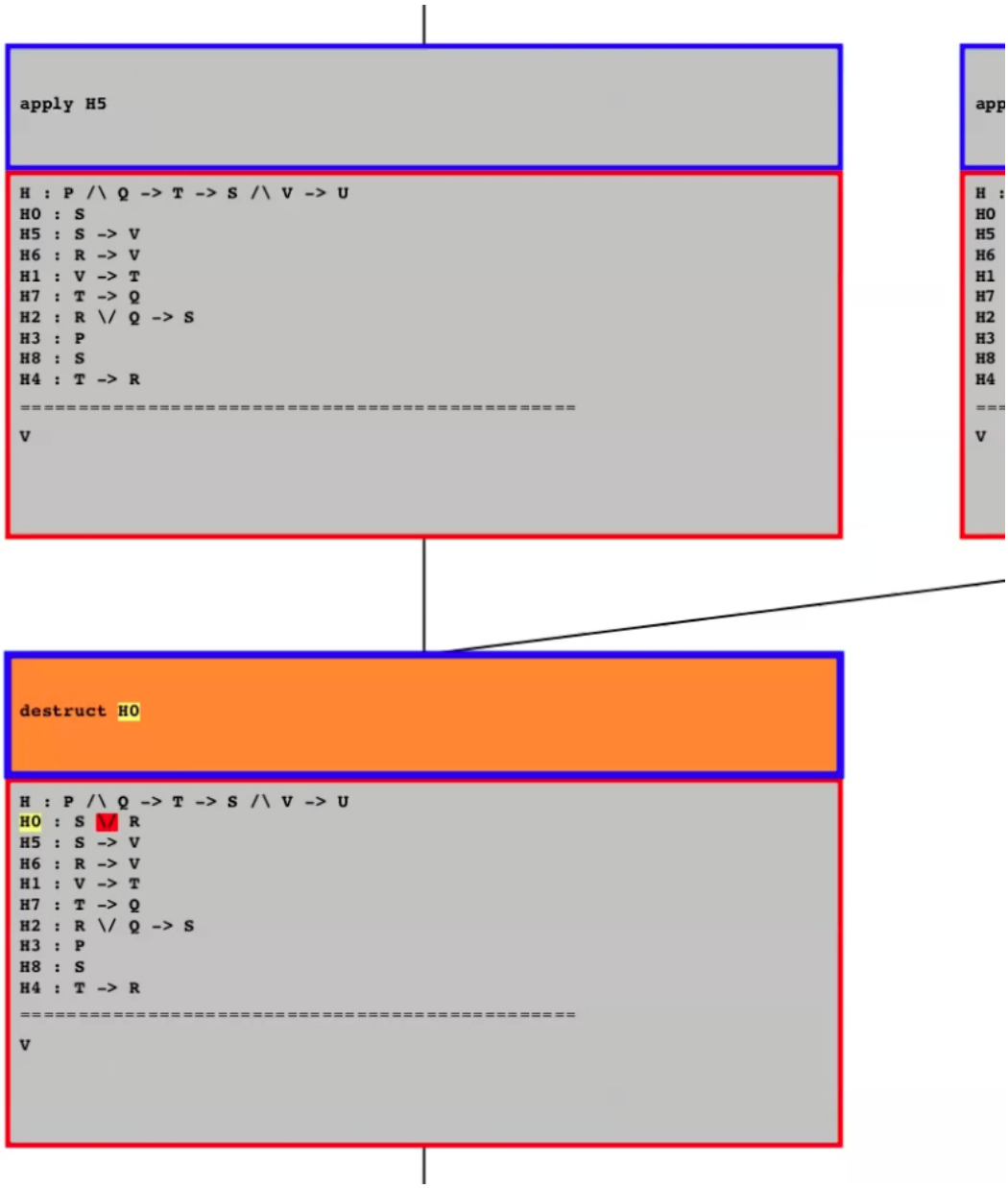

Figure 3.11. Blinking yellow text is used to show the context item matching the tactic argument.

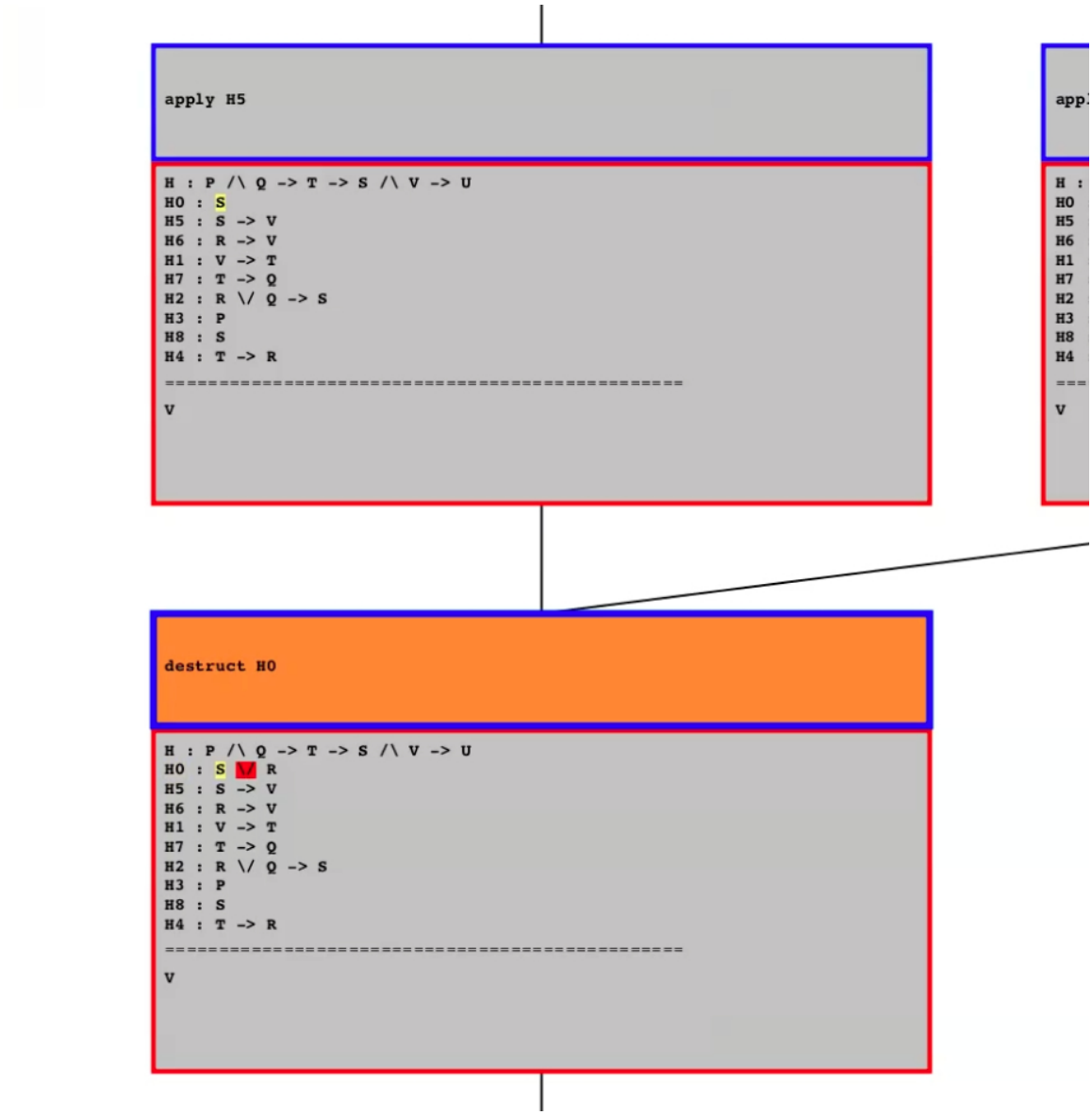

Figure 3.12. Blinking yellow text is used to show matching (sub)formulas in the parent and left child contexts.

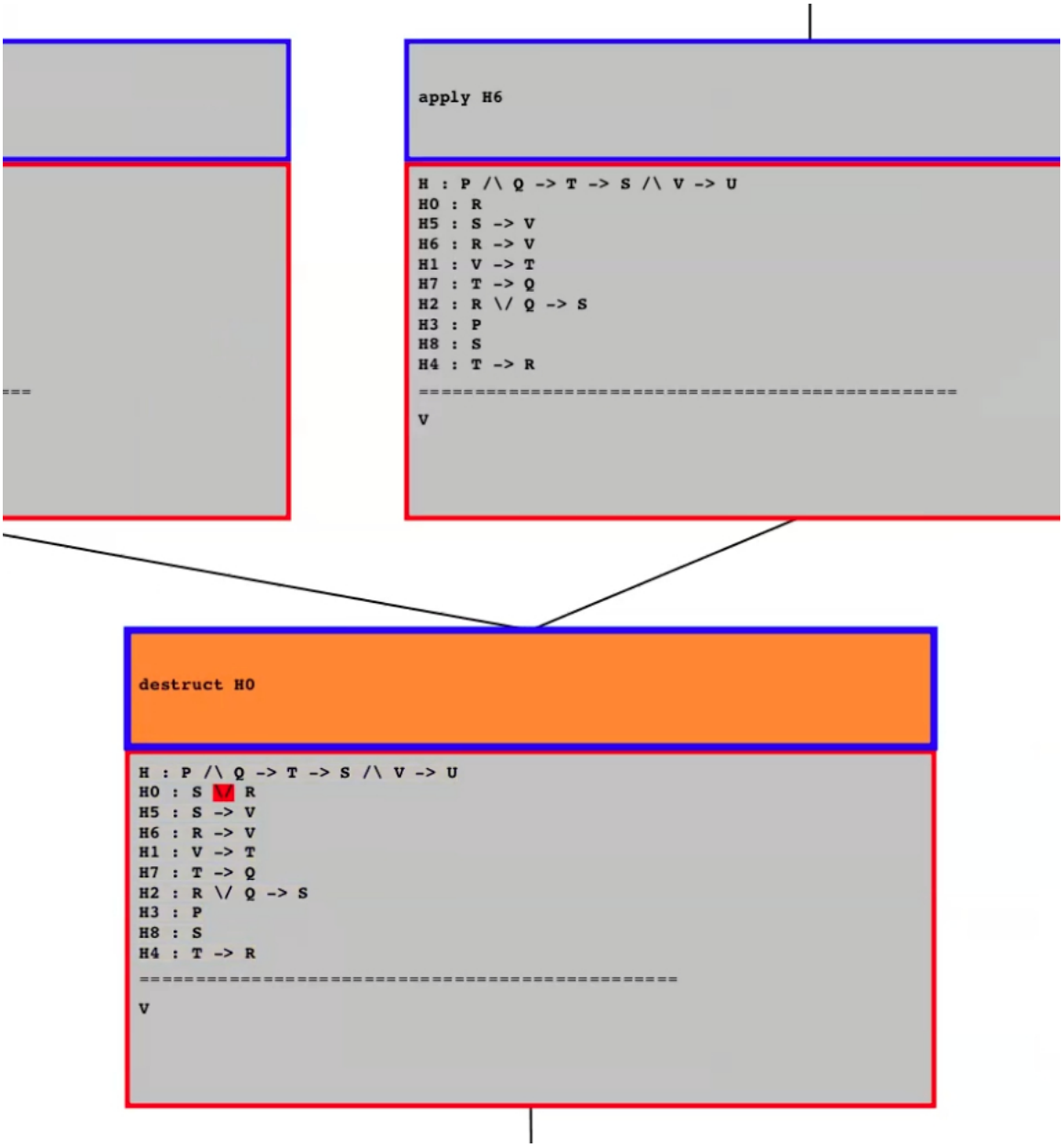

Figure 3.13. The transition between Figure 3.12 and Figure 3.14 is animated

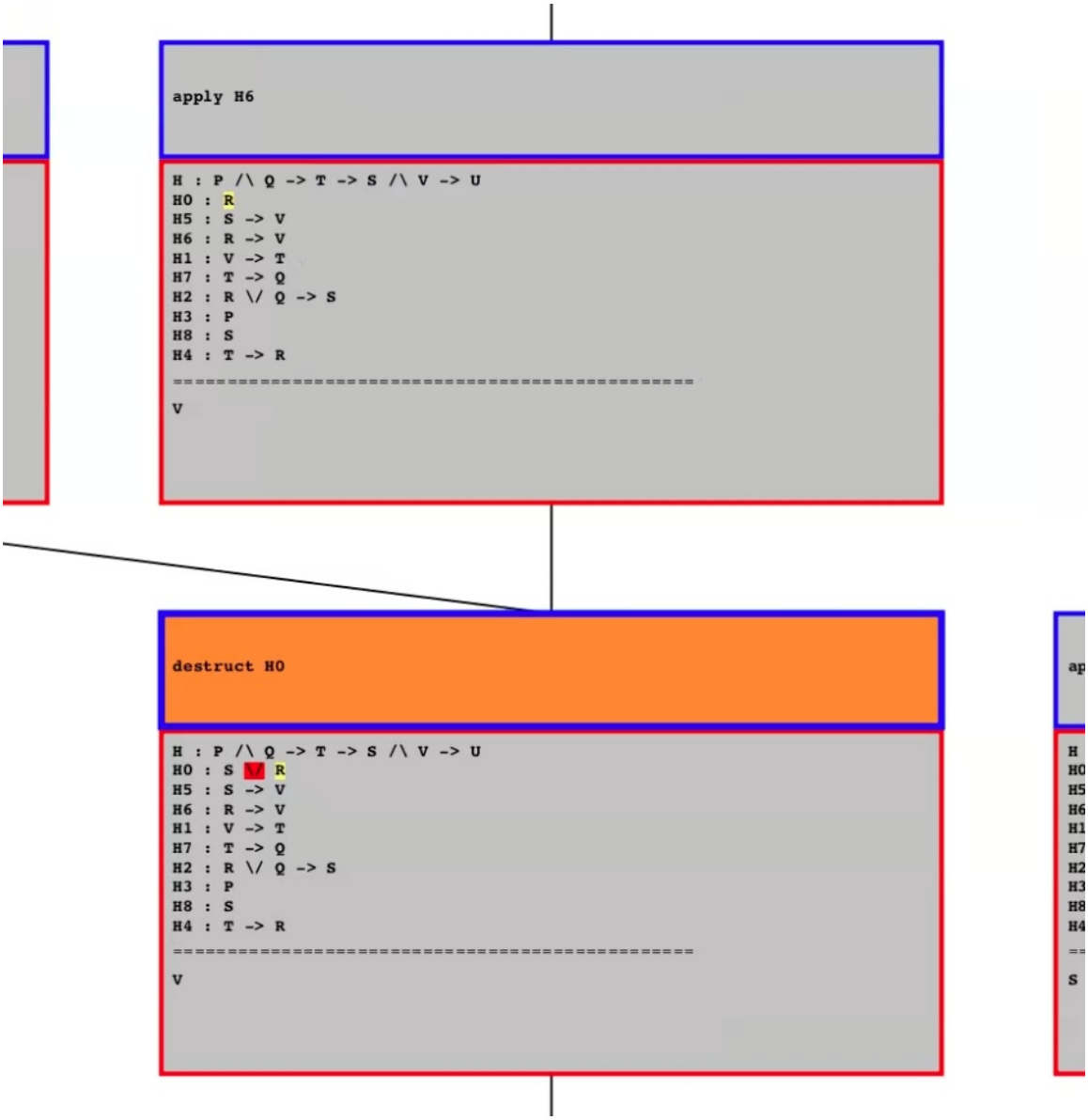

Figure 3.14. Blinking yellow text is used to show matching (sub)formulas in the parent and right child contexts.

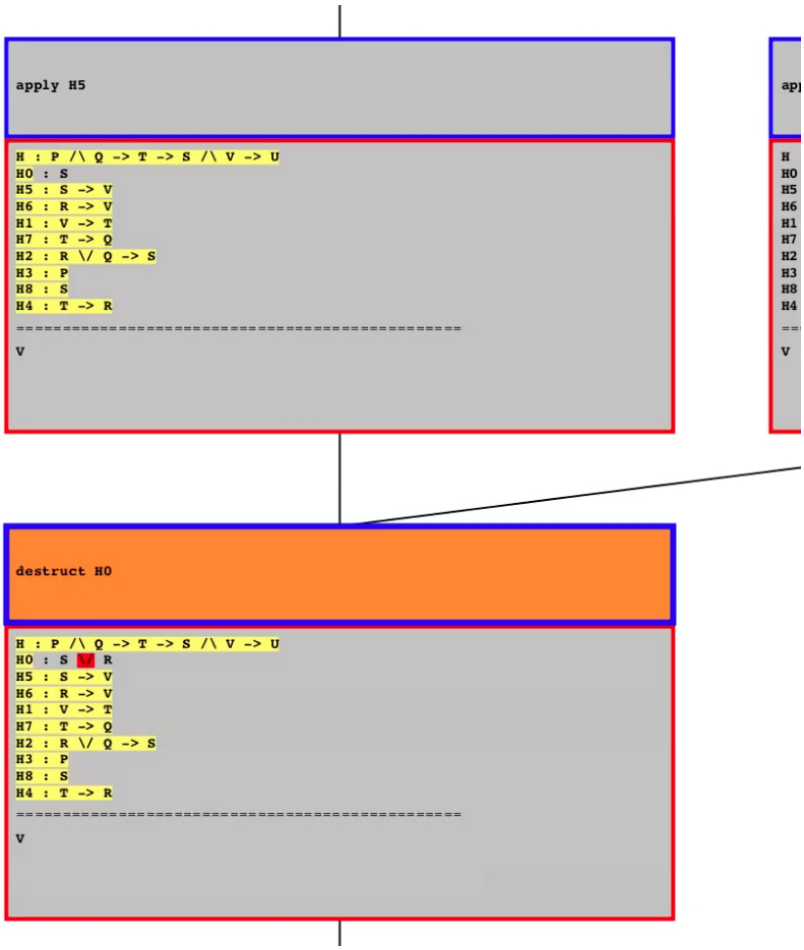

Figure 3.15. Blinking yellow text is used to show matching parts of the parent and left child contexts.

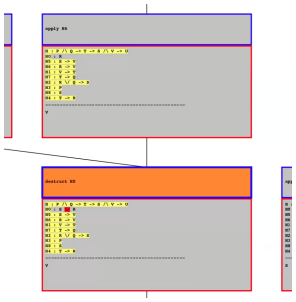

Figure 3.16. Blinking yellow text to show matching parts of the parent and right child contexts.

Figure 3.17. Blinking yellow text is used to show matching consequents for parent and left child nodes. Blinking yellow text is also used to show matching consequents between the parent and the right child nodes.

What was the rationale for implementing these designs? First, we wanted to confirm our hypothesis that a zoomable node-link view of a proof tree would help users understand the relationships between nodes and the effects of tactics–did a

particular tactic produce two sibling goals or did it only produce one child goal? We also envision(ed) such visualizations as potentially even more useful when further features are added (e.g. hiding less important branches when presenting a proof, cut/copy/paste of branches, version control for particular branches, bookmarking nodes, filtering nodes, etc.) and wanted to better understand how users deal with basic navigational issues as a preliminary step.

Second, with the transition highlighting and animation (i.e. blinking yellow text), we wanted to explore a potential way to make tactic effects easier to understand. We were also interested in this feature because of its generalizability: one can think of this feature as a visualization of an extended version of the UNIX "diff" command wherein the actual editing commands, including copy as well as insert and delete, used to change document A into document B are recorded.[7]

3.1.3 Proof Previews

"Proof Previews" is probably a more familiar sort of extension than Proof Transitions for most programmers as it is a variant of the standard "Content Assist" feature of Eclipse[3] and other IDEs for programming. Like Proof Transitions, it is currently quite limited compared to a fully developed plugin, but, unlike Proof Transitions, it is already set up as jEdit/CoqEdit plugin (giving evidence for the general feasibility of the modular interactive theorem prover user interface approach).

In Figure 3.18 we see the result of invoking the "Get Suggestions" menu item

[7]diff is a commonly used utility for comparing versions of documents, but only makes a best guess at where text was inserted or deleted and does not show where text was cut or copied and then pasted.

in the jEdit plugin: a popup listing of tactics that do not (immediately) produce errors when entered at the end of the green section.[8] I expect (and observed in the experiment described in the next section) this to generally be invoked with a keyboard shortcut that, like the basic CoqEdit menu items, can be customized.[9]

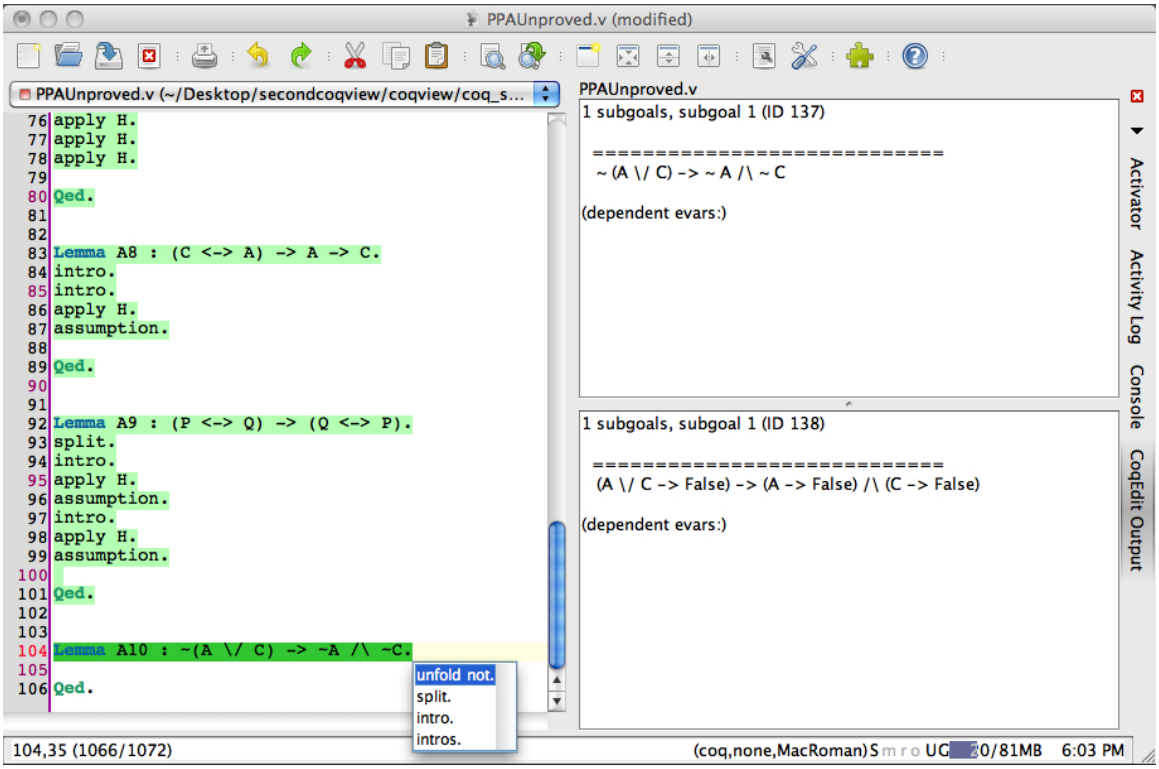

Figure 3.18. CoqEdit, with the Proof Previews plugin, after selecting the "Get Suggestions" menu item (or using an associated shortcut). Note the popup window and the bottom-right sub-window showing the result of evaluating the tactic selected in the popup window.

[8] In addition to creating the popup, it moves the dark green highlighting to the end of the green section.

[9] In the experiment, this was set to Ctrl-Space, matching the default for Content Assist in Eclipse.

The menu item only produces a popup when the last green sentence has the system in "proof mode" (i.e. after one has evaluated "Lemma ... ." but before evaluating the corresponding "Qed.") A more fully developed plugin might give users the option of starting a lemma, for instance, outside of proof mode. The menu item also does not appear if the plugin cannot find any tactics that would not immediately produce errors (which is not a guarantee that the current goal is impossible to prove).

In the bottom-right window in Figure 3.18 and Figure 3.19, a "preview" of the result of entering and evaluating the selected tactic is displayed (hence the name "Proof Previews"). Users can use the arrow keys to move between popup menu items, and press the ESC key to close the popup window (using Proof Previews does not preclude normal tactic entry and evaluation).

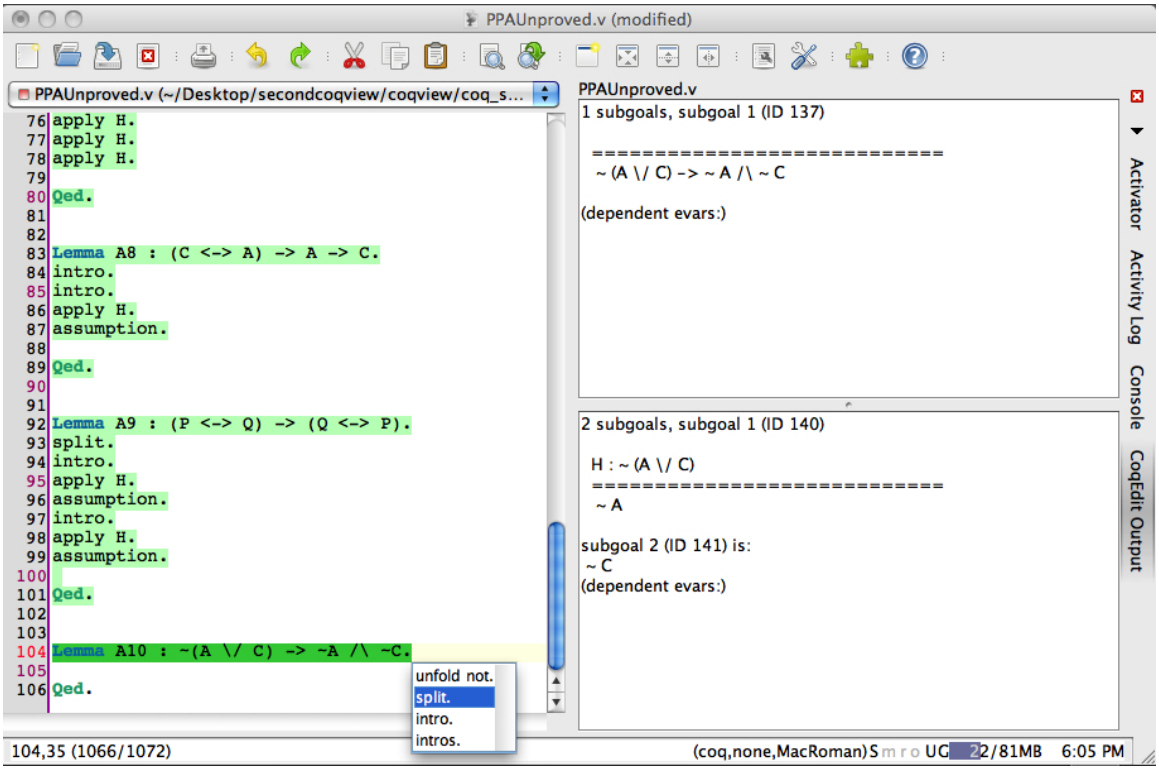

Figure 3.19. The result of pressing the down arrow from the state shown in Figure 3.18. Note that the bottom-right sub-window has changed to reflect the change in the popup window's selected item.

Pressing ENTER when the popup window is displayed will insert the selected tactic into the buffer on a new line at the end of the green region and will evaluate it (thus moving the dark green highlighting to the new tactic and the cursor to the end of the tactic's sentence); this is shown in Figure 3.20. As a result, some proofs can be completed simply by pressing Ctrl-Space (or whatever shortcut is used for the menu item), ENTER, Ctrl-Space, ENTER, Ctrl-Space, ENTER, etc., until there are no more subgoals.

The current version of Proof Previews, while sufficient for the experiment of the

next section, is quite limited in its ability to search for potential tactics. In particular, it does not look for theorems and lemmas from any libraries, or from the environment, to apply (it only tries to apply hypotheses in the context). More advanced version of the plugin might search over indexed subsets of the standard library, hide any tactics producing goals with automatically provable negations, and/or do parallel searches. The main goal of the current Proof Previews is to get a sense for how effective the user interface of an advanced version might be. Can Proof Previews improve the speed of text entry, and can it help users, particularly novice users, know their options options and make decisions?

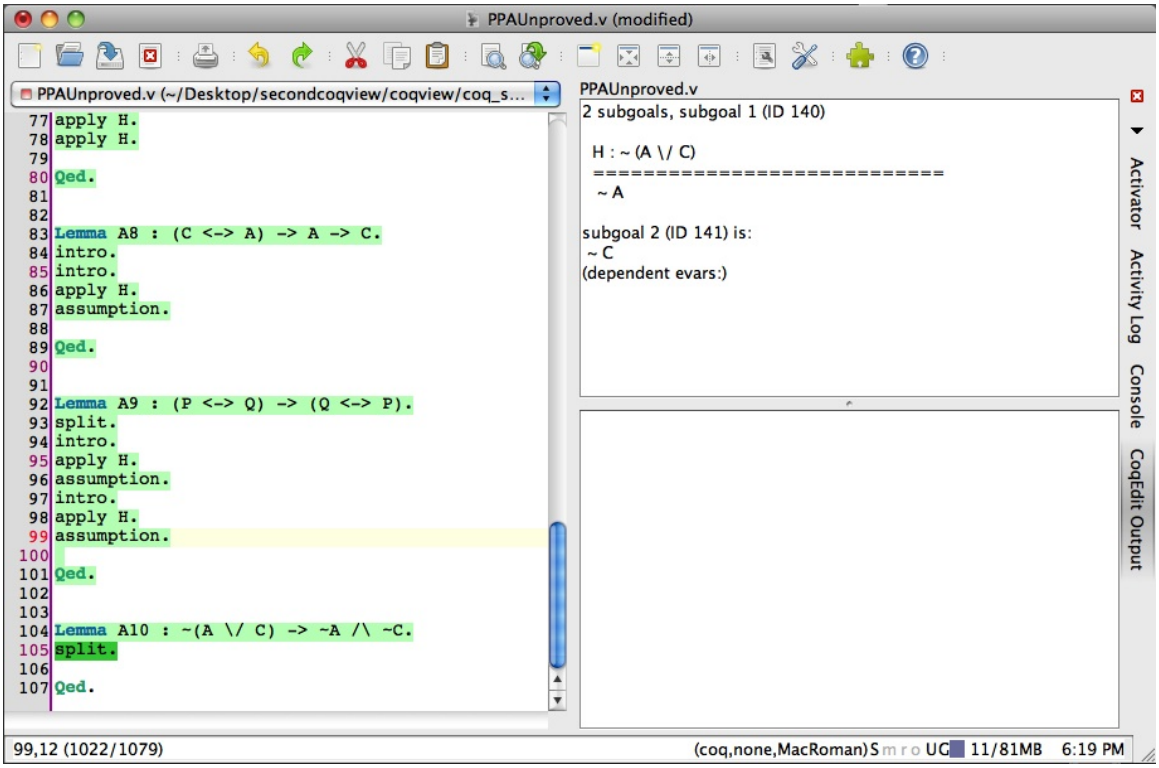

Figure 3.20. The result of pressing ENTER from the state shown in Figure 3.19. Note that the contents of the bottom-right sub-window in Figure 3.19 is now in the top-right sub-window and that the tactic that was selected in the popup window of Figure 3.19 is now inserted on a new line in the buffer and is evaluated.

3.2 Experiment

The goal of the experiment described below was to evaluate the effectiveness of Proof Transitions and Proof Previews in helping novice Coq users to understand proofs others have written and to write their own proofs.

3.2.1 Experiment Design

I conducted a user study with 16 participants that compared using 1) Proof Transitions with the basic CoqEdit interface for reading/interpreting proofs, and 2) Proof Previews with the basic CoqEdit interface for writing proofs (i.e. the inde-

pendent variable was either "Using the the experimental interface" or "Not using the experimental interface"). The comparison was within-subjects (each participant used both the basic and experimental interfaces) and counterbalanced (i.e. equal numbers of participants used each task-ordering/interface-ordering combination), and each participant did the Proof Transitions comparison followed by the Proof Previews comparison. The dependent variables were, for the Proof Transitions portion of the experiment, the fraction of opportunities taken to point out where a branch splits as the result of a destruct tactic and, for the Proof Previews portion, the number of proofs written (in a maximum of 15 minutes) and the amount of time actually taken (to attempt to write 10 proofs).

The participants in the experiment were recruited from University of Iowa Computer Science and Computer Engineering graduate and undergraduate classes and were required to have taken second semester undergraduate coursework.[10] The participants ranged in age from 19 to 37 (mean=23.4, standard deviation=4.1). Three of the participants were women. An additional 17th participant's data was excluded from the analysis as it included outlier values[11][12]

The experiment took place in a faculty member's office (only the participant

[10]Specifically, they were required to have taken 22c:019 "Discrete Math" and 22c:021 "Data Structures" at the University of Iowa, or equivalent coursework.

[11]Values greater than 3*(Q3-Q1) + Q3, where Q1 and Q3 are the first and third quartile values respectively, were considered outliers.

[12]In this particular case, the outlier values may have been attributable either to weak participant English fluency, or the fact that the participant was one of the first scheduled in the experiment.

and I were present). The single session with each participant generally lasted no more than two hours (though some participants opted to take extra time). Participants were compensated with $40. During the session, participants used a 17-inch (1920 x 1200) MacBook Pro with a 2.8 GHz Intel Core 2 Duo Processor and 4 GB 1067 MHz DDR3 memory, running Mac OS X Snow Leopard.

The each session consisted of the following steps:

1. After going through the consent process, participants filled out a payment form (where to send a compensation).

2. Participants filled out a demographic information form. This asked for their ages, genders, levels of confidence in computer science skills and levels of confidence in logic skills (each on a 1-9 scale from "Not at all" to "Very confident"), and which more advanced computer science courses they had passed.

3. Participants watched (and listened to) a training module. This was a 50 minute video on proofs in propositional logic using Coq. It covered the use of all tactics used in the rest of the experiment and presented the material using the basic CoqEdit user interface. It also briefly introduced the Proof Transitions visualization features. Only about 4 minutes were spent viewing proofs with the Proof Transitions interface, compared to about 40 minutes spent doing and viewing proofs using the basic CoqEdit interface (with the remaining time spent viewing slides). Participants were reminded that they could take a break or ask questions at any time.

4. Participants were told (after watching the training video) that it would be a good time to take a break (not all did, and most who did took short ¡ 5 minute breaks).

5. Participants were asked to navigate through two proofs that had already been completed and "verbally explain how the proof works to the best of your ability". These were both presented using either the basic CoqEdit interface or Proof Transitions. The proofs were both using only propositional logic with the first consisting of eleven nodes and the second consisting of twenty-eight nodes. These explanations were recorded as screen-casts.

6. Participants were asked to again explain two (new) completed propositional logic proofs of the same lengths as the previous two. If the participant had previously used Proof Transitions then in this step he used the basic CoqEdit interface and vice versa. For this step and the previous, the four combinations (due to the two orders in which the two pairs of proofs were presented and the two orders in which the two interface types were presented) were balanced across participants.

7. Participants were asked to fill out the first two pages of a four page satisfaction and preferences questionnaire. On the first page, participants were asked to give ratings for the first user interface they used on scales from 1 to 9 along the following axes: terrible to wonderful, frustrating to satisfying, dull to stimulating, difficult to easy, inadequate power to adequate power, and rigid to flexible.

On the second page, participants were asked to rate the second interface they used in the same way. On the second page they were additionally asked to give an overall preference (again on a 1 to 9 scale) between the first and second user interfaces, and a section for comments was given at the bottom of the page.

8. Participants were again prompted to take an optional break.

9. Participants were asked to prove as many of a set of 10 propositional logic lemmas as they could in a 15 minute period, using either just the basic CoqEdit or CoqEdit extended with Proof Previews. With Proof Previews, participants were informed that they could press Ctrl-Space to get a popup window. These lemmas all required between 3 and 10 tactics to prove. Before starting, participants were given a paper "cheat sheet" reminder of tactics they could use and what the tactics would do, taken from the video tutorial. The screen was again recorded during this and the next step.

10. Participants were asked to repeat the previous step with the user interface they had not used in the previous step and with a very similar set of ten lemmas. (In a few cases the same lemmas were given with different variable letters).

11. Participants were asked to fill out the the last two pages of the satisfaction and preferences questionnaire. These contained the same questions as on the first two pages.

Despite having run two pilot sessions[13], there were some problems with the

[13]Ideally, more pilots sessions would have been conducted, but it turned out to be difficult

experiment that only became clear after the first handful of sessions had been conducted. After the first eight participants (for counterbalancing), the above steps were refined in an attempt to get more participants to actually try out features, produce easier-to-evaluate proof explanations, and more comments. The refinements were as follows:

- Before starting the proof explanations, participants were presented with a guide for their explanations. This stated "For each tactic, please try to answer the following: 1) What nodes, if any, does the tactic produce from its goal? 2) How is each child node the same as or different from its parent? 3) Which text segments from the parent node get copied into the child nodes and where do they get copied to?"

- Before starting the proof explanations, participants were also asked if they would like to set a timer for six minutes (generally sufficient) for each proof to ensure that the session did not last more than two hours.

- Before starting the first of the two proof explanations using Proof Transitions, participants were asked to try using each of the features listed at the bottom of the window for at least the first three nodes in the first proof.

- Before starting the first of the two proof explanations using the basic CoqEdit, participants were reminded of the shortcuts used to move forward and backward.

---

to recruit participants–it is unclear if more than one or two additional participants could have been recruited.

- Before filling out each half of the satisfaction and preferences questionnaire, participants were given printed "starter questions" for the comments section. Starter questions common to both comment sections were "Why did you prefer one interface over the other?", "Were there any improvements that you think could be made to either interface? How significant would these improvements be?", "Was there anything in particular that was confusing about either interface?", and "Were there any features of either interface that you could do without? Were there any that were especially helpful?". Specific to the proof interpretation questions were "In the interface with the boxes and flashing highlighting, was having to back away from leaves manually, instead of automatically moving to the next branch, something that made a big difference?" and "In either interface, did you find yourself getting 'lost'?" Specific to the theorem proving questions was "When using the popup window, did you find the preview of the result, in the bottom right, helpful?"

- Before using Proof Previews, participants were walked through how to use the extension to create a three tactic proof of A $\rightarrow$(B $\rightarrow$C) $\rightarrow$(A $\rightarrow$C) $\rightarrow$C. They were also told to use Proof Previews exclusively.

3.2.2 Results

Probably the most disappointing result turned out to be Proof Transitions' transition highlighting and animation, which went largely unused (possibly for the reasons discussed in the next subsection) even in the second set of eight sessions. However, other results were more positive.

	Average	**Std Dev**
Terrible to Wonderful	6.1	1.5
Frustrating to Satisfying	5.4	1.5
Dull to Stimulating	6.5	1.1
Difficult to Easy	5.9	2.5
Inadequate Power to Adequate Power	6.3	2.3
Rigid to Flexible	6.3	1.9

Table 3.1. Satisfaction and preferences questionnaire average results for proof interpretation using Proof Transitions. Values go from 1 to 9 (e.g. a 9 for a Terrible to Wonderful rating would mean completely wonderful).

	Average	**Std Dev**
Terrible to Wonderful	6.8	1.2
Frustrating to Satisfying	7.0	1.6
Dull to Stimulating	5.7	2.0
Difficult to Easy	7.1	2.1
Inadequate Power to Adequate Power	7.1	1.6
Rigid to Flexible	6	2.1

Table 3.2. Satisfaction and preferences questionnaire average results for proof interpretation using the basic CoqEdit interface. Values go from 1 to 9 (e.g. a 9 for a Terrible to Wonderful rating would mean completely wonderful).

Table 3.2.2 and Table 3.2.2 show the questionnaire rating averages for the proof interpretation/explanation with Proof Transitions and the basic CoqEdit interface respectively. The differences were statistically significant only for the Frustrating to Satisfying axis. The overall preference rating average, again on a 1 to 9 scale but this time with 1 indicating preference towards Proof Transitions and 9 indicating preference towards the basic CoqEdit interface for explaining proofs had an average of 5.8. This was, however, not a statistically significant difference from the middle value of 5. In fact, five of the participants gave a number less than 5, indicating a preference towards the Proof Transitions interface, and one more gave a value of 5, indicating no preference.

Comments on Proof Transitions versus basic CoqEdit included:

- "I am a programmer, so I lean towards the [basic CoqEdit interface]. However, if I was to present my findings I would use the [Proof Transitions] interface because people would find it more clear on how the flow of logic works." ... "[In Proof Transitions] I found myself wanting to click left and right when I needed to move up or down"

- "[In Proof Transitions] too much zooming in and out required gets little confusing sometimes. However, the animation of 2nd user interface is really cool and makes it intuitive"

- "[Proof Transitions] was confusing in the sense which way the proof flows"

- "I thought by watching the video that the first UI would be easier to understand

b/c more time was spent on it, but the visual one was more intuitive"

- "I really liked the transition mode of [Proof Transitions], it really helped me visualize the process of solving the proofs. I also enjoyed the first, but the second was a far better visual tool which helped me since I'm more of a visual learner. However, in [Proof Transitions] I did get lost some times when the proof really branched out. Finally, for the [basic CoqEdit user interface] I liked how it listed all the current subgoals which helped out since there were some times I got confused in [Proof Transitions] due to forgetting whether a subgoal had been proved yet or not."

- "The first interface is very confusing, as it is not made clear that when the path branches, new goals have been added. Not being able to see all the goals at once makes it difficult to know how far you have progressed in the total proof."

- "[With Proof Transitions] I found [it] easy to get lost."

- "Would like a zoom level somewhere between all the way out and all the way in."

- "[Proof Transitions] allow us to view the proof more easily so that we can understand it better, but it is difficult to edit it."

- "[The basic CoqEdit user interface] is straightforward"..."[Proof Transitions could] have more scales when do[ing] the zoom in/out"..."For [Proof Transitions], if the proof is complicated, the diagram could be very frustrating"..."If

	Average	Std Dev
Terrible to Wonderful	8.2	1.1
Frustrating to Satisfying	8.0	1.3
Dull to Stimulating	7.4	2.1
Difficult to Easy	8.4	1.1
Inadequate Power to Adequate Power	8.3	0.7
Rigid to Flexible	7.6	1.7

Table 3.3. Satisfaction and preferences questionnaire average results for theorem proving using Proof Previews. Values go from 1 to 9 (e.g. a 9 for a Terrible to Wonderful rating would mean completely wonderful).

there is a mini map, it would help".

- "I wanted both for harder proofs. You can get lost in the leaf paths. I prefer working with the text, out of habit, but the visual helped for backtracking."

- "Should do [Proof Transitions zoom out feature] in steps until user is comfortable in reading text"..."I forgot to use 'visit' option. Might proved to be helpful."

Table 3.2.2 and Table 3.2.2 show the questionnaire rating averages for the proof interpretation/explanation with Proof Transitions and the basic CoqEdit interface respectively. Differences were statistically significant except for the Dull-Stimulating and Rigid-Flexible axes. The overall preference rating average was 3.1 (std dev = 2.4)on a scale from Proof Previews=1 to No Proof Previews=9 (and was statistically signifiant—$p<0.01$).

Comments on Proof Transitions versus basic CoqEdit included:

	Average	**Std Dev**
Terrible to Wonderful	7.1	1.5
Frustrating to Satisfying	6.9	1.7
Dull to Stimulating	7.2	1.2
Difficult to Easy	6.4	1.9
Inadequate Power to Adequate Power	6.8	1.8
Rigid to Flexible	6.7	2.1

Table 3.4. Satisfaction and preferences questionnaire average results for theorem proving using the basic CoqEdit interface. Values go from 1 to 9 (e.g. a 9 for a Terrible to Wonderful rating would mean completely wonderful).

- "The recommendations in the pop-up menu...were incredibly helpful."
- "There really was nothing confusing about either interface...all in all, I think the best option would be the [basic] interface with [Proof Previews] available for when the user gets stuck since it was really helpful at times for me to see what options I had available to solve the problem or figure out an error I had made."
- "The dropdown menu was so useful that I sorely missed it when it was gone and had to look at the manual more frequently."
- "I haven't used the (space+ctrl) as it takes more time to type it. I would prefer typing by hand over the other one."
- "[The command suggestions] are pretty useful."
- "Very intuitive. The options help a lot. Really easy."

- "If it is for educational purposes, I think the second user interface might be too automatic...I don't even need to read the [goals], but proceed right away until encountered a problem that I need to choose the right answer for."
- "Found myself thinking less about the actual proof [with Proof Previews], and instead just exploring options. Was nice to have "suggestions". I did not look much at the preview until I got into the more complicated proofs"
- "[Without Proof Previews] it is really hard to type using control N and return."
- "If a user depends too much on [Proof Previews], it will lead to a dead end."
- "[Proof Previews] almost took the thought and tactics out of theorem solving."
- "Both were great!!"

Proof Transitions and Proof Previews also had some benefits according to more **objective measures**. For Proof Transitions, the dependent variable I found easiest to measure was *opportunities taken to explicitly make the observation that a destruct tactic produced two child subgoals.* Each participant had six opportunities to make this observation, three with Proof Transitions and three without. With Proof Transitions, users were significantly more likely to make this observation when the opportunity presented itself, taking **33 out of 41 opportunities vs 17 out of 38 without** (p=.023). (Some participants ran out of time, or their explanations were inaudible at these points).

Timing results from the Proof Previews part of the experiment are shown in

Figure 3.21 and Figure 3.22. With Proof Previews, participants were able to complete the proofs at an overall average rate of 1.3 minutes per proof vs. 1.9 minutes without. This was a statistically significant result (p=.033).

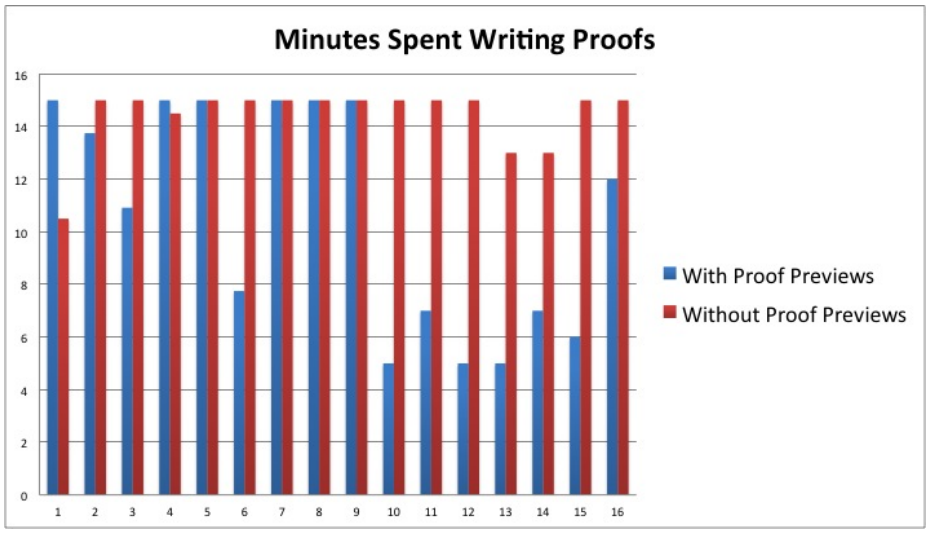

Figure 3.21. The total number of minutes (capped at 15) each participant spent writing proofs, with and without Proof Previews. Note that in the second set of 8, participants were required to use the Proof Preview feature, which had a significant effect (p=.005).

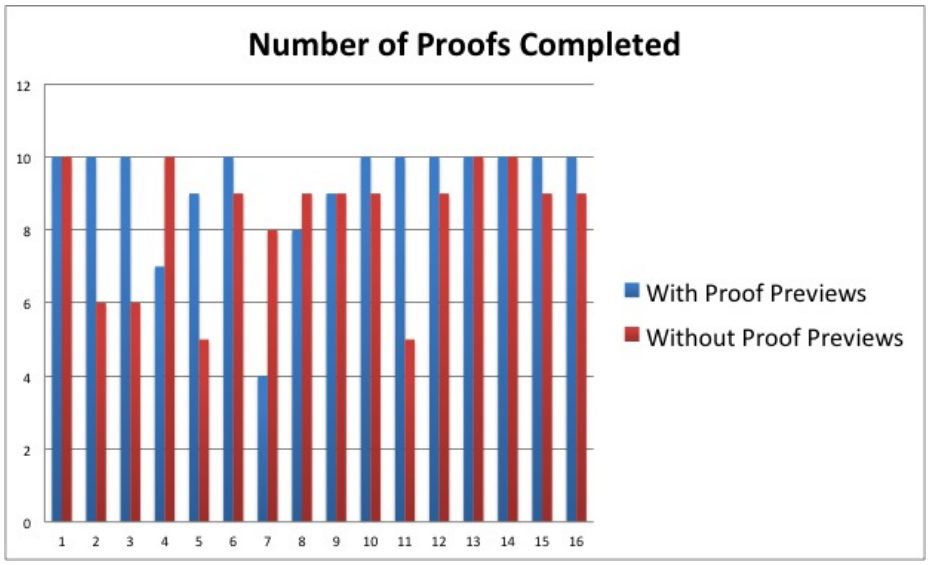

Figure 3.22. The number of proofs each participant was able to complete. The final proof in both the set used with Proof Previews and the set not using Proof Previews was especially challenging. Note that in the second set of 8, participants were required to use the Proof Preview feature. Although having proof previews had a significant effect on the rates at which theorems were proved, the difference in the number of proofs completed was not statistically significant, perhaps because the number of proofs was capped at 10 and it was the final proof that was especially challenging.

3.2.3 Conclusions and Future Work

Although on average, participants rated the proof transitions worse than the basic CoqEdit interface, Proof Transitions appear to be a promising avenue for further research for several reasons. First, a large minority of participants indicated that they actually did prefer Proof Transitions, and it was suggested that Proof Transitions might be useful for certain tasks (like presentations) that would be less suited to the basic CoqEdit. There was also considerable bias against Proof Transitions going into the study because the video training did not spend much time on it and because, as a prototype, it did not allow editing of proofs.

There is also a lot of apparent room for improvement with Proof Transitions. The blinking animation could have been replaced by (as pointed out by Professor Aaron Stump) the ability to toggle through the matching text, and could have been made more salient perhaps by text that floats up to meet the matching text, or lines, or some other way of indicating correspondence. It was pretty clearly too slow for most users. Other fixes might include auto-marking visited nodes, a minimap, and in-order node traversal.

The main problem with Proof Previews appears to be the Paradox of the Active User (see the next chapter)–that users are not necessarily aware of the "Get Suggestions" feature. It does appear to have strong benefits, although how much of these to attribute to the ability to enter tactics more quickly than typing and how much to novice users not needing to look up tactics remains a question for future research.

CHAPTER 4

KEYBOARD-CARD MENUS + SYNTAX TREE HIGHLIGHTING, APPLIED TO FITCH-STYLE NATURAL DEDUCTION PROOFS

The following section, with the exception of the "Future Work" subsection, is taken from a journal paper[29] that my advisor, Professor Juan Pablo Hourcade, and I wrote and which has recently been published. A shorter version of the same paper also appeared in the Interaccion 2013 conference proceedings[28].

The second section in this chapter contains a description of a prototype system that uses Keyboard-Card Menus in combination with a structure editing system. Although the idea can be applied generally, in this case it has been applied to a subset of Coq that might be useful for students studying propositional logic using the widely-used "Logic in Computer Science" by Huth and Ryan [62]. While the prototype is not actually an extension of CoqEdit, it could easily be turned into one.[1]

4.1 Keyboard-Card Menus

4.1.1 Abstract

"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. We describe these menus and their potential applications

[1]The main change needed would be to convert the highlighting used in CoqEdit to underlining, since the extension's highlighting would conflict.

in further detail, along with a study comparing keyboard-card menus' presentation of what are effectively shortcuts with a presentation of these same shortcuts that uses dropdown menus. The data from our study shows that keyboard-card menus have significant advantages over dropdown menus in making the transition to expert use faster.

4.1.2 Introduction

Keyboard-card menus are a new type of menu we have developed which present menu items in 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 [54]). WYSIWYG systems like *Mathematica*[6] and Microsoft *Word*'s[**?**] equation editor allow use of either the mouse or keyboard shortcuts to select symbols from menus and toolbars. Although 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* [72], and an important part of supporting novice users is helping them attain expert status.

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 the study's 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 the process of navigating the menus.

4.1.3 Keyboard-Card Menu Design

Subsection 4.1.3.1 describes how keyboard-card menu work. Subsection 4.1.3.2 describes in more depth the "rolled-chord shortcuts" that keyboard-card menus are used to visualize.

4.1.3.1 Keyboard-Card Menus

Inspired, in part, by work on keyboard-based menus for wearable computers, seen in [80], 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; at least in principle 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 4.1, and in Figure 4.2).

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 lighter in 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.

Initially, when no keys are pressed, only the root card, with top-level menu items, is visible (as in the lower half of Figure 4.1). 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 4.2, which would be displayed while the S key is held down). When one releases the key, the corresponding submenu's keyboard-card disappears. (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

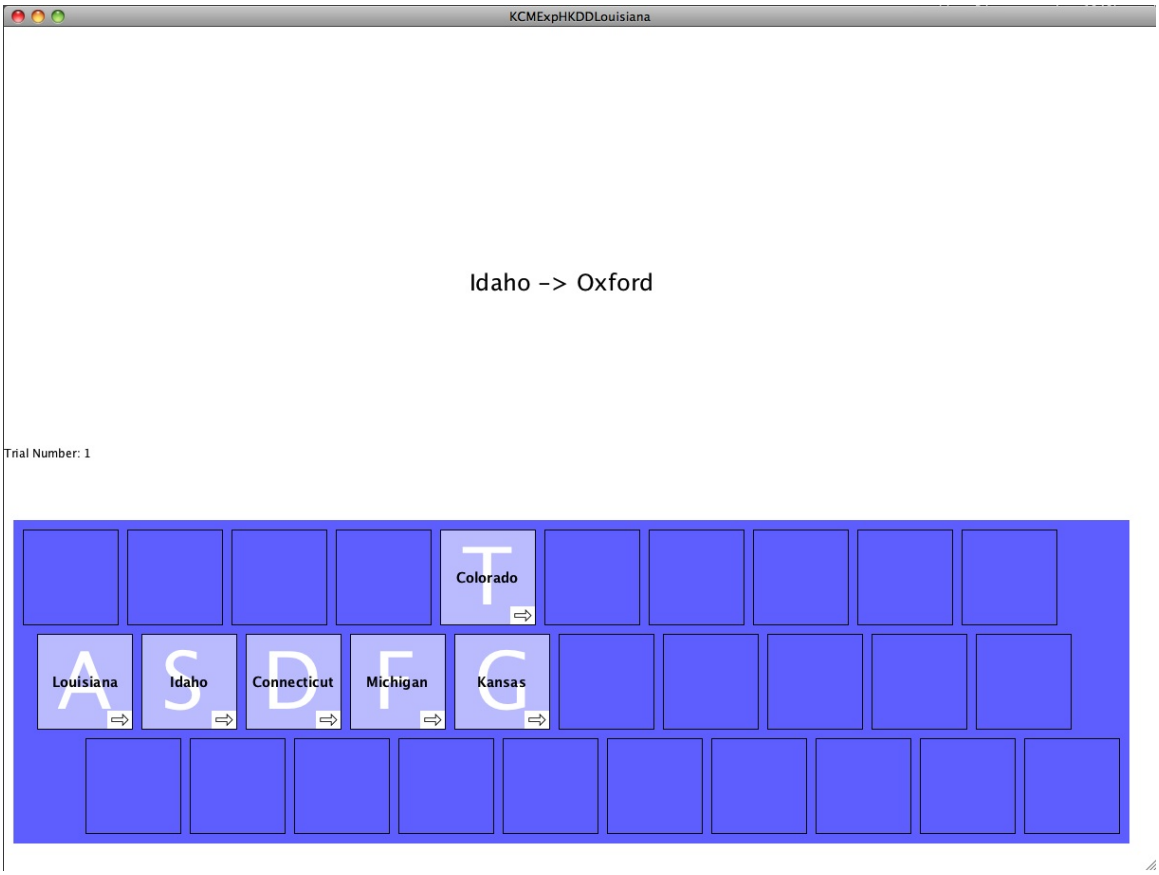

Figure 4.1. In the study, described later in this paper, participants used a keyboard-card menu system, shown in the bottom half of the window, to select state-town pairs. Seen here is actually the "root keyboard-card" of the menu system. The small arrows in the corners of the keys indicate that there is a submenu associated with that key (a detail the participants did not have to rely on). When a user presses one of these keys, a child keyboard-card, such as the one seen in Figure 4.2 appears.

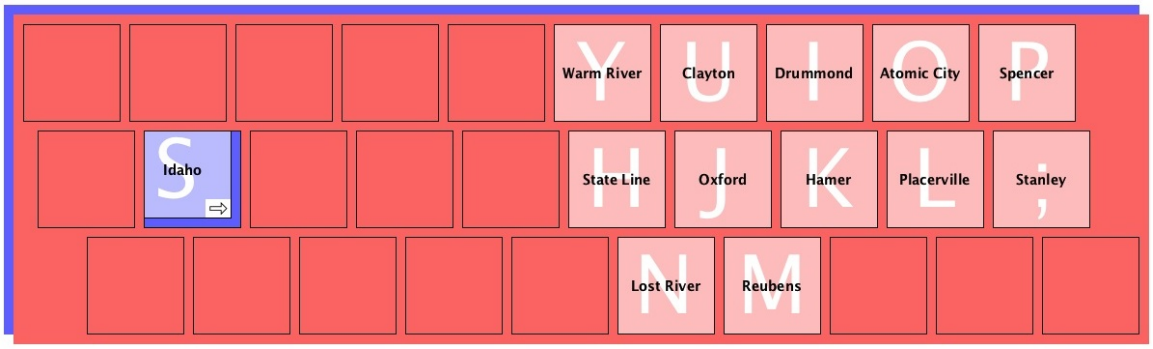

Figure 4.2. The keyboard-card menu from Figure 4.1 with the S key pressed. Note how a hole has been "punched out" of the card.

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 [68, 70].

4.1.3.2 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

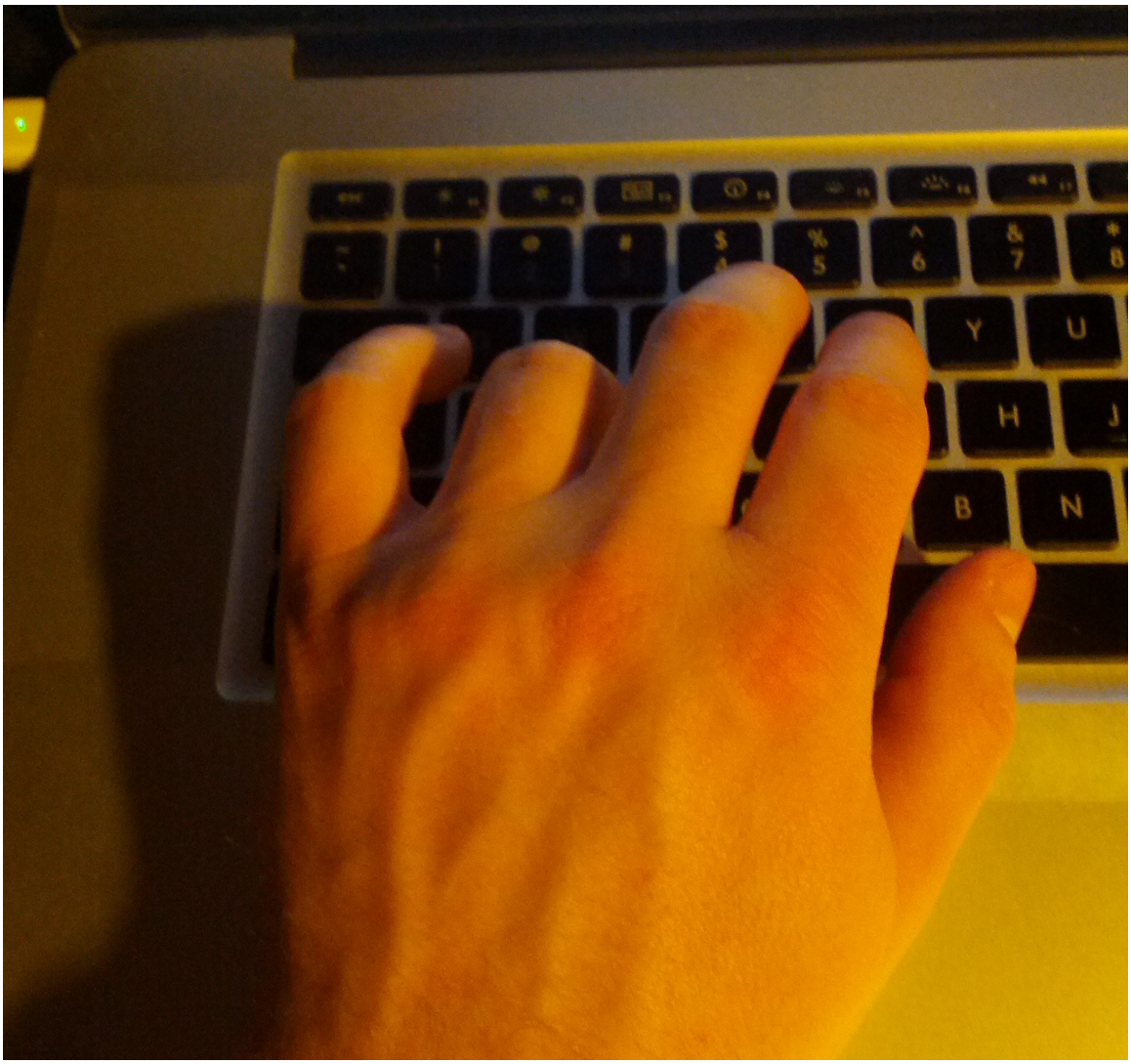

Figure 4.3. Keyboard-card menus are intended to be used with a normal keyboard and monitor, not a touchscreen. The S key is pressed and held down to make Figure 4.2 appear on the screen. Whenever no fingers are held down, the menu appears as it does in the lower half of Figure 4.1, i.e. with a single "keyboard-card" instead of multiple "stacked" cards.

would exist, as with the text editor *vi* or its extension, *Vim* [**?**]). 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 [97]. 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 (we will use square brackets to indicate where keys are held down; e.g. "F-[J, K, J]" indicates that the F key is held down while J, then K, then J, again, are pressed).

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 still provide hundreds of possible shortcuts. To see why, first consider only the shortcuts accessible by pressing two keys. Some pairs of keys are clearly at least relatively awkward to press together. For instance, pressing the W and the X keys together might best be accomplished by holding the W key with the ring finger while using either the middle or index finger to press the X key. Of course this is not a terribly intuitive approach, since normally W and X are both pressed using the ring finger. In addition, using the middle finger can be quite uncomfortable because of the tight connection between the middle and third fingers, and using the index finger may require a larger than normal amount of hand movement.

We avoid these problems altogether if we consider just the two-key rolled-chord shortcuts invokable by pressing a key on one side of the keyboard followed by pressing a key on the other side of the keyboard (using first one hand and then the other; in the study described below we actually consider just a small subset of these). Each side has 15 keys, so the total number of such available shortcuts is (2 sides) * (15 initial-keys/side) * (15 secondary-keys/initial-key) = 450. For comparison, Microsoft Word for Mac 2011 [**?**] contains approximately 230 leaves in its menu hierarchy, if one ignores the Font submenu which adds several hundred more (this is not to say that it would necessarily make sense to add rolled-chord shortcuts to any or all of the items in Microsoft Word's menu hierarchy, but, instead to suggest that many computer end-users and developers have already been exposed to menu hierarchies of such a scale).

We also hypothesize that rolled-chord shortcuts involving three (or perhaps even more) keys may be of practical value, despite being more complicated. First, consider text editing in *vi*, again, and the problem of moving the cursor around in the buffer. Basic cursor movement is accomplished by pressing the H key to move left one character, the L key to move right one character, the J key to move down one line, and the K key to move up one line. In addition, one can move around in larger increments, e.g. forwards or backwards a word, or to the beginning or end of a line, and up and down by paragraph or by page. These larger increments require entirely different keys (e.g. W for forward a word), which we suspect means that many users just avoid pressing the faster movement keys (they may not fully recall what key to press and also wish to avoid accidentally deleting text).

With an alternative text editor using rolled-chord shortcuts, one might use multiple keys in the left hand to select the speed of movement while still using H, J, K and L in the right hand to select the direction. For instance, one might press and hold F to go into a "move" mode and press J to move down a single line followed by L to move forward a character within that next line (recall that this would be "F-[J, L]" , i.e. the F key would not need to be released between J and L presses). One could then add the D key to change the rate of movement to, say, 5 lines/characters per H/J/K/L press (i.e. "F-[D-[... H/J/K/L key presses here ...]]", so "F-[D-[J, L]]" would move the cursor down 5 lines and then over 5 characters). One could, instead of D, add the S key to change the rate to, say, 20 lines/characters (so "F-[S-[J, L]]" would move the cursor down 20 lines and then over 20 characters). Or, instead of D

or S, one could press A to move 80 lines/characters at a time.

Because the F key is used in conjunction with the D, S, and A keys in this way, and the keys are designed to fit under the first, second, third and fourth finger (starting with the index finger), it is likely that

- the first two keys in the three-key sequences are at least relatively easy to press and hold down together, and
- the intensity of the operation is roughly correlated with the center of gravity of the left hand relative to the keyboard and with its degree of rotation.

Because of this correlation, and also because this use of three-key shortcuts could be generalized to applications, besides cursor movement in text editors, where varying degrees of intensity are called for (e.g. zooming and panning in maps, from the keyboard), this style of cursor movement might be considered more organized and intuitive than that of *vi* and other editors. The approach may also be used with other sets of keys (e.g. R, E, W and Q might be used in the alternative text editor to move the cursor while selecting text).

A second instance where having three-key rolled-chord shortcuts might be practical is simply to "extend" the number of keys (rather than intensifying the keys) that can be pressed after pressing the initial key. Suppose, again, that the first key we press is F, but this time the items under the right-hand keys are codes for various medical procedures commonly associated with a hospital department. If we have a list of up to 60 such procedure codes, we could extend the 15 right-hand keys using the

D, S, and A keys, as in the text editing example above, while keeping them associated with the "F" department.

Third, three-key rolled-chord shortcuts may still be very fast, even relative to two-key rolled chord shortcuts. This may be especially true when all three keys come from the center row of the keyboard (e.g. A-[F-[J]]) and so do not require finger joint extensions or contractions.

Rolled-chords also have several important advantages over simple key sequences. First, in sequences like "F-J, F-K, F-J" we may reduce the number of keystrokes needed, as explained above ("F, J, F, K, F, J" vs. "F-[J, K, J]") . 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 [92], 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.

Finally, one should not overlook the possibility of using forests of menu trees, and of using the shortcuts in one collection to move to another. When all rolled-chord shortcuts are single-key, this would simply degenerate to simple key sequences. Many applications might require a balance between one large tree and many small ones.

4.1.4 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* [38] 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 [72], 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 [55].

Our general approach–graphically supporting users in entering commands from the keyboard–has been taken before in a several alternate ways. "GEKA" [58] 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" [67] 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" [95] 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 rec-

ommendations. "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 [81].

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 [70]:

- *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" [26]. Kurtenbach et al. also address the issue of providing large

numbers of quickly accessible menu items in work on the "HotBox" [69].

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" [82], "Multi-Touch Menu" [19], "Finger-count shortcuts" [20], "Multi-touch Marking Menus" [74], and, perhaps most significantly, "Arpege" [25]. An advantage of using a multitouch surface is that thumb mobility may be exploited, as seen in [19] and [82]. However, physical keyboards appear to be faster to type with than virtual keyboards on a multi-touch surface [106], and using them in presenting chords does not create occlusion problems (involving the hands) which, as highlighted in [25], would likely make the presentation rolled chords much more complicated.

4.1.5 Study Description

4.1.5.1 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 4.4, 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 4.4)

- they are more compact (at least for relatively small hierarchies)

Reducing the advantage of compactness, screen resolutions have improved dramatically over the past decade [108] and secondary monitors are often available. Some advantages of keyboard-card menus over dropdown 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 [89]

- 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.

4.1.5.2 Initial Rationale for Experiment Hypothesis:

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.

After the experiment, we thought of a fifth, perhaps most likely, reason keyboard-card menus help users learn shortcuts faster. This is given in our "Discussion" section, below.

4.1.5.3 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 4.4 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 [14] and [55]. 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 SPACEBAR, 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 4.1); 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 4.1, Figure 4.2, and Figure 4.4). 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 [55].

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.

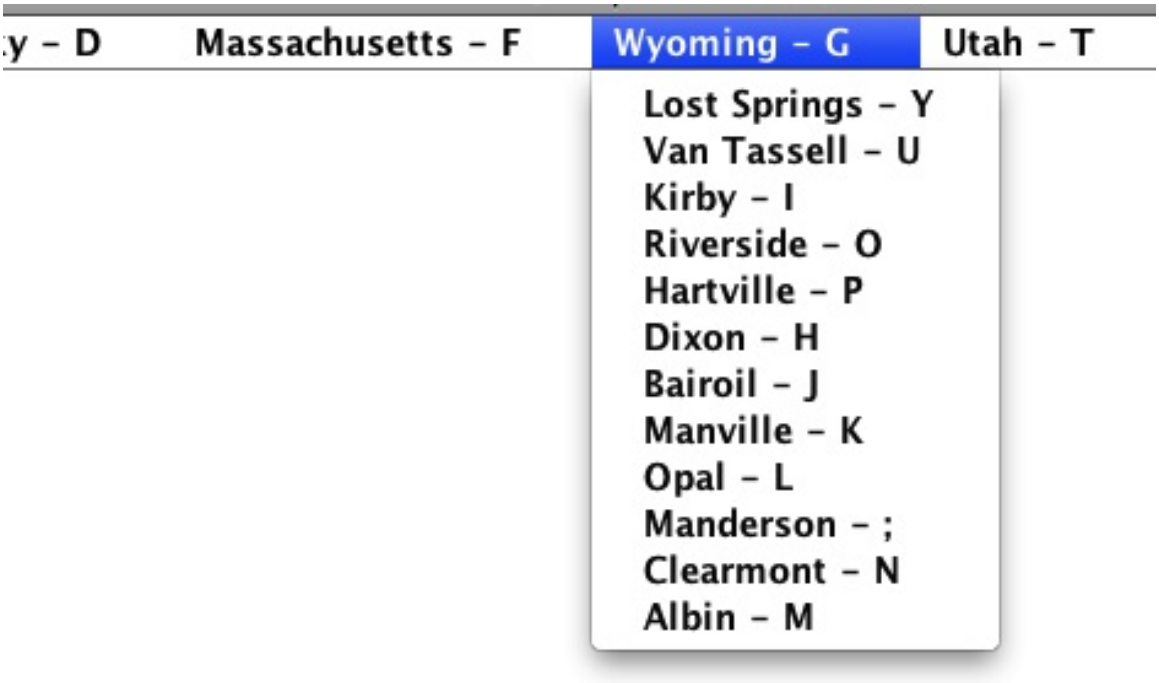

Figure 4.4. Part of a dropdown menu, navigable by pressing and holding keys rather than with the mouse, used in the study; note the letters on each item. Also note that the text is the same font and size as in the keyboard-card menus.

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.

4.1.6 Study Results

The average selection times for the two different menu systems can be seen in Figure 4.5. 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

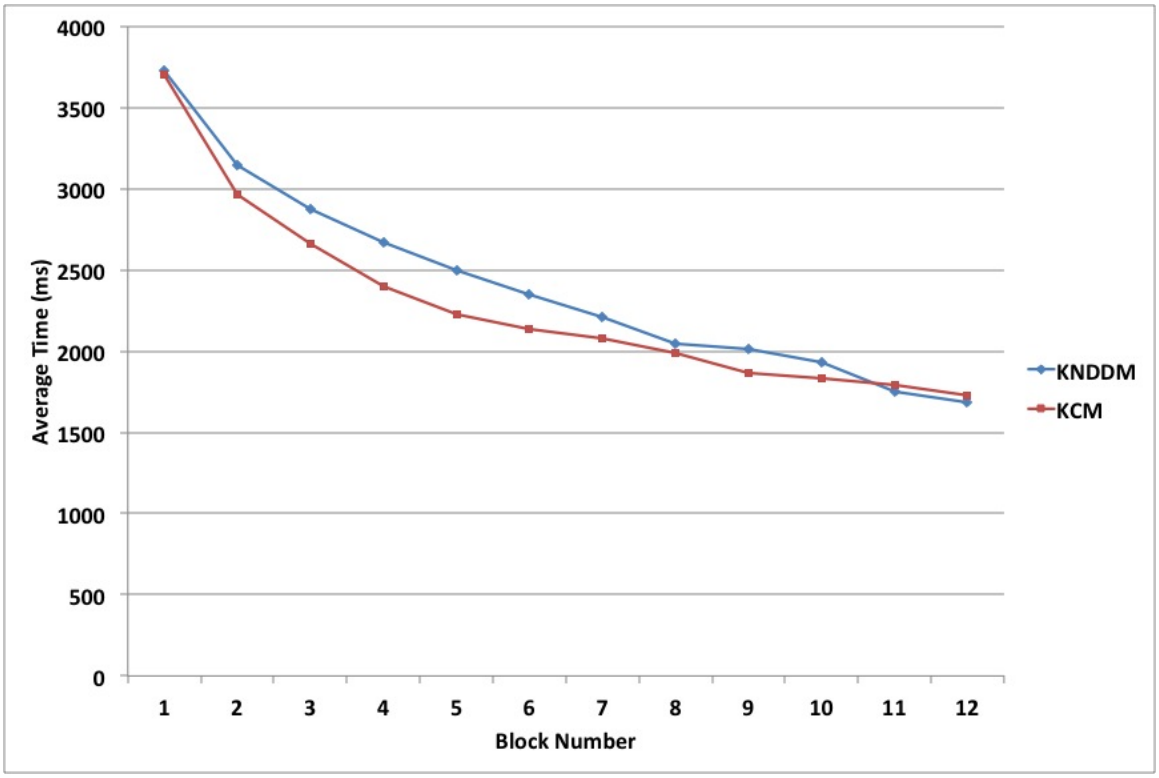

Figure 4.5. Average trial completion times, in milliseconds, for the keyboard-navigated dropdown menus and keyboard-card menus in the study. Note that data is only complete for blocks 1-8

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 fit quadratic equations, as first approximations, 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 (due either to an outlier completion time or to not being able to complete all trials), the result of which was complete data for 323 trials over blocks 1-8; the averages of the times for these individual trials are shown in Figure 4.6. 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).

Average error rates for each trial and menu type are shown Figure 4.7. 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,

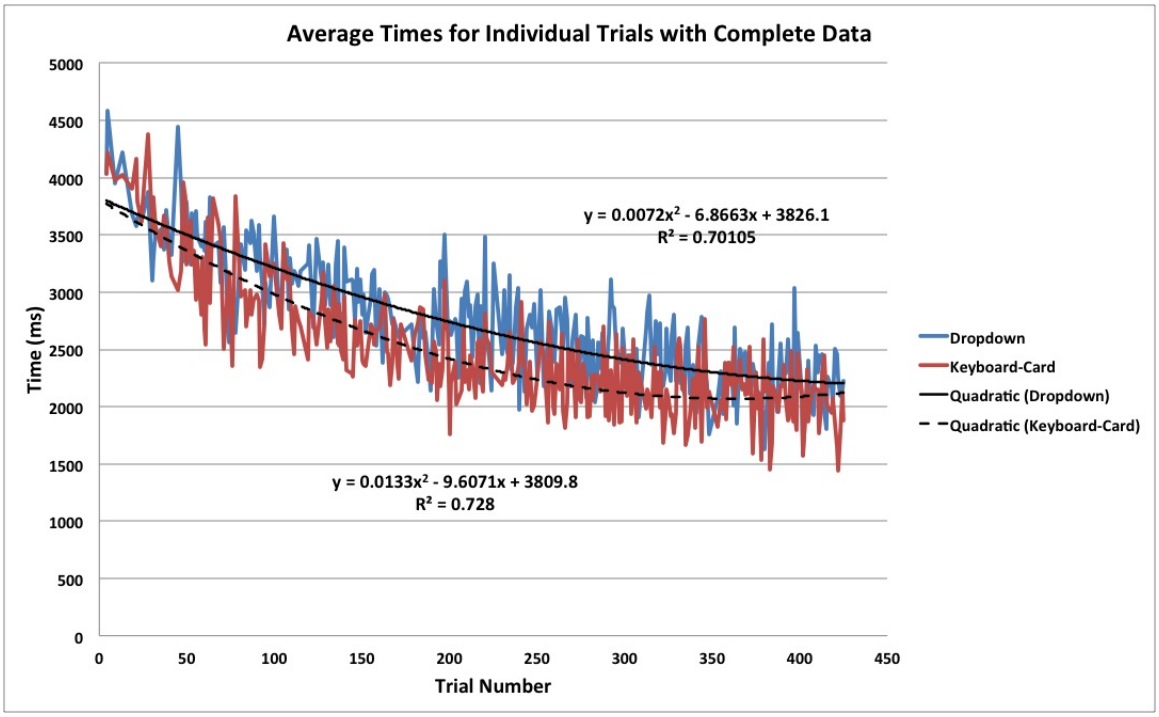

Figure 4.6. Average times across participants at individual trial numbers where data is complete. These individual trials come from blocks 1-8. Also included are quadratic curves fit to the data.

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 4.8) 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 4.9). 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.

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?"

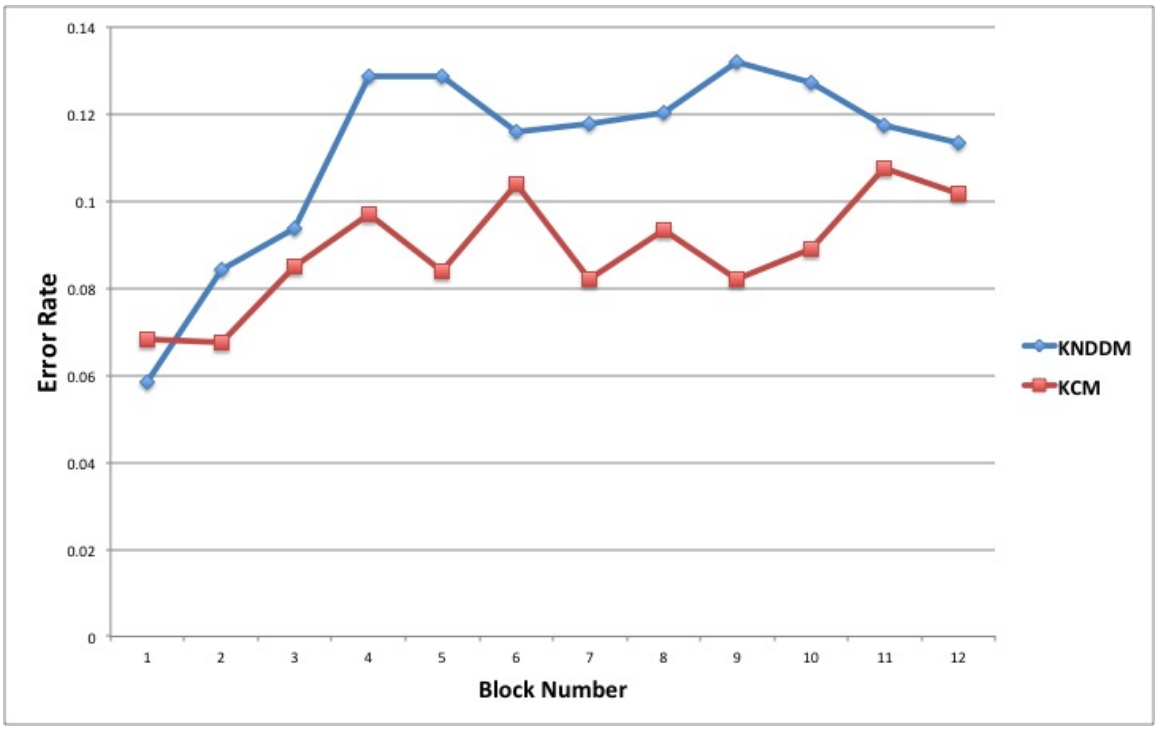

Figure 4.7. Error rates for the keyboard-navigated dropdown menus and keyboard-card menus in the study. Note that data is only complete for blocks 1-8.

- 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 of these ratings 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.

In addition, two open-ended questions were given in the questionnaire:

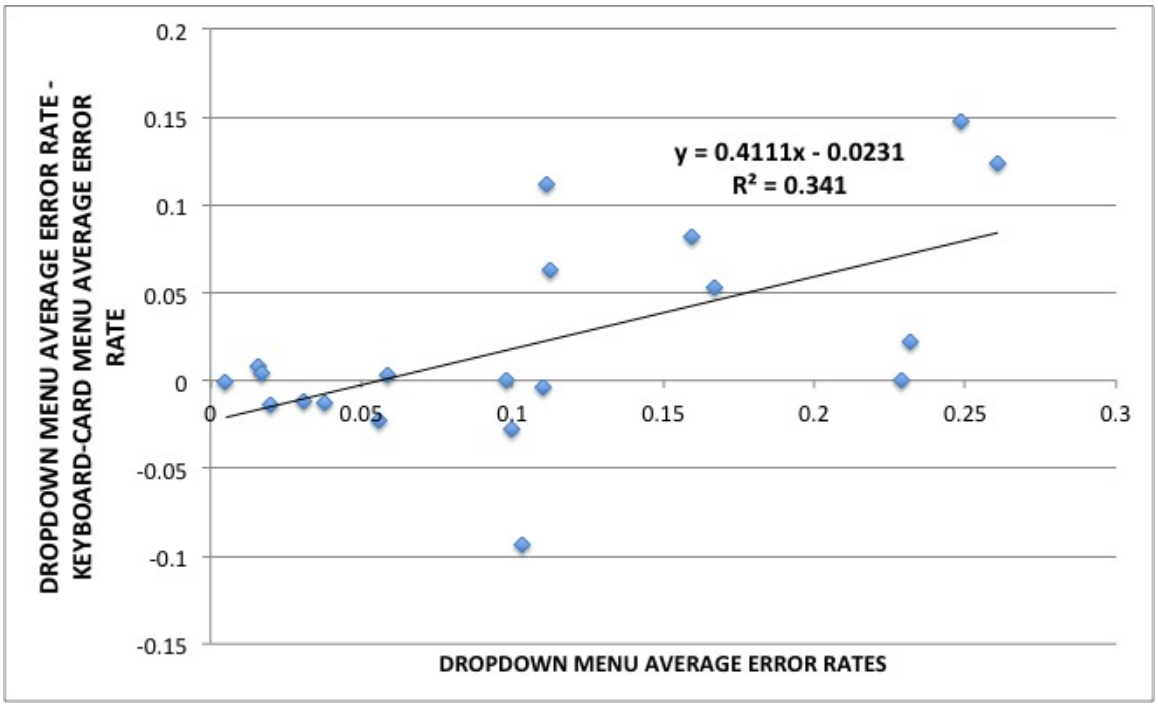

Figure 4.8. The differences between the overall average error rates for dropdown menus and keyboard-card menus, for each participant, plotted against the error rate for dropdown menus.

- Q4 = "For this menu system, did you notice anything that made selecting items difficult? If so, please list and explain."
- Q5 = "Having used both menu systems, which do you prefer? Why?"

Q4 was repeated twice, once in the first half of the questionnaire (taken immediately after using first menu type) and once in the second half at the end of the session. Q5 was the last question in the second half of the questionnaire.

For Q5, 12 of the 20 participants wrote that they preferred the keyboard-card menu, while 8 wrote they preferred the dropdown menus (though with one of the 8, the explanation appeared to give an argument in favor of the keyboard-card menus,

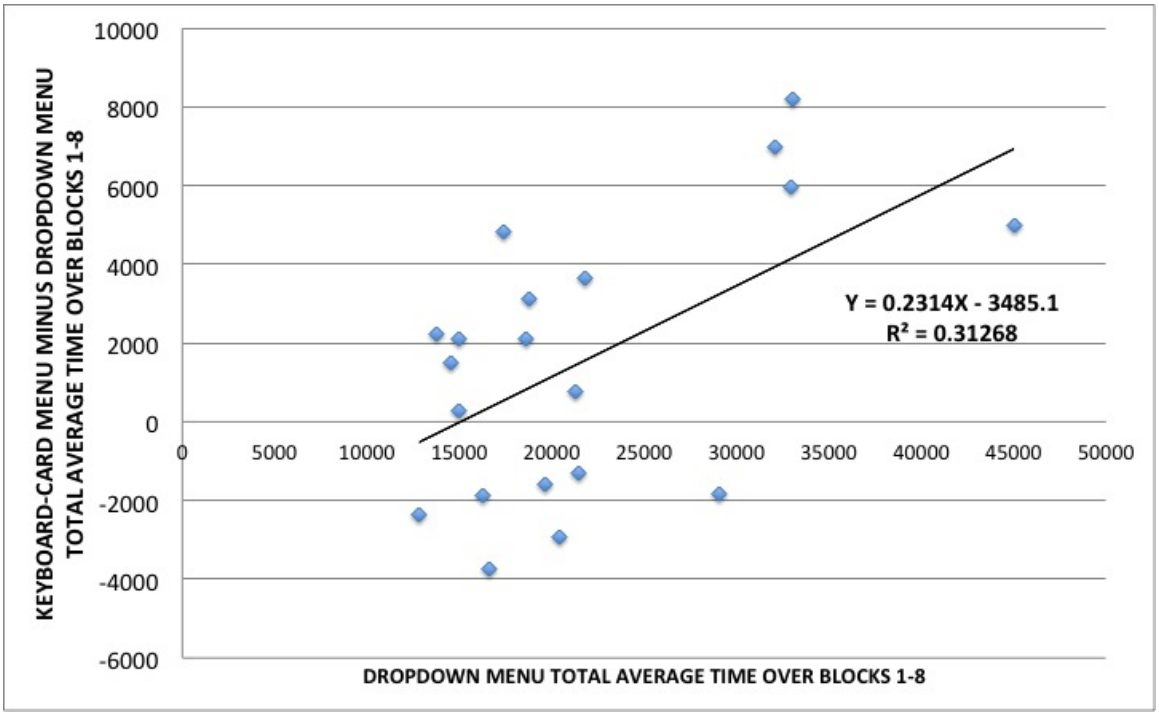

Figure 4.9. The differences between the overall average times for dropdown menus and keyboard-card menus, for each participant, plotted against the time for dropdown menus.

	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 4.1. Questionnaire Results.

so a mistake may have been made in that case which would make the numbers 13 and 7). Explanations for the preference towards the keyboard-card menus included:

- "I seemed to pick up the shortcuts faster. It felt easier to use"
- "There was more visual association in the second"
- "...it gave me a visual of where I needed to type with the keyboard image on the screen. I thought I learned the second menu faster."
- "I felt I got accustomed to it more easily [and] quicker. I believe I finished the 720 trials in the first faster."
- "...it was easier for me to understand it"
- "the layout is more intuitive and it's easier on the eyes."
- "...easier to become familiar with choices"
- "Modeling where to press on the keyboard was helpful, as were the color shifts. It was also easier on the eyes (looking at it was less of a hassle)."

On the other hand, some of the explanations of preference from participants favoring the dropdown menus included:

- "I feel like I was faster with the second procedure. However I believe I made more mistakes along the way."
- "The [keyboard-card menu] might be better for those that use the 'vulture' method of typing as it shows where the key is. The extra space in this layout just makes it take longer to read the flashing colors are distracting once the combinations are memorized."
- "although it's slow starting out, I memorized the keys faster, plus the first one made me feel a little sick D: "
- "more familiar with "dropdown" menu"
- "I like the first menu system because it takes up significantly less space, and it has more flexibility."
- "because then I didn't try to look and check each time."
- "I was able to catch on quicker"

Answers to Q4 after using the keyboard-card menu included:

- "Had trouble w/ Kentucky –¿ might help to alphabetize the key strokes, e.g. 'D' for 'Delaware'
- "I was thrown off that Indiana was 'A' rather than 'I', etc. Also, it was kind of hard to read the words printed over the letters on the screen."
- "I didn't like the change of colors from blue to red –¿ hard to get used to"

- "At the beginning the color changes were distracting toward my goal, but the visual layout soon became helpful."
- "The visual distance between the prompt + the on-screen keyboard"
- "Keys close to each other"
- "I couldn't remember where half the towns were. Seemed to take a long time to locate some cities."
- "I still wanted to sometimes hit the key that corresponded to the letter of the word."
- "Same first letter for states would have made it easier"
- "The background letter images made it harder to read the words."
- "That the names on the left disappeared so if you made the wrong selection you had to depress the key to continue scanning the list."
- "Had to read from left to right at first."
- "The words sometimes mix with the letter in the key"
- "Confusing at first"
- "expanding beyond the home row key set, such as using the key 'T'."
- "having to move up for colorado and right for kansas threw me off a little. fingers had gotten used to staying still"

Answers to Q4 after using the dropdown menu included:

- "Despite the number of trials, it took me forever to learn which letter corresponded to which state (some in the middle I didn't learn). Were menu items ordered in any way? I didn't really pick up on it if they were. Holding down one key and then pressing the other could be awkward when the keys were close together. Maybe the commonly-used items could have moved to the top of each menu to make it easier. Tasks were rapid-fire–I thought I was trying to go a little too fast sometimes"

- "With the state/city combination I was tempted to do the combination of keys that corresponded with the first letters of the word. ex Kentucky -¿ Fairfield I wanted to do K - F."

- "There were way to many choices"

- "beginning letter of state [and] beginning letter of city"

- "I was inclined to associate letters with the names of the places (L - little creek) so the lack of matching made it more difficult. Also, when I had to shift from the standard typing position (G/H for instance) my response time slowed"

- "It's much more difficult to read off a dropdown list than with the keyboard layout. Also, the dropdown list is not in alphabetical order, which oftentimes mislead me to the wrong answer."

- "One thing that I found difficult was when the letters of two options were

close together on the menu. I kept on switching them around in my head and sometimes forgot which was which."

- "I felt like my eyes were doing more work looking around the screen than [with the keyboard-card] trials. Also, having towns on keys which didn't correspond w/ their first letter made it difficult at first."

- "My first instinct would be to hit 'L' for Louisiana rather than 'A', 'C' for Connecticut rather than 'D', etc. Also, the cities under each state's menu were not actually cities in these states, so it was hard to remember their names."

- "It was hard to keep track of the Connecticut/Canaan and Connecticut/Cornwall difference. It was very difficult locating the names on the dropdown lists."

- "They weren't in alphabetical order"

- "At the start, I had to scan the list to find out which key belonged to which city. That lost me some time"

- "The place to look for second menus keep changing and sometimes I would even have to spend time looking for where the menu is."

- "Some menus were in alphabetical order & others weren't. Also as one key had to be held down this slowed me as I became more familiar & memorized combos because sometimes they weren't recognized when typed too fast if my finger didn't stay on the 1st key"

- "Utah -Scofield were right next to one another [using the T and U keys], which made it awkward. Also, there was no alphabetic order to the menu, so it took a long time to find unfamiliar items."

4.1.7 Discussion

The selection time 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 4.5 means that by the 12th block users had (mostly) 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 helped users in finding and recalling shortcuts they had seen but not fully memorized.

Although in Figure 4.5 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 4.5) 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 4.5 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 4.8 and Figure 4.9 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.

After performing the study, we have come up with a relatively simple explanation for the differences between both the average time curves *and* the error rate curves, not among the four given in the "Study Motivation" section above: keyboard-card menus make it easier for users to check their knowledge of the shortcuts. For instance, suppose a user has gained only an intermediate level of experience with the shortcuts and, therefore, wants to check that the shortcut "D-[L]" is the one he wants to use. To check this using the dropdown menu system *without actually invoking the shortcut*, he would have to press and hold the D key and then visually scan through the list of items that appear until he either sees the L key in the list or the menu item itself. In contrast, after pressing the D key using the keyboard-card system, he can move his gaze much more directly to the position of the menu item associated with the L key because he knows where the L key is within the keyboard layout.

From the written responses to the questionnaire, several issues emerged as significant. First, the menu items were not organized either alphabetically or by item frequency. Second, how far users had to move their fingers to invoke a shortcut (e.g. the U key vs. the J key) mattered to them. Third, users' normal expectation is that the shortcut key used will be the same as the first letter of the menu item. Finally, some users found the bright flashing colors bothersome. How much these

issues actually slowed users down, and how much they affect users' overall feelings about the shortcuts, remain to be investigated. However, they may be used as an initial set of heuristics in future keyboard-card menu applications (e.g. given a choice between associating a menu item with the U key or the J key, pick the J key).

4.1.7.1 Future Work

Many issues surrounding keyboard-card menus remain unexplored. Perhaps the most important next step is to test the menu system in an actual application; one such application under consideration is interactive theorem proving with the Coq Proof Assistant[11], since its use involves manipulation of a wide variety mathematical notations using a large set of commands.

Other questions that may be addressed (many of which arise from the results of the study discussed in this paper) 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 (e.g. simple key sequences)?
- What sorts of color schemes should be used? Should bright colors be completely

avoided because they seem to bother some people? Should each card have its own color/texture to make it easier to tell where one is in the hierarchy? (In the study, all sub-cards in the second level were red).

- How much of a difference does it make to assign a menu item to a key that matches the first letter of the menu item (e.g. assigning the "Idaho" submenu to the I key)? And if one were to do so as much as possible, would inconsistencies caused having two items with the same initial letter make the error rate go up? Would removing the large letters from the keys in the keyboard-card menu help reduce confusion?

- Would using dropdown menus alphabetized by shortcut character, with or without extra "inactive" locations, make up the difference in times seen between the two menu types in the study? Would alphabetizing by menu item name (e.g. placing "Atomic City" ahead of "Clayton") in the menus have made any difference?

- What can we do to prevent users from missing shortcut invocations because they type too fast? For instance, would keeping a submenu key "active" for a few milliseconds after it is released fix the problem? Can we try to guess the intended shortcut (e.g. can we sometimes assume that "F, J" was meant to be "F-[J]" if "J" by itself is not a valid shortcut)?

- How effective are these menus for users who do not touch type?

- How much of the difference does using keys other than A, S, D, F, J, K, L, and

semicolon make in expert speed, learnability, accuracy, etc.?

4.1.8 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.

4.1.9 Acknowledgements

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

4.2 Keyboard-Card Menus + Syntax Tree Highlighting: A Case Study with a Subset of Coq

Here I briefly present a prototype structure editing[2] system. The system uses keyboard-card menus to move segments of highlighting within the buffer around, in

[2]In general, this means editing with an awareness of document validity restrictions, in contrast to systems that work by entering text character by character–many existing systems might be viewed as hybrids. Coq, because of its tactic system for building proofs, could itself be considered a structure editor.

accordance with the underlying syntax tree.

There are several major benefits of this approach. First, particularly for systems like Coq that involve complex nesting of (often computer-generated, poorly formatted) code, the approach may help users understand the structure of the text, without having to leave the editing environment. Second, it prevents users from writing many classes of ill-formatted code in a way that may be especially useful to novice users unfamiliar with the rules of the language. Third, it allows boilerplate code to be easily inserted from templates, reducing the number of necessary key strokes. Finally, it comes with the learnability advantages of Keyboard-Card menus, as discussed in the previous section.

Perhaps the best way to understand this system is through an example. First note that the ultimate goal is to produce a Fitch-style proofs using Coq that are in the format of the following proof :

```
Parameter phi : Prop.

Section _1.
  Hypothesis l1 : ~(phi \/ ~phi).
  Section _1_1.
    Hypothesis l2 : phi.
    Fact l3 :  . Proof. apply (or_intro_l l2). Qed.
    Fact l4 : False. Proof. apply (not_elim l3 l1). Qed.
  End _1_1.
  Fact l5 : ~phi. Proof. apply (not_intro l4). Qed.
  Fact l6 : phi \/ ~phi. Proof. apply (or_intro_r l5). Qed.
  Fact l7 : False. Proof. apply (not_elim l6 l1). Qed.
End _1.
Fact l8 : ~~(phi \/ ~phi). Proof. apply (not_intro l7). Qed.
Fact l9 : phi \/ ~phi. Proof. apply (not_not_elim l8). Qed.
```

This is Coq syntax that directly corresponds to a Fitch-style proof as presented

in [62] (some axioms have been added): "Section"s are used in place of the more normal nested boxes used in Fitch-style proofs. [3] This proof can in fact be checked by Coq.

In Figure 4.10 we see the initial state of the system. The dark green text represents the currently selected node. In general, text between "(*" and "*)" is comment text, and throughout this example is being used to show "placeholder" text that either must or could be filled in. In Figure 4.10, this placeholder text indicates that a propositional variable is required. The keyboard-card displayed in the bottom half of the window is already set up so that a propositional variable may be typed in. (In future versions of the keyboard-card menu system, keyboard-cards with different, and possibly larger, sets of keys may be supported so that, for instance, the slash key does not need to be used for backspace).

---

[3] If you are not familiar with such proofs, these indicate where hypotheses are assumed to hold. You may also note that each "Fact" (synonymous to Coq with "Lemma" and "Theorem") is justified using earlier lines in the proof.

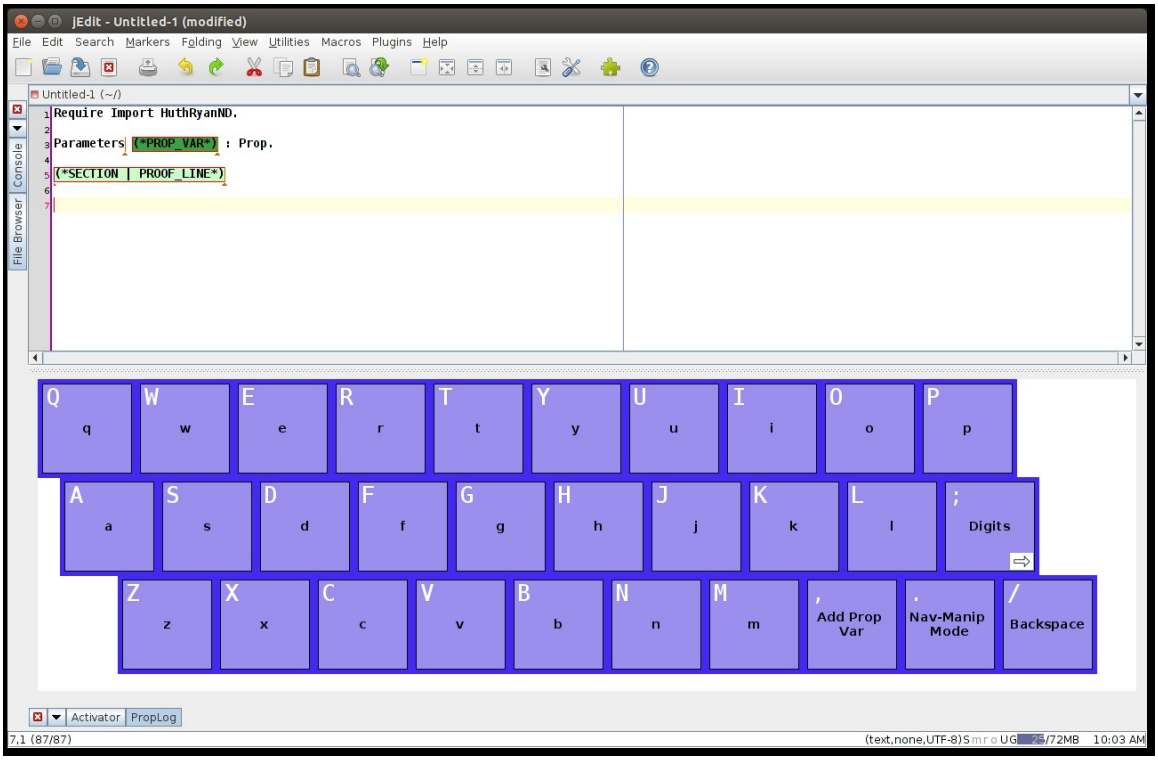

Figure 4.10

Figure 4.11 shows the system after the A key has been pressed but before it has been released (hence the graying out of key labels). Once the A key has been released, other keys can in fact be pressed so as to enter multi-character identifiers.

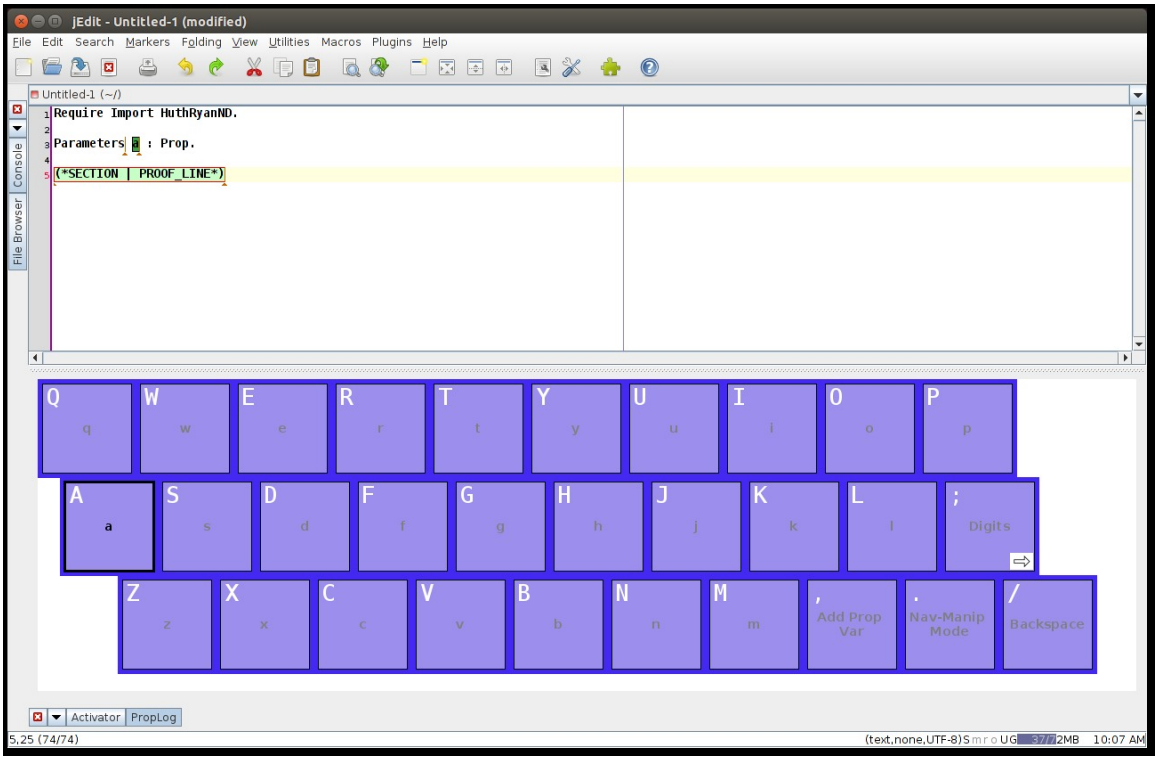

Figure 4.11

Figure 4.12 shows the result of releasing the A key and pressing the period/"Navigate and Manipulate Mode" key. This has the effect of changing the current keyboard-card menu card tree so that the Nav-Manip root card is displayed and active.

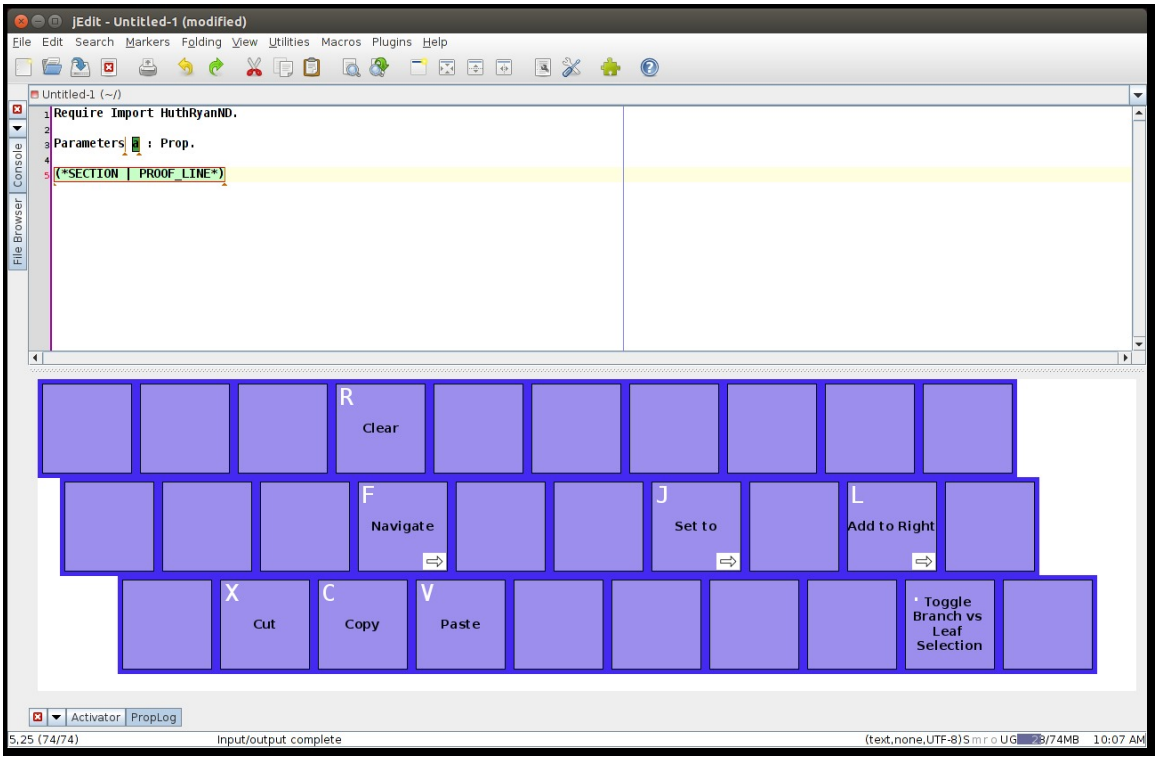

Figure 4.12

In Figure 4.13 we see using the navigation submenu to move to the next leaf in the tree. This is actually a placeholder for an optional list of space-separated propositional variables. Placeholders for optional text only appear in full when they are selected, and are otherwise marked with a small triangle below the line of text and a vertical line within the line of text.

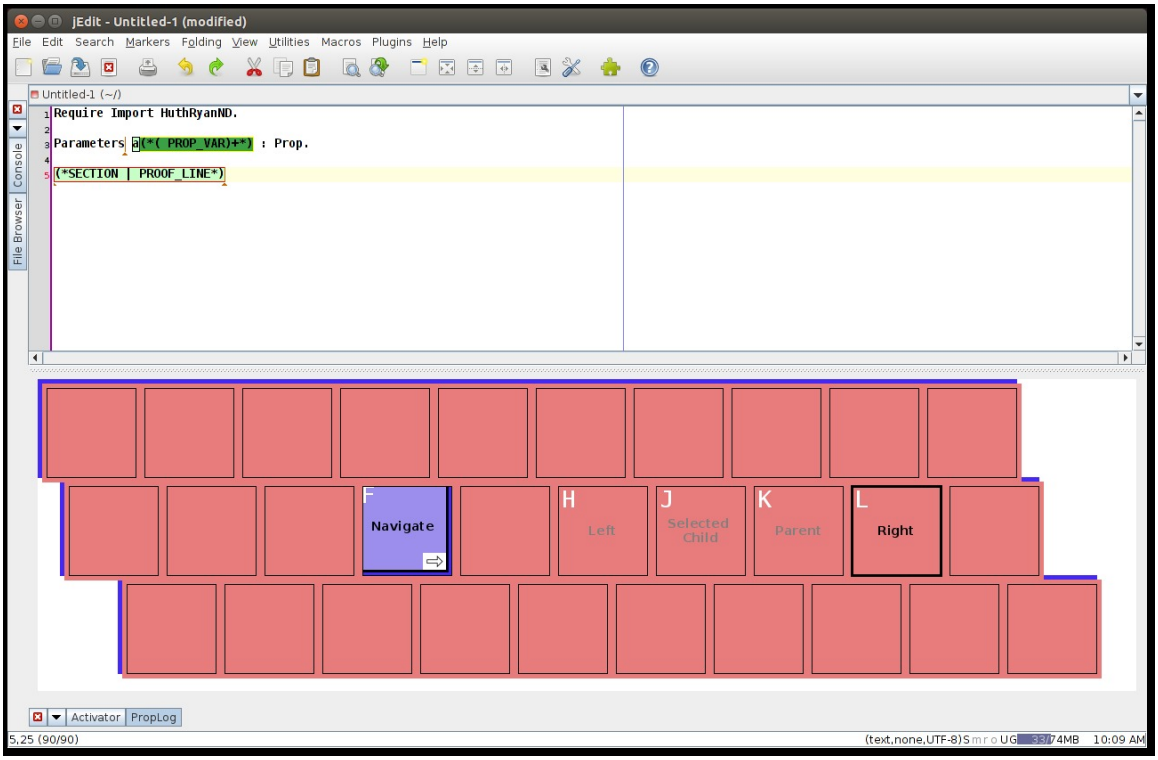

Figure 4.13

Figure 4.14 shows the "Set to" submenu, after the user has moved right two more positions to where "Sections" and "Proof Lines" can be inserted. In future versions of the editor, the other items in the submenu may be grayed out, but at present their selection has no effect.

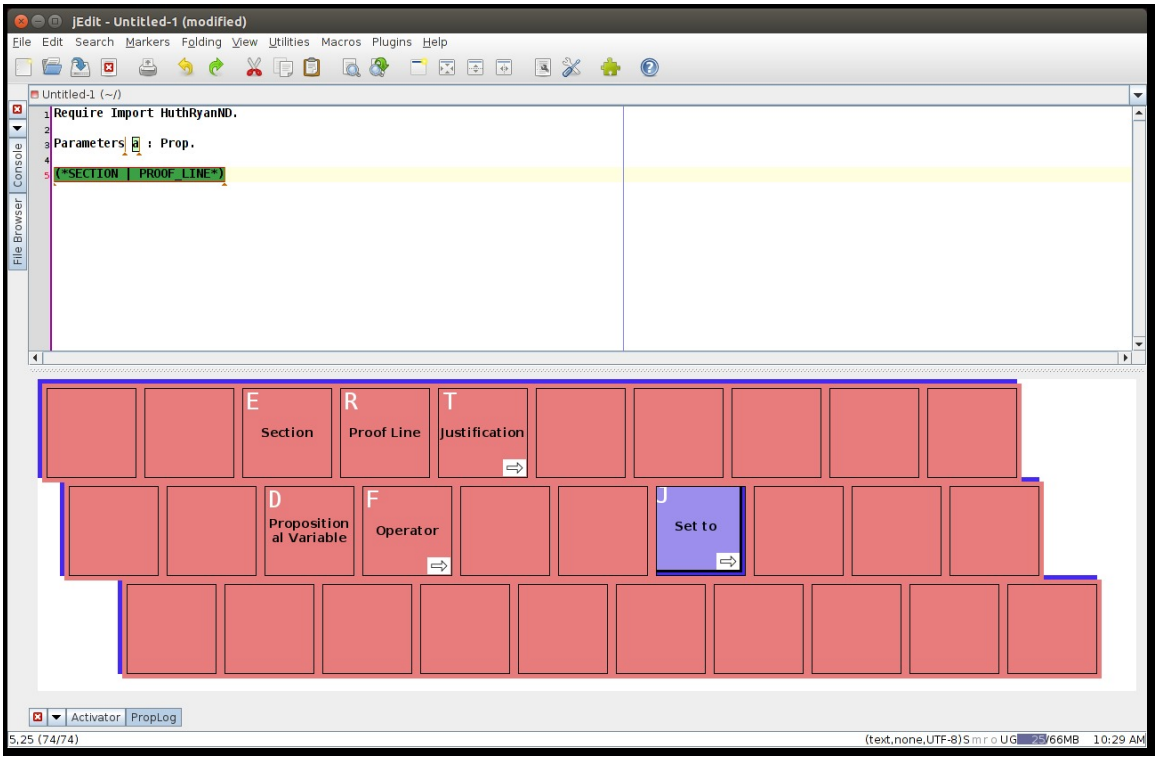

Figure 4.14

Selecting "Proof Line", as in Figure 4.15 has the effect of inserting a line starting with "Fact..". The "Line Identifier" node is selected, and can be edited using a keyboard-card tree similar to that seen in Figure 4.10. The user switches to this keyboard-card tree by pressing the "Set to" submenu key while the Line Identifier placeholder is selected—in this case, the submenu key behaves like a leaf key.

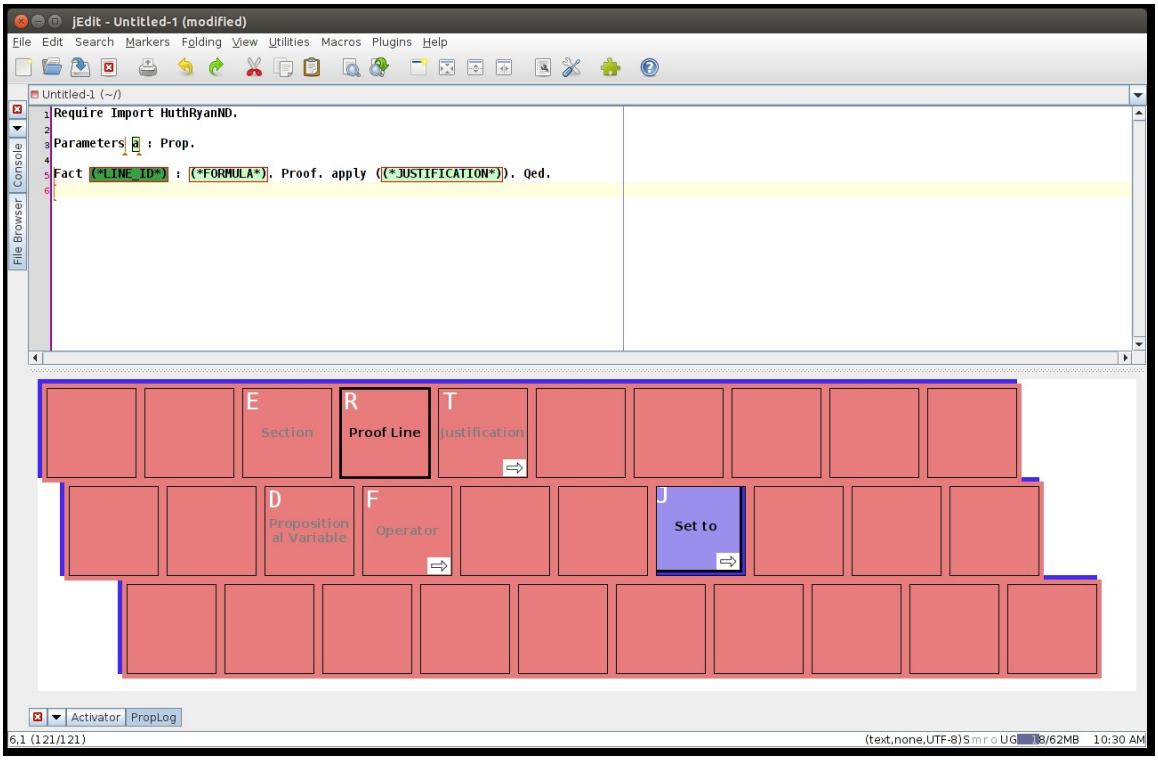

Figure 4.15

After switching back to Nav-Manip mode and moving to the formula placeholder, the "\/" (OR) operator may be inserted along with two formula placeholders. This is accomplished by displaying the "Operator" submenu from the "Set to" submenu (so three fingers are involved).

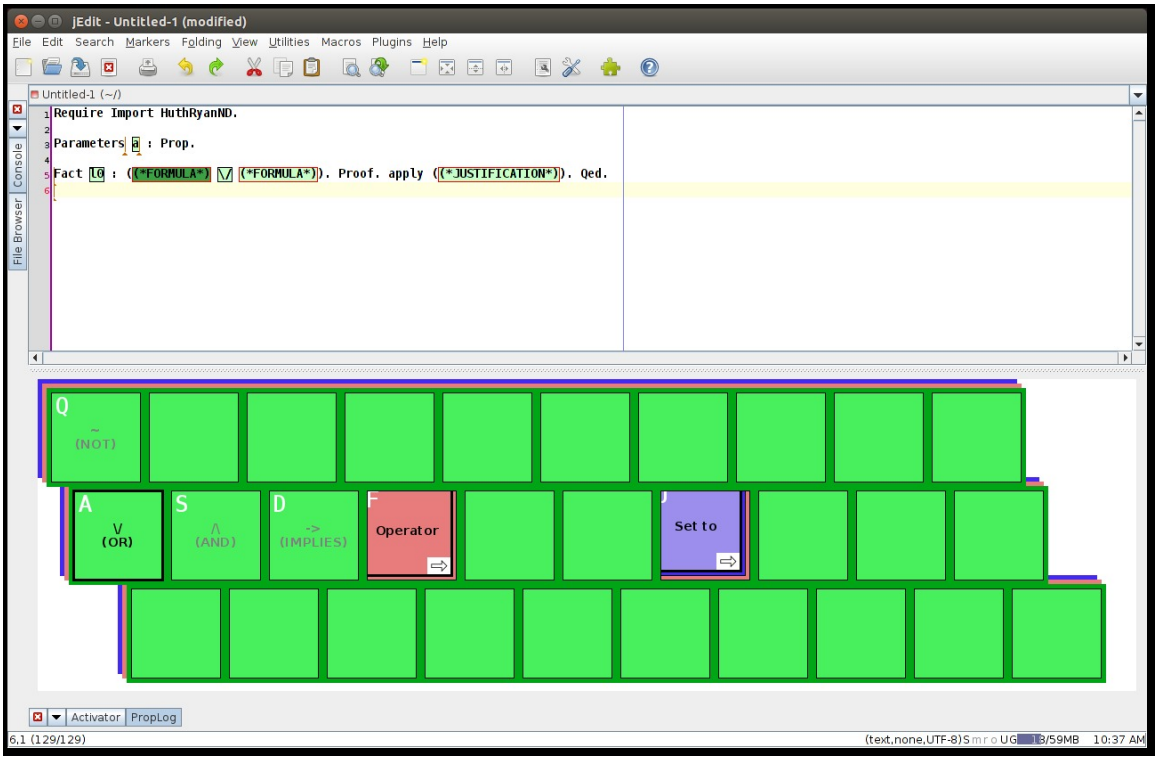

Figure 4.16

Without having to lift one's index fingers, one can then replace the first of the two new formula placeholders with another binary operator and pair of placeholders. Not shown, one could have used the "Add to right" submenu to replace the current formula, call it "A", with a binary operator whose left subformula is A and whose right subformula is selected.

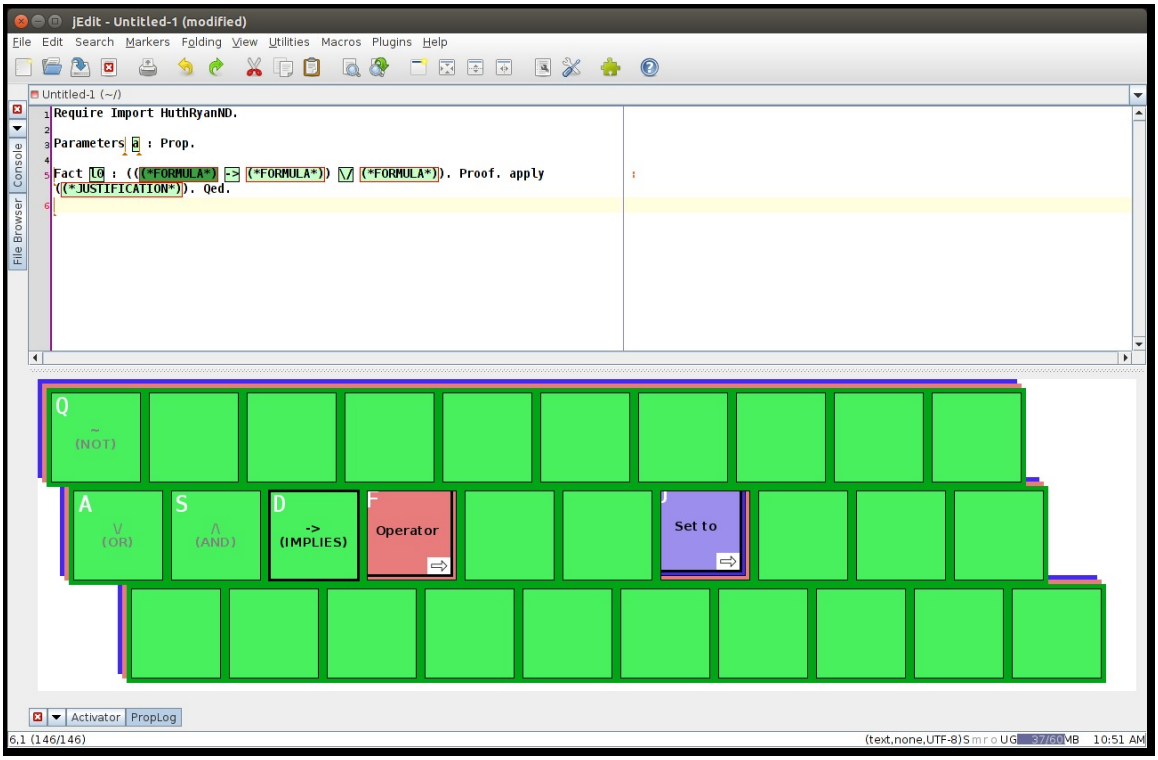

Figure 4.17

There are actually two different selection modes. Leaf mode, which we have seen, draws green highlights over selectable leaves and the user can move left and right among these. Branch mode, seen in Figure 4.18, draws brown highlights over a set of sibling nodes.

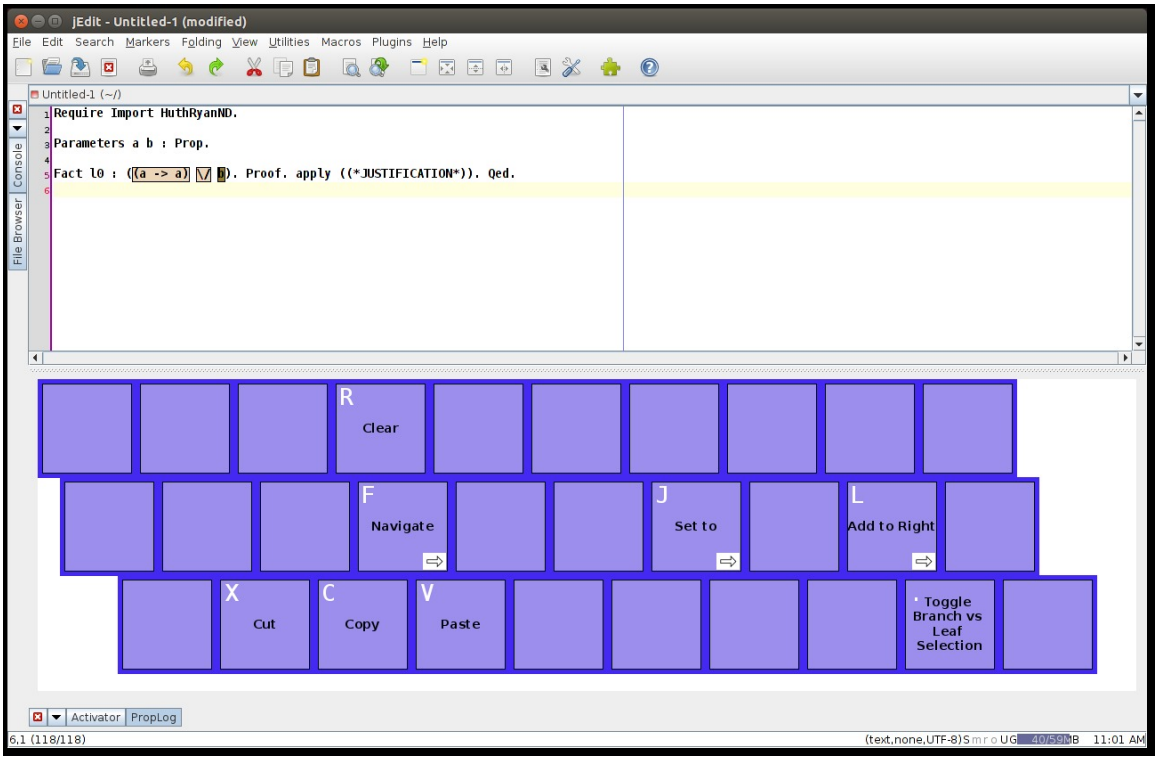

Figure 4.18

Users can move left, as seen in Figure 4.19...

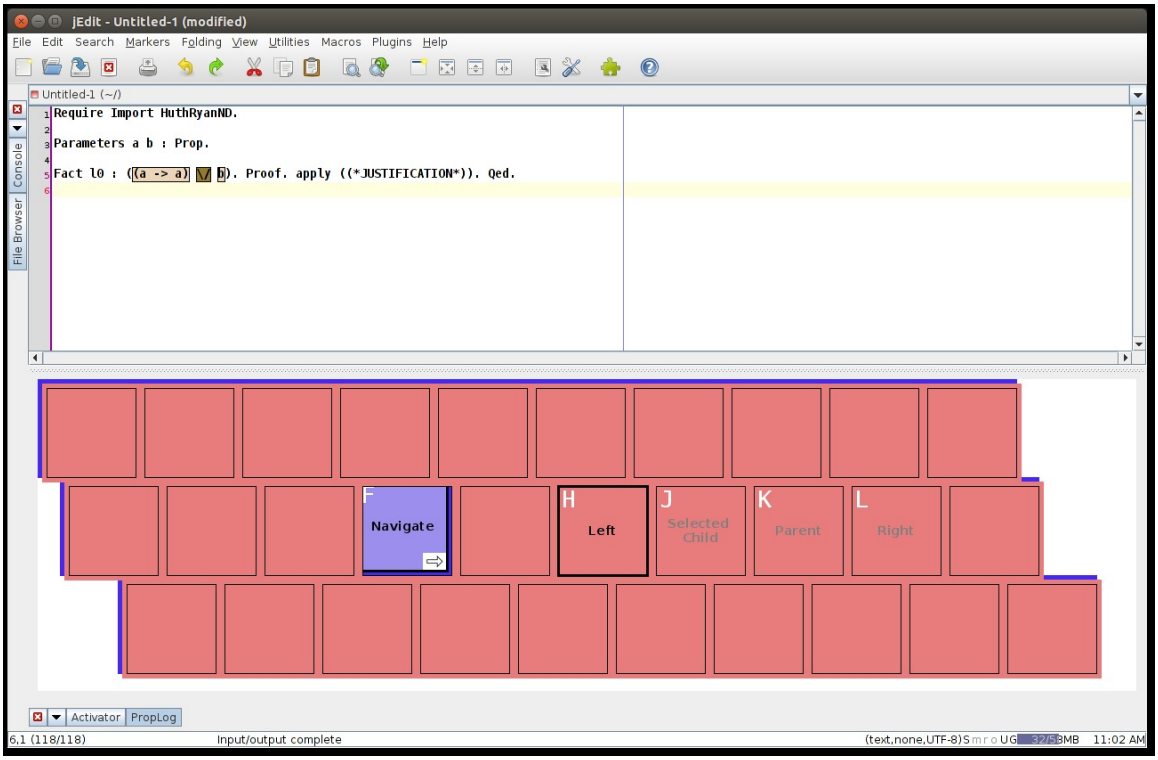

Figure 4.19

...to the parent, as seen in Figure 4.20...

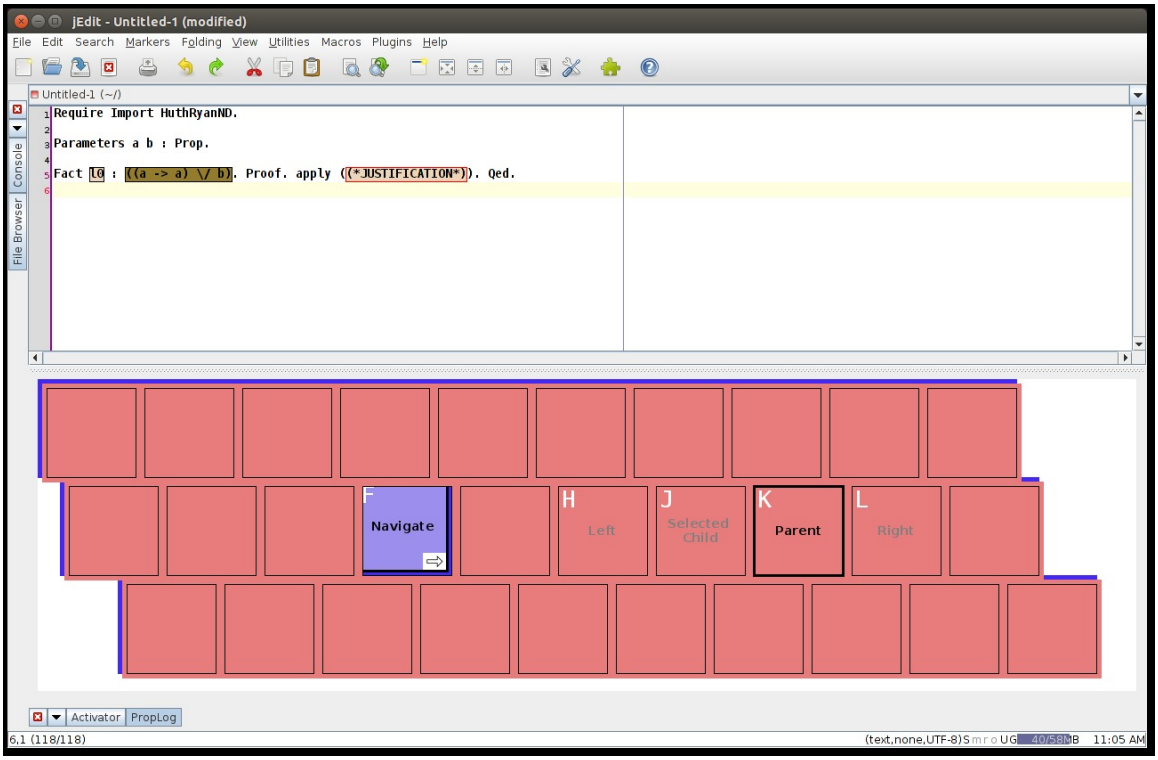

Figure 4.20

...right, as seen in Figure 4.21...

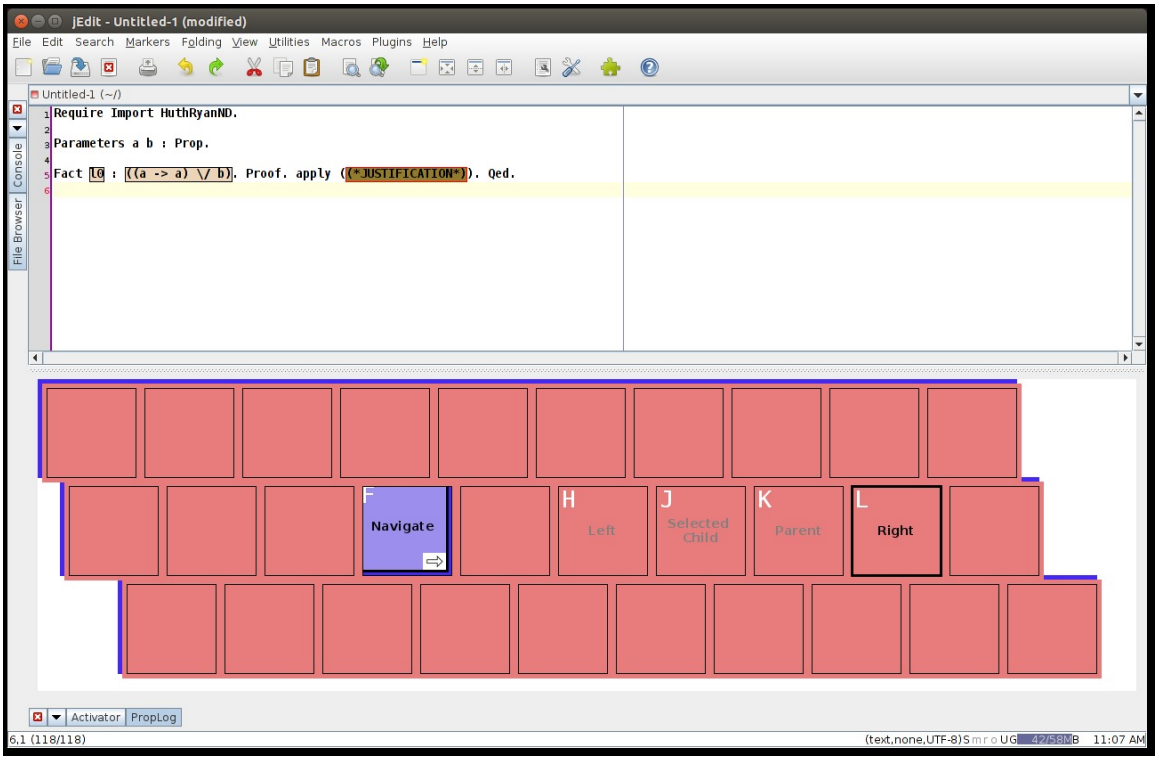

Figure 4.21

...or back to the selected child node. Other selection modes (e.g. movement using in-order or level order traversal) may be considered in future iterations of the design process.

CHAPTER 5

RELATED WORK

The need for effective theorem prover user interfaces has been recognized for over 20 years (e.g. [102]) 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 [87, 93], easier-to-read declarative languages for proof scripts [42], using wikis for creating proof repositories [43], and automating the process of using libraries [17].

One approach taken to improve theorem prover user interfaces has been "Proof by Pointing"[31], an algorithm to build a proof tree by pointing to portions of a goal. This was implemented in the *CtCoq* interface [30], again, later, in the Java-based Pcoq interface [16] and also in the *Jape* system [36]. Note that some schemes for writing and displaying proofs, notably Fitch style proofs, do not display hypotheses with their conclusion, which may lead to ambiguity when using Proof by Pointing [35].

An example of a recent project is *Panoptes* [49], 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

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

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 [21]. 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 where a node represents a statement that is being proved using the node's children. Color coding is used 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 [101]; an

"Interactive Derivation Viewer" for visualizing derivations written in the TPTP language [103]; the LOUI interface for the OMEGA proof assistant, featuring graphical visualization, term browsing, and natural language proof presentation [99]; and the Tecton system, featuring tree visualization combined with hypertext navigation of nodes [64]. Theoretical and methodological work has also been done and can be seen in [46, 32, 79, 107, 52], and [78]. Work aimed at making ITPs more suitable for novices and educational settings includes [104, 85, 87, 33, 45, 57, 96] and [90].

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 [86] 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). One may also note, here, the existence of various lines of work, e.g. the *Frama-C* [4] project, aimed at integrating theorem proving and programming within IDEs, and the *KeY* [27] project (for "Integrated Deductive Software Design") which integrates graph visualization with theorem proving in the Eclipse IDE.

In addition to IDEs, we may consider software visualization tools: [24] reports 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 [39]. Similar to [21], 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* [47], for instance, miniaturizes lines of code with lines of color-coded pixels, and *sv3D* [83] 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 [15, 59] and using texture and 3D object primitives called "geons" to encode additional information [60, 63].

The more general human-computer interaction literature on tree visualization is quite extensive–many techniques have been developed. Four different "common layouts" are listed in [59]: rooted tree (child nodes arranged on a horizontal line above or below their parents), radial trees (nodes of each level of the tree are arranged in concentric circles, with the root node at the center, its children at the first circle out, their children at the next circle out, etc.), balloon trees (each parent's child nodes are arranged radially around the parent), and treemaps (which divide a rectangle into smaller and smaller rectangles with the largest rectangle representing the root of the tree and the smallest representing the leaves, and where division of the rectangles switches between using horizontal and vertical lines when going between tree levels; these were developed in [98]). These common layouts have been extended in various ways. For instance, [88] describes a space-optimized version of balloon trees and [105] describes the the addition of gradients to the rectangles in treemap visualizations, making the structure of the tree easier to see.

[65] presents an extensive review of tree visualization techniques. Along with classifying visualizations as 2D or 3D (or 2.5D when no movement in or manipulation of a 3D visualization is allowed), it divides the visualizations into six overlapping categories: indented list, node-link and tree, zoomable, space-filling, focus+context or distortion, and three-dimensional information landscapes. Indented lists can be seen in file system browsers, e.g. *Microsoft Windows' Explorer*, the rooted and radial

trees mentioned in [59] as common layouts are "node-line and tree" type, and treemaps are an example of the space-filling type. An example of a zoomable visualization is *Grokker* [94] which allows users to click and expand nested circles. An example of the focus+context or distortion category is the *Hyperbolic Browser* [71] which magnifies and centers on the area around a selected node in a radial layout. An example of a three-dimensional information landscape is the *Harmony Information Landscape* [48].

CHAPTER 6

SUMMARY AND CONCLUSIONS

This document contributes in several ways. First and foremost are the development of user interface prototypes and their evaluation. These prototypes include novel applications and implementations of existing user interface ideas to interactive theorem proving: the automatic reorganization of the layout of the tree visualizatio used in Proof Transitions, representation of the cached evaluated section in the basic CoqEdit environment, the presentation of suggestions in Proof Previews. The prototypes also include new schemes for interacting with proofs that may be generalized to work with many other software applications: keyboard-card menus, syntax tree highlighting, and text-manipulation visualization seen in Proof Transitions.

In addition to these user interface ideas, insight into several other issues is provided. One lesson learned in testing Proof Transitions is that the Paradox of the Active User–that users may be too busy using software to explore its features–may extend further than one might expect: not only do inexperienced computer users fail to take advantage of shortcuts when dropdown menus are available, but experienced computer science students often also fail to take advantage of (some) keyboard commands even when interaction using the mouse is not available. This is evidence of the importance of the design principle that one should make taking advantage of features unavoidable (at least by default).

Another lesson learned is on how user interface studies for interactive theorem

provers may be conducted. The original plan for evaluating the effectiveness of Proof Transitions was to do a blind grading of proof explanations. Unfortunately, it turned out (in the pilot testing for the experiment) that the explanations given by participants would make no sense without an accompanying view of the screen (so blind grading became impossible).[1] The solution to evaluating the effectiveness of Proof Transitions turned out to be looking for more objective ways in which Proof Transitions might be better. One attempt at doing this was by encouraging participants to give specific information about the effects of each tactic (e.g. how many nodes does it produce). However, participants usually seemed to find either that these instructions were confusing or that such precise detail about each node was unimportant for their explanations of the proofs (which was probably reasonable, since many tactics produced similar or identical results). The solution that ultimately worked, looking at whether participants stated when a destruct tactic created two child nodes, involved looking at what participants actually included in their proof explanations (rather than what we expected would be in them) and did not require forcing participants to change their behavior.

The final contribution to be mentioned is simply the set of results obtained. With Keyboard-Card Menus, we have evidence that users are capable of interacting with a computer through a new interface and that the presentation used by this

[1]The natural way of explaining the proofs turned out to be to explain what each tactic did as one moved through the proof, rather than describing the overall structure and important points in the proof as we had expected. This was probably the result of limiting the number of tactics to which participants were exposed, and only using propositional logic, while still keeping the proofs complex by having many branches.

interface has learnability advantages over dropdown menus. With the Proof Previews experiment results, we have evidence that the menu of suggestions allow users to complete more proofs in less time, and that users usually prefer using the feature to typing out the proof character by character.

With Proof Transitions, although a majority of participants did not prefer the new interface, one can see positive results when bearing several facts in mind. First, this was not a winner-take-all comparison and a large minority of participants did in fact prefer the Proof Transitions interface; one-size-fits-all is probably not what is needed in theorem prover user interfaces, and, as suggested by both the novice users in the study and experienced users in the earlier survey, many-sizes-to-fit-each may be more appropriate (e.g. a user interface for teaching or presenting proofs vs. a user interface for writing them). Second, this large minority was present despite the experiment being biased against Proof Transitions (since the prototype version of Proof Transitions that was presented to participants did not allow proofs to be written or edited, though a non-prototype version might, and since the tutorial that was seen by participants was presented primarily using the basic interface). Third, there are many ways in which Proof Transitions could be refined to boost user preference for it higher, as discussed below.

This brings us to the question of where to go from here. I see at least four major paths. First is the next iteration of **refinement and testing of Proof Transitions**. This might include:

- Automatically adding visit-marks (the purple squares seen in Figure 3.9) to

nodes as they are visited. The original rationale for not doing so was that users would want to only mark nodes they had actually taken a close look at and felt comfortable they understood (which, as far as I can tell, is still accurate), and it is very easy to navigate through the tree without actually looking at the contents of the nodes. The problem was that users forgot to use the visit-mark feature. Reasonable compromises might be 1) to add the marks only after the user has continuously been at the node, zoomed in, for a few seconds or 2) to gradually make the mark fade in as the user remains at the node.

- Improvements to the transition highlighting and animation (i.e. blinking yellow highlighting). Making the animation play automatically when one moves to a node would be one improvement, since users tended to forget about the animation's existence. Giving the user the ability to move through the matching text at his own pace (e.g. by pressing keys) might be another; one could then allow the animation to be played more quickly when one moves to the node, knowing that the user has the opportunity to go back at a slower pace. It also seems unlikely that blinking highlighting is the best way to match sections of text; other options that may be more salient include drawing lines between matching locations and making a copy of the text float to its matching location.

- One major advantage of the basic CoqEdit interface seems to be the fact that node traversal was linearized–the user is forced to go through the nodes in a particular order. This means faster traversal of the proof tree, as well as making

it easy to tell which nodes have been visited. Adding forwards and backwards commands for both in-order and level-order would be relatively straightforward (though it might be helpful to zoom out during the larger jumps).

- Having a mini-map, rather than forcing users to go through the extra step of zooming out, might make it easier to keep track of where one is within the tree. At the same time, one could allow for more than two zoom levels so that users could inspect branches of the proof tree (as well as pairs of nodes or the entire tree structure).

There are also some longer-term issues to be addressed. Perhaps the most challenging would be a mechanism for getting sets of matching text resulting from tactic use on particular goals. While it may be best to leave to back-end developers the task of providing commands to print out relevant information[2], there are also probably still more design decisions regarding what information is relevant as well has how to present it to the user. For example, a user might be interested in the procedure by which a tactic is executing other tactics (e.g. the fact that the "intros" tactic repeatedly executes the "intro" tactic).

Other longer-term work on Proof Transitions includes:

- Integrating the visualization with the editing environment. Presumably this will follow a model similar to that of Proof Web (see Figure 2.3), but inevitably there will be some complications. One important difference with Proof Web

[2]This would be analogous to a command line debugging tool for a programming language.

that I hope to see is the ability to edit the proof script "from the visualization" at arbitrary nodes in the proof tree. (This would require some automatic code generation and, ideally, automatic code formatting).

- Dealing with nodes containing large amounts of text. One solution here would be to make the nodes scrollable (like ordinary windows). Other solutions that might work better with the text matching visualization might be to shrink the node's text or to expand the box containing the text. Hybrid solutions are another possibility.

- Visualizing advanced tactics that actually have effects on unproved goals other than the current goal. These tactics are not necessary, and users could be prevented from using these, but some relatively minor modifications to the existing visualization could probably be used to represent these.

A second path for future research would be to **fully implement Proof Previews**. One complicating aspect of this is performance: how to index libraries, and perhaps use some parallel processing, to generate a list of possible tactics and theorems to apply in real time. Two additional related problems are how to put the most helpful suggestions at the top of the list and how to avoid not only tactics that immediately produce errors, but also tactics that produce unprovable subgoals. This could be very challenging as it involves techniques from automated theorem proving (a discipline in its own right).

A third path would be to **improve and optimize Keyboard-Card Menus,**

Syntax Tree Highlighting, and their combined use. section 4.1 discusses many possible avenues for future work on Keyboard-Card Menus. Two more, not mentioned there, would be 1) to explore different sets of keys (e.g. consider keyboard-cards that include the SHIFT keys), and 2) to experiment with providing more detailed information about the effects of invoking a command (e.g. pressing and holding a key for several seconds might be equivalent to hovering over an icon with the mouse instead of clicking it). Other areas to work on include:

- Providing alternative ways to traverse the syntax tree, e.g. if one runs out of siblings when moving the right, does moving to an uncle on the right feel seem natural?

- Optimizing the set of commands and their placement in the keyboard-card hierarchies.

- Working with other languages, e.g. 1st order logic, programming languages, or Coq's full "vernacular" language. Part of the rationale for developing Syntax Tree Highlighting was to make clear how to parse expressions in Coq that often involve deep nesting of subexpressions and non-standard or user-defined notations/precedences.

A fourth and final path would be to step back and **re-evaluate the design of traditional text editors**. One can think of the Proof Transitions as a non-traditional editor (the prototype being only available for read-only documents). Given larger amounts of screen space, keyboard-card menus and other new input techniques,

and programming interfaces for zoomable user interfaces, work in this direction seems appropriate.

CHAPTER 7

REFERENCES

[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] Frama-C. http://frama-c.com/.

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

[6] Mathematica. http://www.wolfram.com/mathematica/.

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

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

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

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

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

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

[13] ACM. Software system award. *Announcement at http://awards.acm.org/software_system*, 2013.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[28] Benjamin Berman and Juan Pablo Hourcade. Keyboard card menus: Faster learning of many fast commands. In *Proceedings of the XIV International Congress of Human-Computer Interaction (Interaction 2013), in the Spanish Congress of Informatics (CEDI)*, pages 105–112, 2013.

[29] Benjamin Berman and Juan Pablo Hourcade. Keyboard card menus: A new presentation of non-standard shortcuts. *Journal of Universal Computer Science, Special Issue on Trending Breakthroughs in Human-Computer Interaction*, 2014.

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

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

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

[33] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

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

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] Martin Eyl. The harmony information landscape interactive, three-dimensional navigation through an information space. *Master's thesis, Graz University of Technology, Austria*, 1995.

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

[50] 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.

[51] 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*.

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

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

[54] 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.

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

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

[57] 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.

[58] 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.

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

[60] 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.

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

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

[63] 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.

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

[65] Akrivi Katifori, Constantin Halatsis, George Lepouras, Costas Vassilakis, and Eugenia Giannopoulou. Ontology visualization methodsa survey. *ACM Computing Surveys (CSUR)*, 39(4):10, 2007.

[66] 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.

[67] 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.

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

[69] 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.

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

[71] Jonh Lamping and Ramana Rao. The hyperbolic browser: A focus+ context technique for visualizing large hierarchies. *Journal of Visual Languages & Computing*, 7(1):33–55, 1996.

[72] 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.

[73] 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.

[74] 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.

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

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

[77] 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.

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

[79] 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.

[80] 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.

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

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

[83] 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.

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

[85] 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.

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

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

[88] Quang Vinh Nguyen and Mao Lin Huang. A space-optimized tree visualization. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pages 85–92. IEEE, 2002.

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

[90] 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).

[91] 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.

[92] 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.

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

[94] Walky Rivadeneira and Benjamin B Bederson. A study of search result clustering interfaces: Comparing textual and zoomable user interfaces. *Studies*, 21:5, 2003.

[95] 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.

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

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

[98] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on graphics (TOG)*, 11(1):92–99, 1992.

[99] 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.

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

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

[102] 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.

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

[104] 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.

[105] Jarke J Van Wijk and Huub Van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Information Visualization, 1999.(Info Vis' 99) Proceedings. 1999 IEEE Symposium on*, pages 73–78. IEEE, 1999.

[106] 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¿.

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

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