DEVELOPMENT AND USER TESTING OF NEW USER INTERFACES FOR MATHEMATICS AND PROGRAMMING TOOLS, FOCUSING ON THE COQ PROOF ASSISTANT

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 with the *Coq* proof assistant[10][1]: I intend to show ways to make the difficult task of using Coq easier by improving the user interface. As well as solving serious usability problems for an important and powerful tool for creating machine-checked proofs, many of the techniques I am developing and testing are widely applicable to other forms of coding.

In chapter 2, I first give a description of Coq, including its significance, an example of theorem proving using the tool, a description of current user interfaces,

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

and 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 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 extension. Although this extension 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 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 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

[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.

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

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 [48]. 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 Coq's Significance

The example above is intended to give some sense of what interactive theorem proving with Coq is all about, and the complexities that novice users face, but it barely scratches the surface of Coq's full power and complexity. It also does little to suggest Coq's significance. Most of the applications accounting for this importance

can be divided into those relating (more directly) to computer science and those relating to mathematics.[6]

On the computer science side, Coq has an important place in research on ensuring that computer software and hardware is free of bugs. Given the increasing use of computers in areas where bugs (including security vulnerabilities) can have serious negative consequences (aviation, banking, heath care, etc.), such research is becoming increasingly important. Given, also, that exhaustive testing of the systems involved in these areas is generally infeasible, researchers have recognized the need to actually prove the correctness of these systems (i.e. that the systems conform to their specifications 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. [44, 34]) and can determine the satisfiability of large numbers of large formulas, keeping humans involved in the theorem proving process allows the search for a proof to be tailored to the particular theorem at hand, and therefore allows a wider range, in a sense, of theorems to be proved. Furthermore, contrary to what might have been suggested by the step-by-step detail of the example above, many subproblems can be solved automatically by Coq and other interactive theorem provers, and work is being done to send subproblems of interactive theorem provers to automatic tools [25] in order to combine the best of both worlds.

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

Notable computer science-related achievements, some in industrial contexts, for Coq and other interactive theorem provers include verification of the seL4 microkernel [52] in Isabelle[5], the CompCert verified compiler[55] for Clight (a large subset of the C programming language) in Coq, Java Card EAL7 certification[41] using Coq, and, at higher levels of abstraction, verification of the type safety of a semantics for Standard ML [54] using Twelf[11] and use of the CertiCrypt framework [2] built on top of of Coq to verify cryptographic protocols (e.g. [18]).[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 [43]. Other interactive theorem provers are also having success in general mathematics. For instance, Matita [6], which is closely related to Coq, was used in a proof of Lebesgue's dominated convergence theorem [31]. There are in fact efforts to create libraries of formalized, machine-checked mathematics, the largest of which is the Mizar Mathematical Library [42]. 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 [28].

The potential for transparency also helps make interactive theorem provers, like Coq, a potentially useful tool in mathematics, logic, and computer science ed-

[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.

ucation. 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*[67], 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* [57], *Logic Tutor* [56], *Fitch* (software accompanying the textbook *Language, Proof, and Logic* [19]), and *ProofMood*[8].

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* [30], 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

[8]Basically, this is proof by providing a procedure to get a proof. Coq allows one to prove

awarded both the 2013 ACM SIGPLAN Programming Languages Software Award [73] and the 2013 ACM Software System Award [12].

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[61] lists almost 200 in its tactics index), and Coq allows new tactics to be defined. Other challenges, for both novice and expert users, will be discussed below, but the lack of support for users trying to understand tactic effects is, by itself, probably sufficient justification for the development of new user interfaces.

The two major user interfaces for Coq are currently *Proof General*[7, 16] and *CoqIDE* (which is available from the Coq website[10], and is bundled with Coq). Interacting with Coq using one 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 $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow$

---

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.

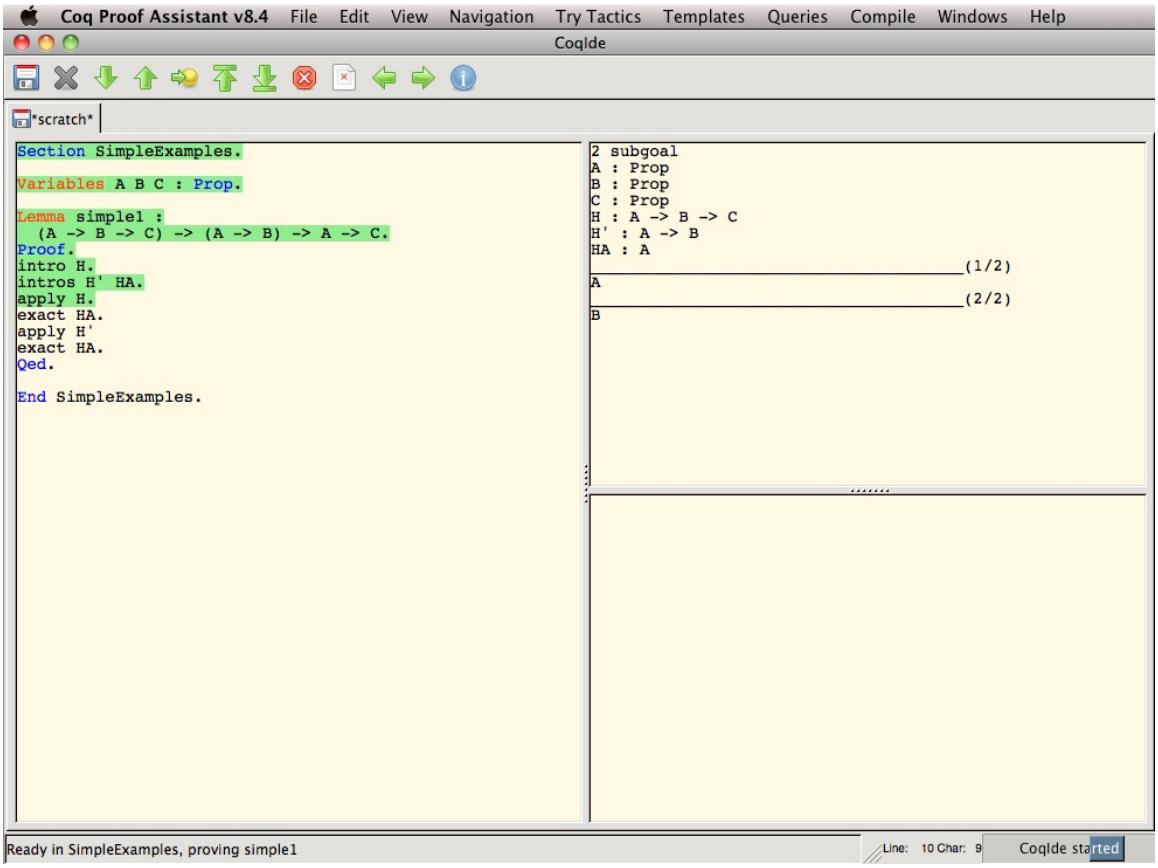

Figure 2.1. CoqIde, displaying the result of entering the tactic "`apply H`" in the top-right panel within the proof of `(A -> B -> C) -> (A -> B) -> A -> C`.

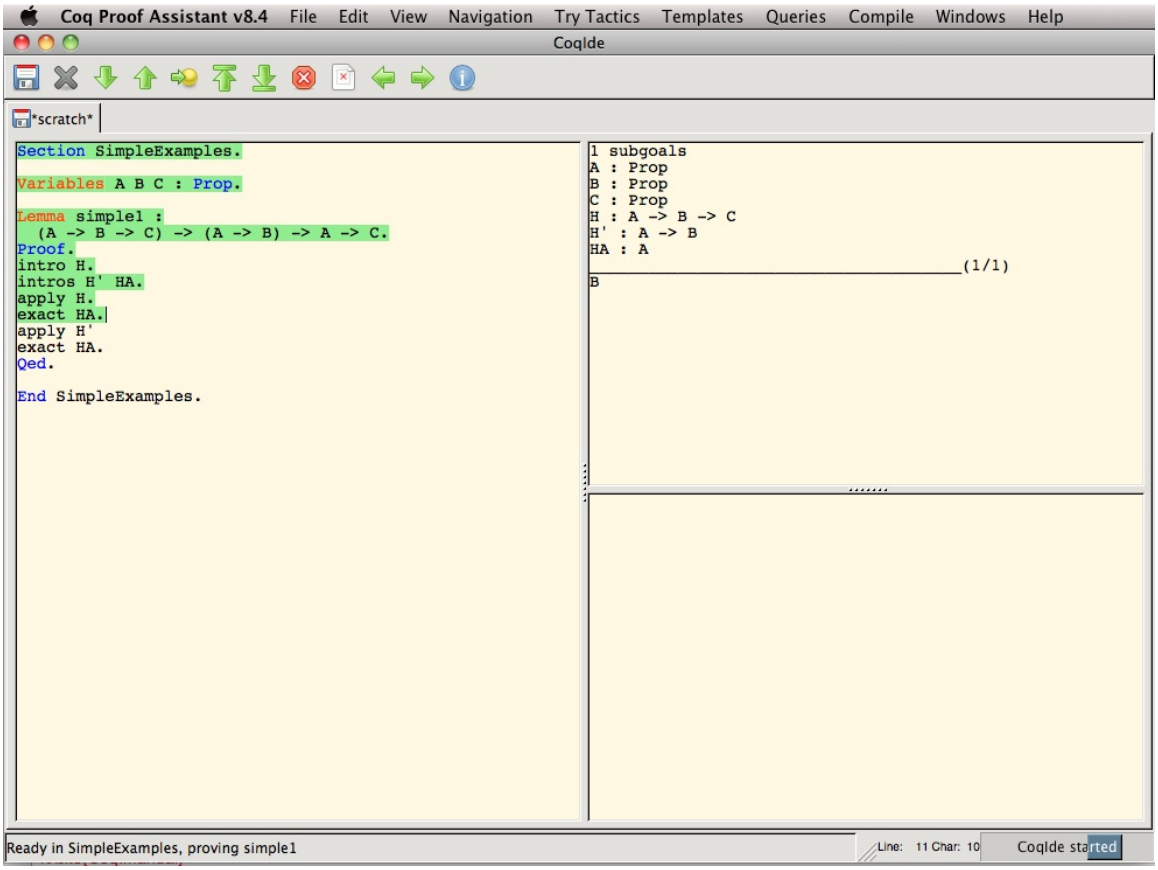

Figure 2.2. CoqIde after moving the end of the evaluated portion of the script forward by one sentence from the state shown in Figure 2.1.

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, *Proof Web*[9] does make a serious attempt. ProofWeb, for the most part, is a web-based version of CoqIde. However, it has a major improvement, shown in the bottom right of Figure 2.3: a visualization of the partially completed proof tree.

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

[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.

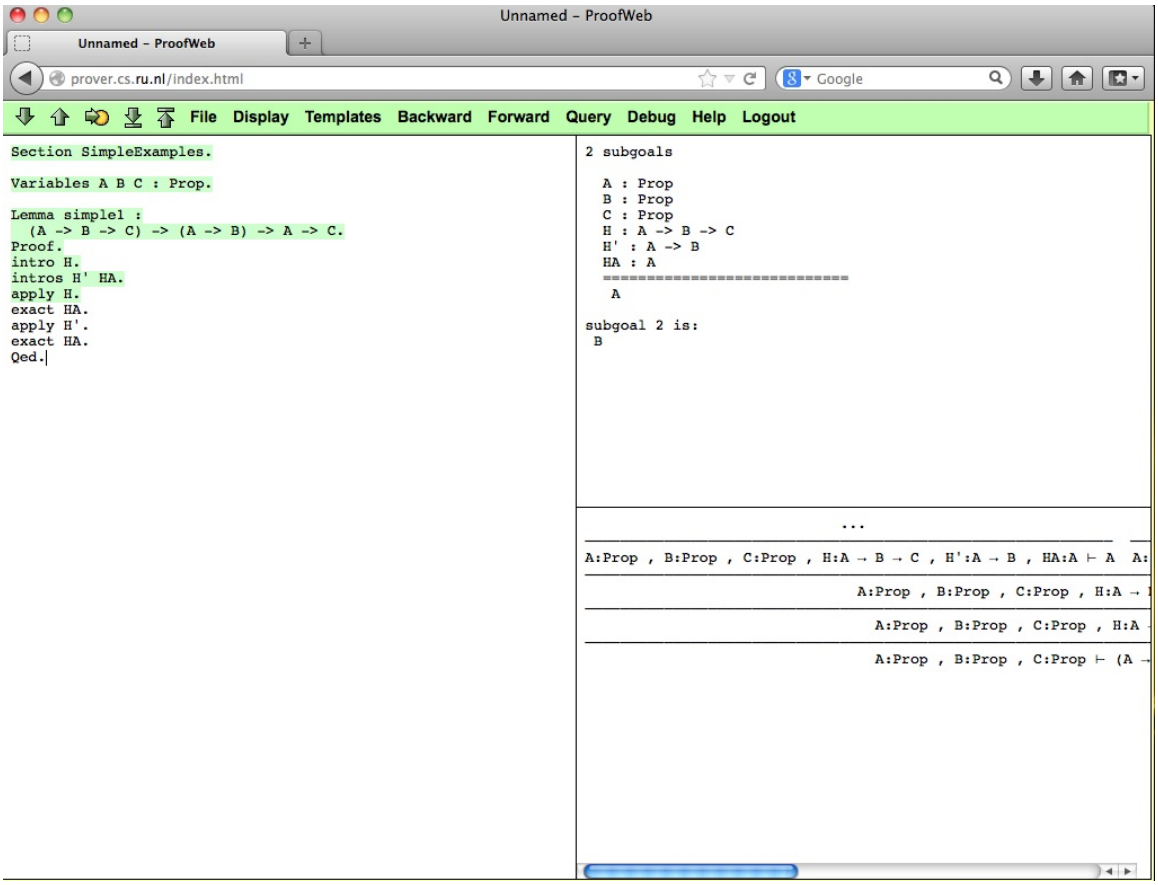

Figure 2.3. ProofWeb, with a partial proof tree displayed in the bottom right.

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).

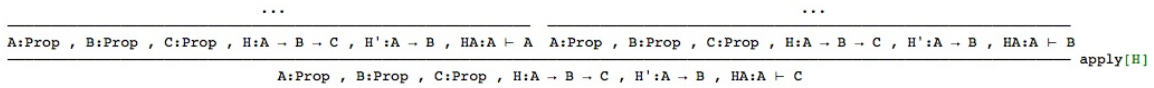

Figure 2.4. The portion of ProofWeb's tree visualization corresponding to the tactic "`apply H`".

[13]This is the Gestalt law of "good continuation; see, for instance, [40].

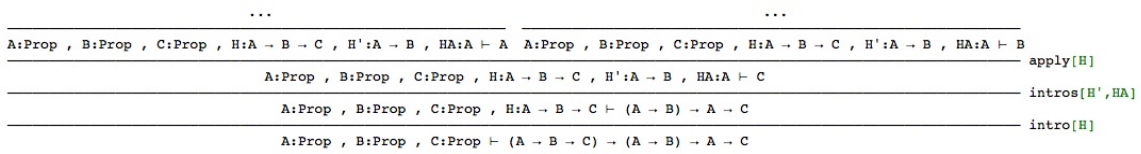

Figure 2.5. The partially completed tree from Figure 2.3, fully displayed.

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/ baber-man/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).

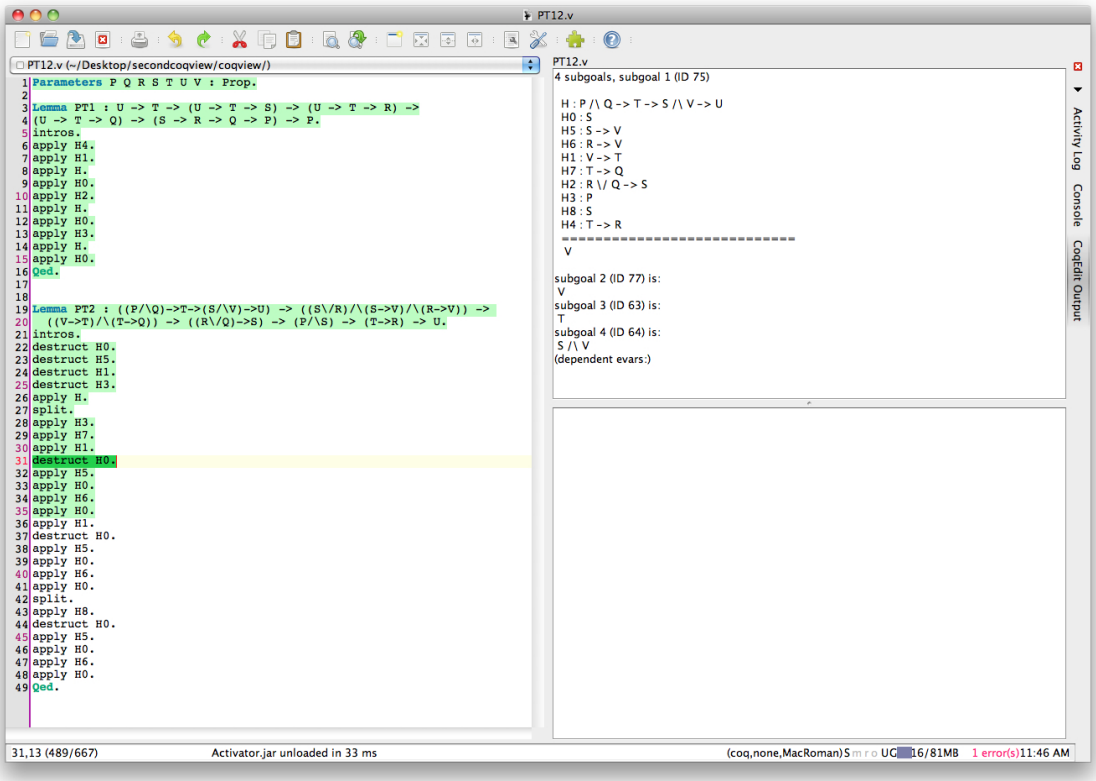

Figure 3.1. The basic CoqEdit user interface

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.

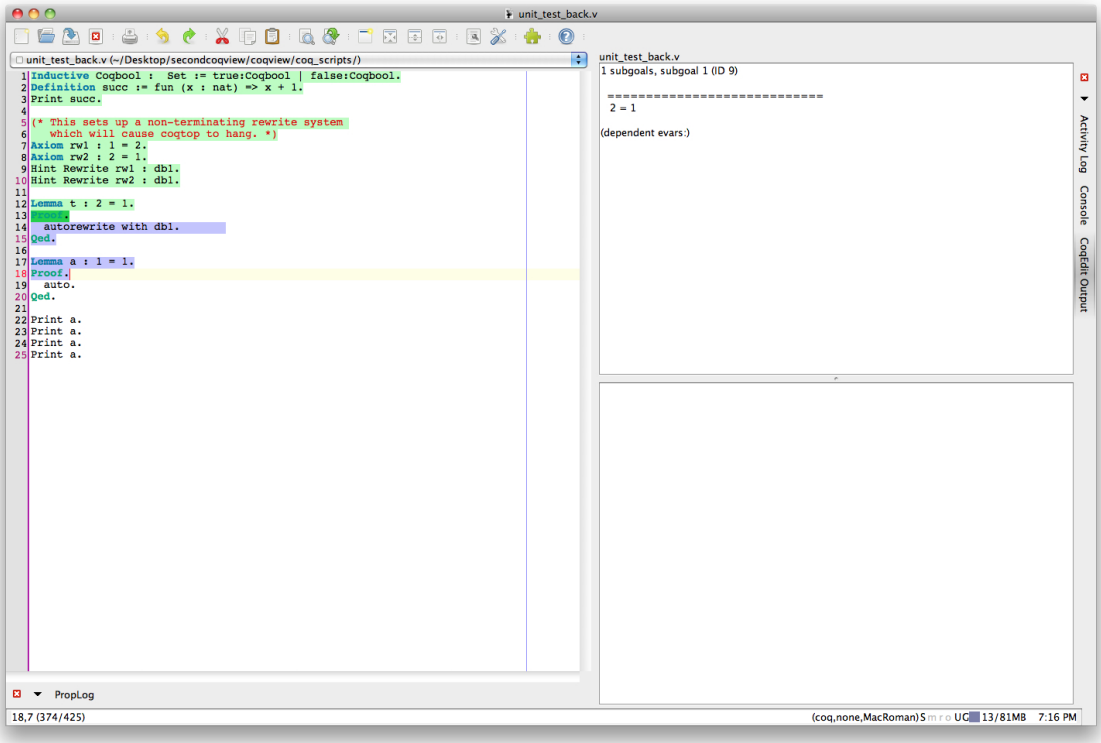

Figure 3.2. The basic CoqEdit user interface, when the first sentence highlighted with purple is being evaluated and the subsequent purple-highlighted sentences are queued for processing. Thanks go to Harley Eades for providing this example of a non-terminating tactic.

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]

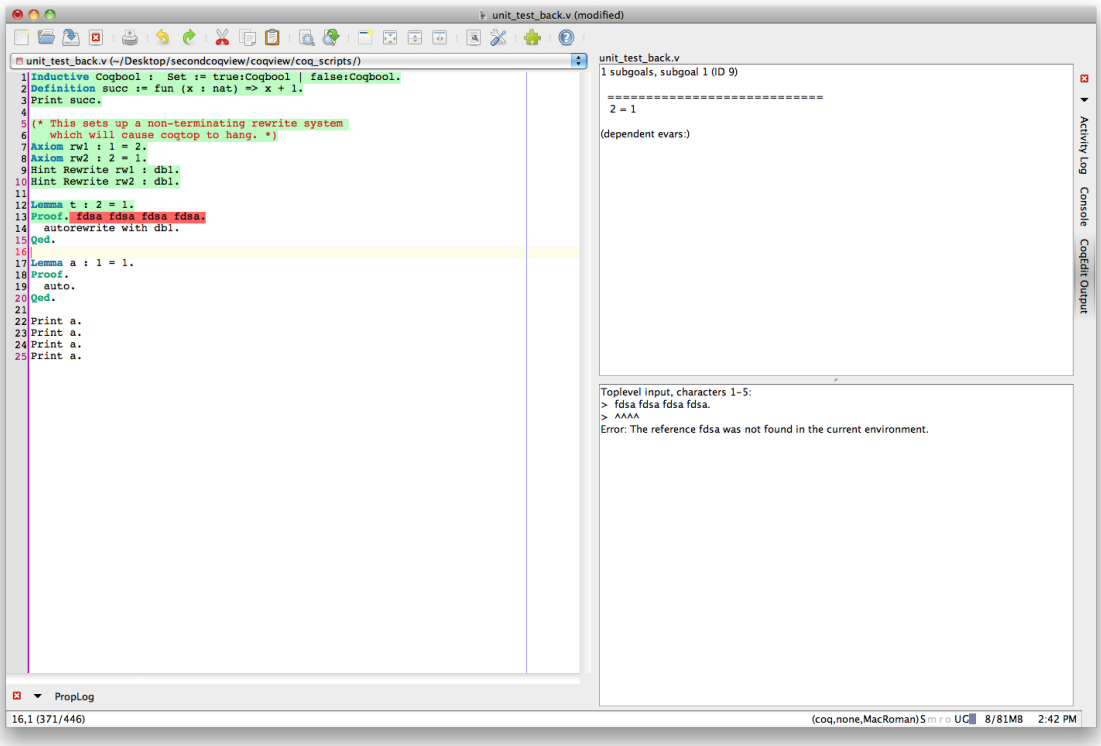

Figure 3.3. The basic CoqEdit user interface, with an error-producing sentence highlighted in red.

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.

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.

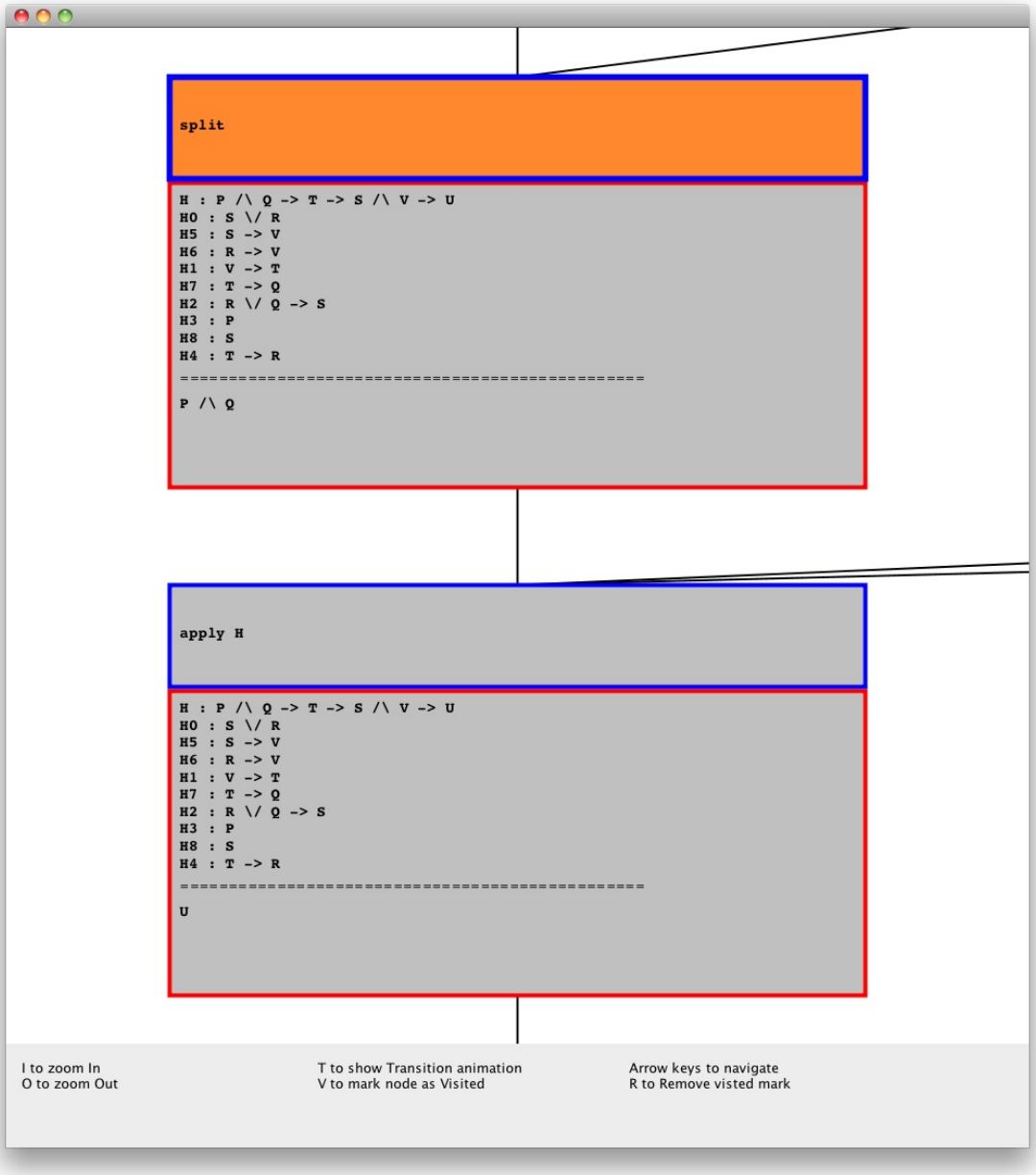

Figure 3.4

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).

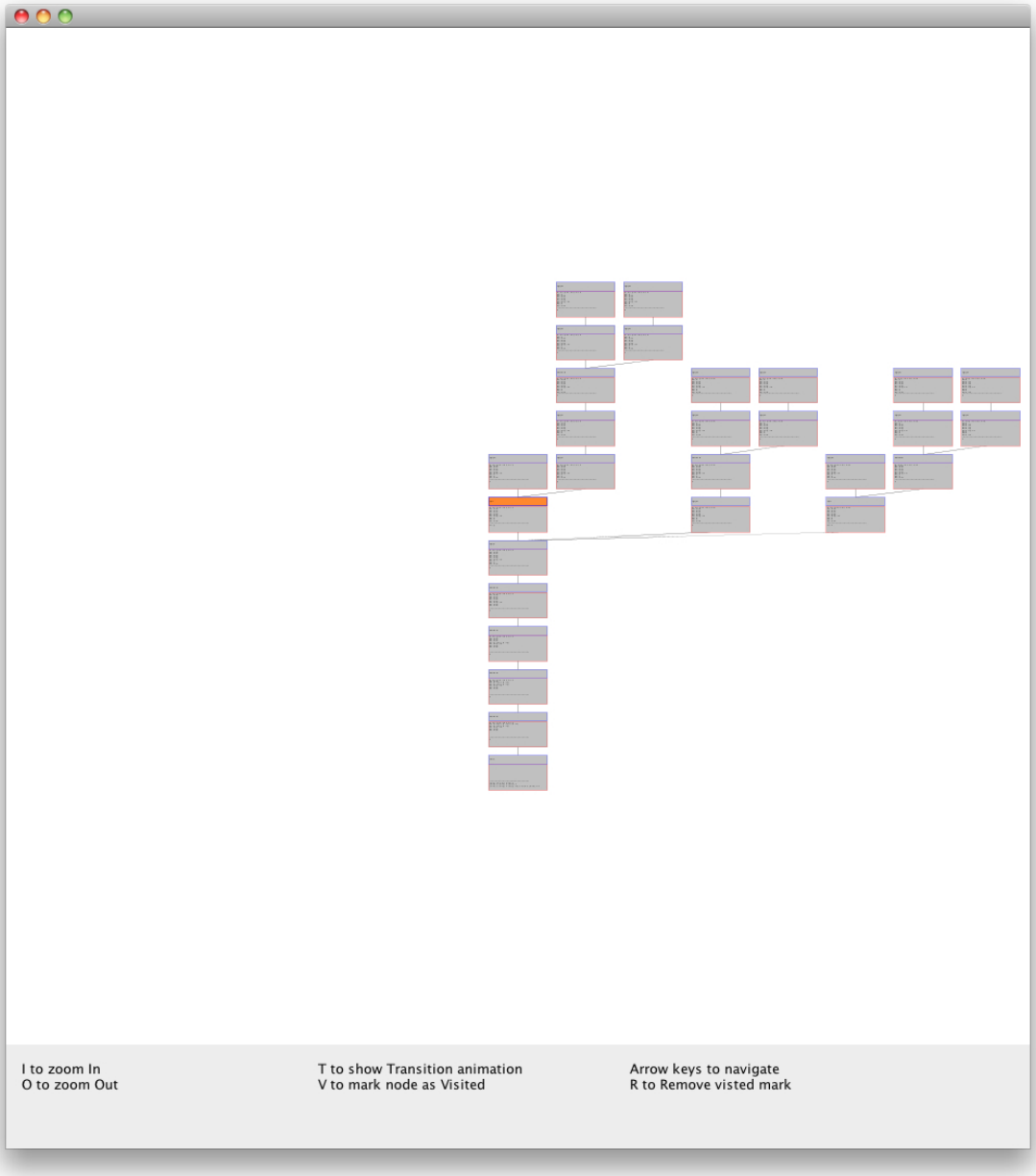
I to zoom In
O to zoom Out
T to show Transition animation
V to mark node as Visited
Arrow keys to navigate
R to Remove visted mark

Figure 3.5

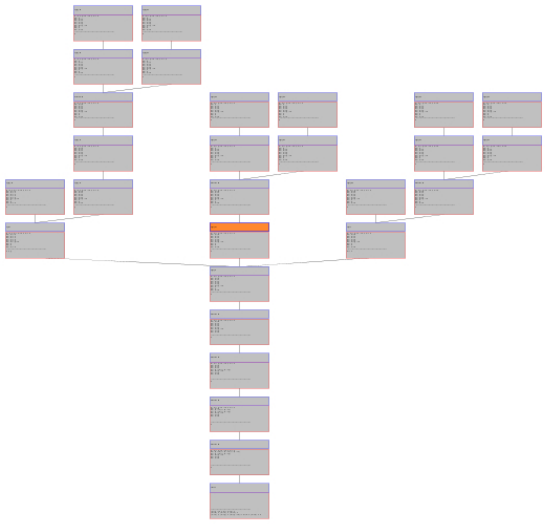

Figure 3.6

Figure 3.7

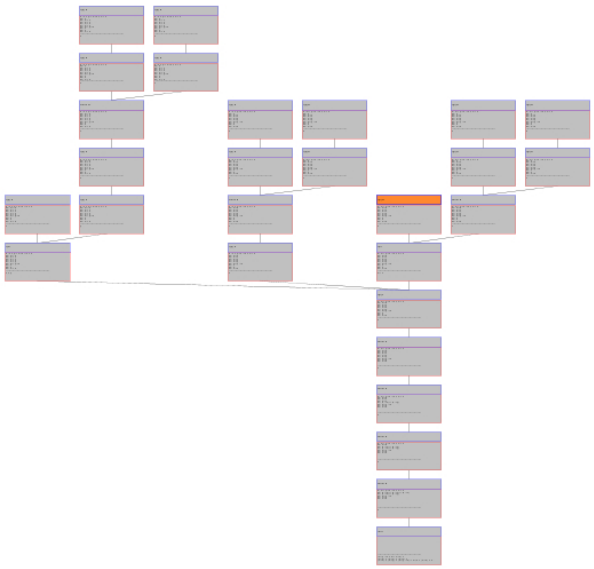

Figure 3.8

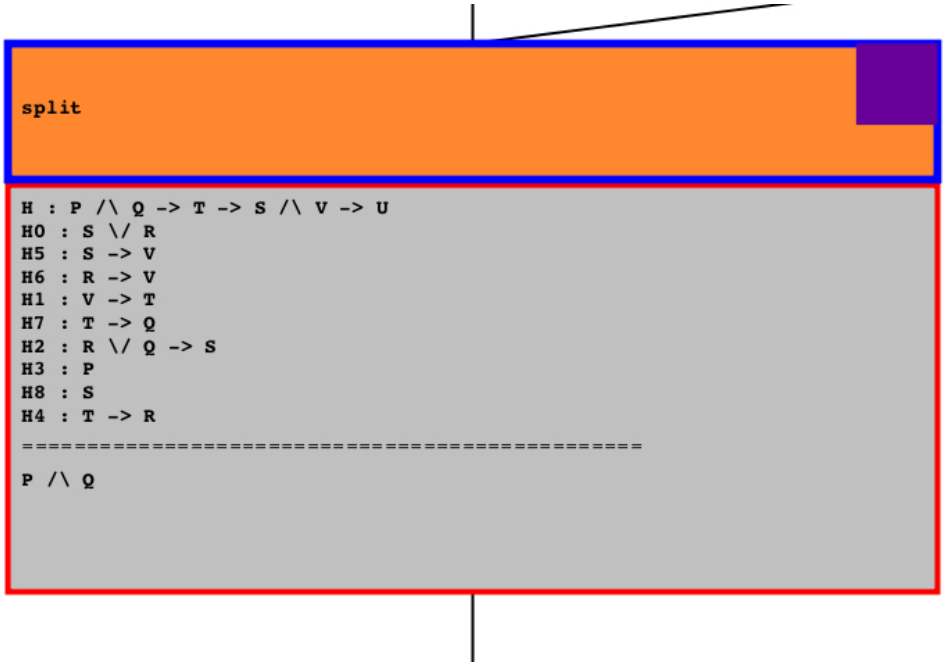

Figure 3.9

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.

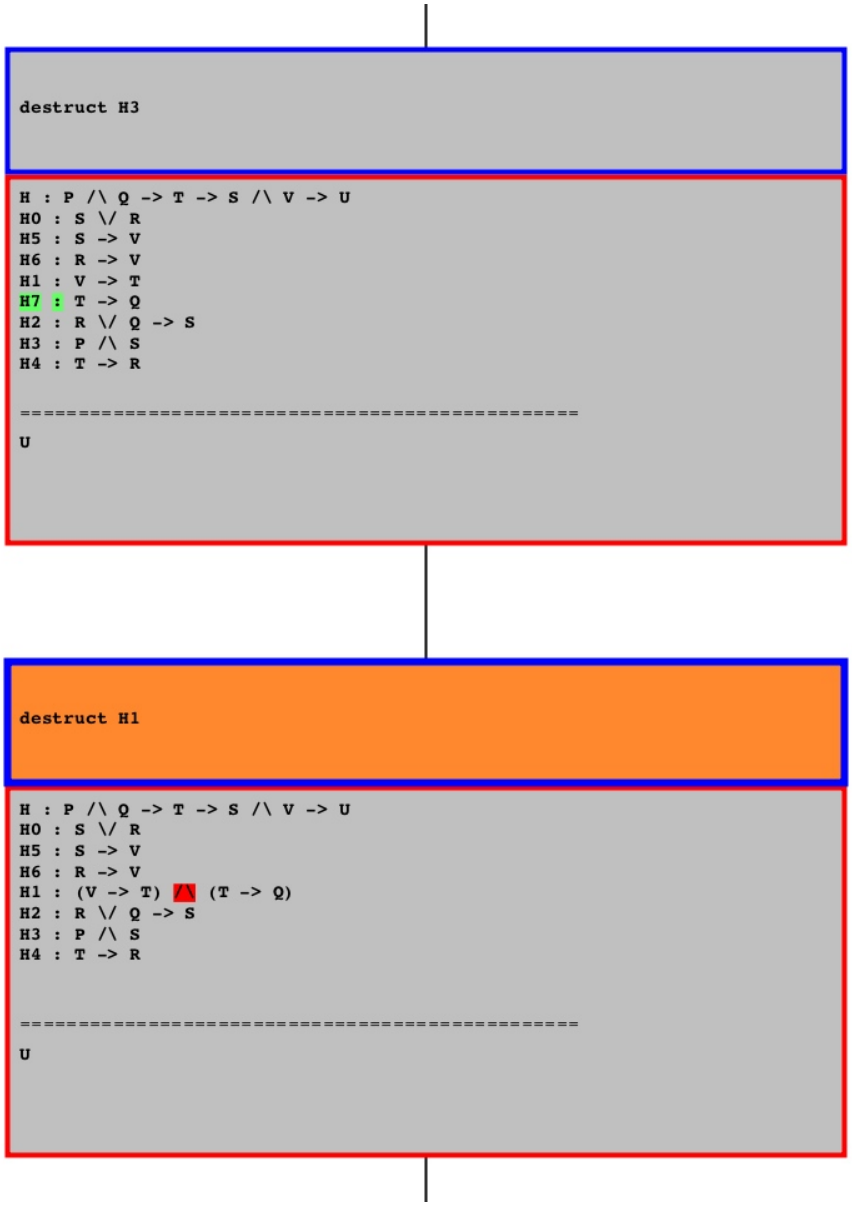
destruct H3
H : P /\ Q -> T -> S /\ V -> U
H0 : S \/ R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P /\ S
H4 : T -> R
U
destruct H1
H : P /\ Q -> T -> S /\ V -> U
H0 : S \/ R
H5 : S -> V
H6 : R -> V
H1 : (V -> T) /\ (T -> Q)
H2 : R \/ Q -> S
H3 : P /\ S
H4 : T -> R
U

Figure 3.10

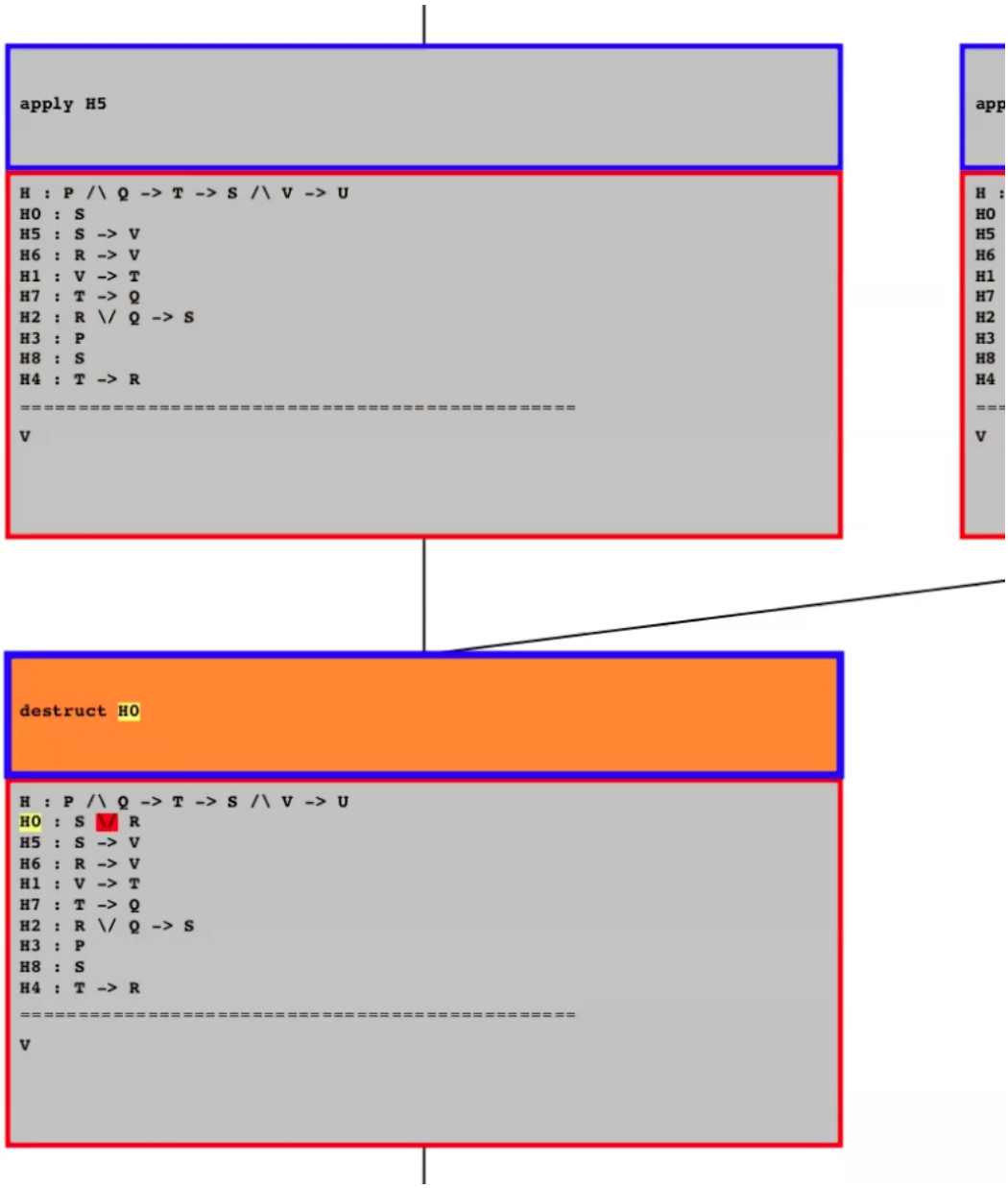
apply H5
H : P /\ Q -> T -> S /\ V -> U
H0 : S
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V
destruct H0
H : P /\ Q -> T -> S /\ V -> U
H0 : S \/ R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V

Figure 3.11

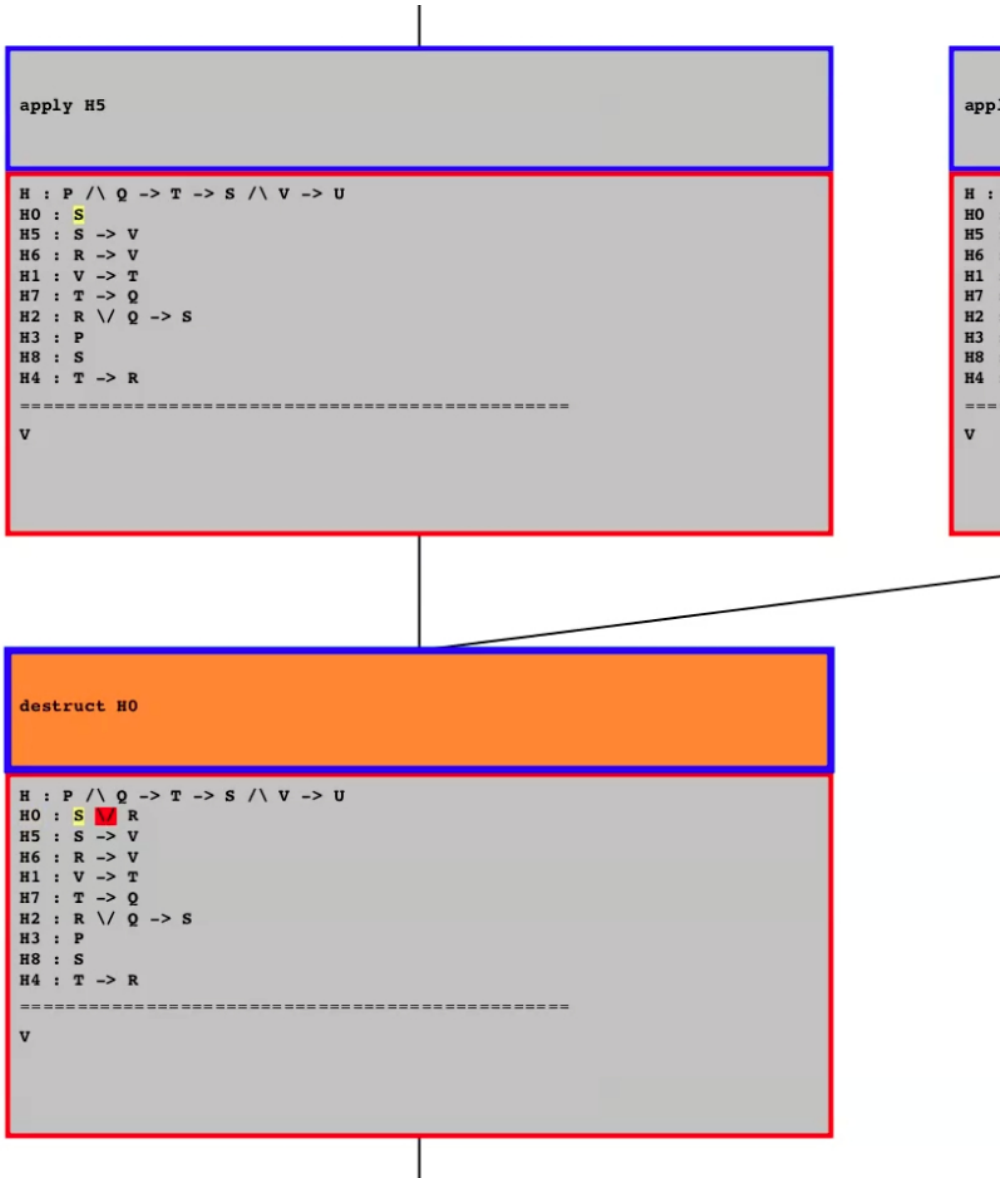
apply H5
H : P /\ Q -> T -> S /\ V -> U
H0 : S
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V
destruct H0
H : P /\ Q -> T -> S /\ V -> U
H0 : S \/ R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V

Figure 3.12

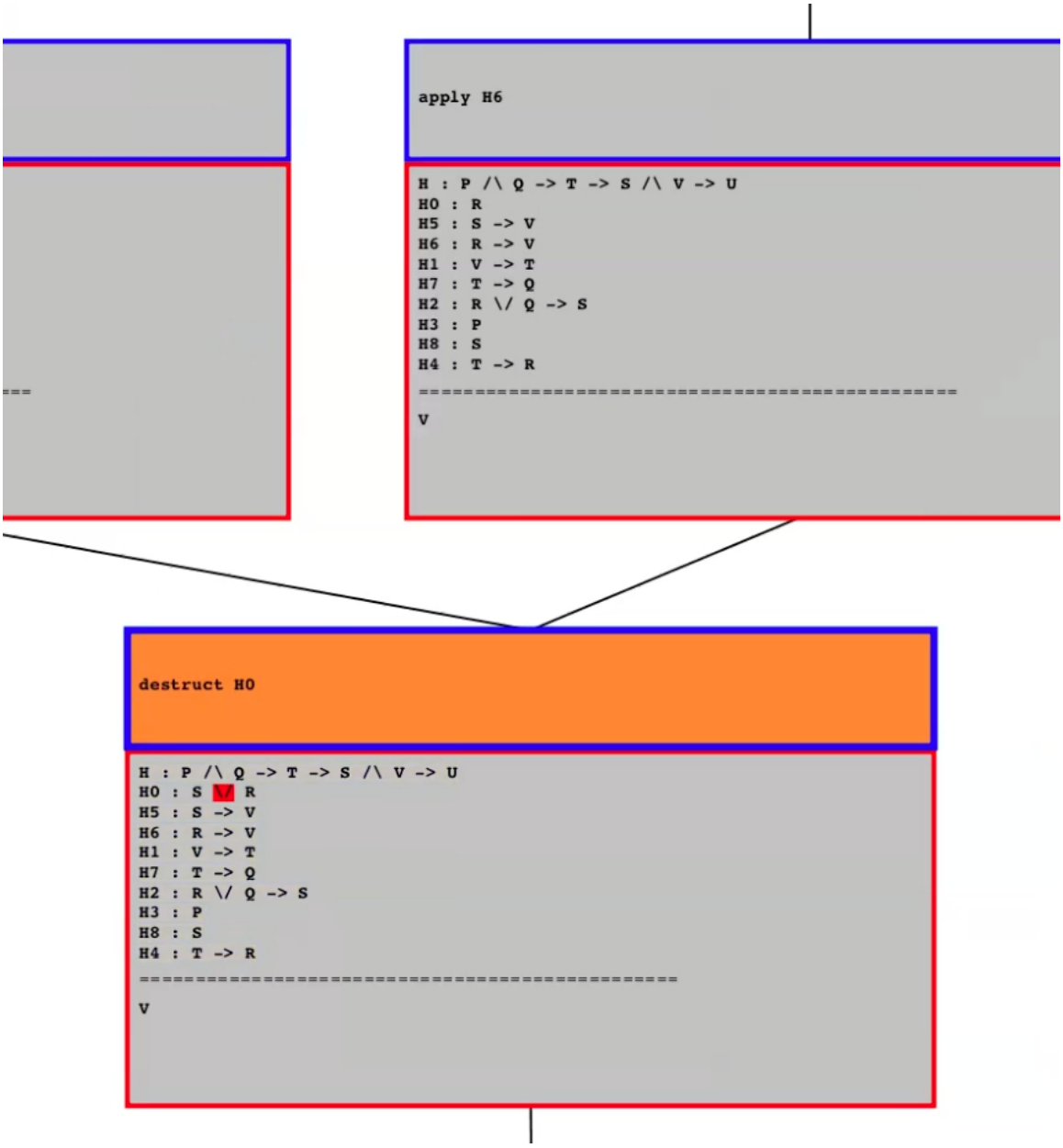
apply H6
H : P /\ Q -> T -> S /\ V -> U
H0 : R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V
destruct H0
H : P /\ Q -> T -> S /\ V -> U
H0 : S \/ R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V

Figure 3.13

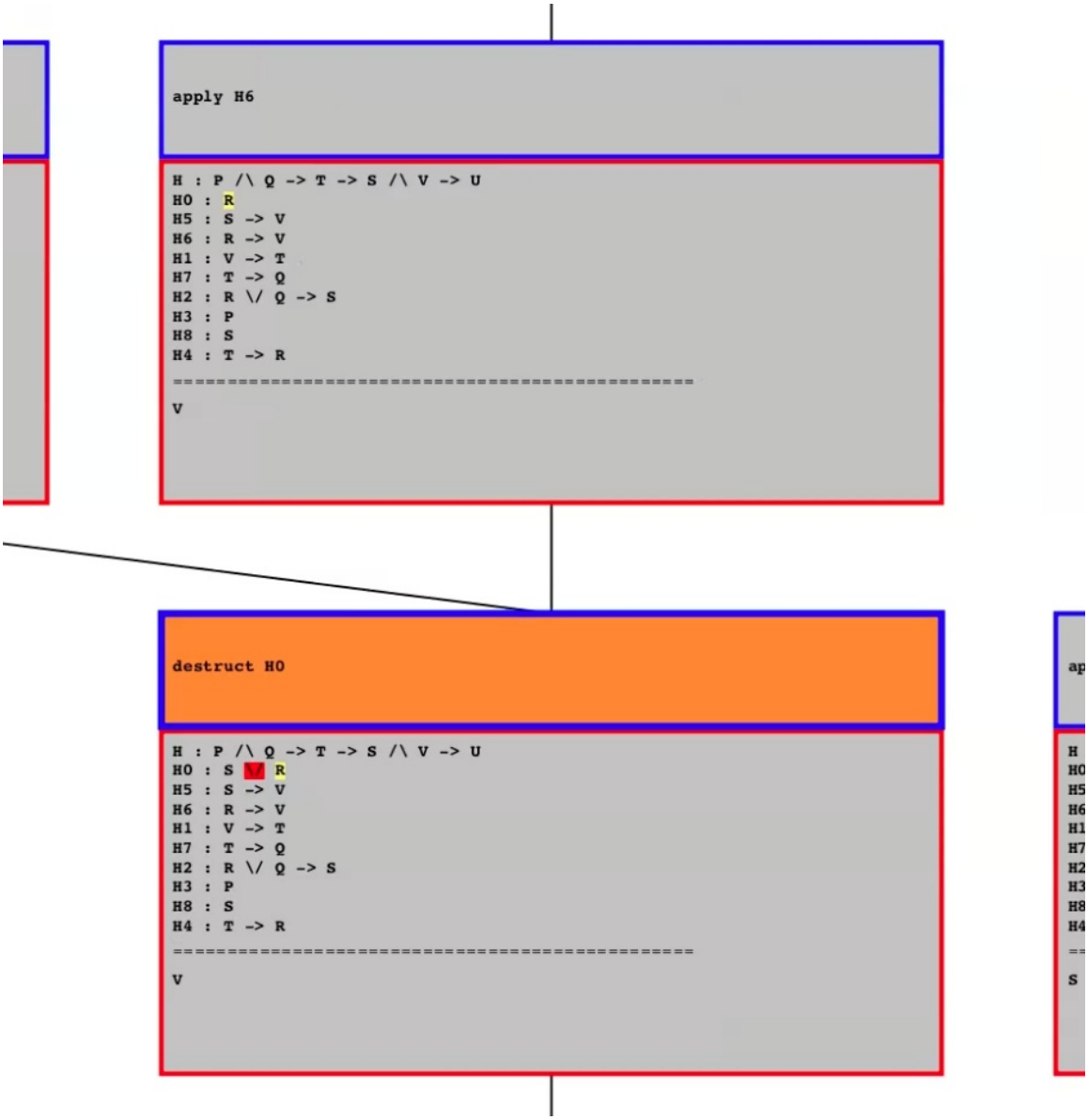
apply H6
H : P /\ Q -> T -> S /\ V -> U
H0 : R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V
destruct H0
H : P /\ Q -> T -> S /\ V -> U
H0 : S \/ R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V

Figure 3.14

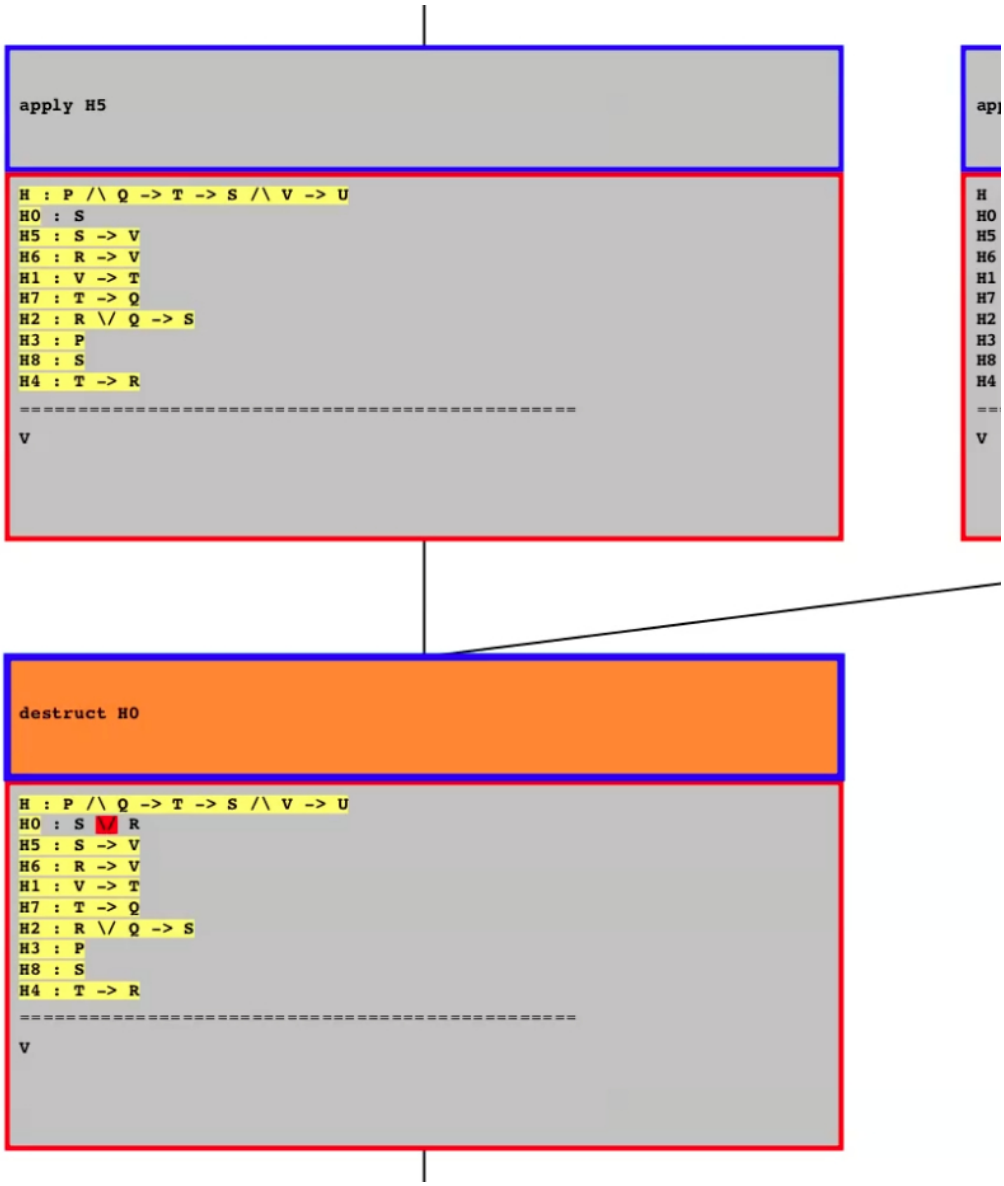
apply H5
H : P /\ Q -> T -> S /\ V -> U
H0 : S
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V
destruct H0
H : P /\ Q -> T -> S /\ V -> U
H0 : S \/ R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V

Figure 3.15


apply H6
H : P /\ Q -> T -> S /\ V -> U
H0 : R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V
destruct H0
H : P /\ Q -> T -> S /\ V -> U
H0 : S \/ R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V

Figure 3.16

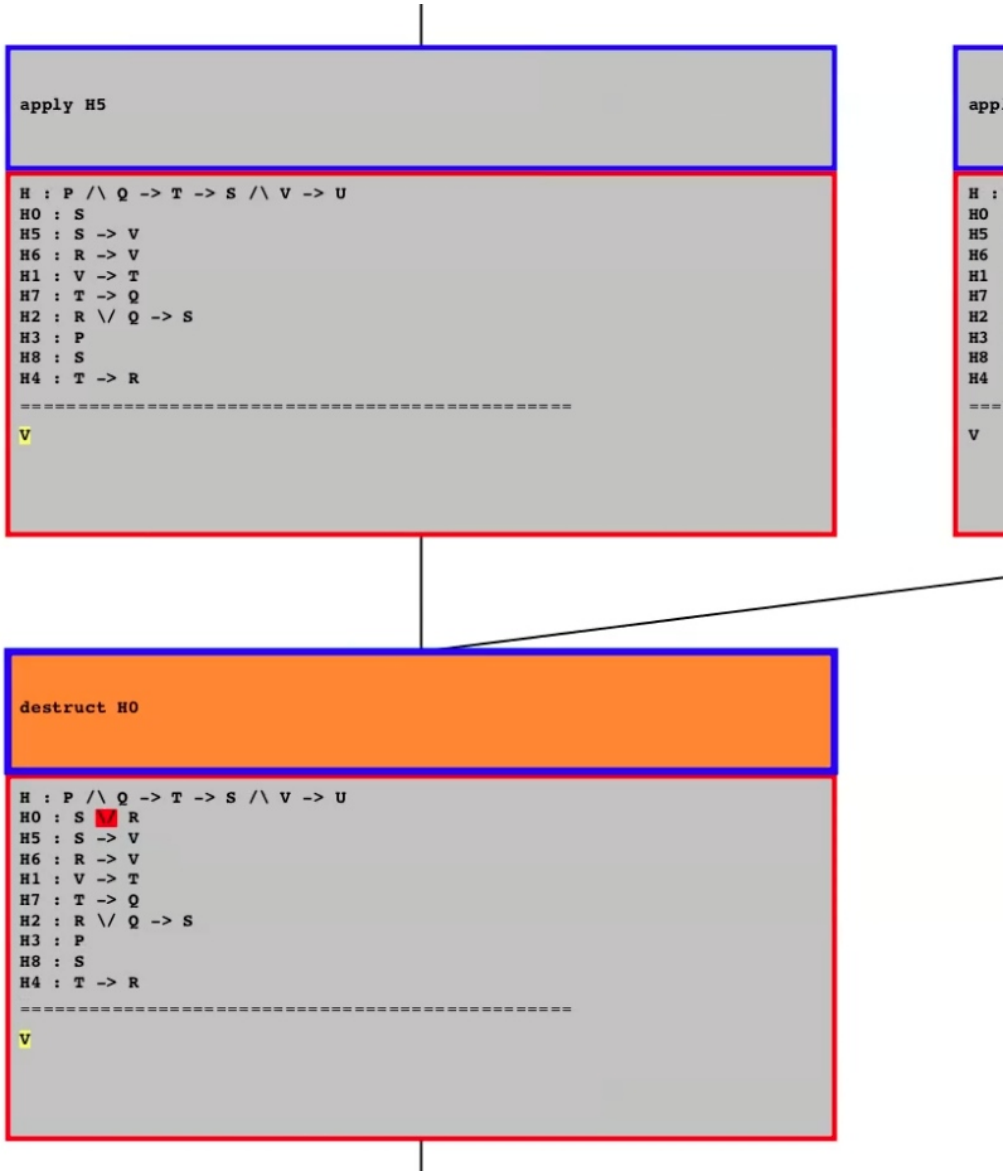
apply H5
H : P /\ Q -> T -> S /\ V -> U
H0 : S
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V
destruct H0
H : P /\ Q -> T -> S /\ V -> U
H0 : S \/ R
H5 : S -> V
H6 : R -> V
H1 : V -> T
H7 : T -> Q
H2 : R \/ Q -> S
H3 : P
H8 : S
H4 : T -> R
V

Figure 3.17

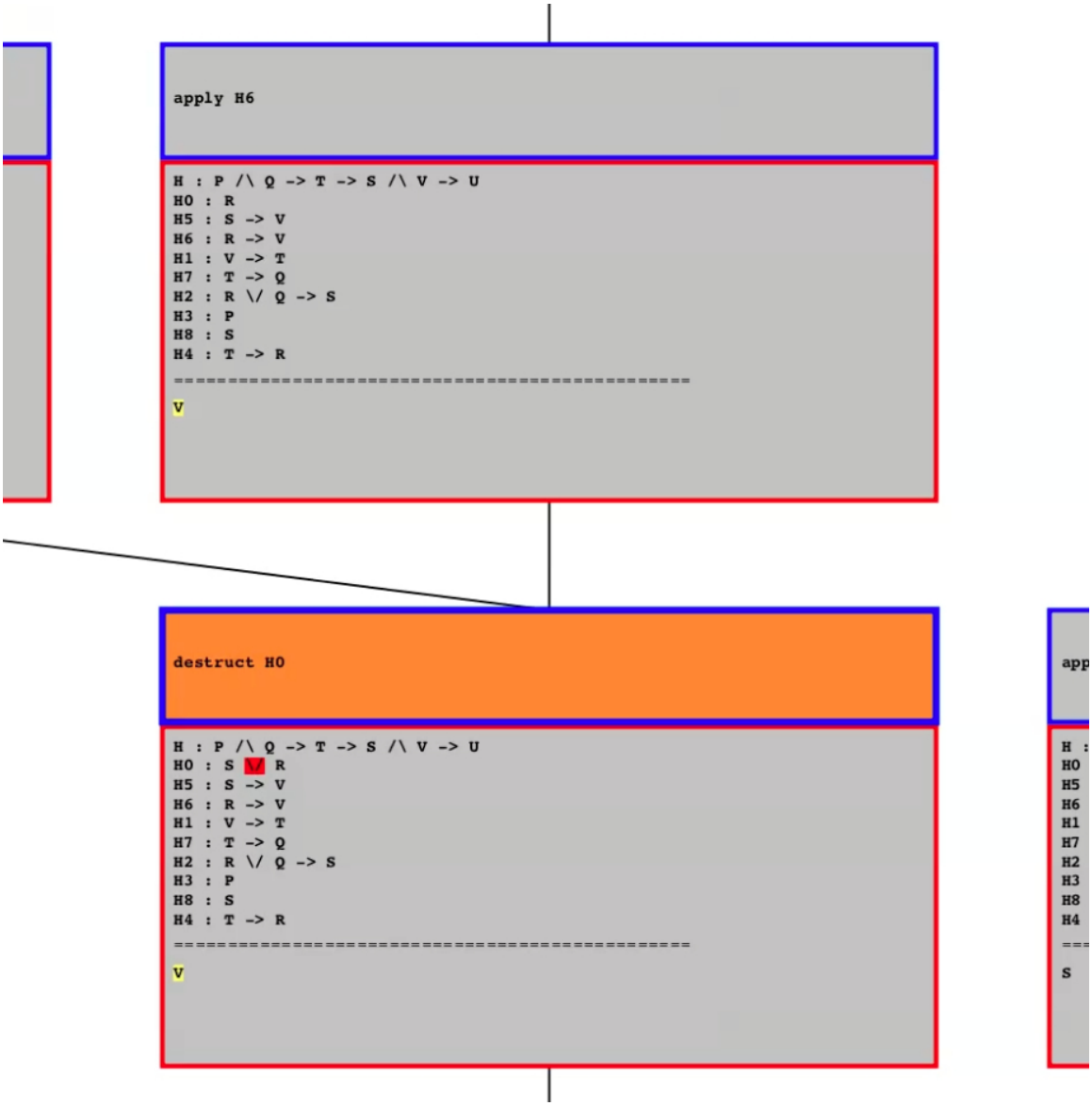

Figure 3.18

3.1.3 Proof Previews

3.2 Experiment Design, Results, Analysis, and Conclusions

CHAPTER 4

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

CHAPTER 5

RELATED WORK

The need for effective theorem prover user interfaces has been recognized for over 20 years (e.g. [75]) 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 [64, 68], easier-to-read declarative languages for proof scripts [32], using wikis for creating proof repositories [33], and automating the process of using libraries [15].

One approach taken to improve theorem prover user interfaces has been "Proof by Pointing"[22], an algorithm to build a proof tree by pointing to portions of a goal. This was implemented in the *CtCoq* interface [21], again, later, in the Java-based Pcoq interface [14] and also in the *Jape* system [27]. 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 [26].

An example of a recent project is *Panoptes* [39], 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 [17]. 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 [74]; an "Inter-

active Derivation Viewer" for visualizing derivations written in the TPTP language [76]; the LOUI interface for the OMEGA proof assistant, featuring graphical visualization, term browsing, and natural language proof presentation [72]; and the Tecton system, featuring tree visualization combined with hypertext navigation of nodes [50]. Theoretical and methodological work has also been done and can be seen in [36, 23, 59, 79, 42], and [58]. Work aimed at making ITPs more suitable for novices and educational settings includes [77, 62, 64, 24, 35, 45, 70] and [66].

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 [63] 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.

In addition to IDEs, we may consider software visualization tools: [20] 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 [29]. Similar to [17], 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* [37], for instance, miniaturizes lines of code with lines of color-coded pixels, and *sv3D* [60] extends this idea using three-dimensional arrays of blocks where information about lines of code is encoded in the height and color of the blocks. Other techniques potentially useful for interactive theorem prover user interfaces include using animation and color gradients to show the direction of relationships between software components [13, 46] and using texture

and 3D object primitives called "geons" to encode additional information [47, 49].

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 [46]: 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 [71]). These common layouts have been extended in various ways. For instance, [65] describes a space-optimized version of balloon trees and [78] describes the the addition of gradients to the rectangles in treemap visualizations, making the structure of the tree easier to see.

[51] 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 [46] 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* [69] which allows users to click and expand nested circles. An example of the focus+context or distortion category is the *Hyperbolic Browser* [53] 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* [38].

CHAPTER 6

SUMMARY AND CONCLUSIONS

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

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

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

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

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

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

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

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

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

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

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

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

[24] 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.

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

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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*.

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

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

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

[45] 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.

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

[47] 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.

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

[49] 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.

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

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

[52] 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.

[53] 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.

[54] 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.

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

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

[57] 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.

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

[59] 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.

[60] 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.

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

[62] 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.

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

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

[65] 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.

[66] 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).

[67] 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.

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

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

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

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

[72] 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.

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

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

[75] 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.

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

[77] 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.

[78] 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.

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