A PROOF-THEORETIC APPROACH TO MATHEMATICAL KNOWLEDGE MANAGEMENT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Kamal Aboul-Hosn

January 2007

A PROOF-THEORETIC APPROACH TO MATHEMATICAL KNOWLEDGE MANAGEMENT

Kamal Aboul-Hosn, Ph.D.

Cornell University 2007

Mathematics is an area of research that is forever growing. Definitions, theorems, axioms, and proofs are integral part of every area of mathematics. The relationships between these elements bring to light the elegant abstractions that bind even the most intricate aspects of math and science.

As the body of mathematics becomes larger and its relationships become richer, the organization of mathematical knowledge becomes more important and more difficult. This emerging area of research is referred to as *mathematical knowledge management* (MKM). The primary issues facing MKM were summarized by Buchberger, one of the organizers of the first Mathematical Knowledge Management Workshop [20].

- How do we retrieve mathematical knowledge from existing and future sources?
- How do we build future mathematical knowledge bases?
- How do we make the mathematical knowledge bases available to mathematicians?

These questions have become particularly relevant with the growing power of and interest in *automated theorem proving*, using computer programs to prove mathematical theorems. Automated theorem provers have been used to formalize

theorems and proofs from all areas of mathematics, resulting in large libraries of mathematical knowledge. However, these libraries are usually implemented at the system level, meaning they are not defined with the same level of formalism as the proofs themselves, which rely on a strong underlying proof theory with rules for their creation.

In this thesis, we develop a proof-theoretic approach to formalizing the relationships between proofs in a library in the same way the steps of a proof are formalized in automated theorem provers. The library defined in this formal way exhibits five desirable properties: independence, structure, an underlying formalism, adaptability, and presentability. The ultimate goal of mathematical knowledge management is to make the vast libraries of mathematical information available to people at all skill levels. The proof-theoretic approach in this thesis provides a strong formal foundation for realizing that goal.

BIOGRAPHICAL SKETCH

Kamal Aboul-Hosn, son of Sydney and Hussein Aboul-Hosn, grew up in Bellefonte, Pennsylvania. With an interest in computers, he entered the Pennsylvania State University in August 1998. While there, he worked on several projects, including his honors thesis on programming with private state, under the guidance of John Hannan, and a parser generator for λProlog, under Dale Miller. He also served as a Technology Learning Assistant, teaching more than twenty faculty members how to effectively use technology in their classrooms. Kamal was graduated from Penn State with an honors B.S. in Computer Science and minor in Mathematics in December 2001.

In August 2002, Kamal entered the Ph.D. program in Computer Science at Cornell University. What started as a final project for a class turned into his thesis work on the formal representation of mathematical knowledge, under the guidance of Dexter Kozen. He has also worked on program verification using Kleene algebra with tests. As a part of the Graduate Student School Outreach Project, Kamal taught an eight-week mini-course on artificial intelligence to local high school students. Kamal was graduated from Cornell University with a Ph.D. in Computer Science and a minor in Economics in January 2007.

Kamal is an avid drummer and photographer. He is also the developer of *Radar In Motion*, an animated weather map widget for Apple's Mac OS X *Dashboard*.

ACKNOWLEDGEMENTS

> "There are things we don't say often enough. Things like what we mean to one another. All of you mean a lot to me. I just want you to know that." –Bill Adama

This thesis would not have been possible without the help of many people. First and foremost, I have to thank my family. I am who I am today because of the love and guidance of my parents, Hussein and Sydney Aboul-Hosn. My sister, Hannah, who is one of the greatest people I know, has been there through everything. I am so proud of you. I must also mention my grandparents, Caroline and Orlen Rice, the latter of whom instilled in me a love of science and engineering at a very young age. You are greatly missed, grandypa. And finally, Sittee, with whom I share a deep bond that words will not do justice. I love all of you.

I cannot possibly convey the gratitude I feel toward Dexter Kozen, my advisor, colleague, and friend. He has been a constant source of inspiration academically, musically, and personally. Many thanks for helping so many of my dreams come true. Vicky Weissman once astutely noted that "for an advisor, you have to pick someone you can see yourself becoming in ten years." I'm very comfortable with the idea of ending up like Dexter.

I also want to thank those who have made my years here at Cornell University so memorable. I have found many lifelong friends in my four and a half years here. I have been extremely fortunate to know people such as Milind Kulkarni and Ganesh Ramanarayanan, with whom I will always remember skipping stones at Stewart Park my first week in Ithaca–my first realization that life here was going to be all right. Siggi Cherem and Chethan Pandarinath have provided many laughs and memorable moments. I look forward to seeing all of you again, even

as our paths now diverge. I can't forget those with whom I played music in my time here, including Dexter, Joel Baines, and Steve Chong. To Becky Stewart, Stephanie Meik, and Kelly Patwell, thank you for all of the guidance, assistance, and fun times. I must also mention Polly Israni, Nikita Kuznetsov, and Terese Damhøj Andersen.

There are several people I've known from earlier in my life I'd like to thank. No one can have better friends than BNG Spicer and Lee Armstrong, who are like brothers to me, and my comrade, Justin Miller, whom I have known for almost 20 years. I am also fortunate to have worked with John Hannan and Dale Miller in my time at Penn State. Several others have also had a deep and meaningful impact on my life, including Matt Weirauch, Sean Fox, and Princess Laura Murphy.

My sincerest thank you to all of you who have supported me, helped me, challenged me, excited me, who have made my life what it is today.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Mathematics is a field that is important in our day-to-day lives. From a very young age, our children are taught the fundamentals of arithmetic. As they progress through middle school and high school, students learn algebra, geometry, trigonometry, and even calculus. In college, the mathematical knowledge we impart to these students becomes more specialized. Economics students learn about derivatives and their use in reasoning about changes in markets. Future physicists use calculus to model the properties of matter and energy.

Those who continue on to advanced courses and graduate degrees learn of the beautiful abstractions that provide a common basis for much of the mathematics they knew most of their lives. It is at this point that they truly understand the intricate hierarchy that binds the entire field of mathematics and all its applications, such as the one see in Figure 1.1. Within each area, the hierarchy gets more specific, branching into many subtopics. For example, the area of differential geometry breaks down further into the geometry of curves, the geometry of surfaces, Riemannian geometry, and several others.

Each part of this hierarchy has its own set of definitions, theorems, axioms, and proofs that are fundamental to that area. Often, theorems in subareas are specialized versions of those that appear higher up in the hierarchy. It may be the case that the proof of a specialized theorem is easier in a subarea because one can take advantage of certain properties that are not true of the more general area. As an example, many mathematical structures with structure-preserving maps including sets, monoids, and rings can be viewed more generally as categories with

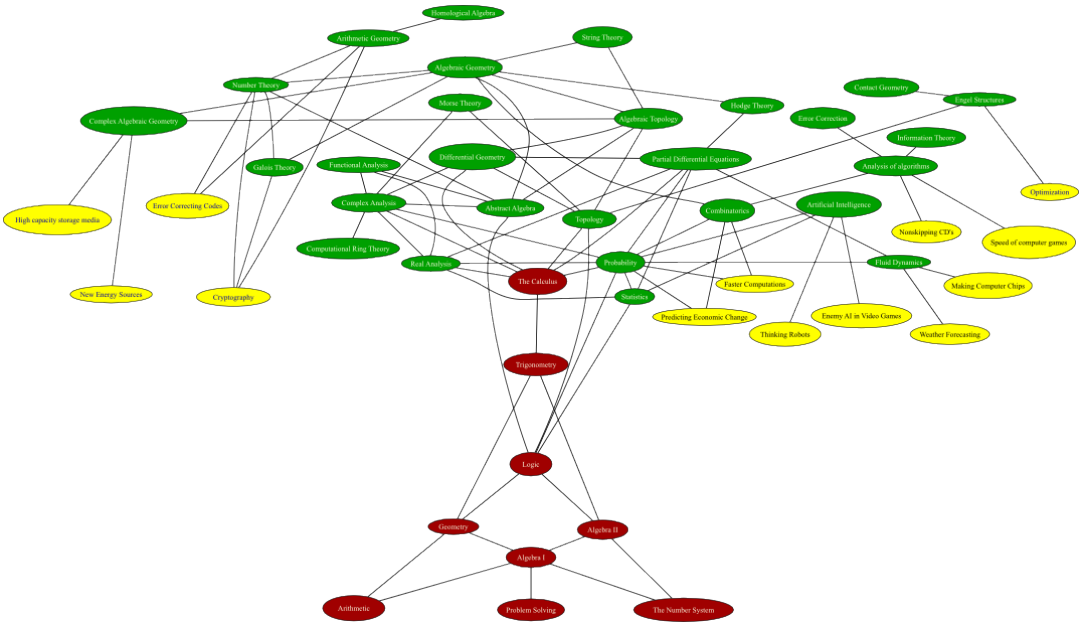


Figure 1.1: A schematic view of the branches of mathematics [101]

morphisms. Some of the properties of the operations on these structures are results regarding morphisms in category theory.

The teaching of mathematics starts with simple, specific concepts and then moves to more general concepts that encompass those already learned as one gets more advanced. Research in mathematics can work in both directions: one starts from more specific ideas and generalizes them; or, one takes general ideas and specializes them to work in a specific instance. It is not always clear which way one is going because the area of mathematics is so large; one may write a paper establishing some new theorems in a subarea of mathematics, only to discover that the work is closely related to or a special case of work in a more general area of which the author was not aware.

The relationships in mathematics are becoming richer as the body of mathematical knowledge is constantly increasing and changing. The number of mathematics journals has continued to increase over the last century and a half, as demon-

strated in Figure 1.2. These journals have continued to become more specific in their topics, indicating that the study of mathematics is becoming more advanced.

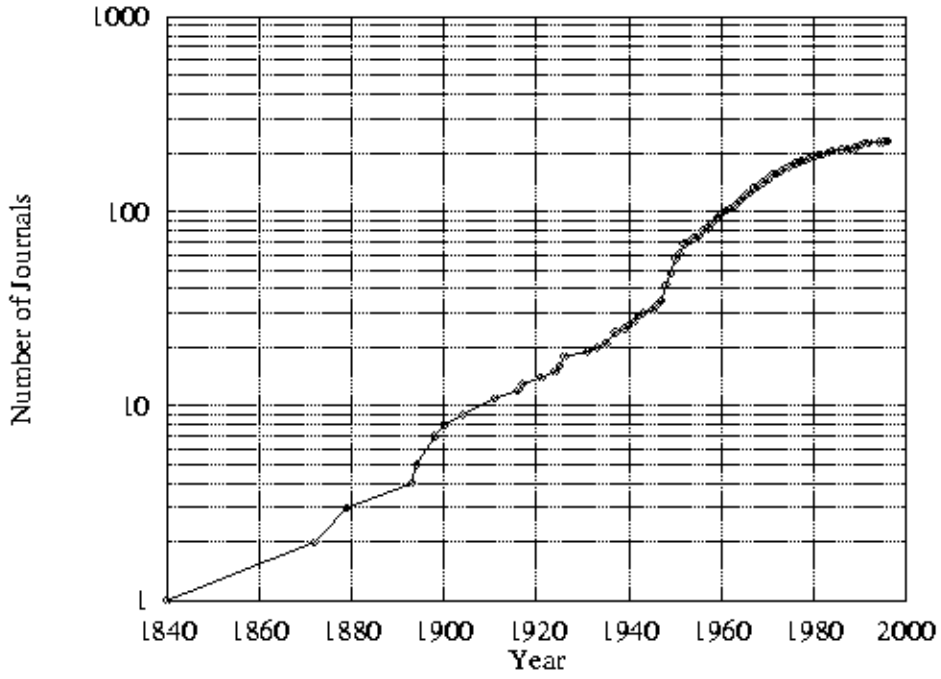


Figure 1.2: Number of mathematics journals [3]

If we look specifically in the field of computer science, where mathematics plays a prominent role, we see a dramatic increase in the number of papers published over the years. For example, the Digital Bibliography and Library Project (DBLP), which provides bibliographic information from papers in major computer science conferences and journals, has seen a dramatic increase in papers, demonstrated in Figure 1.3.

With an increase in the amount and complexity of the information, the organization of mathematical knowledge becomes more important and more difficult. This emerging area of research is referred to as *mathematical knowledge management* (MKM). As summarized by Buchberger, one of the organizers of the first Mathematical Knowledge Management Workshop, the phrase "mathematical

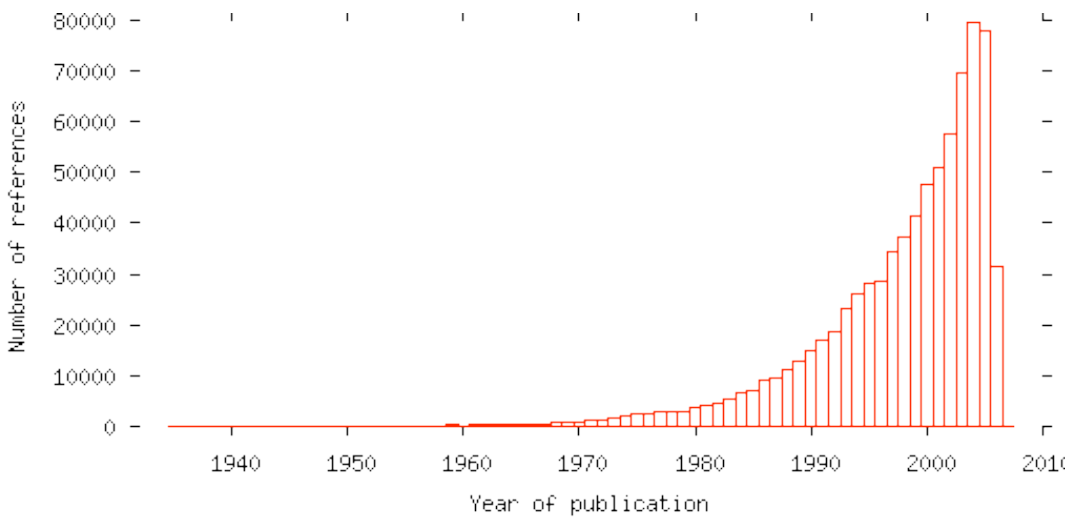


Figure 1.3: Distribution of publication dates for computer science papers [2]

knowledge management" should be parsed as (mathematical knowledge) management as opposed to mathematical (knowledge management), i.e., examining the problem of organizing and disseminating mathematical knowledge [20]. He goes on to summarize the primary issues in the field:

- How do we retrieve mathematical knowledge from existing and future sources?
- How do we build future mathematical knowledge bases?
- How do we make the mathematical knowledge bases available to mathematicians?

At least part of MKM's development has come from the growing power of and interest in *automated theorem proving*, using computer programs to prove mathematical theorems. Using automated theorem provers to find the proofs for theorems offers several advantages. First of all, much of the process can be automated through the use of heuristics called *tactics* and *tacticals*. These heuristics perform basic steps of reasoning, including search for the correct steps to take.

With constantly increasing computer power, more efficient tactics, and research into new search strategies, the portion of the theorem-proving process that can be automated continues to increase.

The primary contribution of automated theorem proving that is relevant to MKM is the formalization of mathematics. A proof written by hand by a mathematician tends to have some steps that are informal or appeal to some intuition on the part of the reader. We even see phrases like "the proof is trivial" or "this step is obvious" in proofs in papers and textbooks. In contrast, a proof produced by a computer program must be rigorous, with every detail justified by a step of reasoning that follows in the domain of the theorem being proven; there is no such thing as "trivial" or "obvious" for an automated theorem prover.

The body of formalized mathematics has continued to increase, with results spanning all major branches of mathematics. As with any large body of information, there is a desire to organize all of these formal theorems and proofs into a digital library. We can then take advantage of the formal structure of these proofs for research and teaching. From a research perspective, we can use the formal library to find theorems useful in a proof we are working on or to discover related theorems based on common proof steps. For teaching, a formalized library of mathematics provides a structured way to organize one's presentation of complex theorems and proofs related to one another.

The basis of any formalized structure for mathematics is a library of proofs and theorems. Large libraries do exist in the automated theorem provers. However, these libraries are usually implemented at the system level, meaning they are not defined with the same level of formalism as the proofs themselves, which rely on a strong underlying proof theory with rules for their creation. In the same way

a proof-theoretic approach can formalize the steps of a proof, we want a proof-theoretic approach that can formalize the relationships between proofs in a library. The library should have the following properties:

1. **Independence** The library should be independent of the underlying logic for which proofs are being done; we should be able to organize proofs for any area of mathematics.

2. **Structure** The formal layout of the library should reflect relationships between theorems. In other words, if we regard a proof to be a lemma used within a larger proof, then the proofs should be such that the relationship is captured inherently in the structure.

3. **An underlying formalism** Proof-theoretic rules should be the basis of manipulating the library. They should formally define the operations of adding a proof to the library, removing a proof from the library, and using one proof in another. These rules should be defined at the same level as the rules used for creating proofs.

4. **Adaptability** The organization of the proofs in the library should be able to change based on the desire to highlight different relationships. For example, one may want to change the structure to group different theorems based on a certain set of lemmas they all use. Changes should be formally described by rules.

5. **Presentability** The formal library needs itself either to be easily read by humans or to be translatable into a format that can be read by humans. The format should reflect the structure of the library and, ideally, be alterable

in a way controlled by the underlying proof-theoretic rules for adapting the library.

The libraries in all of the popular automated theorem provers including Coq [105], NuPRL [71], Isabelle [108], and PVS [89] exhibit the first property. The second property, structure, is found in theorem provers in an informal way. One can declare formulas to be lemmas instead of theorems, however, no distinction is made by the systems themselves. Progress has been made, particularly in Isabelle, toward providing some more structure to the library of theorems.

Properties 3 and 4 are not exhibited by any of the popular theorem provers. As stated, the library is a system-level construct separated from the underlying logic governing the creation of proofs. Therefore, no formalism controls the library. Combined with the fact that there is no structure inherent in the library, adaptability is extremely limited.

Presentability has been addressed by the theorem prover community by taking existing libraries from automated theorem provers and transforming them into a readable format, usually for presentation on the Internet.

In this thesis, we present a proof-theoretic approach to mathematical knowledge management that exhibits all five desired properties. In Chapter 2, we discuss previous work from several aspects of the problem, including proof reuse and library organization. In Chapter 3, we set the basis for a library that exhibits properties 1 and 3 by discussing a *publish-cite* system presented by Kozen and Ramanarayanan [68]. We look at an implementation of this library in an interactive theorem prover for Kleene algebra with tests [63] in Chapter 4. We satisfy properties 2, 4, and 5 by formally defining a hierarchical structure for the mathematical library in Chapter 5. In Chapter 6, we discuss user interfaces for theorem provers and present

a prototype theorem prover for Kleene algebra with tests that presents the library of theorems in an intuitive, structured format. We extend the formalism of the library to include tactics, allowing them to be treated at the same level as proofs and the library itself, in Chapter 7. Finally, we present some future directions for the work and conclusions in Chapters 8 and 9.

Chapter 2

Related Work

The development of formal methods for proof representation and theorem proving has both a rich history and a community that remains active. Much of this work is in automated theorem provers such as Coq [105], NuPRL [71], Isabelle [108], and PVS [89].

The cores of these systems, where issues such as proof representation and the underlying proof logic must be considered, have been well studied and established. However, there are other distinctive characteristics that are paramount to the development of these systems that continue to be important research questions, including proof reuse, proof library representation, and proof tactics. These issues have a serious impact on system usability, both for presenting information to a user and for implementing the system efficiently. We examine the work related to each one of these considerations in detail.

2.1 Proof Reuse

Reusing proofs is important for several reasons. The most obvious is that one does not want to have to perform steps repeatedly when they can be done once and referred to later. From the perspective of an automated theorem prover, time is saved in reusing completed proof steps. The other important reason for proof reuse is that discovering proofs with the same steps helps to establish relationships between theorems, including some that might otherwise go unnoticed.

Carbonell succinctly states the four aspects of problem solving that are relevant to proof reuse, where we transfer information from one proof, called the *source*

proof, to another proof, called the *target proof* [25]:

1. How does one define similarity in proofs?

2. What knowledge is transferred from the source proof to the target proof?

3. How is this transfer accomplished?

4. How does one choose related source proofs given a target proof?

2.1.1 Proof Analogy

A popular method for proof reuse initially explored by mathematicians and artificial intelligence researchers is the idea of *proof analogy*, which tries to map steps from a source proof into steps in a target proof using hints in the relationship between the source theorem and target theorem.

Early work by Kling [56] and Munyer [87] focused on using the source proof to find inference rules that would be relevant for the target proof. Kling's technique can find analogous inference rules for the target proof, but is not designed to use the structure of the source proof to guide the decisions made in the use of these rules. Munyer's work, however, is able to use the order of inference rules in the source proof in order to guide the target proof.

A severe limitation of these approaches is that they define similarity in a purely syntactic sense; syntactic analogy can only discover, for instance, that the proof "if x and y are even, then $x*y$ is even" is related to the proof "if x and y are odd, then $x*y$ is odd." Several others explored other notions of analogy in order to make the technique more powerful, both in finding similar theorems and in applying their proofs.

Carbonell worked on *transformational analogy* and *derivational analogy* in the context of general artificial intelligence problem solving techniques [25]. Carbonell's work dealt primarily with the third element in our list above; but his work also has implications for choosing related theorems and proofs. We talk about his work as it would be applied to theorem proving. Both transformational analogy and derivational analogy attempt to solve a proof by looking at sequences of proof steps that were successful in some previous proof and using them in the target proof. They require the storage of previously completed theorems and their proofs.

Transformational analogy looks for similarity in the statements of theorems, copies the proof for a relevant source theorem, and attempts to adapt the proof to solve the target theorem. The notion of similarity here is vague; it could be as simple as syntactic matching or could use some more complicated metric defined by a user.

In contrast, *derivational analogy* matches source and target proofs instead of theorems. One starts searching for steps in the target proof and then looks for a source proof that has a similar pattern of search. The search procedure for the source proof is then copied to the target proof and used to find a solution. Derivational analogy requires that the proof steps that failed be stored with a proof, in addition to the steps that succeeded. By using the steps from the source proof, one creates a *proof plan*, which guides the steps of searching for a proof of the target theorem [22].

Both of these techniques can be inefficient given a complex similarity metric and large library of previous proofs. The library becomes particularly large when using derivational analogy. Cabonell applied his techniques primarily to natural language

processing and looked at the library of knowledge in that context. Nevertheless, it is obvious that these techniques applied to proof reuse require a well organized library of theorems and proofs.

Melis and Whittle have worked extensively on applying analogy to inductive proofs, particularly for the proof planner CL^AM [79, 110, 81, 80, 82]. They split analogy into two forms: *internal analogy*, which looks for similar subgoals within a single proof, and *external analogy*, which looks for similar theorems outside the context of the current proof. Jamnik demonstrated that Melis and Whittle's technique applies to non-inductive proofs as well [49].

Internal analogy tries to make the search for a proof more efficient by reducing the number of calls to CL^AM's *critic*, which attempts to revise terms on which induction is being performed when the inductive proof can no longer make progress. When CL^AM needs to choose a term on which to perform induction, its analogy system suggests one based on the terms chosen by previous calls to the critic. The suggestions, if successful, prevent the critic from having to search for a term on which to perform induction and prevent the system from performing inductive proofs that will inevitably fail. The use of internal analogy has been able to produce measurable reductions in the time it takes to perform an inductive proof in CL^AM.

External analogy also attempts to reduce the need for search in CL^AM. Melis and Whittle implemented an analogy procedure, ABALONE, on top of CL^AM. ABALONE attempts to find a second-order mapping from source theorems to target theorems. Theorems are represented as syntactic trees in which paths containing existentially quantified variables and induction variables, called *rippling paths*, are marked. Completed theorems–including decisions made in the planning of the theorem's proof, called *justifications*–are maintained in a library. If a useful

second-order mapping from one of these theorems to the target theorem is found, then the proof plan from that theorem is applied to the target theorem. In the event a step of the proof fails, the justifications are used to find lemmas that may be useful to prove in order to continue the proof.

2.1.2 Proof Abstraction

A second approach in research in proof reuse is *proof abstraction*, a refinement of proof by analogy that looks at applying proofs of simpler theorems to more complex ones. One primary difference between proof by analogy and proof abstraction in more recent research is that proof abstraction attempts to abstract important information in a proof in the hope of applying it later without some specific target proof in mind. *Proof abstraction* is "the mapping from one representation of a problem to another which preserves certain desirable properties and which reduces complexity" [45].

Early work can be traced back to Plaisted [93, 94, 95]. His approach is based on abstracting resolution proofs in propositional logic. Plaisted formally defined abstractions and methods for constructing them. These methods can be syntactic in nature, including the renaming of symbols, negation of literals, deletion of arguments to a function, or the turning of functions into propositions. Semantic abstractions based on the underlying domain represented by the atomic propositions are also possible.

The goal is to make a simpler proof through abstraction that has a *resolution proof tree* that can be found by an automated theorem prover. This resolution proof tree is a finite binary tree that finds an assignment of truth values to atomic propositions such that all clauses in a set are true. A resolution proof tree can

be mapped to another tree such that the two trees have the same shape. Then, one can use the resolution proof tree for the abstracted version of a set of clauses to guide the search for a resolution proof for the original set of clauses. Plaisted proved that the proof for the correct abstractions results in the existence of a proof for the original set of clauses. Plaisted further provided a search strategy for abstracting clauses and finding their resolution proof efficiently.

Kolbe and Walther have worked extensively on the problem as well [59, 61, 58, 60]. They were the first to formally develop an explicit notion of a proof library. Their formal system for proof abstraction consists of four important steps: analysis, generalization, retrieval, and reuse. The first stage requires that the inference rules for creating proofs be designed to work on a structure including not only the formula to be proven, but also the "relevant features" that each proof step uses. This additional information, called the *proof catch*, contains a list of axioms used for the proved theorem.

The proof catch and theorem are abstracted during the generalization phase, when function symbols are replaced with function variables. The resulting *schematic conjecture* and second-order *schematic catch* form a *proof shell*, which is stored for use in later proofs. Since it is possible to have many catches for a single conjecture, a *proof volume* is formed, containing a schematic conjecture Φ and a set of schematic catches that, when individually paired with Φ, form a proof shell. These proof volumes are collected together in a library called the *proof dictionary*.

The final two stages, retrieval and reuse, allow us to take advantage of shells stored in the dictionary. Retrieval attempts to find a second-order substitution to instantiate a schematic conjecture to a theorem we are currently trying to prove. The same substitution is applied to a corresponding schematic catch. Since a

single schematic conjecture maps to several schematic catches in a proof volume, it is possible to try several catches with only one search. When a catch is specialized through substitution, it is possible that some of the function variables will not be instantiated. If this is the case, these variables must be instantiated using another substitution. The resulting proof must be verified to make sure that all proof obligations follow from the set of axioms.

Giuchniglia and others have looked into providing a more theoretical basis to abstraction [45, 44]. Giuchniglia and Walsh provided formal definitions of the three main properties of abstractions:

1. An abstraction maps the representation of a problem called the *ground representation* to a new representation, the *abstract representation*.

2. An abstract representation preserves desirable properties of the original problem.

3. An abstract representation is easier to prove.

These properties are formalized through the use of a *formal system*, including a set of axioms, a set of inference rules, and a language for writing formulas.

Abstractions can be classified based on their power and usage. The power of an abstraction relates to its ability to provide usable proofs in the ground representation. Ideally, a formula is provable in the ground representation iff its abstract version is provable in the abstract representation. However, it is possible that the "if" or "only if" part of this statement holds without the other. With regard to use, the authors describe the opposing properties of *deductive uses* and *abductive uses* and *positive uses* and *negative uses*. Deductive uses provide a guarantee that a theorem holds in the ground representation when the abstract version of the

theorem holds in the abstract representation, whereas abductive uses do not provide this guarantee. "Positive" and "negative" uses refer to whether the proof of an abstract formula gives us information about the ground representation of the theorem or its negation. Many abstraction techniques can be classified based on these properties.

2.1.3 Applications of Proof Reuse

Proof reuse has been applied to several formal verification problems. Melis and Schairer applied some of their work in proof by analogy [80]. The proofs with which they work are first-order predicate logic formulas. In their proofs, the nature of the problem is such that subgoals are often very similar, so the reuse of completed proofs is instrumental in reducing the time required to verify programs that may take weeks to do by hand.

The authors have a notion of a lemma, where a proof used in an earlier subgoal can be generalized and reused within later subgoals of the same proof. The system can attempt to detect these similar proofs automatically or the user can specify them. Their analysis indicates that a significant amount of time can be saved when proofs are reused.

Despite the savings, the relationship between these subgoals is never stored in the proof, so a later analysis of the proof would not reflect the fact that similar subgoals were found and reused. Moreover, lemmas are not stored or reusable in different theorems. Given the similarities within proofs, one can imagine that there would also be several similarities between proofs for which storage of some of the more fundamental lemmas could be justified.

Beckert and Klebanov developed a technique for proof reuse that they applied

to correctness proofs for Java programs in the KeY system [17]. Unlike other techniques, which normally attempt to reuse an entire proof, Beckert and Klebanov's procedure reuses only one proof step at a time. Their algorithm considers the current goal in a target proof and analyzes possible uses of proof steps from a single source proof. This source proof need not be complete. Upon successfully applying the proof step to the current goal, the algorithm examines the source proof for steps that followed this proof step and measures there similarity based on the *minimal edit script*, the alterations it would take to turn one program into another.

Beckert and Klebanov have applied their technique to the verification of Java programs. As demonstrated in examples, their algorithm is primarily suited for proofs when the source and target programs are nearly the same. For example, one may start on a proof of the correctness of a program only to discover it cannot be verified. Alterations can be made to the program to correct errors, e.g., not checking for division by zero, and then the correctness of this new program can be verified. The new program is likely to be very similar to the old program, so proof steps from the incomplete proof of correctness for the old program can be duplicated for the proof of correctness of the new program, which can hopefully be verified. With such an approach, the authors did not consider the organization of proofs into any retainable structure.

Pons explored proof generalization and proof reuse in the Coq theorem prover [96]. His goal was to create second-order abstractions of proofs done using Coq so that they could be used in other contexts. In order to create a generalized proof for a theorem regarding some function, one must abstract out the function itself and any properties of that function. For example, generalizing a proof regarding integer

multiplication may require abstracting out uses of associativity and commutativity. The generalized version of the theorem can then be applied to other functions that have the same properties.

Pons proposes a simple algorithm that could be integrated into Coq to generalize proofs. The user could specify a function to be abstracted and then the system could handle discovering properties of this function that also need to be abstracted, creating a generalized version of the proof automatically. However, Pons's algorithm for discovering properties of the abstracted function is based on naming conventions used by creators of proofs; proofs that do not adhere to these naming conventions may fail.

2.2 Library Organization

Proof reuse, particularly proof analogy, may require the maintenance of a library of completed proofs. However, current literature explores the organization of this library strictly in terms of proof search, if it is discussed at all. Organizing a library of proofs without regard to search is an interesting problem in itself, especially with the increasingly large body of mathematics formalized by automated theorem provers and made available on the Internet.

With such vast libraries appearing, library organization is important for several reasons. First of all, automated theorem provers need to be able to deal with hundreds or thousands of theorems and proofs efficiently. Efficiency in a theorem prover can encompass many factors. We have already discussed the importance of proof search, in which library organization plays an important role. Beyond that obvious issue, there is also the desire to group related proofs together into a *theory*. "Related" can mean many things in this context. It can be some informal

notion based on intuition on the part of a user organizing a theory; or, it can be a more formal idea based on the contents of the theorems and proofs being grouped together. It stands to reason that these two concepts are connected. Theorems that make similar assumptions or use the same lemmas in their proofs are likely to be related at some intuitive level.

The other important reason for formal library organization is the users of these libraries. The wealth of mathematical knowledge available on the Internet and in automated theorem provers is not useful if it cannot be presented in a reasonable fashion. The presentation must be dynamic, too, as users may want the information to be organized in different ways that change over time. These organizations are likely to be based on how theorems are related, as discussed above. One person may want the presentation of several theorems to be grouped by their common assumptions; another may choose to focus on a certain group of lemmas that the theorems have in common. These decisions may affection how a person approaches a proof currently being worked on or which theorems one attempts to prove in the first place. Presentation is not simply an issue for a user interface, but represents a fundamental question regarding the organization of mathematical knowledge.

Several people have looked at the problem of library organization. In fact, an entire research community devoted specifically to mathematical knowledge management exists. Creating a large knowledge base for mathematics is a relatively new problem. Mathematical knowledge management is a growing area of research focused on several aspects of the problem. The goal is to discover and express relationships between proofs to form a coherent knowledge base that can aid in teaching and researching mathematics.

2.2.1 Representing Mathematics for Wide Dissemination

A formal language for specifying mathematics is a necessary step for representing the wide range of fields that exist. Unlike presentation-based languages such as LaTeX, a language for mathematics should incorporation semantic information about the formulas and symbols it encodes. Without such information, a computer can only process the mathematics and not actually understand it, a necessary step if the computer is to provide infrastructure for organization and search based on the meaning of formulas.

The MIZAR project provides a language for the formalization of mathematics that is used for the creation of the Mizar Mathematical Library (MML) [98]. Users create *articles*, which contain a set of related theorems and references to other articles they use. Mathematical knowledge management is important in three parts of MIZAR: organizing individual proofs, organizing proofs within an article, and organizing articles in the MML. We focus on the last one.

The MML requires that all articles added to the central database be submitted and undergo several steps before acceptance. A submitted article must have already passed through a verifier, which checks the steps of all proofs in the article. Once the verifier declares that an article has no errors, it can be submitted to the MLL, where it passes through several automated programs. Currently, there is no human intervention at this level to determine the appropriateness of an article; it only need pass technical requirements. Once accepted, an article is added to the *Journal of Formalized Mathematics*, available at the MIZAR website.

An interesting issue MIZAR must deal with is the relationship between individual articles and the rest of the MML, called the *local environment*. This local environment provides a context containing theorems from the MML referred to by

an article. Such an environment might not be necessary if the MML were small; the entire library could be the context. However, with the increasing size of the library, size becomes an issue. MIZAR requires articles to contain declarations for importing elements from other articles called *constructors*. The system's *accommodator* manages the recursive importing of other articles that those constructors need. Work continues on the problem of limiting this recursive process to import only what is needed.

Extensive work has gone into the Open Mathematical Documents (OMDoc) project, which provides a rich language for representing mathematics [57]. OMDoc looks to provide a markup language that can annotate text and formulas to provide structure for use in presenting, archiving, and transmitting mathematical knowledge through the use of content MathML [10] and OPENMATH [23], both based on XML.

Of primary importance in using OMDoc to organize large libraries of formulas is the ability to assign importance to them. Such an assignment is achieved through a `type` attribute, which can be one of several values including `theorem`, `proposition`, `lemma`, or `corollary`. It is important to note that the appropriate use of these terms is still up to users; only informal guidelines are given such as "[a theorem is] an important assertion with a proof" and "[a lemma is] a less important assertion with a proof."

Proof representation in OMDoc has two aspects: the textual representation of a proof step and the justification for the proof step, e.g., a premise whose truth is assumed or already proven or a subproof that provides more detail. OMDoc cannot itself check the validity of these deductions; it is meant only to provide a descriptive language that allows one to specify them. The language is able to

check that premises used are currently in scope, however. This scope includes not only the premises and theorems that can be used, but also local hypotheses that are declared and used to simplify proof steps, similar to the *cut* rule described in Section 4.2.4.

OMDoc also handles the relationships between collections of theorems. These theories are treated as first-class objects that can be structured like documents and broken down into sections. The simplest relationship between theories is inheritance, established with the `import` element, which specifies that a theory accesses elements from another theory. A theory contains the union of the elements explicitly defined and those imported. Care has to be taken to ensure that the inheritance is acyclic. Additional care has to be taken because of the discrepancy between theory names and file names; two files could define different theories with the same name and both could be imported into a third theory, which could consequently be ill-defined.

More complex relationships are possible, too, using a generalized notion of *theory inclusion* which resembles the use of functors as presented in Section 2.2.3. One can define a *morphism* between a source theory and a target theory, a translation of symbols in the source theory to the target theory. Theorems, definitions, and proofs in the source theory can then be translated and used in the target theory. In OMDoc, theory inclusion is treated as a structural property as opposed to a logical property. In the event they are necessary, OMDoc allows the explicit declaration of well-definedness conditions, such as the enforcement of total orderings on sets for use in a list theory that requires comparison.

Both OMDoc and MIZAR provide rich languages for the creation of a library of mathematical formulas. These languages are essential for providing a common

formal description of theorems and proofs across different theorem provers and the Internet.

2.2.2 Creating a Large Mathematical Knowledge Base

Buchberger has worked on the *Theorema* project, a system built on top of Mathematica to add theorem-proving capabilities to the software [20, 91, 21]. However, unlike many theorem provers, *Theorema* places a great deal of focus on the organization of the mathematical library and the presentation of theorems and proofs to users.

Theorema has three primary components: reasoners, organizational tools, and knowledge bases. The latter two are the ones most associated with managing the mathematical theorems that users add to the system. Collections of formulas are organized into *Theorema notebooks*, which can be stored and referred to later. To facilitate the organization of formulas, the system has labels that assign numbers or names to proof steps, as well as hierarchical relationship information through keywords such as "lemma," "theorem," and "definition." These labels do not have any meaning in the underlying logic of the system. Labels provided by the user in writing theorems and proofs are used to organize the notebook when the system creates it. These labels can then be used to refer to theorems in other notebooks or when calling one of *Theorema*'s 30 accessible reasoners.

Theorema does require some organizational information on the part of a user in order to organize notebooks correctly. Specifically, the user must separate text from mathematical formulas and must group formulas under appropriate headings, including "Definitions," "Theorems," "Propositions," etc. [91]. With that information, *Theorema* provides an environment for organizing mathematical knowledge

that makes it easily searchable, extendible, and teachable.

The National Institute of Standards and Technology (NIST) has expended considerable effort in creating its Digital Library of Mathematical Functions (DLMF), what is meant to be a definitive collection of formulas, graphs, and other information pertaining to elementary and higher mathematics [73, 83]. For the first time, the NIST will present a comprehensive list of mathematical formulas online, including hyperlinks to related content and proofs, graphics, and search and download capability. One of the project's goals is to develop general techniques for the organization of large amounts of mathematical data.

The DLMF represents formulas in LaTeX, a language many mathematicians use for the preparation of documents. While the use of LaTeX allows for the reasonable presentation of mathematical knowledge, it does not attach any semantic meaning to it symbols, making difficult the problem of searching the digital library for specific formulas. To aid in search, formulas are annotated with metadata, which serves to disambiguate notation. The metadata also ensures that every formula has a link to a proof. The search problem for the NIST primarily revolves around the ability to create some concrete search syntax for queries that can be used to find symbols inside of mathematical formulas. One may then want to take search results and use them in a computer algebra system or combine them into a customized collection of formulas. The DLMF is meant to make that process as simple as possible for users.

2.2.3 Proof Organization in Automated Theorem Provers

Several theorem provers use ML-style modules as a means by which to organize theories in a hierarchical fashion. Modules have a strong theoretical basis that has

been well studied [74]. Chrząszcz investigated using modules in Coq to provide structure to assumptions, variables, and proofs in the system [9]. Modules are developed interactively by a user in the Coq environment and stored for later use. These modules contain definitions of variables and assumptions, proofs of theorems to be used as lemmas, and even nested modules. They can be required to adhere to a signature, which describes the types of the elements of the module.

The full power of these modules is realized when one employs *functors*, which parameterize modules over signatures. Functors allow one to declare a module abstracted over variables and types that may occur in it. This abstract module, when applied to a specific signature, results in a module with variables, assumptions, and proofs specialized to the elements of that signature, eliminating the need to repeat proofs.

Coq's module system is admittedly quite limited. First of all, once a module is created, it cannot be altered. This means that elements in a submodule cannot be moved to a parent module in order to widen their scope. It is also not possible to have more than one module open at once, which may be desirable if a theory draws its lemmas from several distinct areas contained in separate modules.

Windley developed a package for the HOL theorem prover that allows one to use abstract theories, similar to the use of functors in Coq's module system [111]. Abstract theories use the ML metalanguage in HOL and higher-order logic to provide structures that can be instantiated. The name of the abstract theory is applied to concrete objects to form a specific instance of the theory. The abstract theory has a set of theory obligations declared in ML that become assumptions in the instantiated theory.

Durn and Meeguer developed a module system for the theorem prover Maude

[34]. Their module system takes advantage of Maude's reflectivity to provide users with a module algebra including functors and object-oriented modules. Maude formally defines the operations performed on modules, including renaming and importing. Stressing the importance of performance, Maude compiles modules to a flat, unstructured representation when creating system modules. The underlying system itself does not take advantage of any structure a user has added to help it create theories.

The most advanced organization system in a theorem prover is Isabelle's *locales*, which limit the use of a set of local assumptions and definitions to a current theory [53, 52]. The original intent of locales was to provide a means by which to define syntax and rules whose usefulness did not extend beyond a limited number of proofs. Locales contain variables, assumptions, and definitions. The variables can be viewed as elements that a mathematician would describe as "arbitrary, but fixed" for the purposes of a proof. The local assumptions are properties of these elements. Local definitions are primarily shorthand notation used for large formulas. These definitions may include concrete syntax for better pretty printing.

Locales can be opened for use in proving a theorem, then closed when no longer needed. The stack of active locales forms a *scope* for a proof. Locales can extend other locales, leading to a hierarchical scope with nested locales. Once a theorem is proved in this scope, it can be exported out of the scope of locales, either one at a time through the stack or all at once, the latter resulting in a theorem at the global level. In an export, definitions are made into meta-assumptions and constants are universally quantified. It is important to note that constants and rules defined in a locale that are not used in the proof of a theorem are not included in the export operation.

Wenzel made several improvements to the locales system in his development of Isar, a proof language for Isabelle with a focus on human readability [109]. Ballarin discusses the improvements that were added to the system [12]. Wenzel's locales add the ability to scope theorems through *notes*, which store facts about the constants declared. These notes are theorems in which a locale has been specified as the storage location; theorems are usually added to the global environment. Whenever a locale is opened, its notes are available as rewrite rules, even they are not in the default set used by Isabelle's simplifier. In this way, one can create specialized versions of a theorem, using local constants and local assumptions to instantiate any universal quantifiers of the theorem.

Wenzel's locales also support a richer mechanism for combining locales. Multiple inheritance in nested locales is possible through the normalization of *locale expressions*, a language for combining locales. The most important expression is *merge*, which combines the elements of two locales. Proper combination requires normalization in order to avoid naming conflicts. The existence of the merge command means that using locale expressions is more powerful than simply opening locales and adding their contents to the current scope.

While locales contribute much to library organization in automated theorem provers, they are not without their limitations. Locales are implemented at a level separate from both the declaration of theories and the underlying proofs. While the separation from the declaration of theories allows for more reuse of elements in locales, it also requires a more extensive examination of the relationship between locales and proofs using *development graphs*, which model dependencies in proofs [13].

Another limitation is in the export mechanism. While it is possible to gener-

alize variables, assumptions, and even theorems out of a locale so that they are in a wider scope, it is not possible to move them into a locale to limit their scope. Such an ability can be important if we do not know the organization of a set of theorems before we prove them or if we wish to reorganize the library dynamically to highlight different relationships between theorems through their common structure.

2.2.4 Extracting Libraries from Automated Theorem Provers

With their large sets of proofs, automated theorem provers provide a wealth of mathematical knowledge that can be organized for presentation to users in a variety of fields. Several people have looked at automatically extracting the libraries from these theorem provers to create a cohesive online library.

Asperati et al. worked on the Hypertextual Electronic Library of Mathematics (HELM) project [6], which seeks to use XML to create and maintain an online mathematical library. HELM takes advantage of the structure and maturity of XML to provide better infrastructure for publishing, searching, and modularizing formulas. What separates HELM from other similar projects is that it stresses the importance of proofs as a means by which to organize theorems into a structured hierarchy.

HELM's library structure differs from others in that it separates every theorem and definition into a separate XML file, considering these to be the smallest entity to which one would want to refer. This organizational decision was made in the hope of avoiding the need to import an entire theory to use a single result, which can drive users to simply redefine the result and thus leads to duplication in the library. HELM also wants to avoid a large, flat library structure that can come from

putting too many theorems and definitions into single files; the physical structure of the XML files should reflect the organization of the mathematical library.

HELM distinguishes between *documents*, arbitrary collections of theorems and definitions for presentation, and *theories*, sets of definitions and theorems organized by their formal structure. The organization of theories should be reflected in the underlying organization that the Uniform Resource Identifiers (URIs) used for navigating theorems and definitions. On the other hand, documents, which are meant to be assembled by authors, should be more free in their organization and independent of the organization of the XML files themselves.

Cruz-Filipe et al. worked on the Constructive Coq Repository at Nijmegen (C-CoRN) project, which makes the libraries of Coq available as an online repository [31]. One of the primary desires in developing such a library was *coherency*: related theorems should be grouped together in theories that can be explicitly extended by other theories. Consequently, the library is a tree-like structure. At the lowest level are tactics used for equational reasoning throughout the system. Above the tactics, elements are hierarchically arranged with more complex structures inheriting from simpler ones, e.g., ordered fields are above groups.

Unlike HELM, C-CoRN's library structure groups lemmas into single files for organization. Nevertheless, the system is designed with updates to these files in mind. The hope is to avoid the duplication of lemmas that are used often and the unnecessary repetition of proofs, a problem often encountered in other systems where the overhead in adding a new lemma to a file results in smaller files being added and never changed again.

Like the module system found in Coq, C-CoRN places a lot of importance on abstraction in its hierarchy, as this allows results to be proven once for the abstract

case and then specialized for concrete applications. Abstraction also allows for the reuse of notation in cases where the system may have limitations in allowing overloading. For example, abstraction allows the plus symbol '`[+]`' to be used for real numbers, integers, and natural numbers. However, abstraction may not always be ideal if optimization is a concern; some proofs are more straightforward when they can take advantage of properties of the concrete objects.

Lorigo et al. worked on applying WWW search techniques to obtain information about the structure of libraries of proofs and theorems [72]. They applied Kleinberg's Hyperlink-Induced Topic Search (HITS) algorithm [55] to aid in the search of relationships between mathematical theorems and proofs in the Cornell Formal Digital Library (FDL). The search may wish to discern theorems that are representative of a given collection of theorems or to automatically find collections of theorems based on their contents.

Proofs can be seen as a graph structure where nodes are the names of theorems and directed edges represent the "refers to in proof" relation. In this way, a library of theorems resembles the structure of web pages with hyperlinks to other pages. Applying the HITS algorithm to Cornell's FDL reveals clusters of theorems that could be grouped together in a single theory. Lorigo et al. found that these clusters often reflect theorems grouped together by humans when initially placed in the FDL, indicating that the structure of proofs can in fact provide enough information to group them automatically. However, the approach is meant to be used with already existing libraries of formal mathematics. It gathers information from the library and presents it to the user, but does not reorder the theorems in the library itself into the discovered relationships.

2.3 Tactics

Tactics are computer programs meant to carry out steps of deduction automatically. Tactics can also be combined using *tacticals*, which generally include operations that can compose tactics, perform a conditional test based on the success of a tactic, or repeat a tactic. The primary goal of these tactics is to automate as much of the creation of a proof as possible in a way that is sound with respect to the underlying logic. One of the earliest examples of tactics is Edinburgh LCF, an automated theorem prover developed in the 1970s [46]. In fact, the ML programming language commonly used as tactic language today, and now as a stand-alone programming language, was created specifically for the LCF tactics system.

Effective tactics are a central focus of many modern theorem provers. Most popular theorem provers today contain an ML-like language for tactics, including Coq [105], NuPRL [71], Isabelle [108], and PVS [89]. The Turing-complete language is separate from the underlying language used to represent proofs. Such languages with their strong typing, higher-order constructs, and pattern matching make the implementation of standard tacticals easier. The tactics found in these systems are built from basic inference rules into complex programs that can apply rules, choose between tactics to apply, and analyze the current structure of a proof.

2.3.1 Theoretical Developments in Tactics

Felty looked at implementing tactics in higher-order logic programming languages such as λProlog, which is based on *higher-order hereditary Harrop formulas* [35]. Logic programming languages have built-in infrastructure for unification and search, essential operations in the implementation of tactics. *Clauses* in Prolog-based lan-

guages, where the *body* of a clause implies its *head*, corresponds naturally to the statements of inference rules in which premises imply some conclusion. Additionally, quantification is easily represented using metavariables. These features are in contrast to a typical ML language used for tactics in which features like quantification must be encoded specially.

The main benefit of using a logic programming language is *backtracking*. Backtracking is part of the unification and search mechanism built into logic programming languages. In an attempt to show that a clause is satisfiable in a program, a system like λProlog instantiates variables and attempts to show that all clauses are satisfiable. If a clause fails, the system backtracks to the last successfully satisfied clause and attempts to use a different unification to satisfy it. This process continues until all clauses are satisfied or until all possible unifications are exhausted and the initial clause is deemed unsatisfiable.

One can easily write a set of inference rules for a first-order logic in λProlog. These inference rules can be converted to a set of tactics that can be combined using a set of tacticals. The tacticals defined include composition (`then`), choice (`orelse`), and loop (`repeat`). When implemented with the metalanguage feature *cut* (`!`), which prevents backtracking beyond a certain point, one can obtain the desired operations for tacticals. When using tactics interactively, one has the ability to backup the search for a proof one step at a time; information regarding state during forward and backward operations is handled by the system itself and requires no additional infrastructure.

The use of a language such as λProlog has some other benefits as well. The language's modules allow one to import and use tactics dynamically. Moreover, the implementation allows one to specify different search strategies for use with

the `repeat` and `orelse` tactics. One may use a simple depth-first search if it is sufficient or implement some more complete search strategy if necessary.

Giunchiglia and Traverso worked on correlating tactics in the theorem prover `GETFOL`, considered to be at the *object level*, with terms in a first-order metatheory called MT [42, 43]. They succinctly stated the properties desired for a tactic language: the tactics should be expressions of a logical language in order to facilitate reasoning about them; and, there should be a correspondence between the tactics as represented in this logical language and the programs that implement the tactics. This correspondence is one-to-one between well-formed formulas and computation trees at the object level.

The authors defined both an object theory OT and a metatheory MT. Each theory contains its own language, axioms, and inference rules. The axioms of MT are lifted from the axioms of OT. The original work was admittedly limited; it could only correlate well-formed formulas and primitive object-level tactics, those that could be expressed as a finite sequence of proof steps[42]. Tactics are represented as *sequent trees*, trees of object-level applications of inference rules. One can formally define tactics in the metatheory by defining a notion of *generalization*, replacing constants with variables. As defined, several axioms presented by Giunchiglia and Traverso are tactics; all the tactics in MT correspond to tactics in OT. The relationship allows one to manipulate the tactics at the logic level to alter and optimize the programs that implement them. The authors are able to prove that the correspondence between OT and MT is correct.

Giunchiglia and Traverso further extended their metatheory to represent common tacticals [43]. The difficulty with providing a correspondence between tacticals and a representation in some metatheory is that their execution may not

terminate. In order to represent tacticals, the authors had to add an *if-then-else* construct and function names to the language of the metatheory. Names must be used because both OT and MT are first-order; one would typically use higher-order syntax in a language such as ML to represent tacticals. The authors showed that, even with the extensions, MT's representation of tactics and tacticals is sound.

Syme argued against the use of tactics as we typically see them in theorem provers [104]. He proposed the use of three constructs in a declarative proof system called DECLARE. A proof is *declarative* if the result of a proof step can be understood on its own without appealing to the justification for that step. Popular automated theorem provers tend to be *inferential*, where an appeal to a justification is necessary and automatically interpreted by the system to determine the result of a proof step.

DECLARE uses three constructs for the language of proofs: *decomposition* and *enrichment*, appeals to automation, and *second-order schema application*. Decomposition and enrichment split a proof into several cases and add fact, goals, and constants to the proof environment, respectively. Appeals to automation are hints provided to an automated theorem prover, which is treated as an oracle in this context. These are described by a simple language meant to be declarative in nature. The constructs in the language include highlighting elements from the proof environment, specifying variable instantiations, and specifying case splits.

Syme argued that the declarative approach has several advantages over traditional tactic-based approaches. He argued that while tactics do offer the ability to program more complex and general algorithms for solving proofs, practical tactics tend to be extremely specialized. The simple nature of declarative proofs makes them easier to read and therefore easier to reuse in another setting, including in

the context of different automated theorem provers.

Martin et al. expressed tactics in a general language called *Angel* with a formal semantics that results in a calculus for reasoning about tactics [76]. Angel, independent of the underlying logic for which proofs are done, can be used to prove properties about the tactics themselves, including optimizations for efficiency and readability. The language represents the basic operations we often see in tactics: rule usage, sequence, and choice. The more complex operation of repetition is represented with the recursive μ operator. Several laws regarding the equivalence of tactics can be used to reason about and formulate new tactics. These rules are complete, meaning two tactics can be proven equivalent with these laws if their observable behavior is equivalent.

Martin and Gibbons continued their work, generalizing to a monadic structure that was underlying the list structure used in their semantics [77]. Monads are an established method for modeling intricate programming language features including nondeterminism and side-effects [85]. Useful monads include the list monad (for modeling nondeterminism), the exception monad (for modeling possible failure), the state monad (for modeling a store that can be updated), and the continuation monad (for modeling programs in continuation-passing style).

Martin and Gibbons used a function to convert tactics into functions of the proper type for monadic interpretation. This function also requires an interpretation of primitive rules and an environment for constructing recursive tactics. The interpretation of the semantics in different monads leads to different models of the tactics. As mentioned, using the list monad yields the original Angel semantics. The use of the exception monad results in a semantics very close to those used in Edinburgh LCF. Choice semantics such as those found in Isabelle are most accu-

rately represented by combining state and list monads. Hence, the tactics language in many theorem provers can be modeled by a common underlying formalism.

2.3.2 The Reuse of Tactics

The reuse of tactics is an issue worth mentioning outside the context of simple proof reuse, discussed in Section 2.1. Tactics are programs that can be used as justifications for proof steps; however, applying such a justification to another proof requires special care, since the original use of the tactic may include nondeterminism, backtracking, and search.

Schairer et al. looked at giving users more control over the reuse of tactics in order to increase the likelihood that the tactic succeeds while at the same time reducing the overhead of reusing a tactic [99]. Specifically, the technique improves tactic *replay*, the re-execution of a tactic in the context of a new proof that looks similar to the proof in which the tactic was originally used. In this replay, one does not want to repeat search steps; that the tactic succeeding in the first place means that the correct path is already known and can be remembered for reuse. Current theorem provers allow tactics to succeed or fail, with no ability to stop at a state in the middle of execution. If a small change is needed in a tactic in order for it to succeed, one must either change the code for the tactic and re-execute it from the beginning or must carry out the proof rules manually.

In order to eliminate unnecessary search, Schairer et al.'s technique maintains a trace when evaluating a tactic. This trace keeps track of further calls to tactics and choices made in a search. In the event backtracking is performed, elements from the sequence of steps in this trace can be removed. When reusing a tactic later, one can use the trace to avoid search, as choices are explicitly maintained.

Furthermore, one can use the trace to interactively alter the steps in the replay of a tactic. In the event that the replay of a tactic fails, a user can alter the trace at specific points to make a different choice in a search. One can also replace the call of one tactic with a call to another, referred to as a *callback*. The remainder of the tactic replay can be carried out in the new context formed by making different choices in the search or in calls to other tactics. Consequently, tactics being reused may succeed where they otherwise would have failed.

Felty and Howe looked at harnessing the power of logic programming languages for tactic generalization and reuse [36]. The issue is the same as in the work of Schairer et al.: how does one reuse tactics that are provided as justifications for proof steps? Felty and Howe's technique relies on λProlog's metavariables, variable binding, and backtracking. The system find a minimal unifier, which matches variables in conclusions of proof steps to the variables in premises of subsequent proof steps. Finding this minimal unifier results in a most general version of the tactic justifications, which allows them to apply the tactics in many other proofs.

Chapter 3

Publication-Citation

Formal proof representation is paramount to theorem provers such as Coq [105], NuPRL [71], Isabelle [108], and PVS [89]. However, these systems do not provide a formal basis for remembering and reusing proofs. The proof library is a system-level infrastructure for loading and saving theorems that is separate from the underlying proof theory.

Kozen and Ramanarayanan present a *publish-cite* system, which uses proof rules with an explicit library to formalize the representation and reuse of theorems [68]. The work provides the basis for Chapters 4-7, so it is described in detail in this chapter.

3.1 Motivation: A Classical Proof System

First, we look at a classical proof system for constructive universal equational Horn logic. We build theorems from terms and equations. Consider a set of *individual variables* $X = \{x, y, \ldots\}$ and a first-order signature $\Sigma = \{f, g, \ldots\}$. We use $\overline{x}$ to refer to a sequence of variables $(x_1, \ldots, x_n)$. An *individual term* $s, t, \ldots$ is either a variable $x \in X$ or an expression $ft_1 \ldots t_n$, where f is an n-ary function symbol in Σ and $t_1 \ldots t_n$ are individual terms (referred to with the notation $\overline{t}$). An equation $d, e, \ldots$ is between two individual terms, such as $s = t$. We use the notation $e[x/t]$ to denote the equation e with all free occurrences of x replaced by t.

A *theorem* φ, ψ is a universally quantified Horn formula of the form

$$\forall x_1, \ldots, x_m . d_1 \to d_2 \to \cdots \to d_n \to e \tag{3.1}$$

where the d_i are equations representing *premises*, e is an equation representing the conclusion, and $x_1 \ldots x_m$ are the variables that occur in the equations $d_1, \ldots, d_n, e$. A formula may have zero or more premises. These universally quantified formulas allow arbitrary specialization through term substitution. An example of this specialization can be seen in Section 3.3.

The following is the set of axioms E of classical equational logic with implicit universal quantification.

$$x = x$$

$$x = y \rightarrow y = x$$

$$x = y \rightarrow y = z \rightarrow x = z$$

$$x_1 = y_1 \rightarrow \cdots \rightarrow x_n = y_n \rightarrow f\overline{x} = f\overline{y}$$

where f is an n-ary function in Σ. In addition, there is a set of application-specific axioms Δ.

The deduction rules are in Figure 3.1, where A is a set of equations. The last rule requires that x does not occur in t. This derived rule allows us to use implicit universal quantification.

We may wish to annotate these formulas with simply typed λ-terms in order to remember the steps of deduction. Let $\mathcal{P}$ be a set of *proof variables* $p, q, \ldots$. A *proof* of a theorem is a λ-term abstracted over both the proof variables and the individual terms that appear in the proof. A *proof term* is:

- a variable $p \in \mathcal{P}$
- a constant axiom_φ, referring to an axiom $\varphi \in E \cup \Delta$
- an application $\pi\ \tau$, where π and τ are proof terms
- an application $\pi\ t$, where π is a proof term and t is an individual term

$$\frac{}{\vdash \varphi[\overline{x}/\overline{t}]}\ \varphi \in E \cup \Delta$$

$$\frac{}{e \vdash e}$$

$$\frac{A \vdash \varphi}{A, e \vdash \varphi}$$

$$\frac{A, e \vdash \varphi}{A \vdash e \to \varphi}$$

$$\frac{A \vdash e \to \varphi \quad A \vdash e}{A \vdash \varphi}$$

$$\frac{A \vdash e}{A[x/t] \vdash e[x/t]}$$

Figure 3.1: Rules for a classical proof system

- an abstraction $\lambda p.\tau$, where p is proof variable and τ is a proof term
- an abstraction $\lambda x.\tau$, where x is an individual variable and τ is a proof term

When creating proof terms, we have the typing rules seen in Figure 3.2. These typing rules are what one would expect for a simply-typed λ-calculus. The typing environment Γ maps variables to types. According to the Curry-Howard Isomorphism, the type of a well-typed λ-term corresponds to a theorem in constructive logic and the λ-term itself is the proof of that theorem [100]. For example, a theorem such as (3.1) viewed as a type would be realized by a proof term representing a function that takes an arbitrary substitution for the variables x_i and proofs of the premises d_i and returns a proof of the conclusion e.

We now state the annotated versions of our proof rules in Figure 3.3. Note that the proof terms for term abstraction and application do not yet appear, as we still use implicit universal quantification.

$$\frac{}{\Gamma, p : e \vdash p : e}$$

$$\frac{}{\Gamma \vdash \mathsf{axiom}_\varphi : \varphi}$$

$$\frac{\Gamma \vdash \pi : e \to \varphi \quad \Gamma \vdash \tau : e}{\Gamma \vdash \pi\tau : \varphi}$$

$$\frac{\Gamma \vdash \pi : \forall x.\varphi}{\Gamma \vdash \pi t : \varphi[x/t]}$$

$$\frac{\Gamma, p : e \vdash \tau : \varphi}{\Gamma \vdash \lambda p.\tau : e \to \varphi}$$

$$\frac{\Gamma \vdash \tau : \varphi}{\Gamma \vdash \lambda x.\tau : \forall x.\varphi}$$

Figure 3.2: Typing rules for proof terms

$$\frac{}{\vdash \mathsf{axiom}_\varphi\ \overline{t} : \varphi[\overline{x}/\overline{t}]}\ \varphi \in E \cup \Delta$$

$$\frac{}{p : e \vdash p : e}$$

$$\frac{A \vdash p : \varphi}{A, p : e \vdash \varphi}$$

$$\frac{A, e \vdash \tau : \varphi}{A \vdash \lambda p.\tau : e \to \varphi}$$

$$\frac{A \vdash e \to \pi : \varphi \quad A \vdash \tau : e}{A \vdash \pi\ \tau : \varphi}$$

Figure 3.3: Annotated proof rules

3.2 Explicit Library Representation

In order to represent the set of proofs we can reuse explicitly, we add to the proof system in Section 3.1 a *library* $\mathfrak{L}$. The library is a list $T_1 = \pi_1, \ldots, T_n = \pi_n$, where T_i is a name given to an axiom in $E \cup \Delta$ or to a derived proof and π_i is a proof

term. Unlike the classical system of Section 3.1, we make universal quantification explicit.

In Figure 3.4 are the proof rules in the new system for creating and manipulating proofs. The rules allow one to build proofs constructively. They manipulate a structure of the form $\mathfrak{L};\mathfrak{T}$, where $\mathfrak{L}$ is the library and $\mathfrak{T}$ is a list of annotated *proof tasks* of the form $A \vdash \pi : \varphi$, where A is a set of annotated equations, π is a proof term, and φ is a formula.

$$\textbf{(assume)} \quad \frac{\mathfrak{L} \ ; \ \mathfrak{T}, \quad A \vdash \tau : \psi}{\mathfrak{L} \ ; \ \mathfrak{T}, \quad A, p : e \vdash \tau : \psi}$$

$$\textbf{(ident)} \quad \frac{\mathfrak{L} \ ; \ \mathfrak{T}}{\mathfrak{L} \ ; \ \mathfrak{T}, \quad p : e \vdash p : e}$$

$$\textbf{(mp)} \quad \frac{\mathfrak{L} \ ; \ \mathfrak{T}, \quad A \vdash \pi : e \ \rightarrow \ \psi \quad A \vdash \tau : e}{\mathfrak{L} \ ; \ \mathfrak{T}, \quad A \vdash \pi \ \tau : \psi}$$

$$\textbf{(discharge)} \quad \frac{\mathfrak{L} \ ; \ \mathfrak{T}, \quad A, p : e \vdash \tau : \psi}{\mathfrak{L} \ ; \ \mathfrak{T}, \quad A \vdash \lambda p.\tau : e \ \rightarrow \ \psi}$$

$$\textbf{(publish)} \quad \frac{\mathfrak{L} \qquad\qquad\qquad ; \ \mathfrak{T}, \quad \vdash \pi : \varphi}{\mathfrak{L}, T = \lambda \overline{x}.\pi : \forall \overline{x}.\varphi \ ; \ \mathfrak{T}} \ \overline{x} = FV(\varphi)$$

$$\textbf{(cite)} \quad \frac{\mathfrak{L}_1, T = \pi : \forall \overline{x}.\varphi, \mathfrak{L}_2 \ ; \ \mathfrak{T}}{\mathfrak{L}_1, T = \pi : \forall \overline{x}.\varphi, \mathfrak{L}_2 \ ; \ \mathfrak{T}, \quad \vdash \pi \ t_1 \ldots t_n : \varphi[\overline{x}/\overline{t}]}$$

$$\textbf{(forget)} \quad \frac{\mathfrak{L}_1, T = \pi : \varphi, \mathfrak{L}_2 \ ;}{\mathfrak{L}_1, \mathfrak{L}_2[T/\pi] \qquad ;} \ \varphi \notin E \cup \Delta$$

Figure 3.4: Proof Rules for Basic Theorem Manipulation

The **(publish)**, **(cite)**, and **(forget)** rules allow us to maintain our library

of theorems explicitly. The **(publish)** rule takes a proof task whose assumptions have all been discharged and forms the universal closure of φ and the corresponding λ-closure of π. It then adds the proof to the library $\mathfrak{L}$ with a new name T. Names of theorems must be unique in order to avoid conflicts.

The **(cite)** rule allows us to reuse a proof in the library. We now use names for theorems in addition to the axiom_φ constants. Referring to the proof by name means that we get a pointer to the proof in the library instead of a specialized copy of the proof itself.

It is important not to confuse the specialization of a theorem with the normalization of a proof. The former refers to a formula created by instantiating all universally quantified variables. The latter is a proof term applied to other proofs terms and on which β-reduction has then been performed.

Names are not used in the paper by Kozen and Ramanarayanan in order to avoid namespace management issues. Instead, a *citation token* is added to the proof terms. The proof term pub has the type $\varphi \to \varphi$, which maintains the type of a proof while preventing β-reduction during citation. We do not need the citation token as we use names to refer to theorems.

If we want to remove a theorem from the library, we use **(forget)**. This rule replaces all occurrences of the name of a theorem with its proof. β-reduction then reduces the application of the proof to a normal form. The result is a proof that appears as though we had performed all steps of deduction explicitly. We demonstrate the use of all of these rules in the next section.

3.3 An Example

Consider reasoning about a Boolean algebra $(B, \vee, \wedge, \neg, 0, 1)$. Boolean algebra is an equational theory, thus contains, among its axioms, the axioms of equality and idempotence for $\wedge$:

$$\mathsf{ref} \;:\; \forall x.\quad x = x \tag{3.2}$$

$$\mathsf{sym} \;:\; \forall x, y.\quad x = y \;\rightarrow\; y = x \tag{3.3}$$

$$\mathsf{trans} \;:\; \forall x, y, z.\quad x = y \;\rightarrow\; y = z \;\rightarrow\; x = z \tag{3.4}$$

$$\mathsf{cong}_{\wedge} \;:\; \forall x, y, z.\quad x = y \;\rightarrow\; (z \wedge x) = (z \wedge y) \tag{3.5}$$

$$\mathsf{cong}_{\vee} \;:\; \forall x, y, z.\quad x = y \;\rightarrow\; (z \vee x) = (z \vee y) \tag{3.6}$$

$$\mathsf{cong}_{\neg} \;:\; \forall x, y, z.\quad x = y \;\rightarrow\; \neg x = \neg y \tag{3.7}$$

$$\mathsf{idemp}_{\wedge} \;:\; \forall x.\quad x \wedge x = x \tag{3.8}$$

These axioms are present in our library. Let us prove a simple formula in this algebra:

$$\forall a. \forall b.\quad a \wedge b = a \;\rightarrow\; a \wedge b \wedge b = a \tag{3.9}$$

First, we use **(ident)** to introduce the task

$$p : a \wedge b = a \vdash p \;:\; a \wedge b = a \tag{3.10}$$

Next, we use **(cite)** to use $\mathsf{cong}_{\wedge}$.

$$\vdash \mathsf{cong}_{\wedge}\ (a \wedge b)\ a\ b \;:\; a \wedge b = a \;\rightarrow\; a \wedge b \wedge b = a \wedge b \tag{3.11}$$

We use **(assume)** on (3.11) so that it has the same assumption as (3.10)

$$p : a \wedge b = a \vdash \mathsf{cong}_{\wedge}\ (a \wedge b)\ a\ b \;:\; a \wedge b = a \;\rightarrow\; a \wedge b \wedge b = a \wedge b \tag{3.12}$$

Now we combine (3.12) and (3.10) using **(mp)**.

$$p : a\wedge b = a \vdash \mathsf{cong}_\wedge\ (a\wedge b)\ a\ b\ p\ :\ a\wedge b\wedge b = a\wedge b \tag{3.13}$$

We introduce another copy of our assumption with **(ident)**.

$$p : a\wedge b = a \vdash p\ :\ a\wedge b = a \tag{3.14}$$

Now we wish to use transitivity to conclude $a\wedge b\wedge b = a$ from (3.13) and (3.14). Therefore, we use **(cite)** to introduce a specialized version of trans.

$$\begin{aligned} \vdash \mathsf{trans}\ (a\wedge b\wedge b = a\wedge b)\ (a\wedge b = a)\ (a\wedge b\wedge b = a)\ :\quad & a\wedge b\wedge b = a\wedge b \\ \rightarrow\ & a\wedge b = a \\ \rightarrow\ & a\wedge b\wedge b = a \end{aligned} \tag{3.15}$$

Next, we use **(assume)** to add our single assumption to (3.15).

$$\begin{aligned} p : a\wedge b = a \vdash \mathsf{trans}\ (a\wedge b\wedge b = a\wedge b)\ (a\wedge b = a)\ (a\wedge b\wedge b = a)\ :\quad & a\wedge b\wedge b = a\wedge b \\ \rightarrow\ & a\wedge b = a \\ \rightarrow\ & a\wedge b\wedge b = a \end{aligned} \tag{3.16}$$

Now we apply **(mp)** to (3.16) and (3.13) to get

$$\begin{aligned} p : a\wedge b = a \vdash\ \mathsf{trans}\ & (a\wedge b\wedge b = a\wedge b) \\ & (a\wedge b = a) \\ & (a\wedge b\wedge b = a) \\ & (\mathsf{cong}_\wedge\ (a\wedge b)\ a\ b\ p) \end{aligned} \quad :\ a\wedge b = a\ \rightarrow\ a\wedge b\wedge b = a \tag{3.17}$$

We apply **(mp)** to (3.17) and (3.14) to get

$$\begin{aligned} p : a\wedge b = a \vdash\ \mathsf{trans}\ & (a\wedge b\wedge b = a\wedge b) \\ & (a\wedge b = a) \\ & (a\wedge b\wedge b = a) \\ & (\mathsf{cong}_\wedge\ (a\wedge b)\ a\ b\ p) \\ & p \end{aligned} \quad :\ a\wedge b\wedge b = a \tag{3.18}$$

We now apply **(discharge)** to abstract over the assumption p.

$$\begin{array}{lll} \vdash \lambda p.\ \mathsf{trans} & (a\wedge b\wedge b = a\wedge b) & :\ a\wedge b = a\ \rightarrow\ a\wedge b\wedge b = a \\ & (a\wedge b = a) & \\ & (a\wedge b\wedge b = a) & \\ & (\mathsf{cong}_{\wedge}\ (a\wedge b)\ a\ b\ p) & \\ & p & \end{array} \tag{3.19}$$

Finally, we use the **(publish)** command to add this theorem to our library. The new entry in the library would be as follows.

$$\begin{array}{lllll} T & = & \lambda a.\lambda b.\lambda p. & \mathsf{trans} & (a\wedge b\wedge b = a\wedge b) \\ & & & & (a\wedge b = a) \\ & & & & (a\wedge b\wedge b = a) \\ & & & & (\mathsf{cong}_{\wedge}\ (a\wedge b)\ a\ b\ p) \\ & & & & p \end{array} \tag{3.20}$$

Both T and the proof term it represents have the type

$$\forall a.\forall b.\quad a\wedge b = a\ \rightarrow\ a\wedge b\wedge b = a$$

Now that we have created a new theorem, we may wish to use it in another proof. We can use T to prove

$$\forall x.\forall y.\forall z.\quad (x\vee y)\wedge z = (x\vee y)\ \rightarrow\ (x\vee y)\wedge z\wedge z = (x\vee y) \tag{3.21}$$

We specialize T using the **(cite)** command and the substitution $[a/(x\vee y), b/z]$:

$$\vdash T\ (x\vee y)\ z\ :\ (x\vee y)\wedge z = (x\vee y)\ \rightarrow\ (x\vee y)\wedge z\wedge z = (x\vee y) \tag{3.22}$$

We can publish this theorem with the **(publish)** command, adding to our library the entry

$$U\quad =\quad \lambda x.\lambda y.\lambda z.\quad T\ (x\vee y)\ z\ :\ \forall x.\forall y.\forall z.\quad (x\vee y)\wedge z = (z\vee y)\ \rightarrow\ (x\vee y)\wedge z\wedge z = (x\vee y)$$

We now may choose to remove T from the library with the **(forget)** command. This does not affect the type of U; however, it does replace the single occurrence of T in U's proof with the proof of T:

$$
\begin{array}{lcll}
U & = & \lambda x.\lambda y.\lambda z. & : \quad \forall x.\forall y.\forall z. \\
 & & \left(\begin{array}{ll} \lambda a.\lambda b.\lambda p. & \\ \quad \mathsf{trans} & (a\wedge b\wedge b = a\wedge b) \\ & (a\wedge b = a) \\ & (a\wedge b\wedge b = a) \\ & (\mathsf{cong}_{\wedge}\ (a\wedge b)\ a\ b\ p) \\ & p \end{array}\right) & \quad (x\vee y)\wedge z = (x\vee y) \\
 & & & \quad \rightarrow\ (x\vee y)\wedge z\wedge z = (x\vee y) \\
 & & (x\vee y)\ z &
\end{array}
$$

When we apply β-reduction to this proof to get a normal form, we get a new proof for U:

$$
\begin{array}{lcll}
U & = & \lambda x.\lambda y.\lambda z.\lambda p. & : \quad \forall x.\forall y.\forall z. \\
 & & \quad \mathsf{trans}\ \ ((x\vee y)\wedge z\wedge z = (x\vee y)\wedge z) & \quad (x\vee y)\wedge z = (x\vee y) \\
 & & \quad\quad ((x\vee y)\wedge z = (x\vee y)) & \quad \rightarrow\ (x\vee y)\wedge z\wedge z = (x\vee y) \\
 & & \quad\quad ((x\vee y)\wedge z\wedge z = (x\vee y)) & \\
 & & \quad\quad (\mathsf{cong}_{\wedge}\ ((x\vee y)\wedge z)\ (x\vee y)\ z\ p) & \\
 & & \quad\quad p &
\end{array}
$$

This new proof is the same proof that would result from setting out to prove (3.21) directly instead of using T.

Chapter 4

KAT-ML

Work on the publish-cite system led to the development of KAT-ML, an interactive theorem prover for Kleene algebra with tests. Kleene algebra with tests (KAT), introduced in [63], is an equational system for program verification that combines Kleene algebra (KA), the algebra of regular expressions, with Boolean algebra. KAT has been applied successfully in various low-level verification tasks involving communication protocols, basic safety analysis, source-to-source program transformation, concurrency control, compiler optimization, and dataflow analysis [4, 15, 27, 26, 28, 63, 67]. This system subsumes Hoare logic and is deductively complete for partial correctness over relational models [64].

Much attention has focused on the equational theory of KA and KAT. The axioms of KAT are known to be deductively complete for the equational theory of language-theoretic and relational models. Validity is decidable in *PSPACE* [29, 69]. Because of the practical importance of premises, it is the universal Horn theory that is of more interest; that is, the set of valid sentences of the form

$$p_1 = q_1 \wedge \cdots \wedge p_n = q_n \rightarrow p = q, \tag{4.1}$$

where the atomic symbols are implicitly universally quantified. The premises $p_i = q_i$ are typically assumptions regarding the interaction of atomic programs and tests, and the conclusion $p = q$ represents the equivalence of an optimized and unoptimized program or of an unannotated and annotated program. The necessary premises are obtained by inspection of the program and their validity may depend on properties of the domain of computation, but they are usually quite simple and easy to verify by inspection, since they typically only involve atomic programs and

tests. Once the premises are established, the proof of (4.1) is purely propositional. This ability to introduce premises as needed is one of the features that makes KAT so versatile. By comparison, Hoare logic has only the assignment axiom for introducing non-propositional structure, which is significantly more limited. In addition, this style of reasoning allows a clean separation between first-order interpreted reasoning to justify the premises $p_i = q_i$ and purely propositional reasoning to establish that the conclusion $p = q$ follows from the premises.

The *PSPACE* decision procedure for the equational theory has been implemented by Cohen [27, 26, 28]. Cohen's approach is to try to reduce a Horn formula to an equation, then apply the *PSPACE* decision procedure to verify the resulting equation automatically. However, this reduction is not always possible.

KAT can also be used to reason about flowchart schemes in an algebraic framework. A *flowchart scheme* is a vertex-labeled graph that represents an uninterpreted program. This version of KAT, called *schematic* KAT (SKAT), was introduced in [4]. The semantics of SKAT coincides with the semantics of flowchart schemes over a ranked alphabet Σ. A translation to SKAT from a flowchart scheme is possible by considering the scheme to be a *schematic automaton*, a generalization of automata on guarded strings [66]. The equivalence of schematic automata and SKAT expressions, as well as the soundness of the method for scheme equivalence, are proven in [4].

Our system, KAT-ML, allows the user to develop a proof interactively in a natural human style, keeping track of the details of the proof. An unproven theorem has a number of outstanding *tasks* in the form of unproven Horn formulas. The initial task is the theorem itself. The user applies axioms and lemmas to simplify the tasks, which may introduce new (presumably simpler) tasks. When all tasks

are discharged, the proof is complete.

As the user applies proof rules, the system constructs a representation of the proof in the form of a λ-term. The proof term of an unproven theorem has free task variables corresponding to the undischarged tasks. The completed proof can be verified and exported to LaTeX. The system is based on the publish-cite system described in the previous chapter.

KAT-ML also has the capability of reasoning at the schematic level. One can input simple imperative programs, translate them to KAT, and then use propositional rules and theorems and schematic axioms to reason about the programs. The formal proof maintained in the system can be regarded as verification of the code's behavior. Other extensions of KAT such as von Wright's refinement algebra [107] or Kleene algebra with domain of Desharnais et al. [33] could be supported in the system with few changes.

We have verified formally several known results in the literature, some of which had previously been verified only by hand, including the KAT translation of the Hoare partial correctness rules [64], a verification problem involving a Windows device driver [11], and an intricate scheme equivalence problem [4]. The last is provided in this chapter as an extended example of the system's capabilities.

The system is implemented in Standard ML and is easy to install and use. Source code and executable images for various platforms are available. Several tutorial examples are also provided. The distribution is available from the KAT-ML website [1].

4.1 Preliminary Definitions

4.1.1 Kleene Algebra

Kleene algebra (KA) is the algebra of regular expressions [54, 30]. The axiomatization used here is from [62]. A *Kleene algebra* is an algebraic structure $(K, +, \cdot, {}^*, 0, 1)$ that satisfies the following axioms:

$$(p+q)+r = p+(q+r) \quad (4.2) \qquad (pq)r = p(qr) \quad (4.3)$$

$$p+q = q+p \quad (4.4) \qquad p1 = 1p = p \quad (4.5)$$

$$p+0 = p+p = p \quad (4.6) \qquad 0p = p0 = 0 \quad (4.7)$$

$$p(q+r) = pq+pr \quad (4.8) \qquad (p+q)r = pr+qr \quad (4.9)$$

$$1+pp^* \leq p^* \quad (4.10) \qquad q+pr \leq r \rightarrow p^*q \leq r \quad (4.11)$$

$$1+p^*p \leq p^* \quad (4.12) \qquad q+rp \leq r \rightarrow qp^* \leq r \quad (4.13)$$

This a universal Horn axiomatization. We use pq to represent $p \cdot q$. Axioms (4.2)–(4.9) say that K is an *idempotent semiring* under $+, \cdot, 0, 1$. The adjective *idempotent* refers to the axiom $p+p=p$ (4.6). Axioms (4.10)–(4.13) say that p^*q is the $\leq$-least solution to $q+px \leq x$ and qp^* is the $\leq$-least solution to $q+xp \leq x$, where $\leq$ refers to the natural partial order on K defined by $p \leq q \stackrel{\text{def}}{\Longleftrightarrow} p+q=q$.

Standard models include the family of regular sets over a finite alphabet, the family of binary relations on a set, and the family of $n \times n$ matrices over another Kleene algebra. Other more unusual interpretations include the min,+ algebra, also known as the *tropical semiring*, used in shortest path algorithms, and models consisting of convex polyhedra used in computational geometry.

There are several alternative axiomatizations in the literature, most of them infinitary. For example, a Kleene algebra is called *star-continuous* if it satisfies the infinitary property $pq^*r = \sup_n pq^nr$. This is equivalent to infinitely many

equations

$$pq^n r \quad \leq \quad pq^* r, \quad n \geq 0 \tag{4.14}$$

and the infinitary Horn formula

$$(\bigwedge_{n \geq 0} pq^n r \leq s) \quad \rightarrow \quad pq^* r \leq s. \tag{4.15}$$

All natural models are star-continuous. However, this axiom is much stronger than the finitary Horn axiomatization given above and would be more difficult to implement, since it would require meta-rules to handle the induction needed to establish (4.14) and (4.15).

The completeness result of [62] says that all true identities between regular expressions interpreted as regular sets of strings are derivable from the axioms. In other words, the algebra of regular sets of strings over the finite alphabet P is the free Kleene algebra on generators P. The axioms are also complete for the equational theory of relational models.

See [62] for a more thorough introduction.

4.1.2 Kleene Algebra with Tests

A *Kleene algebra with tests* (KAT) [63] is just a Kleene algebra with an embedded Boolean subalgebra. That is, it is a two-sorted structure $(K, B, +, \cdot, {}^*, {}^-, 0, 1)$ such that

- $(K, +, \cdot, {}^*, 0, 1)$ is a Kleene algebra,
- $(B, +, \cdot, {}^-, 0, 1)$ is a Boolean algebra, and
- $B \subseteq K$.

Elements of B are called *tests*. The Boolean complementation operator $^-$ is defined only on tests. In KAT-ML, variables beginning with an upper-case character denote tests, and those beginning with a lower-case character denote arbitrary Kleene elements.

The axioms of Boolean algebra are purely equational. In addition to the Kleene algebra axioms above, tests satisfy the equations

$$\begin{array}{rclcrcl} BC & = & CB & & BB & = & B \\ B + CD & = & (B + C)(B + D) & & B + 1 & = & 1 \\ \overline{B + C} & = & \overline{B}\,\overline{C} & & \overline{BC} & = & \overline{B} + \overline{C} \\ B + \overline{B} & = & 1 & & B\overline{B} & = & 0 \\ \overline{\overline{B}} & = & B & & & & \end{array}$$

The **while** program constructs are encoded as in propositional Dynamic Logic [37]:

$$\begin{array}{rcl} p\,;\,q & \stackrel{\text{def}}{=} & pq \\ \mathsf{if}\ B\ \mathsf{then}\ p\ \mathsf{else}\ q & \stackrel{\text{def}}{=} & Bp + \overline{B}q \\ \mathsf{while}\ B\ \mathsf{do}\ p & \stackrel{\text{def}}{=} & (Bp)^*\overline{B}. \end{array}$$

The Hoare partial correctness assertion $\{B\}\,p\,\{C\}$ is expressed as an inequality $Bp \leq pC$, or equivalently as an equation $Bp\overline{C} = 0$ or $Bp = BpC$. Intuitively, $Bp\overline{C} = 0$ says that there is no execution of p for which the input state satisfies the precondition B and the output state satisfies the postcondition $\overline{C}$, and $Bp = BpC$ says that the test C is always redundant after the execution of p under precondition B. The usual Hoare rules translate to universal Horn formulas of KAT. Under this translation, all Hoare rules are derivable in KAT; indeed, KAT is deductively complete for relationally valid propositional Hoare-style rules involving partial correctness assertions [64], whereas propositional Hoare logic is not.

The following simple example illustrates how equational reasoning with Horn formulas proceeds in KAT. To illustrate the use of KAT-ML, we will give a mechanical derivation of this proof in Section 4.2.5. The following equations are equivalent in KAT:

$$Cp = C \tag{4.16}$$

$$Cp + \overline{C} = 1 \tag{4.17}$$

$$p = \overline{C}p + C \tag{4.18}$$

Proof. We prove separately the four Horn formulas (4.16) → (4.17), (4.16) → (4.18), (4.17) → (4.16), and (4.18) → (4.16).

For the first, assume that (4.16) holds. Replace Cp by C on the left-hand side of (4.17) and use the Boolean algebra axiom $C + \overline{C} = 1$.

For the second, assume again that (4.16) holds. Replace the second occurrence of C on the right-hand side of (4.18) by Cp and use the distributive law $\overline{C}p + Cp = (\overline{C} + C)p$, the Boolean algebra axiom $\overline{C} + C = 1$, and the multiplicative identity axiom $1p = p$.

Finally, for (4.17) → (4.16) and (4.18) → (4.16), multiply both sides of (4.17) or (4.18) on the left by C and use distributivity and the Boolean algebra axioms $C\overline{C} = 0$ and $CC = C$ as well as (6) and (7).

□

See [63, 64, 70] for a more detailed introduction to KAT.

4.1.3 Schematic KAT

Schematic KAT (SKAT) is a specialization of KAT involving an augmented syntax to handle first-order constructs and restricted semantic actions whose intended se-

mantics coincides with the semantics of first-order flowchart schemes over a ranked alphabet Σ [4]. Atomic actions are assignment operations $\mathsf{x} := \mathsf{t}$, where x is a variable and t is a Σ-term.

Five identities are paramount in proofs using SKAT:

$$\mathsf{x} := \mathsf{s}; \mathsf{y} := \mathsf{t} \;=\; \mathsf{y} := \mathsf{t}[\mathsf{x}/\mathsf{s}]; \mathsf{x} := \mathsf{s} \quad (\mathsf{y} \notin FV(\mathsf{s})) \tag{4.19}$$

$$\mathsf{x} := \mathsf{s}; \mathsf{y} := \mathsf{t} \;=\; \mathsf{x} := \mathsf{s}; \mathsf{y} := \mathsf{t}[\mathsf{x}/\mathsf{s}] \quad (\mathsf{x} \notin FV(\mathsf{s})) \tag{4.20}$$

$$\mathsf{x} := \mathsf{s}; \mathsf{x} := \mathsf{t} \;=\; \mathsf{x} := \mathsf{t}[\mathsf{x}/\mathsf{s}] \tag{4.21}$$

$$\varphi[\mathsf{x}/\mathsf{t}]; \mathsf{x} := \mathsf{t} \;=\; \mathsf{x} := \mathsf{t}; \varphi \tag{4.22}$$

$$\mathsf{x} := \mathsf{x} \;=\; 1 \tag{4.23}$$

where x and y are distinct variables and $FV(\mathsf{s})$ is the set of variables occurring in s in (4.19) and (4.20). The notation $\mathsf{s}[\mathsf{x}/\mathsf{t}]$ denotes the result of substituting t for all occurrences of x in s. As special cases of (4.19) and (4.22), we have

$$\mathsf{x} := \mathsf{s}; \mathsf{y} := \mathsf{t} \;=\; \mathsf{y} := \mathsf{t}; \mathsf{x} := \mathsf{s} \quad (\mathsf{y} \notin FV(\mathsf{s}), \mathsf{x} \notin FV(\mathsf{t})) \tag{4.24}$$

$$\varphi; \mathsf{x} := \mathsf{t} \;=\; \mathsf{x} := \mathsf{t}; \varphi \qquad (\mathsf{x} \notin FV(\varphi)) \tag{4.25}$$

4.2 Description of the System

4.2.1 Rationale for an Independent Implementation

We might have implemented KAT in the context of an existing general-purpose automated deduction system such as NuPRL, Isabelle, or Coq. In fact, Isabelle has already been used to reason about Kleene algebra by several researchers. Struth formalizes Church-Rosser proofs in Kleene algebra and checks them using Isabelle [103, 102]. Kahl also works in Isabelle to create theories that could be used to reason about Kleene algebras [51]. He uses the Isar (Intelligible Semi-Automated Reasoning) language [88, 109, 16] and locales [12] to create and display proofs

for Kleene algebra and heterogeneous relational algebras. Other proof assistants such as PCP (Point and Click Proofs) [50] emphasize human interaction in proof creation over automation. The PCP system is designed with Javascript to run in a web browser. It facilitates the manual creation of proofs in several algebraic theories, including KA. The system is geared specifically towards web-based presentations of proofs in algebra courses, but does not provide any facility for proof reuse.

We initially considered implementing KAT in the context of NuPRL and MetaPRL [48] and expended considerable effort in this direction. However, we discovered that some aspects of these more complex and general systems make them less desirable for our purposes. Because of their complexity, they tend to have steep learning curves that make them impractical for novice users who just want to experiment with KAT by proving a few theorems. Our experience with NuPRL indicated that installing and learning the system require a level of effort that is prohibitive for all but the most determined user, and are difficult without expert assistence. Moreover, encoding KAT requires the translation of the primitive KAT constructs into the (quite different) primitive NuPRL constructs, a task requiring considerable design effort and orthogonal to our main interest. We were interested in providing a lighter-weight tool that would appeal to naive users, allowing them to quickly understand the system and begin proving theorems immediately. Indeed, an early version of KAT-ML was used successfully by students in an undergraduate course on automata theory to understand and manipulate regular expressions.

Furthermore, systems such as MetaPRL are meant to be general tools for reasoning in several different logics. Because of this generality, it is difficult to take advantage of the structure of a specialized logic such as KAT in the internal data

representation. For example, in KAT we know that addition and multiplication are associative, and we can draw advantage from this fact in the form of more efficient data structures for the representation of terms. In systems such as NuPRL, associativity is not built in, but must be programmed as axioms. Thus proofs contain many citations of associativity to rebalance terms, contributing to their complexity. Similarly, because KAT only deals with universal formulas, most of the infrastructure for quantifier manipulation can remain implicit.

For a theorem prover whose goal is to automate as many of steps as possible, these are not serious issues, but if the goal is to faithfully reflect the equational reasoning style specific to KAT used by humans, they are an undesirable distraction.

4.2.2 Overview of KAT-ML

KAT-ML is an interactive theorem prover for Kleene algebra with tests. It is written in Standard ML and is available for several platforms. The system has a command-line interface and a graphical user interface, pictured in Figure 4.1. A user can create and manage libraries of KAT theorems that can be proved and cited by name in later proofs. A few standard libraries containing the axioms of KAT and commonly used lemmas are provided. The system is freely available for downloading from the project website [1].

KAT-ML maintains a library of proofs that can be used easily, even by novices. We have used KAT-ML to verify several proofs in the literature, all of which are explained in detail in the distribution and on the KAT-ML website. KAT-ML has been used by others, including the author of [97], who installed, learned, and used the system to prove a theorem for his paper in only a few hours.

At the core of the KAT theorem prover are the commands *publish* and *cite*.

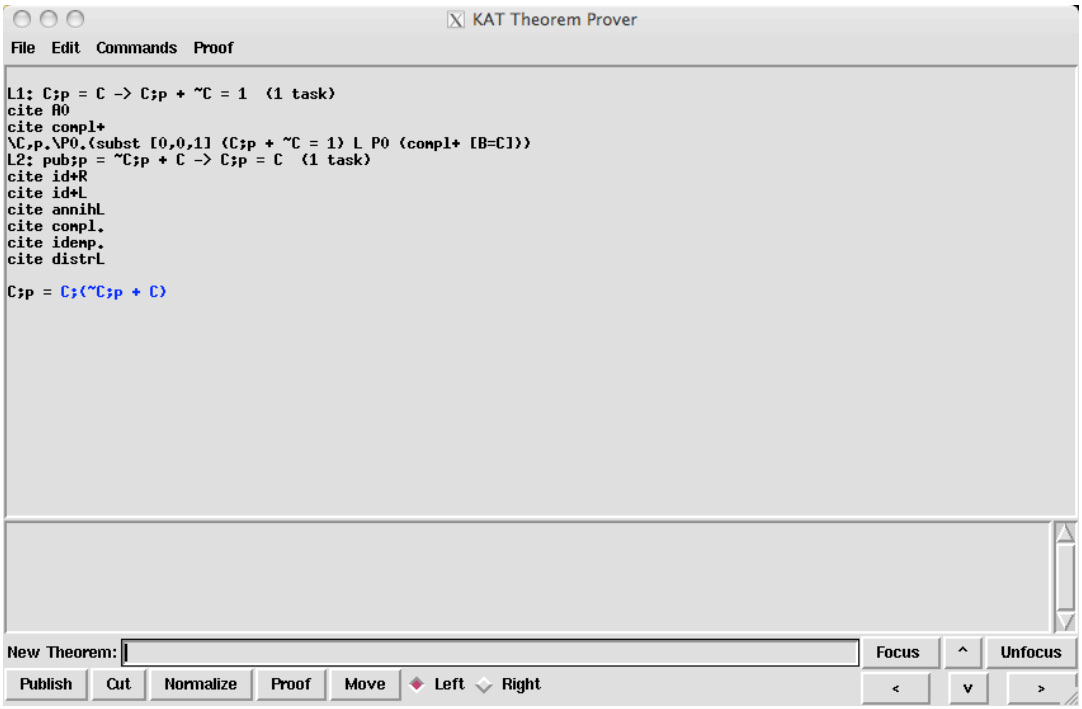


Figure 4.1: KAT-ML main window

Publication is a mechanism for making previous constructions available in an abbreviated form. Citation incorporates previously constructed objects in a proof without having to reconstruct them. All other commands relate to these two in some way. In contrast to other systems, in which these operations are typically implemented at the system level, in KAT-ML they are considered part of the underlying proof theory, as described in Chapter 3.

4.2.3 Representation of Proofs

KAT-ML is a constructive logic in which a theorem is regarded as a type and a proof of that theorem as an object of that type, according to the Curry–Howard Isomorphism [100]. Proofs are represented as λ-terms abstracted over variables $p, q, \ldots$ and $B, C, \ldots$ ranging over individual elements and tests, respectively, and

variables $P_0, P_1, \ldots$ ranging over proofs. If the proof is not complete, the proof term also contains free task variables $T_0, T_1, \ldots$ for the undischarged tasks. The theorem and its proof can be reconstructed from the proof term.

For instance, consider a theorem such a (3.1). Viewed as a type, this theorem would be realized by a proof term representing a function that takes an arbitrary substitution for the variables x_i and proofs of the premises d_j and returns a proof of the conclusion e. Initially, the proof is represented as the λ-term

$$\lambda x_1 \ldots \lambda x_m . \lambda P_1 \ldots \lambda P_n . (T\ P_1 \cdots P_n),$$

where T is a free variable of type $d_1 \to d_2 \to \cdots \to d_n \to e$ representing the main task. Publishing the theorem results in the creation of this initial proof term. As proof rules are applied, the proof term is expanded accordingly. Citing a theorem α in the proof of another theorem β is equivalent to substituting the proof term of α for a free task variable in the proof term of β. The proof of α need not be complete for this to happen; any undischarged tasks of α become undischarged tasks of β.

4.2.4 Citation

Citations are applied to the current task. One may cite a published theorem with the command *cite* or a premise of the current task with the command *use*.

The system allows two forms of citation, *focused* and *unfocused*. In unfocused citation, the conclusion of the cited theorem is matched with the conclusion of the current task, giving a substitution of terms for the individual and test variables of the cited theorem. This substitution is then applied to the premises of the cited theorem, and the current task is replaced with several new (presumably simpler)

tasks, one for each premise of the cited theorem. Each specialized premise of the cited theorem must now be proved under the premises of the original task.

For example, suppose the current task is

```
T6: p < r, q < r, r;r < r |- p;q + q;p < r
```

indicating that one must prove the conclusion $pq+qp \leq r$ under the three premises $p \leq r, q \leq r$, and $rr \leq r$ (in the display, the symbol `<` denotes less-than-or-equal-to $\leq$ and `;` denotes sequential composition). The proof term at this point is

```
\p,q,r.\P0,P1,P2.(T6 (P0,P1,P2))
```
(4.26)

(in the display, `\` represents λ). This means that `T6` should return a proof of $pq + qp \leq r$, given proofs `P0`, `P1`, and `P2` for the three premises.

An appropriate citation at this point would be the theorem

```
sup: x < z -> y < z -> x + y < z
```

The conclusion of `sup`, namely $x + y \leq z$, is matched with the conclusion of the task `T6`, giving the substitution $x = pq,\ \ y = qp,\ \ z = r$. This substitution is then applied to the premises of `sup`, and the old task `T6` is replaced by the new tasks

```
T7: p < r, q < r, r;r < r |- p;q < r

T8: p < r, q < r, r;r < r |- q;p < r
```

This operation is reflected in the proof term as follows:

```
\p,q,r.\P0,P1,P2.(sup [x=p;q y=q;p z=r] (T7 (P0,P1,P2),
                                         T8 (P0,P1,P2)))
```

This new proof term is a function of the same type as (4.26), but its body has been expanded to reflect the application of the theorem `sup`. The free task variables `T7` and `T8` represent the remaining undischarged tasks.

A premise can be cited with the command *use* only when the conclusion is identical to that premise, in which case the corresponding task variable is replaced with the proof variable of the cited premise.

Focused citation is used to implement the proof rule of substitution of equals for equals. In focused citation, a subterm of the conclusion of the current task is specified; this subterm is called the *focus*. The system provides a set of navigation commands to allow the user to focus on any subterm. When there is a current focus, any citation will attempt to match either the left- or the right-hand side of the conclusion of the cited theorem with the focus, then replace it with the specialized other side. As with unfocused citation, new tasks are introduced for the premises of the cited theorem. A corresponding substitution is also made in the proof term. In the event that multiple substitutions are possible, the system prompts the user with the available options and applies the one selected.

For example, suppose that the current task is

```
T0: p;q = 0 |- (p + q)* < q*;p*
```

The axiom

```
*R: x;z + y < z -> x*;y < z
```

would be a good one to cite. However, the system will not allow the citation yet, since there is nothing to match y. If the task were

```
T1: p;q = 0 |- (p + q)*;1 < q*;p*
```

then y would match 1. We can make this change by focusing on the left-hand side of the conclusion of `T0` and citing the axiom

```
id.R: x;1 = x
```

Focusing on the desired subterm gives

```
T0: p;q = 0 |- (p + q)* < q*;p*
               --------
```

where the focus is underlined. Now citing `id.R` matches the right-hand side with the focus and replaces it with the specialized left-hand side of `id.R`, yielding

```
T1: p;q = 0 |- (p + q)*;1 < q*;p*
               ----------
```

At this point we can apply `*R`.

Another useful rule is the *cut* rule. This rule adds a new premise σ to the list of premises of the current task and adds a second task to prove σ under the original premises. Starting from the task $\varphi_1, \ldots, \varphi_n \vdash \psi$, the command `cut` σ yields the new tasks

$$\varphi_1, \ldots, \varphi_n, \sigma \vdash \psi$$

$$\varphi_1, \ldots, \varphi_n \vdash \sigma.$$

4.2.5 An Extended Example

The following is an example of the system in use. It illustrates the interactive development of the implications (4.16)→(4.17) and (4.18)→(4.16) in the proof from Section 4.1.2. In the display, ˜ represents Boolean negation. The proof demonstrates basic publication and citation, focus, and navigation. For more examples of varying complexity, see the Examples directory in the KAT-ML distribution [1]. The command-line interface is used here instead of the graphical user interface for ease of reading.

```
>pub C p = C -> C p + ~C = 1
L0: C;p = C -> C;p + ~C = 1
(1 task)

current task:
T0: C;p = C |- C;p + ~C = 1
```

```
>proof
\C,p.\P0.(T0 P0)

current task:
T0: C;p = C |- C;p + ~C = 1

>focus

current task:
T0: C;p = C |- C;p + ~C = 1

C;p + ~C = 1
--------

>down

current task:
T0: C;p = C |- C;p + ~C = 1

C;p + ~C = 1
---

>use A0 l
cite A0

current task:
T1: C;p = C |- C + ~C = 1


C + ~C = 1
-

>unfocus

current task:
T1: C;p = C |- C + ~C = 1

>cite compl+
cite compl+
task completed

no tasks

>proof
\C,p.\P0.(subst [0,0,1] (C;p + ~C = 1)
   L P0 (compl+ [B=C]))

no tasks

>pub p = ~C p + C -> C p = C
L3: p = ~C;p + C -> C;p = C  (1 task)

current task:
T15: p = ~C;p + C |- C;p = C

>proof
\C,p.\P3.(T15 P3)

current task:
T15: p = ~C;p + C |- C;p = C

>focus

current task:
T15: p = ~C;p + C |- C;p = C

C;p = C
---

>r

current task:
T15: p = ~C;p + C |- C;p = C

C;p = C
      -

>cite id+L r
cite id+L

current task:
T16: p = ~C;p + C |- C;p = 0 + C

C;p = 0 + C
      -----

>d

current task:
T16: p = ~C;p + C |- C;p = 0 + C

C;p = 0 + C
      -

>cite annihL r
cite annihL
x=? p

current task:
T17: p = ~C;p + C |- C;p = 0;p + C

C;p = 0;p + C
      ---

>d

current task:
T17: p = ~C;p + C |- C;p = 0;p + C

C;p = 0;p + C
      -

>cite compl. r
cite compl.
B=? C

current task:
T18: p = ~C;p + C |- C;p = C;~C;p + C

C;p = C;~C;p + C
      ----

>u r

current task:
T18: p = ~C;p + C |- C;p = C;~C;p + C

C;p = C;~C;p + C
                -

>cite idemp. r
```

```
cite idemp.

current task:
T19: p = ~C;p + C |- C;p = C;~C;p + C;C

C;p = C;~C;p + C;C
               ---

>u

current task:
T19: p = ~C;p + C |- C;p = C;~C;p + C;C

C;p = C;~C;p + C;C
      ------------

>cite distrL r
cite distrL

current task:
T20: p = ~C;p + C |- C;p = C;(~C;p + C)

C;p = C;(~C;p + C)
      ------------

>unfocus

current task:
T20: p = ~C;p + C |- C;p = C;(~C;p + C)

>cite cong.L
cite cong.L
cite A0
task completed

no tasks

>proof
\C,p.\P3.(subst [1,1] (C;p = C) R
(id+L [x=C])
(subst [1,0,1] (C;p = 0 + C) R
(annihL [x=p]) (subst [1,0,0,1]
(C;p = 0;p + C) R
(compl. [B=C]) (subst [1,1,1]
(C;p = C;~C;p + C) R
(idemp. [B=C]) (subst [1,1]
(C;p = C;~C;p + C;C) R
(distrL [x=C y=~C;p z=C])
(cong.L [x=C y=p z=~C;p + C] P3))))))

no tasks
```

4.2.6 Heuristics and Reductions

KAT-ML has a set of simple heuristics to aid in proving theorems. It is true that a *PSPACE* decision procedure exists for the equational theory of KAT, including the ability to reduce some Horn formulas to equations, which we could have used to perform more steps automatically. However, its usefulness is limited. Only certain forms of premises can be reduced to equations. In fact, the Horn theory of star-continuous Kleene algebras and relational Kleene algebras is Π^1_1-complete [65, 47]. Even limited to premises of the form $ab = ba$, which express the commutativity of primitive operations and occur frequently in program equivalence proofs [26], these theories are undecidable. In general, the decidability of (not necessarily star-continuous) Kleene algebra with Horn formulas containing premises of this form is unknown. We decided to focus our attention on more practical heuristics for KAT-ML.

The heuristics can automatically perform unfocused citation with premises or

theorems in the library that have no premises (such as reflexivity) that match the current task. The system also provides a list of suggested citations from the library, both focused and unfocused, that match the current task and focus. Currently, the system does not attempt to order the suggestions, but only provides a list of possible citations.

In addition, KAT-ML has a more complex heuristic system called *reductions*. Reductions are sequences of citations of theorems and premises and focus motion carried out by the system. Reductions are derived from MetaPRL tactics for KAT [48]. A user can create new reductions, store them, and apply them manually or automatically. A reduction is *enabled* if it can be applied to the current task at the current focus.

The most basic reduction command either cites a theorem or moves the focus. The former is of the form *theorem side*, where *theorem* is the name of a theorem in the library and *side* is l or r, indicating which side should be used in the matching for a focused citation. The command *move direction* shifts focus left, right, up, or down, when *direction* is l, r, u, or d, respectively. The keyword *premises*, which is enabled if any of the premises of the current task can be used, is also a basic reduction.

Reductions can be combined as follows:

$red_1 + red_2$	is enabled if either red_1 or red_2 is enabled
$red_1\ red_2$	is enabled if red_1 is enabled, and after applying red_1, red_2 is enabled
$(red)^*$	is always enabled; it applies red as many times as possible.

There are several other special reductions for testing the result of other reductions without actually performing them. These reductions do not change the state

of the current task.

fails [red]	is true if *red* is not enabled
succeeds [red]	is true if *red* is enabled
match [term]	is true if the current focus matches the KAT term *term*.

With the addition of 0 and 1, it is not hard to verify that the language of reductions itself satisfies the axioms of KAT. The reductions *match* and *succeeds* are Boolean terms and *fails* has the same effect as the negation operator.

In the system preferences, it is possible to limit the length of time the system tries to apply reductions or specifically limit the number of times a *-reduction is applied to avoid circularities or nonterminating computations. The user has the ability to create and manage reductions and their application with the command `reduce`.

Reductions are meant to encapsulate common sequences of citations and changes of focus that would otherwise be done manually. For example, a standard sequence of citations in KAT uses premises and Boolean commutativity to move a Boolean term in one direction in a sequence of terms as far as possible, then eliminate it with idempotence. One could specify this reduction as

```
((commut. l + premises);move r)*;idemp. l
```

If the current task were

```
T6: A;b = b;A, A;c = c;A |- A;b;c;D;A = b;c;D;A
```

and the current focus were on `A;b`, the user could use the above reduction sequence to automatically get the new task

```
T7: A;b = b;A, A;c = c;A |- b;c;D;A = b;c;D;A
```

which can be completed with reflexivity of equality.

While our heuristics are not as extensive as the tactics present in several existing theorem provers, their simplicity allows them to be created and applied quickly and easily. We describe a more formal representation of tactics in Chapter 7.

4.2.7 Proof Output and Verification

Once a proof is complete, the system can export it in XML format. There is a separate postprocessor that translates the XML file to LaTeX source, which produces human-readable output. The exported proof correctly numbers and references assumptions and tasks and prints every step in the proof. With minimal alteration, one could incorporate the proof in a paper. Examples will be given later.

KAT-ML has a built-in verifier. It checks each step of the proof to make sure that it is valid and that there are no circularities in the library. The verifier also exists as a stand-alone program. One could use it to create a central repository of theorems, uploaded by users and verified by the system so that others could download and use them. We have created and tested a prototype of such a system. It is available on the KAT-ML website.

4.2.8 SKAT in KAT-ML

The KAT theorem prover has the ability to parse simple imperative programming language constructs and translate programs into propositional KAT. One may then cite the schematic axioms (4.19)–(4.23) to create and use premises automatically based on schematic properties. The schematic axioms are used only to establish premises used at the propositional level, where most of the reasoning is done.

The syntax for the imperative language is:

$$
\begin{aligned}
A &::= N \mid S \mid A + A \mid A - A \mid A * A \mid A \;/\; A \mid A \;\%\; A \mid S(L) \mid (A) \\
B &::= \mathsf{true} \mid \mathsf{false} \mid A = A \mid A <= A \mid A >= A \mid A > A \mid A < A \mid !B \\
&\quad \mid B \;\&\&\; B \mid B \;||\; B \mid (B) \\
C &::= S := A \mid \$B \mid \mathsf{if}\ (B)\ \mathsf{then}\ \{C\}\ \mathsf{else}\ \{C\} \mid \mathsf{while}\ (B)\ \mathsf{do}\ \{C\} \mid C; C
\end{aligned}
$$

Here A, B, and C denote arithmetic expressions, Boolean expressions, and imperative commands, respectively. N, S, and L correspond to the natural numbers, strings, and lists of arithmetic expressions, respectively. The arithmetic operations are addition ($+$), subtraction ($-$), multiplication ($*$), division ($/$), and mod ($\%$). $S(L)$ represents a function call with a list of arithmetic arguments. For Booleans, we have standard comparisons for arithmetic expressions ($=, <=, >=, <, >$) and the Boolean operators negation (!), conjunction (&&), and disjunction (||). The operator $\$B$ allows one to execute a Boolean expression as an imperative command or guard. Booleans are programs in KAT, which is very important in the creation of proofs. The \$ is used only to resolve an ambiguity in the grammar. A program is a statement C.

In the system, all commands related to first-order terms are managed in the first-order terms window, as seen in Figure 4.2. One can create a new theorem based on programs entered by the user, with any necessary premises. Upon publication of a theorem, KAT-ML maintains a translation table for the user, mapping KAT primitive propositions to assignments and Boolean tests. Once published, the user can create the proof using any of the applicable propositional axioms and theorems, as well as the schematic first-order axioms.

If a schematic axiom is cited, the system translates the necessary terms back

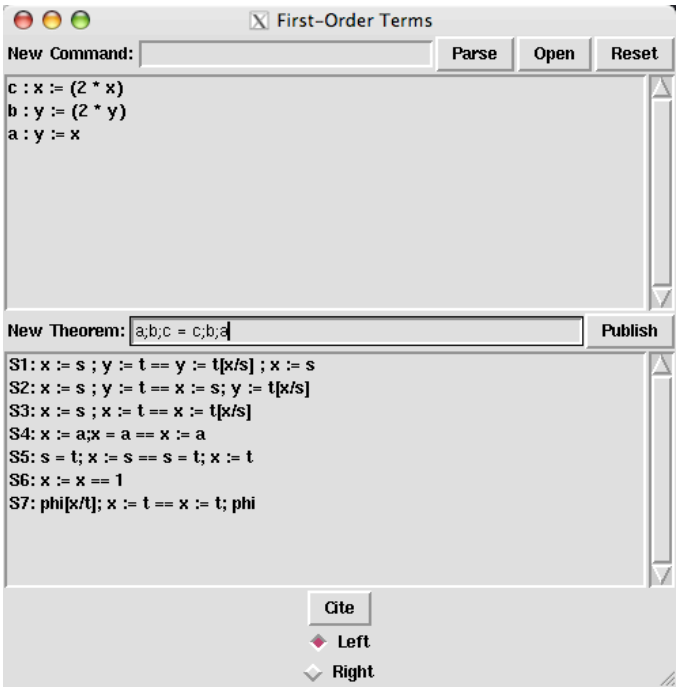


Figure 4.2: KAT-ML first-order window

into the first-order equivalents, matches them with the axiom, checks necessary preconditions (such as $\mathsf{x} \notin FV(\mathsf{s})$), and then replaces the terms with new terms. If necessary, KAT-ML makes propositions out of newly created first-order terms and adds them to the translation table. If the system cannot determine one of the expressions needed in the matching, it prompts the user to fill one in.

The citation of a first-order axiom (4.19)–(4.23) is a shortcut for steps normally done manually. The system creates a new premise φ and performs a cut, thereby creating two new tasks, one for the original conclusion with φ as an additional premise and one for proving the conclusion φ under the original premises. The system immediately proves the latter by replacing all occurrences of it in the proof by an application of the first-order axiom.

For example, consider the program $\mathsf{x} := 5 \ ; \ \mathsf{z} := \mathsf{x} + 7$. Assume that the system already has these assignments translated to propositional terms such that

a represents $\mathsf{x} := \mathsf{5}$ and b represents $\mathsf{z} := \mathsf{x} + \mathsf{7}$. We wish to apply the schematic axiom (4.19) to the term ab by matching ab with the left-hand side of (4.19). KAT-ML looks up a and b to find the first-order terms they represent. Next, it attempts to match the terms with $\mathsf{x} := \mathsf{s}; \mathsf{y} := \mathsf{t}$. It succeeds, matching x with x, s with $\mathsf{5}$, y with z, and t with $\mathsf{x} + \mathsf{7}$. The system then checks any necessary preconditions, in this case that x and z are distinct and that z is not a free variable of $\mathsf{5}$. These conditions are true, so the system creates a new term and makes propositional substitutions.

Now KAT-ML creates a new first-order term representing the right-hand side of the axiom with the appropriate substitutions made, giving $\mathsf{z} := \mathsf{5} + \mathsf{7}\ ;\ \mathsf{x} := \mathsf{5}$. The system creates a new primitive proposition c for $\mathsf{z} := \mathsf{5} + \mathsf{7}$ and translates the new program to the propositional term ca. Now the system performs a cut on the equation $ab = ca$. The first of the two new tasks created is $ab = ca$. It is replaced in the proof term by a special construct including the name of the first-order axiom used and the substitution, thus completing that task. In the other task, ab is replaced with ca using the new premise.

Sometimes first-order unification does not give a unique substitution. Consider trying to replace the assignment $\mathsf{x} := \mathsf{2} + \mathsf{5}$ with $\mathsf{x} := \mathsf{2}\ ;\ \mathsf{x} := \mathsf{x} + \mathsf{5}$ using (4.21). The system can match x with x and $\mathsf{t}[\mathsf{x}/\mathsf{s}]$ with $\mathsf{2} + \mathsf{5}$, but could choose from infinitely many possibilities for s. Consequently, the system asks the user to input the desired value for s, which is $\mathsf{2}$ in this case.

As a longer example, consider the following proof from [78]. We wish to prove the following two programs equivalent:

y := x;	x := 2 * x;
y := 2 * y;	y := 2 * y;
x := 2 * x	y := x

By hand, the proof requires two citations of (4.21) and one citation of (4.19). When we type the programs into KAT-ML, the system creates new propositions a,b, and c, corresponding to y := x, y := 2 * y, and x := 2 * x, respectively. The proof using the system is in Figure 4.3. The command-line interface is used for ease of reading and movement within the equation is suppressed.

We first focus on ab, which is y := x ; y := 2 * y. We then cite (4.21), matching with the left-hand side. This matches x with y, s with x, and t with 2 * y. After the substitution, the right-hand side becomes y := 2 * x, for which the system creates a new term d and uses the appropriate newly created assumption $ab = d$. Next, we move the focus to dc and cite (4.19), matching with the right side. As a result, we get the new assumption $ca = dc$, which is used to replace the focused term. Finally, we want to replace a with ba, so we focus on it and cite (4.21). In this case, the system matches x with y and t[x/s] with x. However, the system cannot find a unique substitution for s, so it asks the user to specify it. We want s to be 2 * y. Finally, we cite reflexivity of equality to complete the proof. Note how the proof term represents the citation of the schematic axioms as a substitution specifying the name of the axiom and the propositional term that represents each statement in the axiom.

The LaTeX output generated by the system for this theorem is in Figure 4.4. Here S1 and S3 refer to axiom (4.19) and (4.21), respectively.

```
current task:
T1:
     -------------------------
     a;b;c = c;b;a

a;b;c = c;b;a

>focus

no premises

current task:
T1:
     -------------------------
     a;b;c = c;b;a

a;b;c = c;b;a
---

>cite S3 l

cite S3
cite A0
no premises

current task:
T4:
     -------------------------
     d;c = c;b;a

d;c = c;b;a
---

>cite S1 r

cite S1
cite A0
no premises

current task:
T7:
     -------------------------
     c;a = c;b;a

c;a = c;b;a
  -
```

```
>cite S3 r

cite S3
s?2 * y
cite A0
no premises

current task:
T10:
     -------------------------
     c;b;a = c;b;a

c;b;a = c;b;a
  ---

>unfocus

no premises

current task:
T10:
     -------------------------
     c;b;a = c;b;a

c;b;a = c;b;a

>cite ref=

cite ref=
task completed

no tasks

>proof

\b,a,c,d.(subst [0,0,2] (a;b;c = c;b;a) L
(S3 [x := s=a x := t=b x :=t[x/s]=d])
(subst [0,1] (d;c = c;b;a) R
(S1 [y := t[x/s]=d x := s=c
x := s=c y := t=a])
(subst [0,1,1] (c;a = c;b;a) R
(S3 [x := t[x/s]=a x := s=b x := t=a])
(ref= [x=c;b;a]))))

no tasks
```

Figure 4.3: Proof steps for theorem from [78]

Theorem 1

$$\mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c} = \mathsf{c} \cdot \mathsf{b} \cdot \mathsf{a}$$

where

$$\begin{aligned} \mathsf{a} &= \mathtt{y} := \mathtt{x} \\ \mathsf{b} &= \mathtt{y} := (2 * \mathtt{y}) \\ \mathsf{c} &= \mathtt{x} := (2 * \mathtt{x}) \\ \mathsf{d} &= \mathtt{y} := (2 * \mathtt{x}) \end{aligned}$$

Proof. By S3, we know that

$$\mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c} = \mathsf{d} \cdot \mathsf{c}$$

By S1, we know that

$$\mathsf{d} \cdot \mathsf{c} = \mathsf{c} \cdot \mathsf{a}$$

By S3, we know that

$$\mathsf{c} \cdot \mathsf{a} = \mathsf{c} \cdot \mathsf{b} \cdot \mathsf{a}$$

By `ref=`, the proof is complete. □

Figure 4.4: Generated LaTeX output

4.2.9 A Schematic Example

Paterson presents the problem of proving the equivalence of the schemes in Figure 4.5. Manna proves the equivalence of the schemes by manipulating the structures of the graphs themselves [75]. Presented in [4] is a proof of the equivalence of the two schemes using the axioms of SKAT and algebraic reasoning. With KAT-ML, the citation of all of the SKAT axioms, with all variable substitutions, is handled

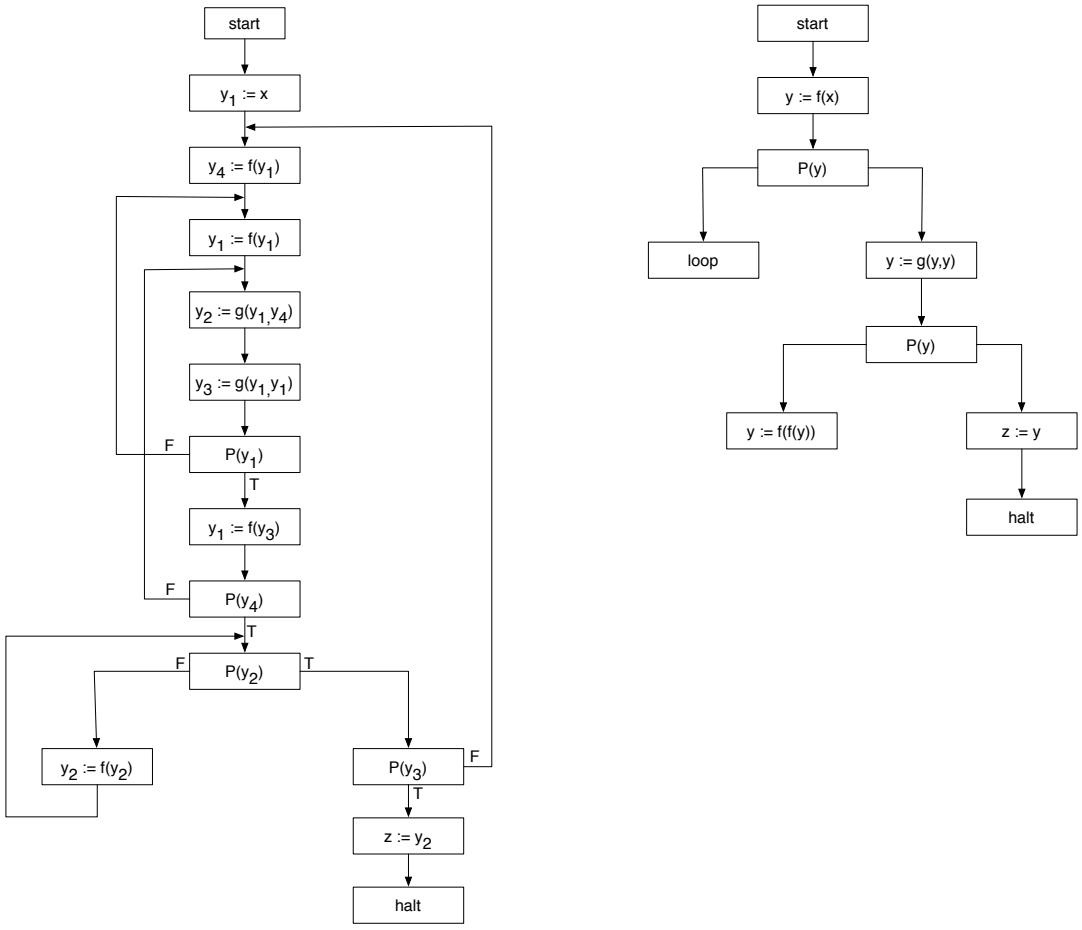


Figure 4.5: Schemes S_{6A} and S_{6E}

by the system.

Without the first-order axioms, it is still possible to prove the equivalence of these schemes. However, it requires that all of the citations of first-order axioms be determined in advance and added as premises to the theorem. The proof was completed successfully without the use of schematic axioms, with a total of 46 premises created manually.

While the proof is correct, it does not explain the origin of the premises. This would be desirable if the proof were distributed and independently verified. With the first-order level of reasoning, the system creates a special substitution in the

proof term to indicate that a first-order axiom was cited.

When using the first-order capabilities, we need only five premises, corresponding to the citation of specific lemmas proven in [4]. Once entered and translated by KAT-ML, the theorem we must prove is in Figure 4.6.

$$I; n; r; (\overline{C}; H; p; s)*; C; i = I; r; (\overline{C}; H; s)*; C; i \tag{4.27}$$

$$b; c; d; e; f; (\overline{B}; d; e; f + B; g; \overline{E}; e; f + B; g; E; (\overline{C}; h)*; C; \overline{D}; c; d; e; f)*; B; g; E;$$
$$(\overline{C}; h)*; C; D; i = b; c; d; e; f; (\overline{B}; d; e; f + B; g; \overline{E}; e; f + B; g; E;$$
$$(\overline{C}; h)*; C; \overline{D}; c; d; e; f)*; B; E; (\overline{C}; h)*; C; D; i \tag{4.28}$$

$$b; (c; d; t; f; B; g; (\overline{C}; h)*; C; \overline{D})*; c; d; t; f; (\overline{C}; h)*; B; C; D; i =$$
$$b; (d; t; f; B; g; (\overline{C}; h)*; C; \overline{D})*; d; t; f; (\overline{C}; h)*; B; C; D; i \tag{4.29}$$

$$n; (B; t; f; \overline{C}; p; (\overline{C}; h)*; C)*; t; f; C; (\overline{C}; h)*; B; C; i =$$
$$n; (B; t; \overline{C}; p; (\overline{C}; h)*; C)*; t; C; (\overline{C}; h)*; B; C; i \tag{4.30}$$

$$o; C; u; (\overline{C}; q; C; u)*; C; i = j; F; k; (\overline{F}; l; F; k)*; F; m \tag{4.31}$$

$$b; c; d; e; f; (\overline{B}; d; e; f)*; B; g; ((\overline{E} + E; (\overline{C}; h)*; C; \overline{D}; c; d); e; f; (\overline{B}; d; e; f)*; B; g)*;$$
$$E; (\overline{C}; h)*; C; D; i = j; F; k; (\overline{F}; l; F; k)*; F; m$$

Figure 4.6: Scheme equivalence theorem

The statement of the theorem is not meant to be read directly. The user enters a program at the first-order level. A translation table created by the system, shown for this example in Figure 4.7, can be used to interpret terms. The translation from automata to KAT expressions applies a generalized version of Kleene's theorem, as described in [4]. The premises (4.27)–(4.31) represent lemmas concerning variable elimination and renaming. These lemmas use properties of homomorphisms of KAT expressions, which cannot be handled by the system.

The proof proceeds exactly as in the original paper [4]. We highlight some of the advantageous uses of the system here.

One task that comes up frequently is of the form $a = a(A \leftrightarrow B)$, which says that A and B are equivalent after executing a. For instance, in the scheme equivalence

$B:$	$\mathsf{P}(\mathsf{y}_1) = 1$	$h:$	$\mathsf{y}_2 := \mathsf{f}(\mathsf{y}_2)$
$C:$	$\mathsf{P}(\mathsf{y}_2) = 1$	$i:$	$\mathsf{z} := \mathsf{y}_2$
$D:$	$\mathsf{P}(\mathsf{y}_3) = 1$	$j:$	$\mathsf{y} := \mathsf{f}(\mathsf{x})$
$E:$	$\mathsf{P}(\mathsf{y}_4) = 1$	$k:$	$\mathsf{y} := \mathsf{g}(\mathsf{y}, \mathsf{y})$
$F:$	$\mathsf{P}(\mathsf{y}) = 1$	$l:$	$\mathsf{y} := \mathsf{f}(\mathsf{f}(\mathsf{y}))$
$G:$	$\mathsf{P}(\mathsf{f}(\mathsf{y}_1)) = 1$	$m:$	$\mathsf{z} := \mathsf{y}$
$H:$	$\mathsf{P}(\mathsf{f}(\mathsf{f}(\mathsf{y}_2))) = 1$	$n:$	$\mathsf{y}_1 := \mathsf{f}(\mathsf{x})$
$I:$	$\mathsf{P}(\mathsf{f}(\mathsf{x})) = 1$	$o:$	$\mathsf{y}_2 := \mathsf{f}(\mathsf{x})$
$b:$	$\mathsf{y}_1 := \mathsf{x}$	$p:$	$\mathsf{y}_1 := \mathsf{f}(\mathsf{f}(\mathsf{y}_2))$
$c:$	$\mathsf{y}_4 := \mathsf{f}(\mathsf{y}_1)$	$q:$	$\mathsf{y}_2 := \mathsf{f}(\mathsf{f}(\mathsf{y}_2))$
$d:$	$\mathsf{y}_1 := \mathsf{f}(\mathsf{y}_1)$	$r:$	$\mathsf{y}_2 := \mathsf{g}(\mathsf{f}(\mathsf{x}), \mathsf{f}(\mathsf{x}))$
$e:$	$\mathsf{y}_2 := \mathsf{g}(\mathsf{y}_1, \mathsf{y}_4)$	$s:$	$\mathsf{y}_2 := \mathsf{g}(\mathsf{f}(\mathsf{f}(\mathsf{y}_2)), \mathsf{f}(\mathsf{f}(\mathsf{y}_2)))$
$f:$	$\mathsf{y}_3 := \mathsf{g}(\mathsf{y}_1, \mathsf{y}_1)$	$t:$	$\mathsf{y}_2 := \mathsf{g}(\mathsf{y}_1, \mathsf{y}_1)$
$g:$	$\mathsf{y}_1 := \mathsf{f}(\mathsf{y}_3)$	$u:$	$\mathsf{y}_2 := \mathsf{g}(\mathsf{y}_2, \mathsf{y}_2)$

Figure 4.7: Translation table for scheme proof

problem above, we need to prove that $tf = tf(C \leftrightarrow D)$, where

$$\begin{aligned} t &\text{ is } \mathsf{y_2} := \mathsf{g}(\mathsf{y_1}, \mathsf{y_1}), \\ f &\text{ is } \mathsf{y_3} := \mathsf{g}(\mathsf{y_1}, \mathsf{y_1}), \\ C &\text{ is } \mathsf{P}(\mathsf{y_2}) = \mathsf{1}, \text{ and} \\ D &\text{ is } \mathsf{P}(\mathsf{y_3}) = \mathsf{1}. \end{aligned}$$

We represent $C \leftrightarrow D$ as $(CD + \overline{C}\ \overline{D})$. The proof steps are given in Figure 4.2.9. Changes in focus have been suppressed.

The proof proceeds by using (4.22) and the laws of Boolean algebra. After citing distributivity, we use (4.22) to commute C and f, which is possible because $\mathsf{y_3} \notin FV(\mathsf{P}(\mathsf{y_2} = \mathsf{1}))$. However, when we apply the axiom to tC, x matches $\mathsf{y_2}$, which is a free variable in the Boolean test. Therefore, $\mathsf{y_2}$ is replaced by t, which is $\mathsf{g}(\mathsf{y_1},\mathsf{y_1})$, creating the new test $\mathsf{P}(\mathsf{g}(\mathsf{y_1},\mathsf{y_1})) = \mathsf{1}$, represented by the new term K. The other citations of (4.22) are similar to these two.

Once we have the Booleans on the left-hand side of each sequence, we use Boolean axioms to get the right-hand side of the equality to match the left-hand side, then cite reflexivity. The proof term (Figure 4.9) reflects our sequence of citations.

When doing the proof manually, it is easy to conclude that $tfC = tfD$, which is the actual step used in the full proof. However, formalizing this equality requires an additional cut and citations of distributivity and some rules related to Booleans.

Another common task is to commute a term through a star under certain assumptions:

$$ab = ba \rightarrow ac = ca \rightarrow (bc)^* a = a(bc)^* \tag{4.32}$$

```
t;f = t;f;(C;D + ~C;~D)

t;f = t;f;(C;D + ~C;~D)
      -----------------

cite distrL

t;f = t;f;C;D + t;f;~C;~D
        ---

cite S7
cite A0

t;f = t;C;f;D + t;f;~C;~D
      ---

cite S7
cite A1

t;f = K;t;f;D + t;f;~C;~D
          ---

cite S7
cite A2

t;f = K;t;K;f + t;f;~C;~D
        ---

cite S7
cite A3

t;f = K;K;t;f + t;f;~C;~D
      ---

cite idemp.

t;f = K;t;f + t;f;~C;~D
              ----

cite S7
cite A4

t;f = K;t;f + t;~C;f;~D
             ----

cite S7
cite A5

t;f = K;t;f + ~K;t;f;~D
                  ----

cite S7
cite A6

t;f = K;t;f + ~K;t;~K;f
                ----

cite S7
cite A7

t;f = K;t;f + ~K;~K;t;f
              -----

cite idemp.

t;f = K;t;f + ~K;t;f
      --------------

cite distrR

t;f = (K + ~K);t;f
      --------

cite compl+

t;f = 1;t;f
      -----

cite id.L

t;f = t;f
      ---

t;f = t;f

cite ref=

task completed
```

Figure 4.8: Proof steps for $tf = tf(C \leftrightarrow D)$

To prove this task, we first need antisymmetry,

$$x \leq y \rightarrow y \leq x \rightarrow x = y$$

Antisymmetry follows from transitivity, symmetry, and the definition of $\leq$. Once

```
\t,f,C,D,K.(subst [1,1] (t;f = t;f;(C;D + ~C;~D)) L (distrL [x=t;f y=C;D z=~C;~D])
(subst [1,0,1,2] (t;f = t;f;C;D + t;f;~C;~D) R (S7 [x := t=f &phi[x//t]=C x := t=f])
(subst [1,0,0,2] (t;f = t;C;f;D + t;f;~C;~D) R (S7 [x := t=t &phi[x//t]=K x := t=t])
(subst [1,0,2,2] (t;f = K;t;f;D + t;f;~C;~D) R (S7 [x := t=f &phi[x//t]=K x := t=f])
(subst [1,0,1,2] (t;f = K;t;K;f + t;f;~C;~D) R (S7 [x := t=t &phi[x//t]=K x := t=t])
(subst [1,0,0,2] (t;f = K;K;t;f + t;f;~C;~D) L (idemp. [B=K])
(subst [1,1,1,2] (t;f = K;t;f + t;f;~C;~D) R (S7 [x := t=f &phi[x//t]=~C x := t=f])
(subst [1,1,0,2] (t;f = K;t;f + t;~C;f;~D) R (S7 [x := t=t &phi[x//t]=~K x := t=t])
(subst [1,1,2,2] (t;f = K;t;f + ~K;t;f;~D) R (S7 [x := t=f &phi[x//t]=~K x := t=f])
(subst [1,1,1,2] (t;f = K;t;f + ~K;t;~K;f) R (S7 [x := t=t &phi[x//t]=~K x := t=t])
(subst [1,1,0,2] (t;f = K;t;f + ~K;~K;t;f) L (idemp. [B=~K])
(subst [1,1] (t;f = K;t;f + ~K;t;f) R (distrR [x=K y=~K z=t;f])
(subst [1,0,1] (t;f = (K + ~K);t;f) L (compl+ [B=K])
(subst [1,1] (t;f = 1;t;f) L (id.L [x=t;f]) (ref= [x=t;f])))))))))))))))
```

Figure 4.9: Proof term for $tf = tf(C \leftrightarrow D)$

we have antisymmetry, it suffices to show

$$\begin{aligned} (bc)^*a &\leq a(bc)^* \\ a(bc)^* &\leq (bc)^*a \end{aligned}$$

for proving (4.32). The proof steps for (4.33) are in Figure 4.10. The proof for (4.33) is similar. Since the task (4.32) is completely propositional in nature, we store it in the library as a separate theorem that we cite 13 times in the scheme equivalence proof.

The complete proof includes more than 50 proven tasks. When exported to LaTeX, the proof is 41 pages, compared to the 9 pages of the original, hand-constructed proof. The increased size is not unreasonable, given that it is a completely formal, mechanically developed and verified proof of one of Manna's most difficult examples.

4.3 Conclusions

We have described an interactive theorem prover for Kleene algebra with tests (KAT) that has as its formal basis the publication-citation system described in

```
(b;c)*;a < a;(b;c)*

cite *R

b;c;a;(b;c)* + a < a;(b;c)*
  ---

cite A1

b;a;c;(b;c)* + a < a;(b;c)*
---

cite A0

a;b;c;(b;c)* + a < a;(b;c)*
               -

cite id.R

a;b;c;(b;c)* + a;1 < a;(b;c)*
------------------
```

```
cite distrL

a;(b;c;(b;c)* + 1) < a;(b;c)*
  ----------------

cite commut+

a;(1 + b;c;(b;c)*) < a;(b;c)*
  ----------------

cite unwindL

a;(b;c)* < a;(b;c)*

cite =<

a;(b;c)* = a;(b;c)*

cite ref=
task completed
```

Figure 4.10: Proof steps for $(bc)^*a \leq a(bc)^*$

Chapter 3. The system provides an intuitive interface with simple commands that allow a user to learn the system quickly. We feel that the most interesting part of this work is not the particular data structures or algorithms we have chosen—these are fairly standard—but rather the design of the mode of interaction between the user and the system. We discuss theorem prover user interfaces in more detail in Chapter 6.

Our main goal was not to automate as much of the reasoning process as possible, but rather to provide support to the user for developing proofs in a natural human style, similar to proofs in KAT found in the literature. KAT is naturally equational, and equational reasoning pervades every aspect of reasoning with KAT. Our system is true to that style. The user can introduce self-evident equational premises describing the interaction of atomic programs and tests using SKAT and reason

under those assumptions to derive the equivalence of more complicated programs. The system performs low-level reasoning and bookkeeping tasks and facilitates sharing of theorems using a proof-theoretic library mechanism, but it is up to the user to develop the main proof strategies. Ultimately, KAT-ML could provide a user-friendly and mathematically sound apparatus for interactive code analysis and verification.

Chapter 5

Hierarchical Math Library Organization

In the scheme equivalence proof in Section 4.2.9, we published the formula (4.32) as a separate theorem in the library. While the theorem is relatively general, it has only been used as a lemma in the context of this larger scheme equivalence proof. Therefore, we may wish to limit its scope to establish its relationship to the entire theorem in the same way one would limit the scope of locally used variables in a program.

The relationship between theorems and lemmas in mathematical reasoning is often vague. What makes a statement a lemma, but not a theorem? One might say that a theorem is "more important," but what does it mean for one statement to be "more important" than another? When writing a proof for a theorem, we often create lemmas as a way to break down the complex proof, so perhaps we expect the proofs of lemmas to be shorter than the proofs of theorems. We also create lemmas when we have a statement that we do not expect to last in readers' minds, i.e., it is not the primary result of our work. The way we make these decisions while reasoning provides an inherent hierarchical structure to the set of statements we prove. However, no formal system exists that explicitly organizes proofs into this hierarchy.

Theorem provers such as Coq [105], NuPRL [71], Isabelle [108], and PVS [89] provide the ability to create lemmas. But their library structures are flat, and no formal distinction exists between lemmas and theorems. Any notion of scoping is only rudimentary in the form of modules, as described in Section 2.2.3. The reasons to distinguish lemmas from theorems in these systems is the same as the reasons

in papers: to ascribe various levels of importance and to introduce dependency or scoping relationships.

We seek to formalize these notions and provide a proof-theoretic means by which to organize a set of proofs in a hierarchical fashion that reflects this natural structure. Our thesis is that the qualitative difference between theorems and lemmas is in their *scope*. Scope already applies to mathematical notation. Never in a paper would one need to define the representation of a set ($\{\ldots\}$) nor operators such as union and intersection. Set notation is standard, thus has a global scope that applies to any proof. However, one often defines operators that are only used for a single paper; the author does not intend for the notation to exist in other papers with the same meaning without being defined again. Similarly, a *theorem* is a statement that can be used in any other proof. Its scope is global, just as set notation. A *lemma* is a statement with a local scope limited to a particular set of proofs. We want a system that represents and manipulates scope formally through the structure of the library of proofs.

In this chapter, we provide a representation that allows us to formalize the scoping of theorems, variables, and assumptions. The ability to create and manage complex scoping and dependency relationships among proofs will allow systems for formalized mathematics to more accurately reflect the natural structure of mathematical knowledge.

5.1 A Motivating Example

Consider reasoning about a Boolean algebra as in Section 3.3. Suppose we wanted to prove the following elementary fact:

$$\forall a \forall b \forall c \forall z. \quad a = b \;\rightarrow\; a = c \;\rightarrow\; z \vee (a \wedge b) = z \vee (a \wedge c) \tag{5.1}$$

Here is how a proof might go. First, we could prove a lemma

$$\forall x \forall y \forall z. \quad x = y \ \rightarrow \ z \vee (x \wedge x) = z \vee (x \wedge y) \tag{5.2}$$

Using $a = b$ and $a = c$ from the statement of our theorem, we could apply the lemma under the substitutions $[x/a, y/b, z/z]$ and $[x/a, y/c, z/z]$ to deduce

$$z \vee (a \wedge a) \ = \ z \vee (a \wedge b) \tag{5.3}$$

$$z \vee (a \wedge a) \ = \ z \vee (a \wedge c) \tag{5.4}$$

Next, we know from applying symmetry to (5.3) that

$$z \vee (a \wedge b) = z \vee (a \wedge a) \tag{5.5}$$

Finally we conclude from transitivity, (5.3), and (5.5) that

$$z \vee (a \wedge b) = z \vee (a \wedge c)$$

which is what our theorem states.

We may decide that (5.2) does not apply to theorems other than (5.1), and consequently, should only have a scope limited to the proof of (5.1). Our representation of proofs makes explicit the limited scope of (5.2).

Another important observation is that in all places we use (5.2), the variable z from (5.1) is always used for the variable z in the lemma. We may wish not to universally quantify z for both (5.1) and (5.2) individually, but instead universally quantify z once and for all so that it can be used by both proofs:

$$\begin{aligned} \forall z. \qquad & \forall a \forall b \forall c. \quad a = b \ \rightarrow \ a = c \ \rightarrow \ z \vee (a \wedge b) = z \vee (a \wedge c) \\ & \text{and } \forall x \forall y. \quad x = y \ \rightarrow \ z \vee (x \wedge x) = z \vee (x \wedge y) \end{aligned} \tag{5.6}$$

Moving the quantifier for z looks like a simple task, applying the first order logic rule

$$(\forall z. \varphi) \wedge (\forall z. \psi) \ \equiv \ \forall z. (\varphi \wedge \psi)$$

However, the proof of the lemma itself must also change, as must any proof that is dependent on this lemma.

Although either version of the lemma can be used to prove the theorem, note that their meanings are subtly different because of the placement of the quantification. Placing a separate quantification of z as in (5.2) makes the lemma read: "Lemma 1: For all x, y, and z,..." In this case, z is a variable in the lemma for which we expect there to be a substitution whenever the lemma is used in a proof. Using one quantification for both the theorem and the lemma as in (5.6) makes the lemma read: "Let z be an arbitrary, but fixed boolean value. Lemma 1: For all x and y..." In this case, z is a fixed constant for the lemma.

In this simple example, using (5.2) or (5.6) does not matter. However, in other cases, the choices made for quantification may reflect a general style in one's proofs. One may like lemmas to be as general as possible, universally quantifying any variables that appear in the lemma and relying on no constants. On the other hand, one may want to make lemmas as specific as possible, applying only in a select few proofs in order to minimize the number of quantifications. We want to capture this subtle difference formally in our representation of proofs in order to allow the user to choose the representation that best fits the intended meaning.

5.2 Proof Representation

In Chapter 3, a library of theorems is represented as a flat list of proof terms. All of the theorems have global scope, i.e., they are able to be cited in any other proof in the library. In this chapter, we use the word "theorem" to mean a theorem, lemma, or axiom.

The goal of this chapter is to provide a scoping discipline so that naming and

using variables can be localized. The proof term itself should tell us in which proofs we can use a lemma. We use a construct similar to the SML let expression, which limits the scope of variables in the same way we wish to limit the scope of lemmas.

In order to represent theorems in a hierarchical fashion, we add two kinds of proof terms to those in Section 3.2:

- a sequence $\tau_1; \ldots; \tau_n$, where $\tau_1, \ldots, \tau_n$ are proof terms. This allows several proofs to use the same lemmas. Sequences cannot occur inside applications.

- an expression $\mathsf{let}\ L_1 = \tau_1 \ldots L_n = \tau_n\ \mathsf{in}\ \tau\ \mathsf{end}$. This term is meant to express the definition of a set of lemmas for use in a proof term τ. The τ_i are proof terms, each bound to an identifier L_i. With the existence of the sequences, each τ_i may define the proof for more than one lemma. The identifiers L_i are arrays, where the j^{th} element, denoted $L_i[j]$, is the name of the lemma corresponding to the j^{th} proof in τ_i not bound to a name in τ_i, denoted $\tau_i[j]$.

  The let expression binds names to the proofs and limits their scope to proof terms that appear later in the let expression. In other words, a lemma $L_i[j]$ can appear in any proof $\tau_k, k > i$, or in τ. The name of a lemma has the same type as the proof to which it corresponds. This scoping discipline for lemmas corresponds exactly to the variable scoping used in SML let expressions.

These new rules have corresponding typing rules, in Figure 5.1.

The rule for a sequence of proof terms is relatively straightforward; the type of a sequence is the conjunction of the types of the proof terms in the sequence. The typing rule for the let expression is based on the scoping of the proofs. We must be able to prove that each proof τ_k has type φ_k under the assumption that all variables $L_i, i < k$ have the type φ_i, where τ_i is assigned to L_i. Finally, we must

$$\frac{\Gamma \vdash \tau_1 : \varphi_1 \quad \ldots \quad \Gamma \vdash \tau_n : \varphi_n}{\Gamma \vdash \tau_1; \ldots; \tau_n : \varphi_1 \wedge \ldots \wedge \varphi_n}$$

$$\frac{\begin{array}{l}\Gamma \vdash \tau_1 : \varphi_1 \\ \Gamma, L_1 : \varphi_1 \vdash \tau_2 : \varphi_2 \\ \ldots \\ \Gamma, L_1 : \varphi_1, \ldots, L_{n-1} : \varphi_{n-1} \vdash \tau_n : \varphi_n \\ \Gamma, L_1 : \varphi_1, \ldots, L_n : \varphi_n \vdash \tau : \varphi\end{array}}{\Gamma \vdash \mathsf{let}\ L_1 = \tau_1 \ldots L_n = \tau_n\ \mathsf{in}\ \tau\ \mathsf{end} : \varphi}$$

Figure 5.1: Typing rules for proof terms

be able to prove that τ has the type φ under the assumption that every L_i has type φ_i.

As an example, we represent the proofs of (5.1) and (5.2) as

$$\begin{array}{l}\mathsf{thm} = \\ \quad \mathsf{let}\ \mathsf{lem} = \lambda x \lambda y \lambda z \lambda p.(\text{Proof of lemma}) \\ \quad \mathsf{in} \\ \quad\quad \lambda a \lambda b \lambda c \lambda z \lambda q \lambda r.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ q))\ (\mathsf{lem}\ r) \\ \quad \mathsf{end}\end{array} \tag{5.7}$$

where thm is the name assigned to (5.1) and lem is the name assigned to (5.2). For ease of reading, we have omitted the applications of proof terms to individual terms, which represent the substitution for individual variables. The proof variables p, q, and r are proofs of type $x = y$, $a = b$, and $a = c$, respectively.

If we choose to universally quantify z only once as in (5.6), we represent the

proof as

$$\begin{aligned}&\mathsf{thm} = \\ &\quad \lambda z.\mathsf{let\ lem} = \lambda x\lambda y\lambda p.(\text{Proof of lemma}) \\ &\quad\quad \mathsf{in} \\ &\quad\quad\quad \lambda a\lambda b\lambda c\lambda q\lambda r.\mathsf{trans\ (sym\ (lem}\ q))\ (\mathsf{lem}\ r) \\ &\quad\quad \mathsf{end}\end{aligned} \tag{5.8}$$

As we can see, there is a one-to-one correspondence between the positions of λ-abstractions and where individual variables are universally quantified. We formally develop the proof terms for thm and lem in Section 5.6.

It is interesting to note that we take the let construct to be primitive in our proof term language. An alternative approach would have been to translate in a standard way, i.e.

$$\mathsf{let}\ L = \pi\ \mathsf{in}\ \tau\ \mathsf{end}\ \equiv\ (\lambda L.\tau)\pi$$

In order to allow such a translation, we would have had to allow abstractions over arbitrary formulas instead of only equations. In fact, such an approach will be useful in Chapter 7 when we look at tactics. For the purposes of local theorem scoping, there is a subtle difference between making the let expression primitive and translating it to an application of a λ-abstraction. The former provides a specific proof for a theorems and binds them to names used in the body of the let expression. The latter replaces all occurrences of the theorem name with a proof of the theorem itself, thus creating a specialized version of the proof through β-reduction.

One of the primary goals set forth in Chapter 3 was to use a library with named theorems in order to avoid β-reduction so that we could keep track of the reuse of

theorems. Using a primitive let expression allows us to stay true to that goal.

From the perspective of presentation, the two approaches are different. Consider a let expression

$$\mathsf{let}\ L = \pi : \varphi\ \mathsf{in}\ \tau : \psi\ \mathsf{end}$$

The primitive let is equivalent to giveing φ the name L, proving it via π, and then proving ψ via τ with references to L. The translation $((\lambda L : \varphi.\tau)\pi) : \psi$ would be equivalent to saying that one can prove ψ via τ given a proof of φ. The proof of φ provided in this case is π, although we could choose to provide any proof with the type φ.

5.3 Proof Rules

We provide several rules for creating and manipulating proofs. The rules allow one to build proofs constructively. They manipulate a structure of the form $\mathfrak{L}; \mathfrak{C}; \mathfrak{T}$, where

- $\mathfrak{L}$ is the library of theorems, $T_1 = \pi_1, \ldots, T_n = \pi_n$, where T_i is an array of identifiers with the j^{th} element denoted $T_i[j]$, naming the j^{th} proof in π_i, denoted $\pi_i[j]$,
- $\mathfrak{C}$ is the list of lemmas currently in scope, $L_1 = \tau_1, \ldots, L_m = \tau_m$, with components defined as they are for $\mathfrak{L}$, and
- $\mathfrak{T}$ is a list of annotated *proof tasks* of the form $A \vdash \pi : \varphi$, where A is a set of assumptions, π is a proof term, and φ is an unquantified Horn formula.

In these rules, we use the following notational conventions:

- α and β are proof variables or individual variables.

- $\overline{X}$ is a set of elements $\{X_1, \ldots, X_n\}$, where X_i can be an individual variable or a proof variable.

- $T = \pi$ binds a proof term π to an identifier T. The term π may define the proof for more than one theorem. Therefore, the identifier T is an array, where the j^{th} element, denoted $T[j]$, is the name of the theorem corresponding to the j^{th} proof in π not bound to a name in π, denoted $\pi[j]$.

- $\overline{T = \pi}$ is a sequence of bindings $T_1 = \pi_1, \ldots, T_n = \pi_n$.

- $\overline{T : \varphi}$ is a sequence of type bindings $T_1 : \varphi_1, \ldots, T_n : \varphi_n$.

- $\pi[\overline{x}/\overline{t}]$ means for all i, replace element $x_i \in \overline{x}$ in π with $t_i \in \overline{t}$.

- Given a binding $T = \pi$, $X[T/\pi]$ means for all i, replace $T[i]$ with $\pi[i]$ in X, where X is a proof term, a list of theorems, or a list of proof tasks.

- For a proof term π, a sequence of identifiers $\overline{T} = T_1 \ldots T_n$, and a variable α, $\pi[\overline{T}/\overline{T\ \alpha}]$ means for all i and j, replace $T_i[j]$ with $T_i[j]\ \alpha$, where juxtaposition represents functional application. We use $\pi[\overline{T\ \alpha}/\overline{T}]$ to denote this substitution in the other direction.

- Given a binding $T = \ldots \lambda\alpha_i \lambda\alpha_j \ldots \pi$, $\mathcal{C}[T(i,j)/T(j,i)]$ means for all k, swap the i^{th} and j^{th} term or proof to which $T[k]$ is applied in $\mathcal{C}$.

- $FV(\varphi)$ is the set of free individual variables in the Horn formula φ.

The structure $\mathfrak{L}; \mathcal{C}; \mathcal{T}$ must also be well typed, according to the rules in Figure 5.2. The typing rules enforce an order on the list of theorems and lemmas. The rules look very similar to the rules for the let expression.

$$\frac{\begin{array}{l}\Gamma \vdash \pi_1 : \varphi_1 \\ \Gamma, T_1 : \varphi_1 \vdash \pi_2 : \varphi_2 \\ \ldots \\ \Gamma, T_1 : \varphi_1, \ldots, T_{n-1} : \varphi_{n-1} \vdash \pi_n : \varphi_n\end{array}}{\Gamma \vdash \overline{T = \pi} : \varphi_1 \to \ldots \to \varphi_n}$$

$$\frac{\begin{array}{l}\Gamma \vdash \overline{T = \pi} : \varphi_{T1} \to \cdots \to \varphi_{Tn} \\ \Gamma, \overline{T : \varphi_T} \vdash \overline{L = \tau} : \varphi_{L1} \to \cdots \to \varphi_{Lm} \\ \Gamma, \overline{T : \varphi_T}, \overline{L : \varphi_L} \vdash \mathfrak{T} : \psi\end{array}}{\Gamma \vdash \overline{T = \pi}; \overline{L = \tau}; \mathfrak{T} : \varphi_{T1} \to \cdots \to \varphi_{Tn} \to \varphi_{L1} \to \cdots \to \varphi_{Lm} \to \psi}$$

Figure 5.2: Typing rules for proof library

The proof rules fit into two categories: rules that manipulate the proof tasks and rules that manipulate the structure of proof terms that appear in $\mathcal{C}$.

5.3.1 Rules for Manipulating Proof Tasks

The first set of rules is in Figure 5.3. These first four rules are very similar to the ones in Chapter 3. Rules dealing with the manipulation of the proof library are in Figure 5.4. Note that the **(reorder)** rule has a side condition $(*)$ explained below.

The **(collect)** rule works on a set of tasks with no further assumptions, i.e., tasks with completed proofs. The rule

1. gives the collection of the tasks a new name L that does not appear in the library or the current list of lemmas,

2. forms the universal closures of the φ_is and the corresponding λ-closures of

$$\textbf{(assume)}\quad \frac{\mathfrak{L}\ ;\ \mathcal{C}\ ;\ \mathcal{T},\quad A \vdash \tau : e}{\mathfrak{L}\ ;\ \mathcal{C}\ ;\ \mathcal{T},\quad A, p : d \vdash \tau : e}$$

$$\textbf{(ident)}\quad \frac{\mathfrak{L}\ ;\ \mathcal{C}\ ;\ \mathcal{T}}{\mathfrak{L}\ ;\ \mathcal{C}\ ;\ \mathcal{T},\quad p : e \vdash p : e}$$

$$\textbf{(mp)}\quad \frac{\mathfrak{L}\ ;\ \mathcal{C}\ ;\ \mathcal{T},\quad A \vdash \pi : e \to \varphi \quad A \vdash \tau : e}{\mathfrak{L}\ ;\ \mathcal{C}\ ;\ \mathcal{T},\quad A \vdash \pi\ \tau : \varphi}$$

$$\textbf{(discharge)}\quad \frac{\mathfrak{L}\ ;\ \mathcal{C}\ ;\ \mathcal{T},\quad A, p : e \vdash \tau : \varphi}{\mathfrak{L}\ ;\ \mathcal{C}\ ;\ \mathcal{T},\quad A \vdash \lambda p.\tau : e \to \varphi}$$

Figure 5.3: Rules for manipulating proof tasks

the τ_is, and

3. moves the proofs to the list of lemmas currently in scope.

Any lemmas that were in scope for the proof tasks are explicitly made lemmas with the let statement. These lemmas are no longer immediately available to proof tasks. However, one can access a lemma moved into a let by using the **(promote)** rule. If no lemmas currently exist, a let expression is not created and instead the name L is bound to the λ-closures of the τ_is.

The **(publish)** rule moves the current lemmas to the library, at which point they become theorems.

The **(tcite)** rule is the elimination rule for the universal quantifier for theorems in the library. This rule specializes the theorem with a given substitution $[\overline{x}/\overline{t}]$. It is important to note that the proof $\pi_i[j]$ of $T_i[j]$ is not copied into the proof tasks. As in Section 3.2, the name of the theorem serves as a citation token, with the same type as the proof itself. The **(lcite)** rule does the same for lemmas from $\mathcal{C}$.

$$\textbf{(collect)}\quad \frac{\mathcal{L}\ ;\ \overline{M=\pi}\ ;\ \vdash \tau_1 : \varphi_1 \ldots \vdash \tau_n : \varphi_n}{\mathcal{L}\ ;\ L = \ \mathsf{let}\ \overline{M=\pi}\ \mathsf{in}\ \lambda\overline{x}_1.\tau_1; \ldots ; \lambda\overline{x}_n.\tau_n\ \mathsf{end}\ ;} \quad \overline{x}_i = FV(\varphi_i)$$

$$\textbf{(publish)}\quad \frac{\mathcal{L}\ ;\ \overline{L=\tau}\ ;}{\mathcal{L}, \overline{L=\tau}\ ;\ ;}$$

$$\textbf{(tcite)}\quad \frac{\mathcal{L}_1, T=\pi, \mathcal{L}_2\ ;\ \mathcal{C}\ ;\ \mathcal{T}}{\mathcal{L}_1, T=\pi, \mathcal{L}_2\ ;\ \mathcal{C}\ ;\ \mathcal{T},\ \ \vdash T[j]\ \overline{t} : \varphi[\overline{x}/\overline{t}]} \quad T[j] : \forall\overline{x}.\varphi$$

$$\textbf{(lcite)}\quad \frac{\mathcal{L}\ ;\ \mathcal{C}_1, L=\pi, \mathcal{C}_2\ ;\ \mathcal{T}}{\mathcal{L}\ ;\ \mathcal{C}_1, L=\pi, \mathcal{C}_2\ ;\ \mathcal{T},\ \ \vdash L[j]\ \overline{t} : \varphi[\overline{x}/\overline{t}]} \quad L[j] : \forall\overline{x}.\varphi$$

$$\textbf{(tforget)}\quad \frac{\mathcal{L}_1, T=\pi, \mathcal{L}_2\ ;\ \mathcal{C}\ ;\ \mathcal{T}}{\mathcal{L}_1, \mathcal{L}_2[T/\pi]\ ;\ \mathcal{C}[T/\pi]\ ;\ \mathcal{T}[T/\pi]}$$

$$\textbf{(lforget)}\quad \frac{\mathcal{L}\ ;\ \mathcal{C}_1, L=\pi, \mathcal{C}_2\ ;\ \mathcal{T}}{\mathcal{L}\ ;\ \mathcal{C}_1, \mathcal{C}_2[L/\pi]\ ;\ \mathcal{T}[L/\pi]}$$

$$\textbf{(promote)}\quad \frac{\mathcal{L}\ ;\ \mathcal{L}_1, L = \mathsf{let}\ \overline{M=\tau}\ \mathsf{in}\ \pi\ \mathsf{end}, \mathcal{L}_2\ ;}{\mathcal{L}\ ;\ \mathcal{L}_1, \overline{M=\tau}, L=\pi, \mathcal{L}_2\ ;}$$

$$\textbf{(reorder)}\quad \frac{\mathcal{L}\ ;\ \mathcal{C}_1, L = \lambda\alpha_1 \ldots \lambda\alpha_i\lambda\alpha_j \ldots \lambda_n.\pi, \mathcal{C}_2\ ;}{\mathcal{L}\ ;\ \mathcal{C}_1, L = \lambda\alpha_1 \ldots \lambda\alpha_j\lambda\alpha_i \ldots \lambda_n.\pi, \mathcal{C}_2[L(i,j)/L(j,i)]\ ;} \quad (*)$$

Figure 5.4: Rules for manipulating the proof library

The **(tforget)** rule removes all citations of the forgotten theorems and replaces them with the proofs of the theorems. All citations of the theorems $T[1], \ldots, T[n]$ are replaced with a specialized version of the corresponding proof $\pi[1], \ldots, \pi[n]$. The **(lforget)** rule does the same for lemmas in $\mathcal{C}$.

The **(promote)** rule moves a set of lemmas from inside a let expression to the

list of lemmas currently in scope. This makes these lemmas again available to be cited.

The **(reorder)** rule changes the order of abstractions in a proof term. Correspondingly, citations of any lemmas defined by that proof term must be changed to have the order of their applications changed. The condition $(*)$ is that if α_i is an individual variable and α_j is a proof variable with type φ, then α_i does not occur anywhere in φ. If α_i did occur in φ and we performed **(reorder)**, φ would contain an unbound variable.

5.3.2 Rules for Manipulating Proof Terms

The set of rules for manipulating proof terms that appear in $\mathcal{C}$ is in Figure 5.5. These rules do not change any proofs of theorems currently in scope for the proof tasks, so we know that any changes in proofs do not have to be reflected in the current tasks. Some of these rules have side conditions, which are marked with a symbol in $(\cdot)$ and explained below.

The **(push)** rule moves an abstraction from the front of a sequence to each proof in the sequence. This rule does not change the types of the proofs; it only duplicates $\lambda\alpha$. One would anticipate using this rule after performing a **(generalize)**.

The **(pull)** rule is the inverse of the **(push)** rule. It moves an abstraction from the front of every proof in a sequence to the front of the entire sequence. This rule would most likely be used before a **(specialize)**.

The **(generalize)** rule moves an abstraction from the outside of a let statement to each proof term in the list of defined lemmas and to the proof term τ. This does not change any theorem whose proof is in τ. The proofs and types of the lemmas $\overline{L}$ do change, because they are now abstracted over another variable.

$$\textbf{(push)} \quad \frac{\lambda\alpha.(\pi_1;\ldots;\pi_n)}{\lambda\alpha.\pi_1;\ldots;\lambda\alpha.\pi_n}$$

$$\textbf{(pull)} \quad \frac{\lambda\alpha.\pi_1;\ldots;\lambda\alpha.\pi_n}{\lambda\alpha.(\pi_1;\ldots;\pi_n)}$$

$$\textbf{(generalize)} \quad \frac{\lambda\alpha.\mathsf{let}\ \overline{L=\pi}\ \mathsf{in}\ \tau\ \mathsf{end}}{\mathsf{let}\ \overline{L=\lambda\alpha.\pi[\overline{L}/\overline{L\ \alpha}]}\ \mathsf{in}\ \lambda\alpha.\tau[\overline{L}/\overline{L\ \alpha}]\ \mathsf{end}}$$

$$\textbf{(specialize)} \quad \frac{\mathsf{let}\ \overline{L=\lambda\alpha.\pi}\ \mathsf{in}\ \lambda\alpha.\tau\ \mathsf{end}}{\lambda\alpha.\mathsf{let}\ \overline{L=\pi[\overline{L\ \alpha}/\overline{L}]}\ \mathsf{in}\ \tau[\overline{L\ \alpha}/\overline{L}]\ \mathsf{end}}\ (**)$$

$$\textbf{(split)} \quad \frac{\mathsf{let}\ \overline{L=\pi_L}, \overline{M=\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}}{\mathsf{let}\ \overline{L=\pi_L}\ \mathsf{in}\ \mathsf{let}\ \overline{M=\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}\ \mathsf{end}}$$

$$\textbf{(merge)} \quad \frac{\mathsf{let}\ \overline{L=\pi_L}\ \mathsf{in}\ \mathsf{let}\ \overline{M=\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}\ \mathsf{end}}{\mathsf{let}\ \overline{L=\pi_L}, \overline{M=\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}}$$

$$\textbf{(rename)} \quad \frac{\lambda\alpha.\pi}{\lambda\beta.\pi[\alpha/\beta]}\ (\#)$$

Figure 5.5: Rules for manipulating proof terms in $\mathcal{C}$

Correspondingly, we have to change any citations of the lemmas. From the scoping discipline, we know exactly where these citations can be: in the proofs of the lemmas, $\overline{\pi}$, or in the proof τ. Before performing **(generalize)**, all the lemmas and τ referred to the same α. Now, the first abstraction for any of the lemmas is over α. Consequently, any citation of the lemmas must be changed to have the first application be to a term that matches α explicitly. Since all of the proofs referred to the same α before the operation, we can simply use the α in the applications and replace all occurrences of $L_i[j]$ with $L_i[j]\ \alpha$.

The types of the L_is and π_is also change. If α is an individual variable, we add another universal quantification to the front of the type. If α is a proof variable, we add another implication, corresponding to a premise.

The **(specialize)** rule does the opposite of **(generalize)**. A variable that was universally quantified for the lemmas L now becomes a constant for them when we move α to the outside of the let. As stated, the rule requires $\lambda\alpha$ to precede every proof π. This is not actually a requirement for correctness, but it makes stating the side condition easier. The side condition $(**)$ is that any citation of a lemma $L_i[j]$ is of the form $L_i[j]\ \alpha$. In other words, the same variable used in the λ-abstraction for the lemma must be the first variable to which the lemma is applied. Otherwise, the proof may no longer be correct, since another term used in the place of α may have different assumptions than those of α. Given this condition and the scoping discipline, we know exactly which citations need to change: those of the form $L_i[j]\ \alpha$ that appear in the π_is or in τ.

The **(split)** rule takes a list of lemma definitions and separates them into two sets of definitions, one in the same place and one nested in a new let expression within the in part of the original let. The proofs of the lemmas do not change at all, so no citations need to change. The **(merge)** rule is the inverse of the **(split)** rule.

The **(rename)** rule changes the name of a single variable. The side condition $(\#)$ is that the new name β must not occur anywhere in π. This corresponds to α-conversion.

Soundness for the proof system requires that a sequence of applications of the rules transforms a proof term of a type φ into a new proof term of a type ψ that is equivalent modulo first-order equivalence. Let $\pi \Rightarrow \tau$ mean that the proof term

τ is derivable from π using our proof rules in one step.

Theorem 5.1 *If $\pi \Rightarrow \tau$ and $\Gamma \vdash \pi : \varphi$, then $\Gamma \vdash \tau : \psi$, where φ and ψ are equivalent modulo first-order equivalence.*

Proof. The proof is by induction on the proof terms. It can be found in Appendix A. □

5.4 A Tree Structure Represention of Proof Terms

The structure we have presented thus far provides a formal representation for theorems and proofs that a theorem prover could use as its internal representation of a proof library. However, the relationships between theorems, assumptions, and variables may not be clear when presented to an end user, particularly for a large library. Given the importance we place on usability, it is necessary to have a representation we can present to individuals that gives an intuitive understanding of the library structure.

Fortunately, there is a natural correspondence between our proof terms and a nested tree structure that makes the relationships between theorems, assumptions, and variables obvious. A *nested tree* is a tree in which nodes may themselves represent trees. A *proof tree* is a nested tree that represents a library of theorems. A proof tree contains two kinds of nodes:

- *Proof nodes* are leaf nodes that contain proof terms $\pi : e$, where π is a proof term not containing any let expressions or sequences and e is an equation.
- *Collection nodes* are internal nodes that contain a list of names L and a proof tree T, where L is the list of lemmas whose proofs are represented in

T. Element i of L is the name given to the theorem represented by leaf node i in an in-order traversal of T. The proof tree T is considered to be rooted at the collection node that contains it.

Figure 5.6 provides a one-to-one correspondence between proof terms and proof trees. A parabola containing a proof term τ indicates that τ is recursively examined and converted to a proof tree. An ellipse containing a proof term τ indicates τ should be put inside a proof node as is.

For a proof term with several λ-abstractions in immediate succession, we represent the abstractions on a single edge. We represent a library of theorems as a collection node call the *library node*. The names in the list in this collection node correspond to theorems; any names in collection nodes inside this node are lemmas. There is a proof node for every theorem or lemma in the library node.

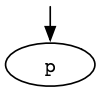
The proof of theorem is formed by following a path from the root of a proof tree to a proof node P, collecting abstractions on the edges along the path and using as the body the proof in P. Any collection nodes encountered along the way are turned int let expressions. Abstractions that are on the path starting at the library node and going to the collection node containing P are constants for the proof.

As an example, we can represent the proof terms (5.7) and (5.8) as the trees in Figures 5.7 and 5.8, respectively.

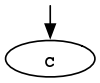
5.5 Proof Term Manipulations on Trees

Our proof term rules in Figure 5.5, as well as the **(promote)** rule from Figure 5.3, can be viewed as alterations made to proof trees. These manipulations include moving edge labels, changing edges between nodes, and moving subtrees in and

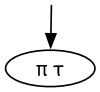
- a variable $p \in P$



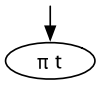
- a constant c



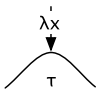
- $\pi\ \tau$



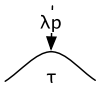
- $\pi\ t$



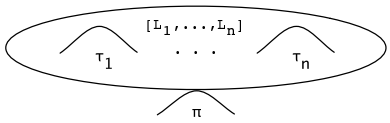
- $\lambda x.\tau$



- $\lambda p.\tau$



- let $x_1, ..., x_n = \tau_1, ..., \tau_n$ in π end



- $\pi_1; \ldots; \pi_n$

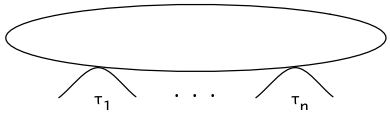


Figure 5.6: Translation between proof terms and proof trees

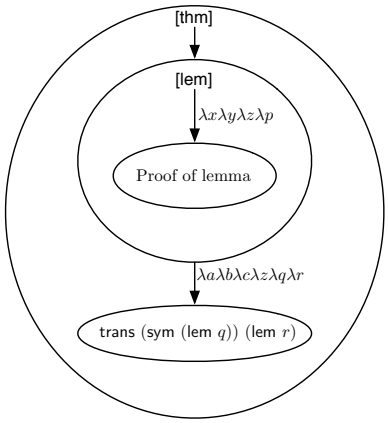


Figure 5.7: (5.7) as a proof tree

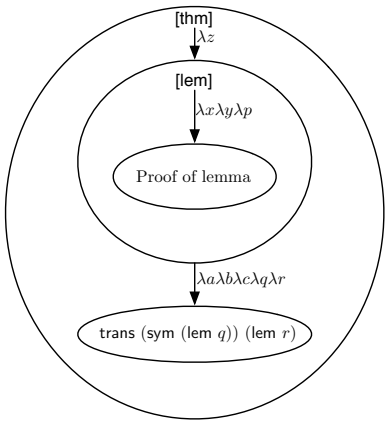


Figure 5.8: (5.8) as a proof tree

out of collection nodes. The tree manipulations corresponding to the proof rules are given in Figures 5.9-5.12. Changes are highlighted in red.

5.6 A Constructive Example

To demonstrate the use of the proof rules, we develop the proofs of (5.2) and (5.1). We use the axioms presented in Section 3.3. Until we need them, we omit both $\mathfrak{L}$ and $\mathfrak{C}$ for readability. We also omit term substitutions when performing cites.

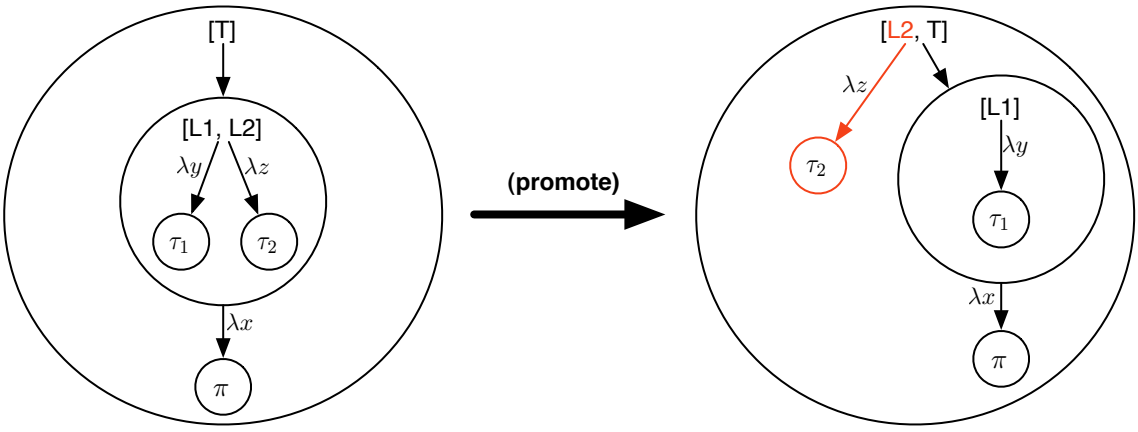


Figure 5.9: **(promote)** as a tree manipulaiton

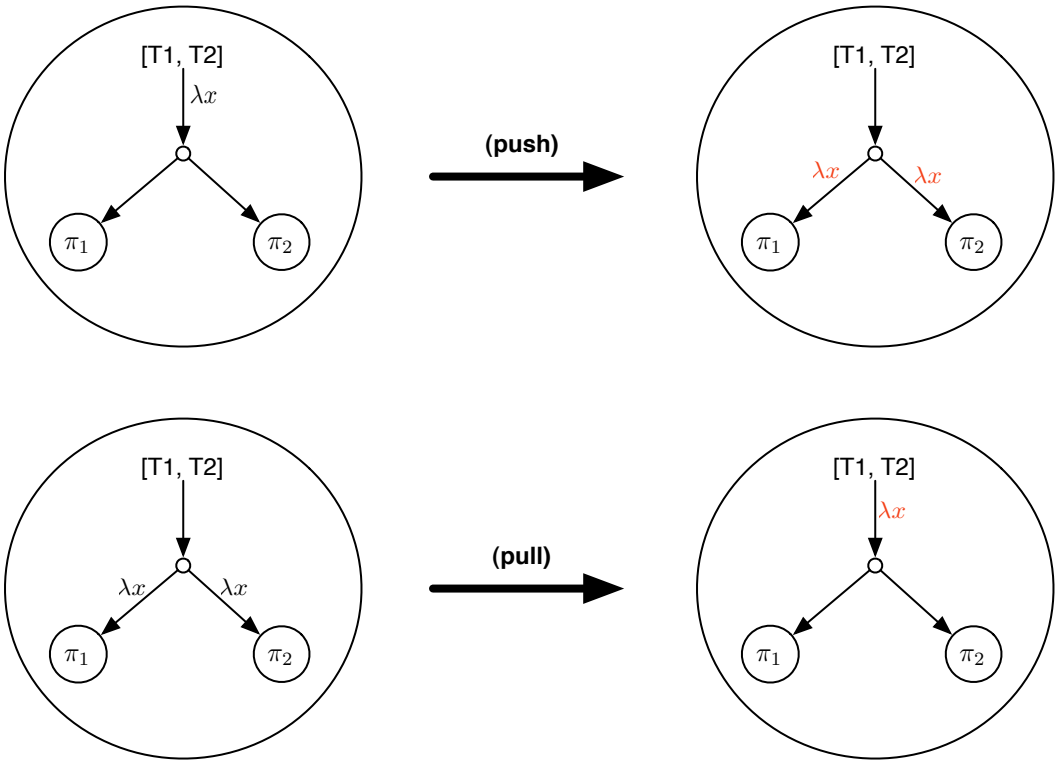


Figure 5.10: **(push)** and **(pull)** as tree manipulaitons

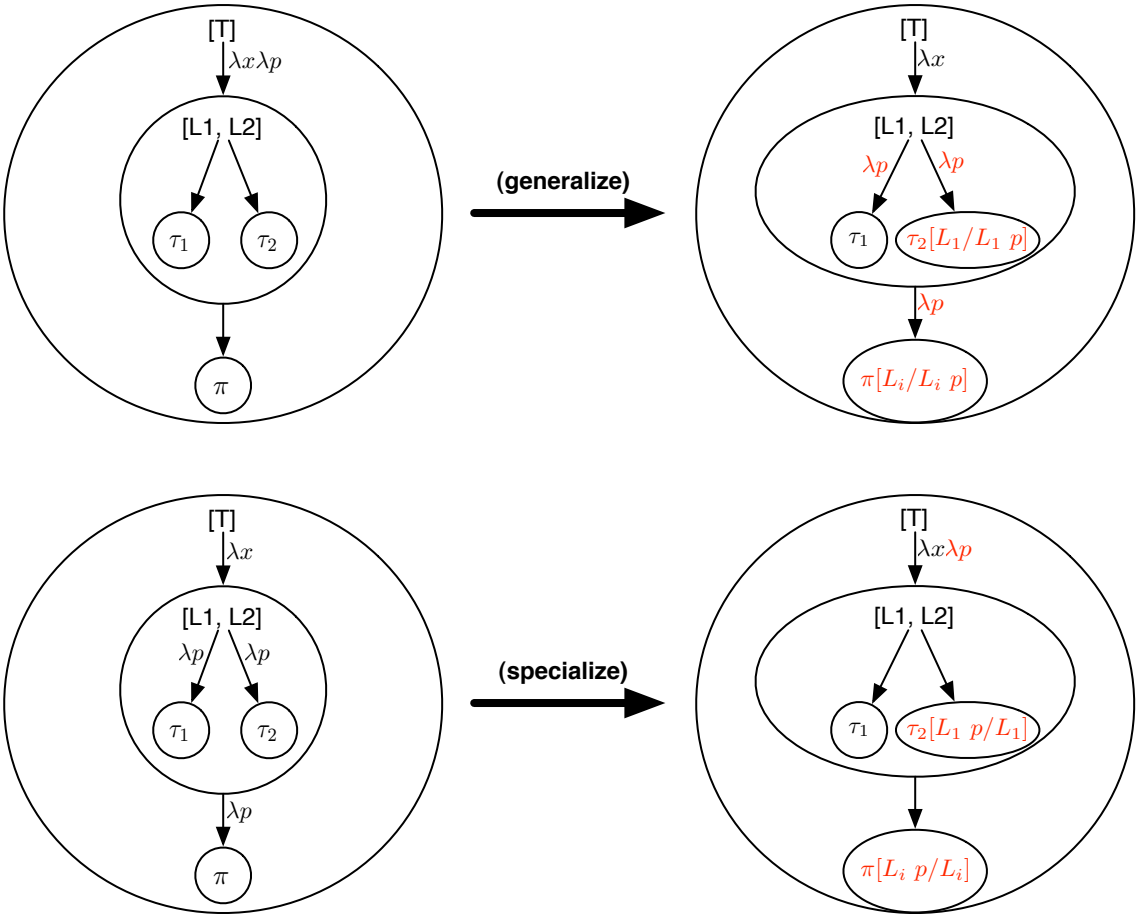


Figure 5.11: **(generalize)** and **(specialize)** as tree manipulaitons

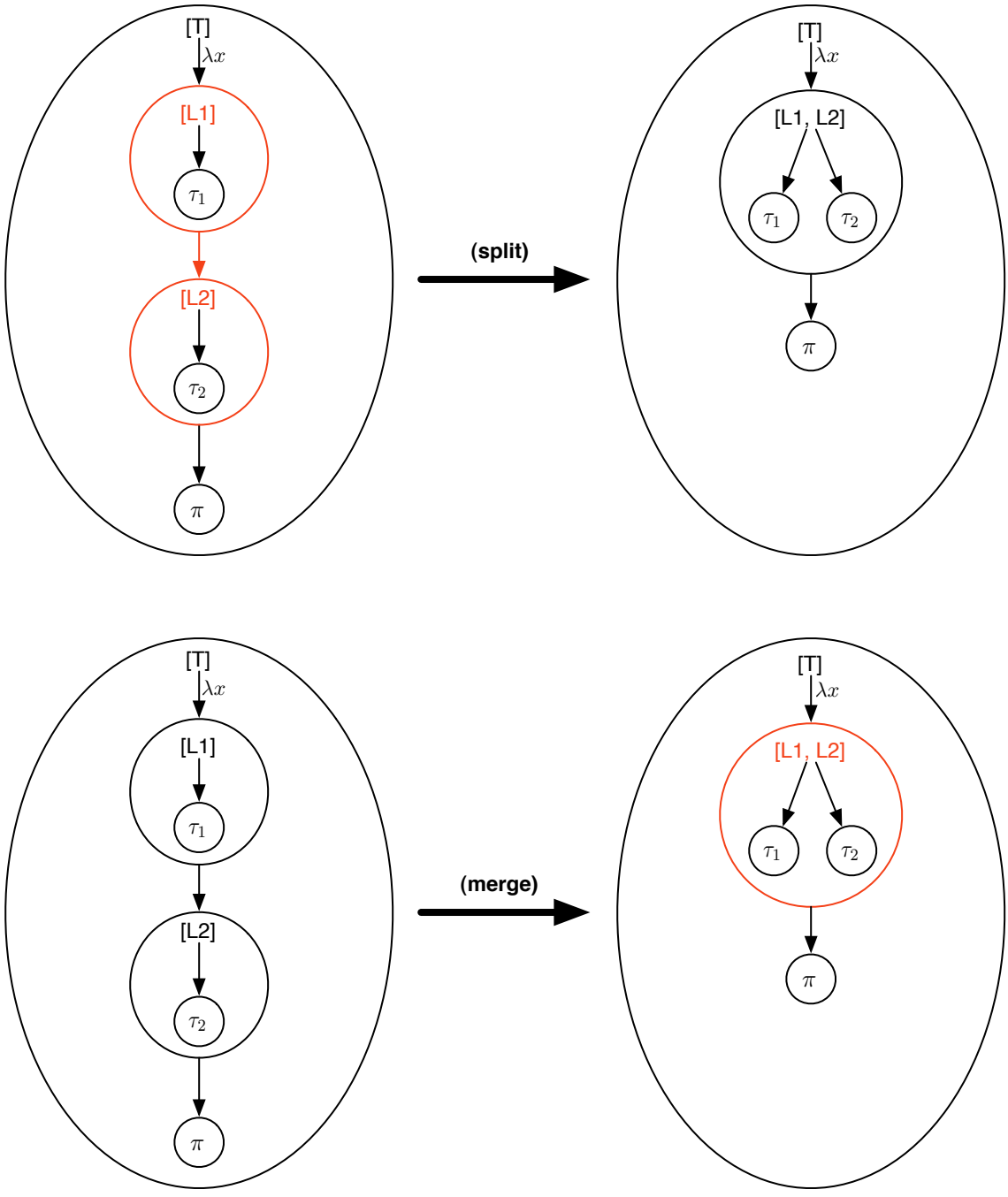


Figure 5.12: **(split)** and **(merge)** as tree manipulaitons

First, we prove the lemma. By **(ident)**, we have

$$p : x = y \vdash p : x = y \tag{5.9}$$

We use **(tcite)** with the substitutions $[x/x, y/y, z/x]$ and **(assume)** to add

$$p : x = y \vdash \mathsf{cong}_\wedge : x = y \ \rightarrow \ (x \wedge x) = (x \wedge y) \tag{5.10}$$

Applying **(mp)** to (5.9) and (5.10) gives

$$p : x = y \vdash \mathsf{cong}_\wedge \ p : (x \wedge x) = (x \wedge y) \tag{5.11}$$

We use **(tcite)** with the substitutions $[x/x \wedge x, y/x \wedge y, z/z]$ and **(assume)** to add

$$p : x = y \vdash \mathsf{cong}_\vee : (x \wedge x) = (x \wedge y) \ \rightarrow \ z \vee (x \wedge x) = z \vee (x \wedge y) \tag{5.12}$$

Applying **(mp)** to (5.11) and (5.12) gives

$$p : x = y \vdash \mathsf{cong}_\vee \ \mathsf{cong}_\wedge \ p : z \vee (x \wedge x) = z \vee (x \wedge y) \tag{5.13}$$

Now we apply **(discharge)** to (5.13) to get

$$\vdash \lambda p.\mathsf{cong}_\vee \ \mathsf{cong}_\wedge \ p : x = y \ \rightarrow \ z \vee (x \wedge x) = z \vee (x \wedge y) \tag{5.14}$$

We can use the **(collect)** rule to add (5.14) to our current term, given it the name lem. Our entire state is

$$\mathcal{L}; \mathsf{lem} = \lambda x \lambda y \lambda z \lambda p.\mathsf{cong}_\vee \ \mathsf{cong}_\wedge \ p : \forall x, y, z.x = y \ \rightarrow \ z \vee (x \wedge x) = z \vee (x \wedge y);$$

Now we start on the proof of the theorem. First we use **(ident)** to add the task

$$q : a = b \vdash q : a = b \tag{5.15}$$

Next, we use **(lcite)** with the substitutions $[x/a, y/b, z/z]$ and **(assume)** to get our lemma from the current term

$$q : a = b \vdash \mathsf{lem} : a = b \ \rightarrow \ z \vee (a \wedge a) = z \vee (a \wedge b) \tag{5.16}$$

Applying **(mp)** to (5.15) and (5.16) gives

$$q : a = b \vdash \mathsf{lem}\ q : z \vee (a \wedge a) = z \vee (a \wedge b) \tag{5.17}$$

We now use **(cite)** with the substitutions $[x/z \vee (a \wedge a), y/z \vee (a \wedge b)]$ and **(assume)** to introduce

$$q : a = b \vdash \mathsf{sym} : z \vee (a \wedge a) = z \vee (a \wedge b) \ \rightarrow\ z \vee (a \wedge b) = z \vee (a \wedge a) \tag{5.18}$$

Applying **(mp)** to (5.17) and (5.18) gives

$$q : a = b \vdash \mathsf{sym}\ (\mathsf{lem}\ q) : z \vee (a \wedge b) = z \vee (a \wedge a) \tag{5.19}$$

Next, we use **(ident)** to introduce

$$r : a = c \vdash r : a = c \tag{5.20}$$

Next, we use **(lcite)** with the substitutions $[x/a, y/c, z/z]$ and **(assume)** to get our lemma from the current term again

$$r : a = c \vdash \mathsf{lem} : a = c \ \rightarrow\ z \vee (a \wedge a) = z \vee (a \wedge c) \tag{5.21}$$

Applying **(mp)** to (5.20) and (5.21) gives

$$r : a = c \vdash \mathsf{lem}\ r : z \vee (a \wedge a) = z \vee (a \wedge c) \tag{5.22}$$

Applying **(tcite)** with the substitutions $[x/z \vee (a \wedge b), y/z \vee (a \wedge a), z/z \vee (a \wedge c)]$ allows us to add

$$\vdash \mathsf{trans} : z \vee (a \wedge b) = z \vee (a \wedge a) \ \rightarrow\ z \vee (a \wedge a) = z \vee (a \wedge c) \ \rightarrow\ z \vee (a \wedge b) = z \vee (a \wedge c) \tag{5.23}$$

Applying **(assume)** to (5.19), (5.22), and (5.23) gives

$$q : a = b, r : a = c \ \vdash\ \mathsf{sym}\ (\mathsf{lem}\ q) : z \vee (a \wedge b) = z \vee (a \wedge a) \tag{5.24}$$

$$q : a = b, r : a = c \ \vdash\ \mathsf{lem}\ r : z \vee (a \wedge a) = z \vee (a \wedge c) \tag{5.25}$$

$$q : a = b, r : a = c \ \vdash\ \mathsf{trans} : (a \wedge b) = z \vee (a \wedge a) \tag{5.26}$$

$$\rightarrow\ z \vee (a \wedge a) = z \vee (a \wedge c) \ \rightarrow\ z \vee (a \wedge b) = z \vee (a \wedge c)$$

Two applications of **(mp)** using (5.24), (5.25), and (5.26) gives

$$q : a = b, r : a = c \vdash \mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ q))\ (\mathsf{lem}\ r) : z \vee (a \wedge b) = z \vee (a \wedge c) \quad (5.27)$$

We use **(discharge)** on each assumption in (5.27) to get

$$\vdash \lambda q.\lambda r.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ q))\ (\mathsf{lem}\ r) : a = b \ \rightarrow\ a = c \ \rightarrow\ z \vee (a \wedge b) = z \vee (a \wedge c) \quad (5.28)$$

We can use the **(collect)** rule to add (5.28) to our current term, give it the name thm, and make lem a lemma by introducing a let expression. Our new $\mathcal{C}$ term is

$$\begin{array}{l}
\mathsf{thm}\ = \\
\quad \mathsf{let}\ \mathsf{lem} = \lambda x \lambda y \lambda z \lambda p.\mathsf{cong}_{\vee}\ \mathsf{cong}_{\wedge}\ p : \forall x, y, z.x = y \ \rightarrow\ z \vee (x \wedge x) = z \vee (x \wedge y) \\
\quad \mathsf{in} \\
\qquad \lambda a \lambda b \lambda c \lambda z \lambda q.\lambda r.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ q))\ (\mathsf{lem}\ r) \ : \ \forall a, b, c, z.a = b \ \rightarrow\ a = c \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow\ z \vee (a \wedge b) = z \vee (a \wedge c) \\
\quad \mathsf{end}
\end{array}$$

At this point, we could apply **(publish)** to add thm to the library. However, we may first wish to make thm and lem use the same z. To do this, we apply **(reorder)** to the term several times to get

$$\begin{array}{l}
\mathsf{thm}\ = \\
\quad \mathsf{let}\ \mathsf{lem} = \lambda z \lambda x \lambda y \lambda p.\mathsf{cong}_{\vee}\ \mathsf{cong}_{\wedge}\ p : \forall z, x, y.x = y \ \rightarrow\ z \vee (x \wedge x) = z \vee (x \wedge y) \\
\quad \mathsf{in} \\
\qquad \lambda z \lambda a \lambda b \lambda c \lambda q.\lambda r.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ q))\ (\mathsf{lem}\ r) \ : \ \forall z, a, b, c.a = b \ \rightarrow\ a = c \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow\ z \vee (a \wedge b) = z \vee (a \wedge c) \\
\quad \mathsf{end}
\end{array}$$

We now apply **(specialize)** to move λz to the front of the **let** expression

$$\begin{array}{l}\mathsf{thm}\ = \\ \quad \lambda z.\mathsf{let}\ \mathsf{lem} = \lambda x\lambda y\lambda p.\mathsf{cong}_{\vee}\ \mathsf{cong}_{\wedge}\ p : \forall x, y.x = y\ \rightarrow\ z\vee(x\wedge x) = z\vee(x\wedge y) \\ \quad \mathsf{in} \\ \qquad \lambda a\lambda b\lambda c\lambda q.\lambda r.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ q))\ (\mathsf{lem}\ r)\ \ :\ \ \forall z, a, b, c.a = b\ \rightarrow\ a = c \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow\ z\vee(a\wedge b) = z\vee(a\wedge c) \\ \quad \mathsf{end}\end{array}$$

5.7 Conclusions

We see many benefits to an automated theorem prover using a library with such a formal hierarchical structure. First of all, we would expect the structure of the library to indicate which theorems are more closely related–theorems that use the same variables, assumptions, or lemmas would be grouped together in let expressions and share abstractions. Large mathematical libraries could naturally be broken down into smaller parts based on these groupings.

One can imagine several heuristics that could be improved by the structure of the library. A system could first look at citing lemmas currently in scope before searching the entire library. The number of lemmas in scope is likely to be smaller than the number of theorems. Heuristics that automatically detect similar subproofs and create lemmas from them should also be possible. Given the formal structure of proofs, finding shared lemmas is a form of common subexpression elimination. In discovering these lemmas automatically, the library takes on the structure natural to the theorems proven. It could also provide guidance to a user proving a new theorem, knowing that the current proof being worked on and other theorems already proven share a few lemmas.

Chapter 6

User Interfaces

As theorem provers become more powerful and more useful to mathematicians, their user interfaces must be designed with a wider audience in mind. The mathematical knowledge management community has a particular interest in the design of user interfaces. As discussed in Sections 4.2.1 and 4.2.2, we placed a good deal of importance on the user interface for KAT-ML and continue to do so as we develop the underlying theory. In this chapter, we explore issues related to graphical representation of mathematical libraries.

6.1 Proofs Represented as Graphs

Representing proofs as graphs is an important method for making the relationships in proofs explicit and presentable. In contrast to our tree representation in Section 5.4, popular methods for graphical proof representation actually capture the reasoning in a proof with the graph structure; we use the trees simply to represent the structure of the entire proof library. Nevertheless, several other techniques allow one to capture notions of scope in a picture representation of a proof. Whereas our approach derives the tree representation from the formal underlying theory, the approaches discussed here start with graphs and provide a formalization of the operations on them.

Girard presented proof nets as a graphical representation of linear logic proofs [41]. *Proof structures* contain formulas with links between them, where formulas are the conclusion of exactly one link. Logical soundness of the proof structure is determined using a notion of a *trip*, where formulas are visited in a particular way,

given their links. The proof structures that are logically sound are called *proof nets*.

Proof nets are sufficient for representing the multiplicative fragment of linear logic. With the addition of *proof boxes*, one can represent all of linear logic. A proof box contains a proof net with conclusions $A_1, \ldots, A_n$. The proof box is considered a black box proof of these formulas $A_1, \ldots, A_n$, which can be linked to other formulas. Proof boxes give us a scope for formulas; formulas inside the proof box that are not the conclusions are limited in scope to the box itself.

Milner uses bigraphs to model several different mathematical formalizations, including the π-calculus, Petri nets, and the λ-calculus [84]. Bigraphs contain nodes, which can be nested, and *ports*, which are linked to join nodes. Milner provides a categorical axiomatization of the formation of bigraphs that is complete with respect to equations on expressions representing bigraphs. Since bigraphs can represent the λ-calculus, they are powerful enough to represent the proof terms discussed in Chapter 3. The nesting of bigraph nodes allows them to represent the scoping of theorems; however, this scoping is limited to very simple cases without the inclusion of some notion of sequencing, which allows one to use a lemma in multiple theorems.

Deduction graphs, developed by Geuvers and Loeb, represent logical proofs with a scoping mechanism available for formulas and subproofs [39]. Scope is captured with boxes, which are particularly useful for the $\rightarrow$-introduction rule of both Gentzen-Prawitz style deduction and Fitch-style deduction. These deduction graphs can be translated to a λ-term with a let expression used for scoping. It is the let expression that allows one to show which parts of the graph are shared and repeated. Geuvers and Loeb also defined several rewrite rules on these λ-terms

that allow one to manipulate the scope of formulas, but in a relatively simplistic manner.

6.2 Current User Interfaces

Work in the area of user interfaces for theorem provers continues to grow as developers try to make the systems more accessible to mathematicians. The work addresses graphical, point-and-click user interfaces for several important aspects of theorem provers, including development, organization, and search. Many of these approaches build on top of an existing theorem prover and are thus limited by the underlying formalism of the theorem prover itself. We describe some of the ongoing work in this area, in particular how it relates to the organization of a theorem library.

Théry, Bertot, and Kahn explain the necessity of developing good user interfaces for theorem provers [106]. With more fields of computer science and software development benefiting from formalized mathematics, there is a need to have usable interfaces for theorem provers so that those outside the development community can easily learn the system and apply it to their work. Unlike symbolic algebra systems including Maple and Mathematica, theorem provers must deal with planning and managing proofs, requiring special considerations for user interfaces.

One aspect of the interface for managing proofs is the theorem library. Théry et al. stated that "the success of a particular theorem prover may depend more on the availability of a large number of well organized theories than any other factor." The theories, containing several related theorems, are grouped in a hierarchical fashion based on dependencies, as seen in Figure 6.1. The authors proposed that one should be able to click on a theory and get a menu of the theorems it contains.

However, as presented, the hierarchical structure does not expand into theories to organize the theorems themselves.

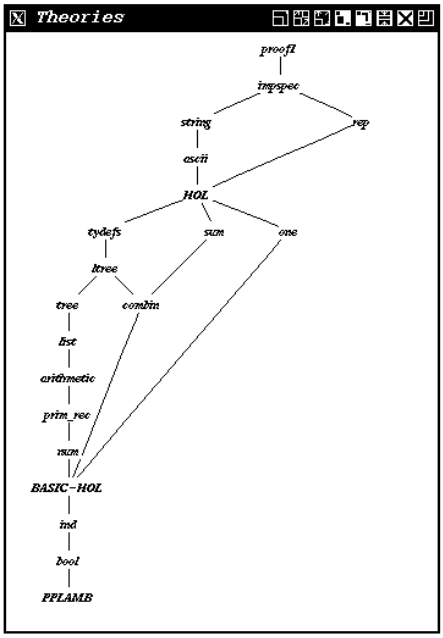


Figure 6.1: An example theory layout in HOL [106]

Bertot, Kahn, and Théry discussed several important aspects of tree-based approaches to theorem prover user interfaces [18, 19]. They advocated the graphical manipulation of theorems using a technique called *proof by pointing*, where one uses the mouse to select and edit structured terms. Steps performed when one clicks on a term are guided by an underlying interpretation of Gentzel rules for natural deduction. For example, if the current goal is an implication of the form $a \vee b \rightarrow c$ and one clicks on the a, the result is two new implications $a \rightarrow c$ and $b \rightarrow c$, where the first is now the current goal.

The style of deduction in Bertot et al.'s system lends itself well to a tree representation of proofs. The authors found, however, that trees tend to grow too wide for practical presentation. Other methods of presentation are possible, in-

cluding a more vertical linear representation that assigns numbers to each line of the deduction and refers to them in subsequent steps. The authors also discussed translation of proofs into readable English, a well-studied problem.

Bertot and Théry further explored the formalization of other aspects of a user interface, including menus, the declaration of new rules, proof script management, and the interaction between different modules of the system. However, left out of this formal description is the library of proven theorems. The authors assumed the existence of a list of theorems that can be used in the same way as assumptions; clicking on a theorem applies it to the current goal.

Most theorem provers today use Emacs as their preferred user interface. The Emacs interface provides a powerful and scriptable infrastructure for these systems, allowing one to edit proofs and test them in an interactive environment. A popular way to use theorem provers in Emacs is through the *Proof General* interface, a generic environment for Emacs that works with many popular systems including Coq, Isabelle, and HOL, pictured in Figure 6.2 [7].

Proof General is geared specifically toward advanced users developing large libraries of theorems in proof assistants that have an interactive command line. The system takes proof scripts and commands entered by the user in an Emacs buffer, sends them to the theorem prover using this command line, and then returns the output in a separate window. Proof General also manages complex proof scripts across multiple files, automatically maintaining dependency relationships.

Aspinall et al. have continued work based on the Proof General system to create the *Proof General Kit* (PG Kit) [8]. The system is based on the idea of *document-centered authoring*, where one produces a single document that can be viewed in several ways, including as a proof script for a theorem prover and as a

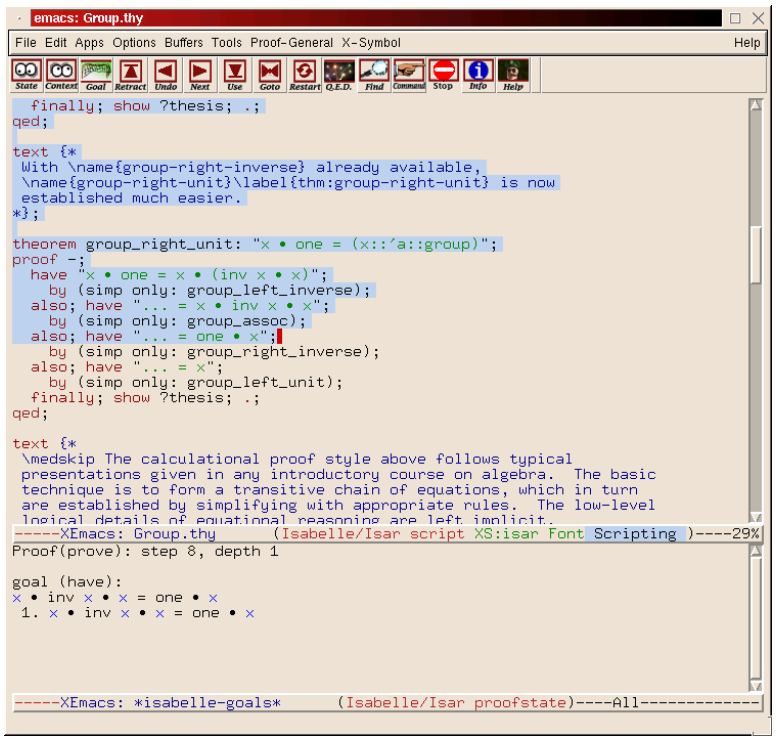


Figure 6.2: Replaying a proof in Isar in Proof General [7]

human-readable LaTeX document. Novel in their approach is the fact that external tools that alter the different views of the document (e.g., LaTeX adding references or a theorem prover filling in proof cases) can send these changes to the central document, a process called *backflow*. Currently, the authors have both Emacs and Eclipse versions.

Piroi described several user interface features of *Theorema*, a system built on top of *Mathematica* that provides infrastructure for formalizing and proving mathematical theorems [92]. *Theorema* allows one to label formulas and collections of formulas so that they can be stored in units called *notebooks* for later referral in other proofs. The proofs themselves are hierarchical in nature, with subproofs labeled with their own numbers and displayed by clicking on a hyperlink. The user interface provides a mechanism for viewing theorems in a human-readable format

and interactively navigating a proof and performing steps of deduction.

Cairns developed *Alcor*, a user interface for Mizar that stresses search [24]. We have already discussed the Mizar Mathematical Library (MML) in Section 2.2.1, where one can submit theorems to be added to an online repository of proofs. As the library grows, the issue of search becomes increasingly important; one wants to avoid repetition in proofs by using ones that already exist whenever possible and wants to avoid attempting to add a proof to the library that already exists.

A user can click on a term in the proof being developed and then search for it in the entire MML. The search results pane allows one to click on individual matches and see them in the context of the article in which they appear. Further searches can be conducted on terms in that article. While currently limited in its usefulness, Alcor has the potential to provide an important resource for proof developers in systems with libraries growing increasingly complex.

Geuvers and Mamane worked on tmEgg, a LaTeX-oriented front end to Coq implemented as a plugin to TeXmacs [40]. TeXmacs provides the ability to annotate a LaTeX document with tags that can be not only printed, but also automatically passed as commands to an external program. The authors adapted the software to work with Coq, passing tags as commands that could be used to formally prove a theorem being typeset. Therefore, articles produced with tmEgg can be seen as a mathematical document with an underlying formalism provided by Coq.

An important issue in incorporating Coq into a LaTeX document is *document-consistency*: insuring that commands are executed in Coq in the same context and order in which they appear in the document. Coq's backtracking support becomes necessary in order to maintain this consistency, as one may not type commands

in a top-down fashion, instead going back through a document and adding formal justifications later.

One uses the structure of theorems in Coq in the LaTeX document, creating new lemmas, definitions, etc., using the keywords of the theorem prover itself. Consequently, the structure of the resulting mathematical article is the same as the structure in the underlying formalism. The details of the proof can be hidden, although the authors state that they would like to expand this ability in the future, allowing one to hide several lemmas used by Coq to prove a main lemma that should appear in the document.

6.3 A Proof-Theoretic User Interface

As discussed in Section 5.4, there is a natural correspondence between our proof representation and a nested tree structure that provides a nice graphical view of a proof library. To demonstrate the tree representation's usefulness, we have constructed a proof-of-concept implementation of a theorem prover for KAT that allows one to view and manipulate a proof library graphically. The Grappa graph drawing tool for Java makes it easy to create a tree-like structure of theorems with which one can interact with mouse clicks and menus [14].

Figure 6.3 is an example of the possible organization of the library of theorems for the Hoare logic rules, which have already been verified using KAT-ML. The tree structure looks very similar to the one in Section 5.4. Proof nodes are represented by elliptical nodes with the name of a theorem in them. Abstractions, described in Section 5.4 as edge labels, are represented as diamond-shaped nodes containing the individual variable or proof variable abstraction. Each abstraction is represented in its own node; they are not grouped together when one immediately follows

another. Representing them as individual nodes makes their manipulation easier in Grappa. Rectangular nodes with a list of theorem names represent collections nodes. One can double-click on these nodes to see the proofs they contain.

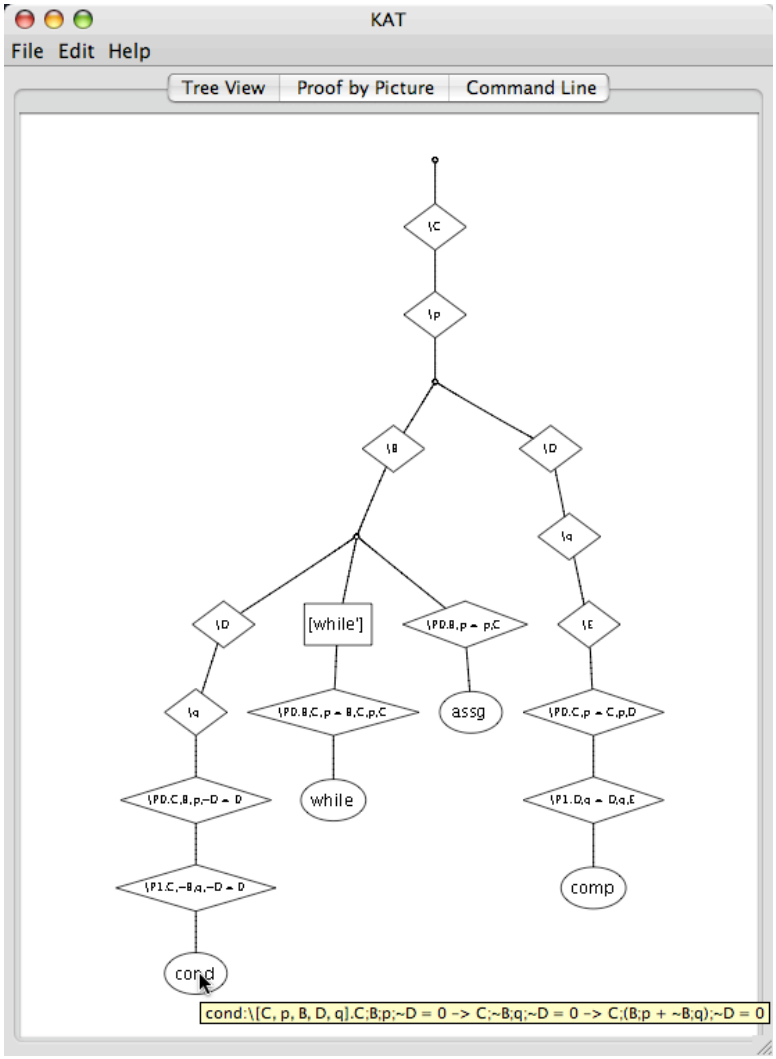


Figure 6.3: Hoare logic rules arranged in hierarchical fashion

In this particular example, we have theorems for the Hoare conditional, while, assignment, and composition rules. All of the theorems refer to a condition C and a program p, so we have used the **(pull)** rule to abstract over these variables once for all of the theorems. The while theorem uses a lemma, `while'`, which is represented

as a collection node. The variables C, p, and B are arbitrary, fixed constants for the proof of `while'`. Double-clicking on the collection node containing the lemma reveals that it is abstracted over a variable q and an assumption $P1$.

Right-clicking on any node reveals a list of commands that one can run, as shown in Figure 6.4. The "Publish New Theorem..." and "Publish New Lemma..." commands are available at any node. The former creates a new top-level theorem specified by the user. The latter creates a new lemma. If the node selected is a collection node, then the lemma is added to that node. If it is any other kind of node, then a new collection node is created above that node.

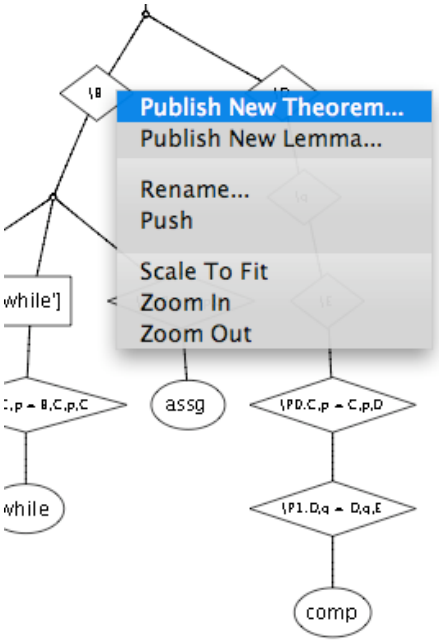


Figure 6.4: Right-click options

Depending on the node on which one right-clicks, other options may also be available. For example, in Figure 6.4, the user is presented with the options "Rename..." and "Push," which apply the **(rename)** rule and **(push)** rule to the corresponding proof term, respectively. These options are only presented if

necessary preconditions are met. For example, the command corresponding to the **(reorder)** rule will only be presented if changing the order of abstractions does not cause a variable to be moved out of scope improperly, as described in Section 5.3.2.

The interface presented here is currently quite basic, but offers insight into the ease with which one can develop a useful graphical user interface for managing a complex library of theorems. As repositories of formalized mathematics become larger, new ways of visualizing the relationships between different proofs need to be considered a priority.

Chapter 7

Tactics

In Section 4.2.6, we presented a set of simple heuristics to aid in proving theorems. These heuristics were system-level constructs designed specifically with the KAT-ML prover in mind. They are able to take advantage of the proof representation and KAT itself. However, the heuristics are still separate from these underlying formalisms in the same way they are in most theorem provers. Theorem provers such as Coq, NuPRL, and Isabelle provide extensive tools for users to create proofs quickly with automated methods.

Fundamental to these systems is the use of *tactics* and *tacticals*, programs that represent and execute several steps of deduction. The language used for tactics is typically a full-scale programming language, separate from the language used to represent proofs. Consequently, there is also a separation between the use of theorems in proofs and the use of tactics.

Despite being implemented at different levels, theorems and tactics have much in common. Both store and repeat proof steps. They represent generalized proof techniques used often within the theory in which they exist. Moreover, both provide guidance and hints to a user regarding the completion of a proof; proofs that share a few tactics or theorems are likely to share more. Nevertheless, work in the area of tactics and tacticals focuses on developing automated proof steps at the system level, separate from the underlying logic in which they work.

The power of the separate tactics language comes at a price, as explained by Delahaye [32]. Separating the two languages requires a user to learn two languages when creating proofs and the developer to create a separate infrastructure for

debugging and validating tactics.

The separation between tactics and theorems also inhibits our flexibility in proof representation. While tactics may be used to automate proof steps, they are not represented in the completed proof; the tactics merely apply a sequence of elementary inference rules that a user would perform manually without the tactics. There are times when formally representing the tactics at the same level as proofs can be useful, particularly when transferring a proof to a paper. If a step in the proof is repeated several times by a tactic, we may want to perform the step explicitly the first time and then say that the step is "repeated several times in the same way." We want to be able to represent such a statement formally in the proof itself.

Our goal is to represent tactics in a way that allows them to be treated at the same formal level as proofs and theorems, independent of their system-level implementation. Many very useful tactics on commonly used algebras only require simple constructs that can be represented easily in the same way as theorems, not needing Turing-complete languages used in theorem provers. For example, a tactic for substitution of equals for equals requires congruence rules for each operation in the algebra, the ability to iterate through several steps of using different congruence rules, and the ability choose the appropriate congruence rule at each step.

We also want a representation that allows us to easily translate tactics into the proof steps they represent using proof-theoretic, formal rules. Such a representation gives us the flexibility to make proofs more general by using the tactics in the representation or more specific by using some or all of the individual proofs steps.

Finally, the representation should be independent of search techniques and algorithms used to implement automated proof search. While these issues are

important for a theorem prover, they are system-level decisions orthogonal to the choices made in representing tactics.

In this chapter, we propose such a representation. We extend the proof system in Chapter 3 to represent tactics at the same level as theorems and move freely from tactics to proof steps. We formalize several common tactics and propose a way to represent them in our proof system. We then provide formal rules for creating and manipulating tactics and their use in proofs. Finally, we provide an extended example for creating a simple tactic and using it.

7.1 A Motivating Example

Consider reasoning about a Boolean algebra as in Section 3.3. Let us look at a particular form of tactic. It is easy to see that the axiom $\mathsf{idemp}_\wedge$ allows us to prove

$$\forall a. \quad a \wedge a \wedge a = a \tag{7.1}$$

Once we have the proof of (7.1), we can use it to prove

$$\forall a. \quad a \wedge a \wedge a \wedge a = a \tag{7.2}$$

in the following way. From (7.1) and $\mathsf{cong}_\wedge$ with the substitution $[x/a \wedge a \wedge a, y/a, z/a]$, we can deduce

$$a \wedge a \wedge a \wedge a = a \wedge a \tag{7.3}$$

We then use $\mathsf{idemp}_\wedge$ to get

$$a \wedge a = a \tag{7.4}$$

Finally, we apply trans to (7.3) and (7.4) with the substitution $[x/a \wedge a \wedge a \wedge a, y/a \wedge a, z/a]$ to conclude

$$a \wedge a \wedge a \wedge a = a$$

which is true for arbitrary a, yielding our desired conclusion (7.2). We can continue to prove a theorem like this for $n + 1$ occurrences of a using the proof for n occurrences of a.

The form of this proof is typical: an inductive argument where we use the result from one proof to prove a step in the next proof. We wish to generalize this kind of proof as a tactic that allows one to represent the execution of several steps of the proof either with the tactic itself or with the individual proof steps.

The need to recover the steps is important, particularly for presentation. Imagine one proves a theorem such as (7.2). Given that the proof steps are similar and repeated, one may wish to state the proof step explicitly once and then capture the rest of the iterations with one statement.

7.2 Tactic Representation

For representing tactics, we extend the proof representation developed in chapter 3. In order to use tactics, we introduce a few new proof terms:

- A *case statement*,

$$\begin{aligned} \mathsf{case}\ \delta\ \mathsf{of}\ \ & {=}\varphi_1 \Rightarrow \pi_1 \\ & \ldots \\ & {=}\varphi_n \Rightarrow \pi_n \\ & \psi_1 \Rightarrow \tau_1 \\ & \ldots \\ & \psi_m \Rightarrow \tau_m \end{aligned}$$

  where $\delta, \varphi_1, \ldots, \varphi_n, \psi_1, \ldots, \psi_m$ are formulas and $\pi_1, \ldots, \pi_n, \tau_1, \ldots, \tau_m$ are proof terms. The case statement is very similar to the one in Standard ML. We look at the structure of δ and match it against the types in the body of the

statement. There are two kinds of matches that can occur. We can exactly match the type δ with a type φ_i, signified by the $=$, or we match a type δ against a possible unification, ψ_j. The difference is that a type δ matches a case $=\varphi_i$ if $\delta = \varphi_i$, whereas it matches a case ψ_j if there exists a substitution such that $\delta = \psi_j[\overline{x}/\overline{t}]$. The proof to the right of the $\Rightarrow$ of the matched case is a proof of the type δ, as enforced by the type system. For simplicity, we assume that the ψ_i cases always come after the $=\varphi_i$ cases.

We use the notation $\overline{=\varphi \Rightarrow \pi}$ to represent $=\varphi_1 \Rightarrow \pi_1 \ldots =\varphi_n \Rightarrow \pi_n$ and $\overline{\psi \Rightarrow \tau}$ to represent $\psi_1 \Rightarrow \tau_1 \ldots \psi_n \Rightarrow \tau_m$.

- A *formula variable* X, representing a quantified or unquantified formula
- A *formula abstraction* $\lambda X.\pi$, where X is a formula variable and π is a proof term. We need this proof term in order to abstract over the δ found in the case statement.

To support tactics, we extend formulas with *recursive types* [86],[90, Ch. 20]. We require the addition of three types:

- A *formula variable* X.
- A *recursive formula* $\mu X.\varphi$, where X is a formula variable and φ is a formula.
- A *sum formula* $\{\delta : \ =\varphi_1 + \ldots + =\varphi_n + \psi_1 + \ldots + \psi_m\}$ where $\delta, \varphi_1, \ldots, \varphi_n, \psi_1, \ldots, \psi_m$ are formulas. We use the notation $\{\delta : \ \overline{=\varphi} + \overline{\psi}\}$ to represent $\{\delta : \ =\varphi_1 + \ldots + =\varphi_n + \psi_1 + \ldots + \psi_m\}$. The sum formula is closely related to the case statement, as will be apparent when examining the typing rules. In fact, we refer to an individual $=\varphi_i$ or ψ_j in a sum formula as a case.

The typing rules for the proof terms are in Figure 7.1. The typing rules for all proof terms are given, as we have changed the type system to allow proof variables to be arbitrary formulas rather than simply equations. While we will not fully harness the power of this change with our new proof rules, this representation makes our rules for the new proof terms easier.

With the presence of abstraction over type variables, we need to type the formulas with kinds [90, Ch. 29]. The kinds primarily provide information for matching a formula with a case in a sum formula. Kinds are built from a base kind $*$ and the first-order signature $\Sigma = \{f, g, \ldots\}$. A *kind term* s_*, t_* is a *base kind* $*$ or an expression $f\ t_{1*} \ldots t_{n*}$ where f is an n-ary function symbol in Σ and $t_{1*}, \ldots, t_{n*}$ are kind terms. A kind equation d_*, e_* is between two kind terms, such as $s_* = t_*$.

For the most part, kind information is implicit; the kind $s_* = t_*$ of an equation $s = t$ is formed by replacing all variables in s and t with $*$. However, we may want to be explicit about kind information when the kind is more specific than the type. For example, the type $x = y$ implicitly has the kind $* = *$. If we mean for it to represent a more specific kind, say, $*\vee* = *\wedge*$ in our Boolean algebra example, we would have to specify the kind explicitly with the notation $(x = y : *\vee* = *\wedge*)$. A type's explicit kind can never be less specific than its implicit kind, i.e., $x \wedge y = y$ cannot have the kind $* = *$. We use the explicit kinds to match formulas with cases in the sum formula.

The type of a case statement with a formula variable X is the sum formula formed from the types of the proofs in the body of the statement. The second and third typing rules allow us to be more specific about a proof with a sum formula type. The type of a proof with a formula δ is δ if either δ is equal to one of the φ_i

$$\frac{}{\Gamma, p : \varphi \vdash p : \varphi}$$

$$\frac{}{\Gamma, c : \varphi \vdash c : \varphi}$$

$$\frac{\Gamma \vdash \pi : \varphi \to \psi \quad \Gamma \vdash \tau : \varphi}{\Gamma \vdash \pi\tau : \psi}$$

$$\frac{\Gamma \vdash \pi : \forall x.\varphi}{\Gamma \vdash \pi t : \varphi[x/t]}$$

$$\frac{\Gamma, p : \varphi \vdash \tau : \psi}{\Gamma \vdash \lambda p.\tau : \varphi \to \psi}$$

$$\frac{\Gamma \vdash \tau : \varphi}{\Gamma \vdash \lambda x.\tau : \forall x.\varphi}$$

$$\frac{\Gamma \vdash \pi_1 : \varphi_1 \quad \ldots \quad \Gamma \vdash \pi_n : \varphi_n \quad \Gamma \vdash \tau_1 : \psi_1 \quad \ldots \quad \Gamma \vdash \tau_m : \psi_m}{\Gamma \vdash \mathsf{case}\ X\ \mathsf{of}\ \overline{\equiv\varphi \Rightarrow \pi}, \overline{\psi \Rightarrow \tau} : \{X : \overline{\varphi} + \overline{\psi}\}}$$

$$\frac{\Gamma \vdash \pi : \{\delta : \overline{\equiv\varphi} + \overline{\psi}\}}{\Gamma \vdash \pi : \delta}\ \varphi_i = \delta$$

$$\frac{\Gamma \vdash \pi : \{\delta : \overline{\equiv\varphi} + \overline{\psi}\}}{\Gamma \vdash \pi : \delta} \quad \begin{array}{l} \psi_i[\overline{x}/\overline{t}] = \delta \text{ or} \\ \delta : e_*, \psi_i : e_* \end{array}$$

$$\frac{\Gamma \vdash \pi : \psi}{\Gamma \vdash \lambda X.\pi : \forall X.\psi}$$

$$\frac{\Gamma \vdash \pi : \forall X.\psi}{\Gamma \vdash \pi\ \varphi : \psi[X/\varphi]}$$

$$\frac{\Gamma \vdash \lambda p.\pi : \mu X.\varphi}{\Gamma \vdash \pi[p/\lambda p.\pi] : \mu X.\varphi}$$

$$\frac{\Gamma \vdash \pi[p/\lambda p.\pi] : \mu X.\varphi}{\Gamma \vdash \lambda p.\pi : \mu X.\varphi}$$

Figure 7.1: Typing rules for new proof terms

or δ unifies with or has the same kind as one of the ψ_i.

The type of the formula abstraction is the universal quantification over that formula. It is important to note that this is not the same as an abstraction over a proof variable p with the type φ. A term $\lambda p : \varphi.\pi$ would have the type $\varphi \rightarrow \psi$, where ψ is the type of π. When typing the application of a formula abstraction, the replacement of X with φ requires us to use the kind information. The only place such type variables appear is in case statements.

Finally, we have typing rules for proof terms with recursive types. The two typing rules correspond to unfolding and folding the proof term. We take an equi-recursive approach to the recursive types. In other words, $\mu X.\varphi$ is equivalent to $\varphi[X/\mu X.\varphi]$.

From the standpoint of an automated theorem prover, it is our type system that does most of the work of finding the correct steps to apply from a tactic. Most of this work is in choosing the correct case when applying a case statement to a type δ. Without any restrictions, δ may match several cases, requiring the type system to search though an exponential number of possible proofs. It is this search problem that makes implementing theorem prover tactics difficult. We regard the search problem as an implementation issue separate from the issue of formally representing tactics that we deal with in this chapter. For the sake of this chapter, we assume that when matching a type against possible cases in a case statement, we only explore the first match found, which removes the need for search at all.

We provide several rules for creating and manipulating proofs. The rules allow one to build proofs constructively. They manipulate a structure of the form $\mathfrak{L}; \mathfrak{T}$, as described in Chapter 3. The proof rules can easily be extended to handle theorem scoping as in Chapter 5.

In Figure 7.2, we present the rules for basic proof manipulation. The rules

$$\textbf{(assume)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \tau : \psi}{\mathcal{L}\ ;\ \mathcal{T},\quad A, p : \varphi \vdash \tau : \psi}$$

$$\textbf{(ident)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T}}{\mathcal{L}\ ;\ \mathcal{T},\quad p : \varphi \vdash p : \varphi}$$

$$\textbf{(mp)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi : \varphi \ \rightarrow\ \psi \quad A \vdash \tau : \varphi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi\ \tau : \psi}$$

$$\textbf{(discharge)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A, p : e \vdash \tau : \psi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \lambda p.\tau : e\ \rightarrow\ \psi}$$

$$\textbf{(publish)}\quad \frac{\mathcal{L}\qquad\qquad\qquad\qquad ;\ \mathcal{T},\quad \vdash \pi : \varphi}{\mathcal{L}, T = \lambda\overline{x}.\pi : \forall\overline{x}.\varphi\ ;\ \mathcal{T}}$$

$$\textbf{(cite)}\quad \frac{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T}}{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T},\quad \vdash \pi : \varphi}$$

$$\textbf{(inst)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi : \forall x.\varphi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi\ t : \varphi[x/t]}$$

$$\textbf{(norm}_t\textbf{)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash (\lambda x.\pi)\ t}{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash \pi[x/t] : \varphi}$$

$$\textbf{(norm}_p\textbf{)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash (\lambda p.\pi)\ \tau}{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash \pi[p/\tau] : \varphi}$$

$$\textbf{(forget)}\quad \frac{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T}}{\mathcal{L}_1, \mathcal{L}_2[T/\pi]\qquad ;\ \mathcal{T}[T/\pi]}$$

Figure 7.2: Proof Rules for Basic Theorem Manipulation

are very similar to the ones in Chapter 3. One difference is that the **(ident)** and **(assume)** rules allow one to introduce assumptions with formula types and not

just equations. We also add the **(inst)** rule, which allows us to instantiate variables over which a proof term is abstracted. Before, this was handled by the **(cite)** rule, but new rules give us the ability to have term abstractions in proof tasks, so we need to instantiate explicitly.

We also have **(norm$_t$)** and **(norm$_p$)** rules for performing β-reduction on applications of λ-abstractions over terms and proofs, respectively. It is important to note that the **(norm$_t$)** rule does not replace x in a proof in a case of a case statement where we perform unification if x occurs in the type for that case. In other words, for the proof term

$$\mathsf{case}\ X\ \mathsf{of}\ \overline{=\varphi \Rightarrow \pi}, \overline{\psi \Rightarrow \tau} : \{X :\ \overline{=\varphi}\ +\ \overline{\psi}\}$$

we do not replace x in τ_i if it occurs in ψ_i. We do, however, replace x in any of the π_i and φ_i in which they occur. This behavior is not unlike the case statement in Standard-ML. The **(forget)** rule allows us to remove a theorem from the library. With the possibility of recursive proof terms, the **(forget)** rule must perform its replacement of T with π and normalize repeatedly until T no longer appears.

In Figure 7.3, we introduce the proof rules to create, use, and manipulate theorems and tactics. The **(case)** rule combines existing proof tasks into a case statement. The types variable X can be unified with one of the types $\varphi_1, \ldots, \varphi_n$ or matched exactly with one of types of the assumptions $p_1, \ldots, p_m$. These types must be equations. The **(decase$^=$)** and **(decase)** allow us to determine which case the type δ matches and replace the case statement with the proof term for that specific case.

The rules **(fold)** and **(unfold)** are standard rules one would expect for dealing with recursive types. The **(publish^r)** rule allows us to publish recursive proof terms. In other words, these are tactics that use themselves in the proof. Recursion

$$\textbf{(case)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A, \overline{p : e} \vdash \pi_1 : \varphi_1 \quad \ldots \quad A, \overline{p : e} \vdash \pi_n : \varphi_n}{\mathcal{L}\ ;\ \mathcal{T},\quad \vdash \mathsf{case}\ X\ \mathsf{of}\ \overline{=e \Rightarrow p}, \overline{\varphi \Rightarrow \pi} : \{X :\ \overline{=e}\ +\ \overline{\varphi}\}}$$

$$\textbf{(decase}^{=}\textbf{)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \mathsf{case}\ \delta\ \mathsf{of}\ \overline{=\varphi \Rightarrow \pi}, \overline{\psi \Rightarrow \tau} : \delta}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi_i : \delta}\ \varphi_i = \delta$$

$$\textbf{(decase)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \mathsf{case}\ \delta\ \mathsf{of}\ \overline{=\varphi \Rightarrow \pi}, \overline{\psi \Rightarrow \tau} : \delta}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \tau_i[\overline{x}/\overline{t}] : \delta}\ \psi_i[\overline{x}/\overline{t}] = \delta$$

$$\textbf{(fold)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi[p/\lambda p.\pi] : \mu X.\varphi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \lambda p.\pi : \mu X.\varphi}$$

$$\textbf{(unfold)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \lambda p.\pi : \mu X.\varphi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi[p/\lambda p.\pi] : \mu X.\varphi}$$

$$\textbf{(publish}^{r}\textbf{)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad p : \mu X.\psi \vdash \pi : \varphi}{\mathcal{L}, p = \lambda\overline{x}.\lambda p.\pi : \forall\overline{x}.\mu X\varphi\ ;\ \mathcal{T}}\ \mu X.\psi = \forall\overline{x}.\mu X\varphi$$

$$\textbf{(forget}_1\textbf{)}\quad \frac{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T},\quad A \vdash T\ \tau : \psi}{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T},\quad A \vdash \pi\ \tau : \psi}$$

$$\textbf{(norm}_f\textbf{)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T}\ \ A \vdash (\lambda X.\pi)\ \psi}{\mathcal{L}\ ;\ \mathcal{T}\ \ A \vdash \pi[X/\psi] : \varphi}$$

Figure 7.3: Proof Rules for Tactics

of this nature is very important for tactics; we want to be able to repeat proof steps several times, such as in our example in Section 7.1. The rule takes a proof task with a single assumption of a recursive type and moves it to the library. The name assigned to the theorem is the same as the proof variable in the assumption. It is also necessary that the type of the proof variable and the type of the proof term added to the library are equivalent.

We add the **(forget$_1$)** rule, which functions much like **(forget)**, except we replace a theorem name with the proof of that theorem in only a single application in a single proof task and we do not remove the theorem from the library. This rule allows us to make explicit one step in the application of a tactic. Finally, the **(norm$_f$)** rule performs β-reduction on applications of λ-abstractions over formulas.

The steps in creating a tactic with several cases that recursively call the tactic would be as follows:

1. Use the **(assume)** and **(ident)** rules to add a proof variable with the type of the tactic to be created.

2. Create the proof terms for the cases of the tactic, using the assumption added in step 1 for the recursive calls.

3. Use the **(case)** rule to combine the proof terms created in step 2 into a single case statement.

4. Use the **(publish^r)** rule to publish the new tactic.

7.3 A Constructive Example

We can provide a tactic for our example in Section 7.1. First, we give a general description of the proof steps in our tactic. For a given x and a, if we want to prove $x \wedge a = a$, we use a recursive tactic that is quantified over an equation Y. If Y is of the form $x = x$, then we use ref to prove the equation true. If Y is of the form $x \wedge a = a$, then it suffices to apply trans to proofs of $x \wedge a = a \wedge a$ and $a \wedge a = a$. The latter follows directly from idemp$_\wedge$. For the former, we use cong$_\wedge$ on a proof of $x = a$, which we obtain by recursively calling the tactic.

Let

$$\varphi_R \quad = \quad \mu X.\forall x.\forall a.\forall Y. \quad X \ \rightarrow \ \{Y : x = x \ + \ x \wedge a = a\}$$

First, we use **(ident)** to create a proof task

$$R : \varphi_R \vdash R : \varphi_R \tag{7.5}$$

Next, let us create the cases of our tactic. We first create what will be the "base case" for our recursion. We use **(cite)**, **(inst)**, and **(assume)** to get the proof task

$$R : \varphi_R \vdash \mathsf{ref} \ x : x = x \tag{7.6}$$

For the recursive case, we use **(inst)** on (7.5) and the fact that we use equi-recursive types to get

$$\begin{aligned} R : \varphi_R \ \vdash \ & R \ x \ a \ (x = a : * \wedge * = *) \\ & : \varphi_R \ \rightarrow \ \{(x = a : * \wedge * = *) : \ x = x \ + \ x \wedge a = a\} \end{aligned} \tag{7.7}$$

We have made the kind of $x = a$ explicit in order to make sure it matches the $x \wedge a = a$ case in our sum formula type in φ_R. Next, we use **(mp)** on (7.7) and (7.5) to get

$$R : \varphi_R \vdash R \ x \ a \ (x = a : * \wedge * = *) \ R : (x = a : * \wedge * = *) \tag{7.8}$$

For the rest of the example, we do not show the kind of $x = a$ for readability. To use congruence of $\wedge$, we use **(cite)**, **(inst)**, and **(assume)** to add the task

$$R : \varphi_R \vdash \mathsf{cong}_\wedge \ x \ a \ a : x = a \ \rightarrow \ x \wedge a = a \wedge a \tag{7.9}$$

We combine (7.9) and (7.8) using **(mp)** to get

$$R : \varphi_R \vdash \mathsf{cong}_\wedge \ x \ a \ a \ (R \ x \ a \ (x = a) \ R) : x \wedge a = a \wedge a \tag{7.10}$$

For the proof of $a \wedge a = a$, we use **(cite)**, **(inst)**, and **(assume)** to add the proof task

$$R : \varphi_R \vdash \mathsf{idemp}_\wedge\ a : a \wedge a = a \tag{7.11}$$

We introduce transitivity with **(cite)**, **(inst)**, and **(assume)**

$$\begin{aligned} R : \varphi_R \vdash \mathsf{trans}\ (x \wedge a)\ (a \wedge a)\ a :&\ \ x \wedge a = a \wedge a \\ &\rightarrow\ a \wedge a = a \\ &\rightarrow\ x \wedge a = a \end{aligned} \tag{7.12}$$

Two applications of **(mp)** with (7.12),(7.10), and (7.11) give the completed recursive case for our tactic:

$$\begin{aligned} R : \varphi_R \vdash \mathsf{trans}\ &(x \wedge a)\ (a \wedge a)\ a && : x \wedge a = a \\ &(\mathsf{cong}_\wedge\ x\ a\ a\ (R\ x\ a\ (x = a)\ R)) \\ &(\mathsf{idemp}_\wedge\ a) \end{aligned} \tag{7.13}$$

Now we use the **(case)** rule to combine (7.6) and (7.13) for our tactic:

$$\begin{aligned} R : \varphi_R \vdash\ &\mathsf{case}\ Y\ \mathsf{of} && : \{Y :\ x = x\ +\ x \wedge a = a\} \\ &(x = x) \Rightarrow \mathsf{ref}\ x \\ &(x \wedge a = a) \Rightarrow \mathsf{trans}\ \ (x \wedge a)\ (a \wedge a)\ a \\ &\qquad\qquad\qquad\quad (\mathsf{cong}_\wedge\ x\ a\ a \\ &\qquad\qquad\qquad\qquad\ (R\ x\ a\ (x = a)\ R)) \\ &\qquad\qquad\qquad\quad (\mathsf{idemp}_\wedge\ a) \end{aligned}$$

Finally, we use the **(publish^r)** rule to publish the tactic as

$$
\begin{aligned}
R = \lambda x.\lambda a.\lambda Y.\lambda R.\ &\mathsf{case}\ Y\ \mathsf{of} \\
&(x = x) \Rightarrow \mathsf{ref}\ x \\
&(x{\wedge}a = a) \Rightarrow \mathsf{trans}\ \ (x{\wedge}a)\ (a{\wedge}a)\ a \\
&\qquad\qquad\qquad\quad (\mathsf{cong}_{\wedge}\ x\ a\ a\ (R\ x\ a\ (x = a)\ R)) \\
&\qquad\qquad\qquad\quad (\mathsf{idemp}_{\wedge}\ a)
\end{aligned}
$$

The type of this tactic is

$$\forall x.\forall a.\forall Y.\quad \varphi_R\ \rightarrow\ \{Y :\ x = x\ +\ x{\wedge}a = a\}$$

Notice that $\forall x.\forall a.\forall Y.\quad \varphi_R\ \rightarrow\ \{Y : x = x\ +\ x{\wedge}a = a\}$ is equal to φ_R, which is necessary for applying the rule.

We now have a tactic that given an x of the form $a{\wedge}\ldots{\wedge}a$ will provide a proof of $a{\wedge}\ldots{\wedge}a = a$. If applied to a term that is not of this form, the tactic will not have a type.

We can now apply the tactic to create a new proof. We use the **(cite)** and **(inst)** rules just as we do on theorems to create the proof task

$$\vdash R\ (b{\wedge}b{\wedge}b)\ b\ (b{\wedge}b{\wedge}b{\wedge}b = b) : \varphi_R\ \rightarrow\ b{\wedge}b{\wedge}b{\wedge}b = b$$

We then use **(cite)** and **(mp)** to get the conclusion we desire.

$$\vdash R\ (b{\wedge}b{\wedge}b)\ b\ (b{\wedge}b{\wedge}b{\wedge}b = b)\ R : b{\wedge}b{\wedge}b{\wedge}b = b \tag{7.14}$$

We may want to make one step of the application of the tactic R explicit. First, we use the **(forget$_1$)** rule on (7.14) to replace the name of the tactic with its body

and then use the normalize rules to perform β-reduction to get

$$\begin{array}{lll} \vdash \ \mathsf{case}\ (b{\wedge}b{\wedge}b{\wedge}b = b)\ \mathsf{of} & & : b{\wedge}b{\wedge}b{\wedge}b = b \quad (7.15) \\ \quad (x = x) \Rightarrow \mathsf{ref}\ x & & \\ \quad (x{\wedge}a = a) \Rightarrow \mathsf{trans}\ \ (x{\wedge}a)\ (a{\wedge}a)\ a & & \\ \qquad\qquad\qquad\qquad (\mathsf{cong}_{\wedge}\ x\ a\ a\ (R\ x\ a\ (x = a)\ R)) & & \\ \qquad\qquad\qquad\qquad (\mathsf{idemp}_{\wedge}\ a) & & \end{array}$$

We can then use **(decase)** to replace the case statement with the specific case that is matched, where $b{\wedge}b{\wedge}b{\wedge}b = b$ unifies with $x{\wedge}a = a$ under the substitution $[x/b{\wedge}b{\wedge}b, a/b]$.

$$\begin{array}{lll} \vdash \ \mathsf{trans}\ \ (b{\wedge}b{\wedge}b{\wedge}b)\ (b{\wedge}b)\ b & & : b{\wedge}b{\wedge}b{\wedge}b = b \quad (7.16) \\ \qquad\qquad (\mathsf{cong}_{\wedge}\ (b{\wedge}b{\wedge}b)\ b\ b & & \\ \qquad\qquad\quad (R\ (b{\wedge}b{\wedge}b)\ b\ (b{\wedge}b{\wedge}b = b)\ R)) & & \\ \qquad\qquad (\mathsf{idemp}_{\wedge}\ b) & & \end{array}$$

Now one of the steps of the proof is explicit whil e the others are implicitly captured in the application of the tactic R.

7.4 Conclusions

We have presented a proof-theoretic approach in which tactics are treated at the same level as theorems and proofs. The proof rules allow us to create, manipulate, and apply tactics in a way that is completely formal and independent of system-level decisions regarding proof search. Many important tactics can be represented in the relatively simple system we have demonstrated, particularly in algebras such as our Boolean example.

Representing tactics at this level has several advantages for automated theorem

provers, from both the perspective of a user and a developer. For users, powerful tactics can be created without needing to learn a separate tactics language. However, the power of the language used to implement the theorem prover can be harnessed to make proof search as complete and efficient as desired. Additionally, when combined with the work in Chapter 5, tactics can be put into a local scope and abstractions can be manipulated just as theorems can be, a powerful ability that current theorem provers lack.

Chapter 8

Future Work

There are several promising directions of future work based on the formalism presented in this thesis. These directions include both theoretical work and application-based work.

8.1 Proof Refactorization

One of the primary purposes of a formal representation of proofs is to facilitate the reuse of proofs. With the formalization based on the λ-calculus used here, we are able to take advantage of techniques for code reuse in programs. The repeated use of lemmas in proofs is similar to the reuse of locally defined constants in a computer program. For a user creating a small set of theorems from scratch, it might be possible to fathom all of the subproofs that should be lemmas that are proved once and then referenced in other proofs. However, for larger sets of proofs with more intricate subproofs, it may be difficult to know what constitutes a good set of lemmas to be reused in several proofs. For proofs that have already been completed, there may not even be the opportunity for the user to create a set of lemmas.

With these ideas in mind, an open area of research is the discovery of common subproofs, a process we call *proof refactorization*. The idea is similar to that of *code refactorization* found in integrated development environments such as Eclipse [38]. Code refactorization allows one to take a block of code and convert it to a function by passing local variables in as parameters. The function is then available for calling in other parts of the program. One could apply this same technique

to proofs, making subproofs available as lemmas by abstracting out variables and assumptions.

One step further is to try to automate the process by using *common subexpression elimination*, where common terms are discovered and abstracted out. The technique is already used in compilers to optimize the operation of code by temporarily storing the results of a computation preformed repeatedly. The elimination of common subproofs is similar. We want to automatically discover proofs that are the same and make them into lemmas. Ideally, we should not only find proofs that are syntactically identical, but proofs that are specialized versions of some lemma we can abstract out.

8.2 Tactic Type Systems

The representation of tactics presented in Chapter 7 depends on a type system to apply the tactics in a sound way. From an implementation standpoint, this type system is responsible for the search for the correct steps of deduction to take. We stated that the type system is an issue separate from the formal representation of the tactics and therefore is a problem beyond the scope of this thesis. Several aspects of the type system could be explored, particularly with respect to the search it performs in applying a case statement.

One important question is how powerful to make the search so that it is reasonably useful. The most powerful search would explore every possible case in a case statement exhaustively until a proper typing derivation is found or until no more cases can be explored. However, such a search is potentially very slow, if it even terminates at all. On the other extreme is a type system that performs no search, only exploring the first match in a case statement. While very efficient, it

is possible that such a strategy will not be able to apply a tactic that could be used if a more ambitious search were applied. Is there some search strategy in between these two that is ideal? Does the ideal search strategy depend on the domain of theorems and proofs?

We may also be able to design our tactics in such a way that a less complex search strategy suffices. There are several approaches we could take:

- **Equations on which tactics are recursively called should get syntactically shorter.** If equations get shorter, then termination can be guaranteed, thus eliminating the need to detect cycles or use some sort of depth limit in the search.

- **Ensure that an equation can only match one case in a tactic.** Using tactics in which only one match is possible limits the branching factor in our search to 1, making the search much more manageable. There are several tactics that are of this structure, including substitution using congruence and transitivity, an important tactic in KAT.

- **Order the cases to minimize search.** It stands to reason that the search strategy is going to start from the first case and work its way to the last. Therefore, we should design our tactics intelligently to put cases that are likely to perform less searching (e.g., ones that reduce the size of an equation significantly in recursive calls) before those that require more searching.

8.3 Implementation

We have provided the basic infrastructure that could be used to build a general-purpose interactive theorem prover. The engineering of such a project is large

enough to be a thesis itself. The importance in any implementation is to maintain a strong relationship with the underlying formalism described in this thesis.

One issue of interest is the creation of useful rules that build on those in Figure 5.5. The rules as presented are sound transformations that make very specific alterations to proof terms one step at a time. To be useful, the system should have meta-steps that perform several steps at once with the system internally justifying each step with our rules. For example, in order to use the **(specialize)** rule, one would likely use the **(reorder)**, **(rename)**, and **(pull)** rules several times to get the proof term into the correct form. Ideally, a user should be able to perform a specialize operation that automates this process. The same is true for the calling of **(merge)**, which can require the use of the **(generalize)** rule.

Another issue is the representation of the data in some distributable format. KAT-ML uses its own XML encoding for the data that takes advantage of proof terms specialized for KAT. XML formats continue to gain popularity in the representation of mathematics, as described throughout Chapter 2. With its extensive support for mathematics, including attributes for assigning importance to proofs, OMDoc [57] would be a reasonable language for representing proofs for wider dissemination.

8.4 Online Library Sharing

As described in Section 4.2.7, one of the goals in the KAT-ML theorem prover was the creation of a central repository of KAT theorems. A general-purpose theorem prover could also benefit from an online library of theorems. Libraries of theorem provers currently available online are static objects separate from the theorem provers themselves. The formal representation of proofs presented in this thesis

lends itself well to a much more active relationship between an online repository of theorems and the theorem prover itself.

An intriguing idea is to use the online repository as a shared resource of all the proven theorems that could be updated and accessed by all those using a theorem prover. Working with such a prover would have the following mode of interaction:

1. A user starts up the system, which automatically downloads newly added theorems.

2. The user continues work on a proof that has so far been elusive. Looking through the new data downloaded from the online repository, the user notices that one of the new theorems is exactly what is necessary to finish the proof.

3. The user finishes that proof and marks it to be sent to the online repository.

4. Before the theorem prover is closed, it uploads marked, completed theorems to the central repository, where they are verified and added to the library.

Access to a constantly changing online repository like this allows all users to benefit from the work of others.

Another interesting mode of interaction would be peer-to-peer interaction between the users of the theorem prover without a central repository. The goal would be to allow users to share proofs, both complete and incomplete, with other users in an attempt to make their work available to others and to seek help on proofs with which they are having difficulty. The sharing could look similar to the music library sharing feature available in Apple's iTunes software, in which users can make their library available online for streaming by others within their own subnet [5].

Chapter 9

Conclusions

We have presented a proof-theoretic approach to mathematical knowledge management that exhibits several desired properties. We represent the relationship between proofs, the library of proofs and theorems, and proof tactics in a way that allows them to be treated at the same formal level as proofs themselves. Consequently, what have until now been system-level constructs can be integrated into the underlying proof logic, where more complex constructs such as scoping and tactics can be represented with well-studied parts of the typed λ-calculus.

Fundamental to the design of this proof representation have been the five properties discussed in Chapter 1: independence, structure, an underlying formalism, adaptability, and presentability. Adherence to these five properties is a good measure for any representation for proofs. While the representation of proofs in popular theorem provers has been able to provide a subset of these properties, no system has been able to provide them all. The proof library representation we have described in this thesis does exhibit all five desired properties.

Up until now, considerations for mathematical knowledge management have been secondary to work in expanding the automation of theorem provers so that more proofs could be completed with less human interaction. This work has resulted in the expanding of formal digital libraries to include much of the basis of mathematics, as well as many more specific topics in computer science. It is because of this expansion that the issues of effectively representing proof libraries must now be paramount.

One of the goals of theorem provers is to formalize mathematics in a way that

makes it accessible at all levels, particularly advanced students and researchers. In order to succeed at this goal, theorem provers need to stay true to that formalization as much as possible, as its benefits have already been proven. What we previously viewed as implementation details or informal notions can now be seen for what they really are: elements with inherent structure that can be captured by an underlying proof theory.

The work in mathematical knowledge management is in its infancy, with several aspects still to be explored and understood. Some approaches are more formal than others. All of the approaches have one thing in common: they are trying to provide a way to organize mathematical information in such a way that it can be understood and used by everyone, from young children just learning arithmetic to professors at the forefront of mathematical research. The proof-theoretic representation of proof libraries presented in this thesis provides a strong formal foundation for realizing mathematical knowledge management's ultimate goals.

Appendix A

Library Organization Soundness

Soundness for the proof system requires that a sequence of applications of the rules transforms a proof term of a type φ into a new proof term of a type ψ that is equivalent modulo first-order equivalence. Let $\pi \Rightarrow \tau$ mean that the proof term τ is derivable from π using our proof rules in one step.

We make use of the following identities and properties from first-order logic.

$$a \to (b \wedge c) \quad \equiv \quad (a \to b) \wedge (a \to c) \tag{A.1}$$

$$\forall x.a \wedge \forall x.b \quad \equiv \quad \forall x.(a \wedge b) \tag{A.2}$$

Theorem A.1 *If $\pi \Rightarrow \tau$ and $\Gamma \vdash \pi : \varphi$, then $\Gamma \vdash \tau : \psi$, where φ and ψ are equivalent modulo first-order equivalence.*

Proof. The proof is by induction on deductions Π of the form $\Gamma \vdash \pi : \varphi$.

- Let Π be a deduction of the form

$$\Gamma \vdash \lambda x. \quad (\pi_1; \ldots; \pi_n) \ : \ \forall x. \quad (\varphi_1 \wedge \ldots \wedge \varphi_n) \tag{A.3}$$

We can use our **(push)** rule to get a proof term of the form

$$\lambda x.\pi_1; \ldots; \lambda x.\pi_n \tag{A.4}$$

The typing derivation for (A.3) must have been of the form

$$\dfrac{\dfrac{\dfrac{\Pi_1 \qquad\qquad \Pi_n}{\Gamma \vdash \pi_1 \ : \ \varphi_1 \quad \cdots \quad \Gamma \vdash \pi_1 \ : \ \varphi_n}}{\Gamma \vdash \pi_1; \ldots; \pi_n \ : \ (\varphi_1 \wedge \ldots \wedge \varphi_n)}}{\Gamma \vdash \lambda x. \quad (\pi_1; \ldots; \pi_n) \ : \ \forall x. \quad (\varphi_1 \wedge \ldots \wedge \varphi_n)}$$

From the deductions $\Pi_1, \ldots, \Pi_n$, we can deduce the following.

$$\dfrac{\dfrac{\begin{matrix}\Pi_1\\ \Gamma \vdash \pi_1 \ : \ \varphi_1\end{matrix}}{\Gamma \vdash \lambda x.\pi_1 \ : \ \forall x.\varphi_1} \quad \cdots \quad \dfrac{\begin{matrix}\Pi_n\\ \Gamma \vdash \pi_n \ : \ \varphi_n\end{matrix}}{\Gamma \vdash \lambda x.\pi_n \ : \ \forall x.\varphi_n}}{\Gamma \vdash \lambda x.\pi_1; \ldots; \lambda x.\pi_n \ : \ \forall x.\varphi_1 \wedge \ldots \wedge \forall x.\varphi_n}$$

This is a deduction of the type of (A.4). Furthermore, we know that the types $\forall x.(\varphi_1 \wedge \ldots \wedge \varphi_n)$ and $\forall x.\varphi_1 \wedge \ldots \wedge \forall x.\varphi_n$ are equivalent by (A.2).

- Let Π be a deduction of the form

$$\Gamma \vdash \lambda p. \quad (\pi_1; \ldots; \pi_n) \ : \ e \to (\varphi_1 \wedge \ldots \wedge \varphi_n) \tag{A.5}$$

We can use our **(push)** rule to get a proof term of the form

$$\lambda p.\pi_1; \ldots; \lambda p.\pi_n \tag{A.6}$$

The typing derivation for (A.5) must be of the form

$$\dfrac{\dfrac{\begin{matrix}\Pi_1\\ \Gamma, p : e \vdash \pi_1 \ : \ \varphi_1\end{matrix} \quad \cdots \quad \begin{matrix}\Pi_n\\ \Gamma, P : e \vdash \pi_1 \ : \ \varphi_n\end{matrix}}{\Gamma, p : e \vdash \pi_1; \ldots; \pi_n \ : \ (\varphi_1 \wedge \ldots \wedge \varphi_n)}}{\Gamma \vdash \lambda p. \quad (\pi_1; \ldots; \pi_n) \ : \ e \to (\varphi_1 \wedge \ldots \wedge \varphi_n)}$$

From the deductions $\Pi_1, \ldots, \Pi_n$, we can deduce the following.

$$\dfrac{\dfrac{\begin{matrix}\Pi_1\\ \Gamma, P : e \vdash \pi_1 \ : \ \varphi_1\end{matrix}}{\Gamma \vdash \lambda P.\pi_1 \ : \ e \to \varphi_1} \quad \cdots \quad \dfrac{\begin{matrix}\Pi_n\\ \Gamma, P : e \vdash \pi_n \ : \ \varphi_n\end{matrix}}{\Gamma \vdash \lambda P.\pi_n \ : \ e \to \varphi_n}}{\Gamma \vdash \lambda P.\pi_1; \ldots; \lambda P.\pi_n \ : \ e \to \varphi_1 \wedge \ldots \wedge e \to \varphi_n}$$

This is the deduction of the type of (A.6). Furthermore, we know that the types $e \to (\varphi_1 \wedge \ldots \wedge \varphi_n)$ and $e \to \varphi_1 \wedge \ldots \wedge e \to \varphi_n$ are equivalent by (A.1).

- Let Π be a deduction of the form

$$\Gamma \vdash \lambda x.\pi_1; \ldots; \lambda x.\pi_n : \forall x.\varphi_1 \wedge \ldots \wedge \forall x.\varphi_n \tag{A.7}$$

We can use our **(pull)** rule to get a proof term of the form

$$\lambda x. \quad (\pi_1; \ldots; \pi_n) \tag{A.8}$$

The typing derivation for (A.7) must be of the form

$$\dfrac{\dfrac{\begin{matrix}\Pi_1\\ \Gamma \vdash \pi_1 \ : \ \varphi_1\end{matrix}}{\Gamma \vdash \lambda x.\pi_1 \ : \ \forall x.\varphi_1} \quad \cdots \quad \dfrac{\begin{matrix}\Pi_n\\ \Gamma \vdash \pi_n \ : \ \varphi_n\end{matrix}}{\Gamma \vdash \lambda x.\pi_n \ : \ \forall x.\varphi_n}}{\Gamma \vdash \lambda x.\pi_1; \ldots; \lambda x.\pi_n \ : \ \forall x.\varphi_1 \wedge \ldots \wedge \forall x.\varphi_n}$$

From the deductions $\Pi_1, \ldots, \Pi_n$, we can deduce the following.

$$\dfrac{\dfrac{\begin{matrix}\Pi_1 & & \Pi_n\\ \Gamma \vdash \pi_1 \ : \ \varphi_1 & \cdots & \Gamma \vdash \pi_1 \ : \ \varphi_n\end{matrix}}{\Gamma \vdash \pi_1; \ldots; \pi_n \ : \ (\varphi_1 \wedge \ldots \wedge \varphi_n)}}{\Gamma \vdash \lambda x. \quad (\pi_1; \ldots; \pi_n) \ : \ \forall x. \quad (\varphi_1 \wedge \ldots \wedge \varphi_n)}$$

This is the deduction of the type of (A.8). Furthermore, we know that the types $\forall x.\varphi_1 \wedge \ldots \wedge \forall x.\varphi_n$ and $\forall x.(\varphi_1 \wedge \ldots \wedge \varphi_n)$ are equivalent by (A.2).

- Let Π be a deduction of the form

$$\Gamma \vdash \lambda p.\pi_1; \ldots; \lambda p.\pi_n : e \to \varphi_1 \wedge \ldots \wedge e \to \varphi_n \tag{A.9}$$

We can use our **(pull)** rule to get a proof term of the form

$$\lambda p. \quad (\pi_1; \ldots; \pi_n) \tag{A.10}$$

The typing derivation for (A.9) must be of the form

$$\dfrac{\dfrac{\begin{matrix}\Pi_1\\ \Gamma, p : e \vdash \pi_1 \ : \ \varphi_1\end{matrix}}{\Gamma \vdash \lambda P.\pi_1 \ : \ e \to \varphi_1} \quad \cdots \quad \dfrac{\begin{matrix}\Pi_n\\ \Gamma, p : e \vdash \pi_n \ : \ \varphi_n\end{matrix}}{\Gamma \vdash \lambda p.\pi_n \ : \ e \to \varphi_n}}{\Gamma \vdash \lambda p.\pi_1; \ldots; \lambda p.\pi_n \ : \ e \to \varphi_1 \wedge \ldots \wedge e \to \varphi_n}$$

From the deductions $\Pi_1, \ldots, \Pi_n$, we can deduce the following.

$$\dfrac{\dfrac{\begin{matrix}\Pi_1 & & \Pi_n\\ \Gamma, P : e \vdash \pi_1 \ : \ \varphi_1 & \cdots & \Gamma, P : e \vdash \pi_1 \ : \ \varphi_n\end{matrix}}{\Gamma, P : e \vdash \pi_1; \ldots; \pi_n \ : \ (\varphi_1 \wedge \ldots \wedge \varphi_n)}}{\Gamma \vdash \lambda P. \quad (\pi_1; \ldots; \pi_n) \ : \ e \to (\varphi_1 \wedge \ldots \wedge \varphi_n)}$$

This is the deduction of the type for (A.10). Furthermore, we know that the types $e \to \varphi_1 \wedge \ldots \wedge e \to \varphi_n$ and $e \to (\varphi_1 \wedge \ldots \wedge \varphi_n)$ are equivalent by (A.1).

- Let Π be a deduction of the form

$$\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\pi_L}, \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end} : \psi \tag{A.11}$$

Using our **(split)** rule, we can get a proof term of the form

$$\mathsf{let}\ \overline{L} = \overline{\pi_L}\ \mathsf{in}\ \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}\ \mathsf{end} \tag{A.12}$$

The typing derivation for (A.11) must be as follows.

$$\begin{array}{rcl}
\Pi_1 & : & \Gamma \vdash \pi_{L_1} : \varphi_{L_1} \\
\Pi_2 & : & \Gamma, L_1 : \varphi_{L_1} \vdash \pi_{L_2} : \varphi_{L_2} \\
\ldots & & \\
\Pi_n & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_{n-1} : \varphi_{L_{n-1}} \vdash \pi_{L_n} : \varphi_{L_n} \\
\Xi_1 & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n} \vdash \pi_{M_1} : \varphi_{M_1} \\
\Xi_2 & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1} \vdash \pi_{M_2} : \varphi_{M_2} \\
\ldots & & \\
\Xi_m & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1}, \ldots, M_{m-1} : \varphi_{M_{m-1}} \vdash \pi_{M_m} : \varphi_{M_m} \\
\Xi & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1}, \ldots, M_m : \varphi_{M_m} \vdash \tau : \psi \\
\hline
 & & \Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\pi_L}, \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end} : \psi
\end{array}$$

First, we use $\Xi_1, \ldots, \Xi_m, \Xi$ to get a derivation Π_M

$$\begin{array}{rcl}
\Xi_1 & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n} \vdash \pi_{M_1} : \varphi_{M_1} \\
\Xi_2 & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1} \vdash \pi_{M_2} : \varphi_{M_2} \\
\ldots & & \\
\Xi_m & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1}, \ldots, M_{m-1} : \varphi_{M_{m-1}} \vdash \pi_{M_m} : \varphi_{M_m} \\
\Xi & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1}, \ldots, M_m : \varphi_{M_m} \vdash \tau : \psi \\
\hline
 & & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n \vdash \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end} : \psi
\end{array}$$

We combine Π_M with $\Pi_1, \ldots, \Pi_n$ to get

$$\dfrac{\begin{array}{rcl}\Pi_1 & : & \Gamma \vdash \pi_{L_1} : \varphi_{L_1} \\ \Pi_2 & : & \Gamma, L_1 : \varphi_{L_1} \vdash \pi_{L_2} : \varphi_{L_2} \\ \ldots & & \\ \Pi_n & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_{n-1} : \varphi_{L_{n-1}} \vdash \pi_{L_n} : \varphi_{L_n} \\ \Pi_M & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n \vdash \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end} : \psi\end{array}}{\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\pi_L}\ \mathsf{in}\ \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}\ \mathsf{end} : \psi}$$

This is a derivation for the type of (A.12), which is the same as the type in (A.11).

- Let Π be a deduction of the form

$$\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\pi_L}\ \mathsf{in}\ \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}\ \mathsf{end} : \psi \tag{A.13}$$

Using our **(merge)** rule, we can get a proof term of the form

$$\mathsf{let}\ \overline{L} = \overline{\pi_L}, \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end} \tag{A.14}$$

The typing derivation for (A.13) must be as follows.

$$\dfrac{\begin{array}{rcl}\Pi_1 & : & \Gamma \vdash \pi_{L_1} : \varphi_{L_1} \\ \Pi_2 & : & \Gamma, L_1 : \varphi_{L_1} \vdash \pi_{L_2} : \varphi_{L_2} \\ \ldots & & \\ \Pi_n & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_{n-1} : \varphi_{L_{n-1}} \vdash \pi_{L_n} : \varphi_{L_n} \\ \Pi_M & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n \vdash \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end} : \psi\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\pi_L}\ \mathsf{in}\ \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}\ \mathsf{end} : \ \varphi_{L_1} \to \ldots \to \varphi_{L_n} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \varphi_{M_1} \to \ldots \to \varphi_{M_m} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \psi\end{array}}$$

where Π_M is of the form

$$\begin{array}{rcl}
\Xi_1 & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n} \vdash \pi_{M_1} : \varphi_{M_1} \\
\Xi_2 & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1} \vdash \pi_{M_2} : \varphi_{M_2} \\
\ldots & & \\
\Xi_m & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1}, \ldots, M_{m-1} : \varphi_{M_{m-1}} \vdash \pi_{M_m} : \varphi_{M_m} \\
\Xi & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1}, \ldots, M_m : \varphi_{M_m} \vdash \tau : \psi \\
\hline
\multicolumn{3}{c}{\Gamma, L_1 : \varphi_{L_1}, \ldots, L_n \vdash \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end} : \psi}
\end{array}$$

We use $\Pi_1, \ldots, \Pi_n, \Xi_1, \ldots, \Xi_m,$, and Ξ to get a derivation

$$\begin{array}{rcl}
\Pi_1 & : & \Gamma \vdash \pi_{L_1} : \varphi_{L_1} \\
\Pi_2 & : & \Gamma, L_1 : \varphi_{L_1} \vdash \pi_{L_2} : \varphi_{L_2} \\
\ldots & & \\
\Pi_n & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_{n-1} : \varphi_{L_{n-1}} \vdash \pi_{L_n} : \varphi_{L_n} \\
\Xi_1 & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n} \vdash \pi_{M_1} : \varphi_{M_1} \\
\Xi_2 & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1} \vdash \pi_{M_2} : \varphi_{M_2} \\
\ldots & & \\
\Xi_m & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1}, \ldots, M_{m-1} : \varphi_{M_{m-1}} \vdash \pi_{M_m} : \varphi_{M_m} \\
\Xi & : & \Gamma, L_1 : \varphi_{L_1}, \ldots, L_n : \varphi_{L_n}, M_1 : \varphi_{M_1}, \ldots, M_m : \varphi_{M_m} \vdash \tau : \psi \\
\hline
\multicolumn{3}{c}{\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\pi_L}, \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end} :\ \varphi_{L_1} \to \ldots \to \varphi_{L_n}} \\
\multicolumn{3}{c}{\to \varphi_{M_1} \to \ldots \to \varphi_{M_m}} \\
\multicolumn{3}{c}{\to \psi}
\end{array}$$

This is a derivation for the type of (A.14), which is the same as the type in (A.13).

Before we can prove soundness using the **(generalize)** and **(specialize)** rules, we must prove several lemmas regarding substitution. We state the lemmas as meta-typing rules.

Lemma A.2

$$\frac{\Gamma, p : \varphi_p, L : \varphi \vdash \tau : \psi}{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \tau[L/L\ p] : \psi}$$

where $L = \pi$ does not appear in τ.

Proof. The proof is by induction on type derivations. Assume $\Gamma, p : \varphi_p, L : \varphi \vdash \tau : \psi$.

- $\tau = p$: The case follows trivially from the assumption.
- $\tau = q, q \neq q$: The case follows trivially from the assumption.
- $\tau = M, M \neq L$: The case follows trivially from the assumption.
- $\tau = L$: From our assumption, we know

  $$\Gamma, p : \varphi_p, L : \varphi \vdash L : \varphi$$

  We need to type $L[L/L\ p]$, which is $L\ p$. The type derivation for this term is

  $$\frac{\overline{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash\ \ L : \varphi_p \to \varphi} \quad \overline{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash p : \varphi_p}}{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash L\ p : \varphi}$$

  This is what we needed to show.

- $\tau = \lambda x.\pi$: The typing derivation is

  $$\frac{\begin{matrix}\Pi \\ \Gamma, p : \varphi_p, L : \varphi \vdash \pi : \psi\end{matrix}}{\Gamma, p : \varphi_p, L : \varphi \vdash \lambda x.\pi : \forall x.\psi}$$

  By induction on Π, we have the deduction

  $$\Pi' : \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \pi[L/L\ p] : \psi$$

  From our typing rule for term abstractions, we have the deduction

  $$\frac{\begin{matrix}\Pi' \\ \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \pi[L/L\ p] : \psi\end{matrix}}{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \lambda x.\pi[L/L\ p] : \forall x.\psi}$$

  which is what we needed.

- $\tau = \lambda q.\pi$: The typing derivation is

$$\frac{\begin{array}{c}\Pi \\ \Gamma, p : \varphi_p, L : \varphi, q : d \vdash \pi : \psi\end{array}}{\Gamma, p : \varphi_p, L : \varphi \vdash \lambda q.\pi : d \to \psi}$$

  By induction on Π, we have a deduction

$$\Pi' : \Gamma, p : \varphi_p, L : \varphi_p \to \varphi, q : d \vdash \pi[L/L\ p] : \psi$$

  From our typing rule for term abstractions, we have the deduction

$$\frac{\begin{array}{c}\Pi' \\ \Gamma, p : \varphi_p, L : \varphi_p \to \varphi, q : d \vdash \pi[L/L\ p] : \psi\end{array}}{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash (\lambda q.\pi)[L/L\ p] : d \to \psi}$$

  which is what we needed.

- $\tau = \pi t$: The typing rule in this case is

$$\frac{\begin{array}{c}\Pi \\ \Gamma, p : \varphi_p, L : \varphi \vdash \pi : \forall x.\psi\end{array}}{\Gamma, p : \varphi_p, L : \varphi \vdash \pi t : \psi[x/t]}$$

  By induction on Π, we have a typing derivation

$$\Pi' : \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \pi[L/L\ p] : \forall x.\psi$$

  We then use Π' to deduce

$$\frac{\begin{array}{c}\Pi' \\ \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \pi[L/L\ p] : \forall x.\psi\end{array}}{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \pi[L/L\ p]\ t : \psi}$$

  Since $\pi[L/L\ p]\ t = (\pi\ t)[L/L\ p]$, we have the proof we needed.

- $\tau = \pi_1\pi_2$: The typing rule for proof application is

$$\frac{\begin{array}{c}\Pi_1 \\ \Gamma, p : \varphi_p, L : \varphi \vdash \pi_1 : e \to \psi\end{array} \quad \begin{array}{c}\Pi_2 \\ \Gamma, p : \varphi_p, L : \varphi \vdash \pi_2 : e\end{array}}{\Gamma, p : \varphi_p, L : \varphi \vdash \pi_1\pi_2 : \psi}$$

By induction on Π_1 and Π_2, we get the deductions

$$\begin{aligned}\Pi_1' &= \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \pi_1[L/L\ p] : e \to \psi \\ \Pi_2' &= \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \pi_2[L/L\ p] : e\end{aligned}$$

We use Π_1' and Π_2' to create the deduction

$$\frac{\begin{array}{cc}\Pi_1' & \Pi_2' \\ \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \pi_1[L/L\ p] : e \to \psi & \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \pi_2[L/L\ p] : e\end{array}}{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash (\pi_1\pi_2)[L/L\ p] : \psi}$$

which is what we needed to showed.

- $\pi_1; \ldots; \pi_n$: The typing rule for a sequence is

$$\frac{\begin{array}{ccc}\Pi_1 & & \Pi_n \\ \Gamma, p : \varphi_p, L : \varphi \vdash \tau_1 : \varphi_1 & \ldots & \Gamma, p : \varphi_p, L : \varphi \vdash \tau_n : \varphi_n\end{array}}{\Gamma, p : \varphi_p, L : \varphi \vdash \tau_1; \ldots; \tau_n : \varphi_1 \wedge \ldots \wedge \varphi_n}$$

  By induction on the Π_i deductions, we get n deductions of the form

$$\Pi_i' : \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \tau_i[L/L\ p] : \varphi_i$$

  We use the Π_i' to create a deduction

$$\frac{\begin{array}{cc}\Pi_1 & \Pi_n \\ \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \tau_1[L/L\ p] : \varphi_1 & \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \tau_n[L/L\ p] : \varphi_n\end{array}}{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash (\tau_1; \ldots; \tau_n)[L/L\ p] : \varphi_1 \wedge \ldots \wedge \varphi_n}$$

  which is what we wanted to show.

- $\mathsf{let}\ M_1 = \tau_1 \ldots M_n = \tau_n\ \mathsf{in}\ \tau\ \mathsf{end}$: The typing rule for a let expression is

$$\frac{\begin{array}{rcl}\Pi_1 & : & \Gamma, p : \varphi_p, L : \varphi \vdash \tau_1 : \varphi_1 \\ \Pi_2 & : & \Gamma, p : \varphi_p, L : \varphi, M_1 : \varphi_1 \vdash \tau_2 : \varphi_2 \\ \ldots & & \\ \Pi_n & : & \Gamma, p : \varphi_p, L : \varphi, M_1 : \varphi_1, \ldots, M_{n-1} : \varphi_{n-1} \vdash \tau_n : \varphi_n \\ \Pi & : & \Gamma, p : \varphi_p, L : \varphi, M_1 : \varphi_1, \ldots, M_n : \varphi_n \vdash \tau : \varphi\end{array}}{\Gamma, p : \varphi_p, L : \varphi \vdash \mathsf{let}\ M_1 = \tau_1 \ldots M_n = \tau_n\ \mathsf{in}\ \tau\ \mathsf{end} : \varphi}$$

By induction on the Π_i and Π, we get the deductions

$$\begin{aligned}\Pi'_i &\equiv \Gamma, p : \varphi_p, L : \varphi_p \to \varphi, M_1 : \varphi_1, \ldots, M_{i-1} : \varphi_{i-1} \vdash \tau_i[L/L\ p] : \varphi_i \\ \Pi' &\equiv \Gamma, p : \varphi_p, L : \varphi_p \to \varphi, M_1 : \varphi_1, \ldots, M_n : \varphi_n \vdash \tau[L/L\ p] : \varphi\end{aligned}$$

Note that our assumption that L is not reassigned in τ is important. We use the Π'_i and Π' to create the deduction

$$\frac{\begin{array}{rcl}\Pi'_1 & : & \Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \tau_1[L/L\ p] : \varphi_1 \\ \Pi'_2 & : & \Gamma, p : \varphi_p, L : \varphi_p \to \varphi, M_1 : \varphi_1 \vdash \tau_2[L/L\ p] : \varphi_2 \\ \ldots & & \\ \Pi'_n & : & \Gamma, p : \varphi_p, L : \varphi_p \to \varphi, M_1 : \varphi_1, \ldots, M_{n-1} : \varphi_{n-1} \vdash \tau_n[L/L\ p] : \varphi_n \\ \Pi' & : & \Gamma, p : \varphi_p, L : \varphi_p \to \varphi, M_1 : \varphi_1, \ldots, M_n : \varphi_n \vdash \tau[L/L\ p] : \varphi\end{array}}{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash (\mathsf{let}\ M_1 = \tau_1 \ldots M_n = \tau_n\ \mathsf{in}\ \tau\ \mathsf{end})[L/L\ p] : \varphi}$$

□

Lemma A.3

$$\frac{\Gamma, L : \varphi \vdash \tau : \psi}{\Gamma, L : \forall x.\varphi \vdash \tau[L/L\ x] : \psi}$$

where $L = \pi$ does not appear in τ.

Proof. The proof is by induction on type derivations. It looks nearly identical to the proof of Lemma A.2. □

Lemma A.4

$$\frac{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \tau : \psi}{\Gamma, p : \varphi_p, L : \varphi \vdash \tau[L\ p/L] : \psi}$$

where $L = \pi$ does not appear in τ and any occurrence of L in τ is applied to p.

Proof. The proof is by induction on type derivations. It is very similar to the proofs of the previous two lemmas. The second condition is needed in order to

ensure that a proof is well typed. If we allowed L to be applied to an arbitrary proof π', then $\tau[L\ p/L]$ might not type, as the type of L changes. □

Lemma A.5

$$\frac{\Gamma, L : \forall x.\varphi \vdash \tau : \psi}{\Gamma, L : \varphi \vdash \tau[L\ x/L] : \psi}$$

where $L = \pi$ does not appear in τ and any occurrence of L in τ is applied to x.

Proof. The proof is by induction on typing derivations, similar to the previous three lemmas. □

Now we can complete the remaining two cases in the proof of soundness.

- Let Π be a deduction of the form

$$\Gamma \vdash \lambda p.\mathsf{let}\ \overline{L} = \overline{\pi}\ \mathsf{in}\ \tau\ \mathsf{end} : e \to \varphi \tag{A.15}$$

Using our **(generalize)** rule, we can get a derivation of the form

$$\mathsf{let}\ \overline{L} = \overline{\lambda p.\pi[\overline{L}/\overline{L\ p}]}\ \mathsf{in}\ \lambda p.\tau[\overline{L}/\overline{L\ p}]\ \mathsf{end} \tag{A.16}$$

From our typing rules, we know that the derivation Π must be of the form

$$\frac{\begin{array}{lcl}\Pi_1 & : & \Gamma, p : e \vdash \tau_1 : \varphi_1 \\ \Pi_2 & : & \Gamma, p : e, L_1 : \varphi_1 \vdash \tau_2 : \varphi_2 \\ \ldots & & \\ \Pi_n & : & \Gamma, p : e, L_1 : \varphi_1, \ldots, L_{n-1} : \varphi_{n-1} \vdash \tau_n : \varphi_n \\ \Pi_\tau & : & \Gamma, p : e, L_1 : \varphi_1, \ldots, L_n : \varphi_n \vdash \tau : \varphi\end{array}}{\dfrac{\Gamma, p : e \vdash \mathsf{let}\ \overline{L} = \overline{\pi}\ \mathsf{in}\ \tau\ \mathsf{end} : \varphi}{\Gamma \vdash \lambda p.\mathsf{let}\ \overline{L} = \overline{\pi}\ \mathsf{in}\ \tau\ \mathsf{end} : e \to \varphi}}$$

Using Lemma A.2 repeatedly on our deductions $\Pi_2 \ldots \Pi_n$ gives us deductions of the form

$$\Pi'_i : \Gamma, p : e, L_1 : e \to \varphi_1, \ldots, L_{i-1} : e \to \varphi_{i-1} \vdash \tau_i[\overline{L}/\overline{L\ p}] : \varphi_i$$

for $2 \leq i \leq n$. It is important to note that only $L_1, \ldots, L_{i-1}$ appear in the proof term τ_i. For τ_i, we apply Lemma A.2 $(i-1)$ times, once for each of the L_i that can appear in it.

Applying Lemma A.2 to Π_τ gives us the new derivation

$$\Pi'_\tau : \Gamma, p : e, L_1 : e \to \varphi_1, \ldots, L_n : e \to \varphi_n \vdash \tau[\overline{L}/\overline{L\ p}] : \varphi$$

From Π_1, we get a typing derivation

$$\Pi'_1 : \frac{\begin{array}{c}\Pi_1 \\ \Gamma, p : e \vdash \tau_1 : \varphi_1\end{array}}{\Gamma \vdash \lambda p.\tau_1 : e \to \varphi_1}$$

We use the Π'_i to get derivations of the form

$$\Pi''_i : \frac{\begin{array}{c}\Pi'_i \\ \Gamma, p : e, L_1 : e \to \varphi_1, \ldots, L_i : e \to \varphi_i \vdash \tau[L/L\ p] : \varphi\end{array}}{\Gamma, L_1 : e \to \varphi_1, \ldots, L_{i-1} : e \to \varphi_{i-1} \vdash \lambda p.\tau_i[\overline{L}/\overline{L\ p}] : e \to \varphi_i}$$

From Π'_τ, we get a derivation

$$\frac{\begin{array}{c}\Pi'_\tau \\ \Gamma, p : e, L_1 : e \to \varphi_1, \ldots, L_n : e \to \varphi_n \vdash \tau[\overline{L}/\overline{L\ p}] : \varphi\end{array}}{\Pi''_\tau : \Gamma, L_1 : e \to \varphi_1, \ldots, L_n : e \to \varphi_n \vdash \lambda p.\tau[\overline{L}/\overline{L\ p}] : e \to \varphi}$$

Finally, we combine Π'_1, the Π''_i, and Π''_τ to form a derivation

$$\frac{\begin{array}{lcl}\Pi'_1 & : & \Gamma \vdash \lambda p.\tau_1 : e \to \varphi_1 \\ \Pi''_2 & : & \Gamma, L_1 : e \to \varphi_1 \vdash \lambda p.\tau_2[\overline{L}/\overline{L\ p}] : e \to \varphi_2 \\ \ldots & & \\ \Pi''_n & : & \Gamma, L_1 : e \to \varphi_1, \ldots, L_{n-1} : e \to \varphi_{n-1} \vdash \lambda p.\tau_i[\overline{L}/\overline{L\ p}] : e \to \varphi_n \\ \Pi''_\tau & : & \Gamma, L_1 : e \to \varphi_1, \ldots, L_n : e \to \varphi_n \vdash \lambda p.\tau[\overline{L}/\overline{L\ p}] : e \to \varphi\end{array}}{\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\lambda p.\pi[\overline{L}/\overline{L\ p}]}\ \mathsf{in}\ \lambda p.\tau[\overline{L}/\overline{L}\ p]\ \mathsf{end} : e \to \varphi}$$

This is a derivation of the type for (A.16), which has the same type as derived in (A.15). This is what we needed to show.

- Let Π be a deduction of the form

$$\Gamma \vdash \lambda x.\mathsf{let}\ \overline{L} = \overline{\pi}\ \mathsf{in}\ \tau\ \mathsf{end} : \forall x.\varphi \tag{A.17}$$

Using our **(generalize)** rule, we can get a derivation of the form

$$\mathsf{let}\ \overline{L} = \overline{\lambda x.\pi[\overline{L}/\overline{L\ x}]}\ \mathsf{in}\ \lambda x.\tau[\overline{L}/\overline{L\ x}]\ \mathsf{end} \tag{A.18}$$

From our typing rules, we know that the derivation Π must have been of the form

$$\dfrac{\begin{array}{lcl}\Pi_1 & : & \Gamma \vdash \tau_1 : \varphi_1 \\ \Pi_2 & : & \Gamma, L_1 : \varphi_1 \vdash \tau_2 : \varphi_2 \\ \ldots & & \\ \Pi_n & : & \Gamma, L_1 : \varphi_1, \ldots, L_{n-1} : \varphi_{n-1} \vdash \tau_n : \varphi_n \\ \Pi_\tau & : & \Gamma, L_1 : \varphi_1, \ldots, L_n : \varphi_n \vdash \tau : \varphi\end{array}}{\dfrac{\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\pi}\ \mathsf{in}\ \tau\ \mathsf{end} : \varphi}{\Gamma \vdash \lambda x.\mathsf{let}\ \overline{L} = \overline{\pi}\ \mathsf{in}\ \tau\ \mathsf{end} : \forall x.\varphi}}$$

Using Lemma A.3 repeatedly on our deductions $\Pi_2 \ldots \Pi_n$ gives us deductions of the form

$$\Pi'_i : \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_{i-1} : \forall x.\varphi_{i-1} \vdash \tau_i[\overline{L}/\overline{L\ p}] : \varphi_i$$

for $2 \leq i \leq n$. It is important to note that only $L_1, \ldots, L_{i-1}$ appear in the proof term τ_i. For τ_i, we apply Lemma A.3 $(i-1)$ times, once for each of the L_i that can appear in it.

Applying Lemma A.3 to Π_τ gives us the new derivation

$$\Pi'_\tau : \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_n : \forall x.\varphi_n \vdash \tau[\overline{L}/\overline{L\ p}] : \varphi$$

From Π_1, we get a typing derivation

$$\Pi'_1 : \dfrac{\begin{array}{c}\Pi_1 \\ \Gamma \vdash \tau_1 : \varphi_1\end{array}}{\Gamma \vdash \lambda x.\tau_1 : \forall x.\varphi_1}$$

We use the Π'_i to get derivations of the form

$$\Pi''_i : \dfrac{\begin{array}{c}\Pi'_i \\ \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_i : \forall x.\varphi_i \vdash \tau[L/L\ x] : \varphi\end{array}}{\Gamma, L_1 : \forall x.\varphi_1, \ldots, L_{i-1} : \forall x.\varphi_{i-1} \vdash \lambda x.\tau_i[\overline{L}/\overline{L\ x}] : \forall x.\varphi_i}$$

From Π'_τ, we get a derivation

$$\Pi''_\tau : \dfrac{\begin{array}{c}\Pi'_\tau \\ \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_n : \forall x.\varphi_n \vdash \tau[\overline{L}/\overline{L\ x}] : \varphi\end{array}}{\Gamma, L_1 : \forall x.\varphi_1, \ldots, L_n : \forall x.\varphi_n \vdash \lambda x.\tau[\overline{L}/\overline{L\ x}] : \forall x.\varphi}$$

Finally, we combine Π'_1, the Π''_i, and Π''_τ to form a derivation

$$\dfrac{\begin{array}{lcl}\Pi'_1 & : & \Gamma \vdash \lambda p.\tau_1 : \forall x.\varphi_1 \\ \Pi''_2 & : & \Gamma, L_1 : \forall x.\varphi_1 \vdash \lambda x.\tau_2[\overline{L}/\overline{L\ p}] : \forall x.\varphi_2 \\ \ldots & & \\ \Pi''_n & : & \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_{n-1} : \forall x.\varphi_{n-1} \vdash \lambda x.\tau_i[\overline{L}/\overline{L\ x}] : \forall x.\varphi_n \\ \Pi''_\tau & : & \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_n : \forall x.\varphi_n \vdash \lambda x.\tau[\overline{L}/\overline{L\ x}] : \forall x.\varphi\end{array}}{\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\lambda x.\pi[\overline{L}/\overline{L\ x}]}\ \mathsf{in}\ \lambda x.\tau[\overline{L}/\overline{L}\ x]\ \mathsf{end} : \forall x.\varphi}$$

This is a derivation of the type for (A.18), which has the same type as derived in (A.17). This is what we needed to show.

- Let Π be a deduction of the form

$$\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\lambda p.\pi}\ \mathsf{in}\ \lambda p.\tau\ \mathsf{end} : e \to \varphi \tag{A.19}$$

where L is applied to p in all occurrences in τ and the π_i. Using our **(specialize)** rule, we can get a derivation of the form

$$\lambda p.\mathsf{let}\ \overline{L} = \overline{\pi[\overline{L\ p}/\overline{L}]}\ \mathsf{in}\ \tau[\overline{L\ p}/\overline{L}]\ \mathsf{end} \tag{A.20}$$

We know the derivation of (A.19) must be of the form

$$\frac{\begin{array}{lcl}\Pi_1 & : & \Gamma \vdash \lambda p.\tau_1 : e \to \varphi_1 \\ \Pi_2 & : & \Gamma, L_1 : e \to \varphi_1 \vdash \lambda p.\tau_2 : e \to \varphi_2 \\ \ldots & & \\ \Pi_n & : & \Gamma, L_1 : e \to \varphi_1, \ldots, L_{n-1} : e \to \varphi_{n-1} \vdash \lambda p.\tau_n : e \to \varphi_n \\ \Pi_\tau & : & \Gamma, L_1 : e \to \varphi_1, \ldots, L_n : e \to \varphi_n \vdash \lambda p.\tau : e \to \varphi \end{array}}{\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\lambda p.\pi}\ \mathsf{in}\ \lambda p.\tau\ \mathsf{end} : e \to \varphi}$$

The derivations Π_i are of the form

$$\frac{\begin{array}{c}\Pi_i' \\ \Gamma, L_1 : e \to \varphi_1, \ldots, L_{i-1} : e \to \varphi_{i-1}, p : e \vdash \tau_i : \varphi_i\end{array}}{\Gamma, L_1 : e \to \varphi_1, \ldots, L_{i-1} : e \to \varphi_{i-1} \vdash \lambda p.\tau_i : e \to \varphi_i}$$

Using Lemma A.4 repeatedly on our deductions $\Pi_2' \ldots \Pi_n'$ and gives us deductions of the form

$$\Pi_i'' : \Gamma, L_1 : e \to \varphi_1, \ldots, L_{i-1} : e \to \varphi_{i-1}, p : e \vdash \tau_i[\overline{L\ p}/\overline{L}] : \varphi_i$$

for $2 \leq i \leq n$. It is important to note that only $L_1, \ldots, L_{i-1}$ appear in the proof term τ_i. For τ_i, we apply Lemma A.2 $(i-1)$ times, once for each of the L_i that can appear in it.

The derivation Π_τ must be of the form

$$\frac{\begin{array}{c}\Pi_\tau' \\ \Gamma, L_1 : e \to \varphi_1, \ldots, L_n : e \to \varphi_n, p : e \vdash \tau : \varphi\end{array}}{\Gamma, L_1 : e \to \varphi_1, \ldots, L_n : e \to \varphi_n \vdash \lambda p.\tau : e \to \varphi}$$

We use Π'_1, the Π''_i, and Π'_τ to get a derivation

$$\frac{\begin{array}{rcl}\Pi'_1 & : & \Gamma, p : e \vdash \tau_1[\overline{L\ p}/\overline{L}] : \varphi_1 \\ \Pi''_2 & : & \Gamma, p : e, L_1 : \varphi_1 \vdash \tau_2[\overline{L\ p}/\overline{L}] : \varphi_2 \\ \ldots & & \\ \Pi''_n & : & \Gamma, p : e, L_1 : \varphi_1, \ldots, L_{n-1} : \varphi_{n-1} \vdash \tau_n[\overline{L\ p}/\overline{L}] : \varphi_n \\ \Pi'_\tau & : & \Gamma, p : e, L_1 : \varphi_1, \ldots, L_n : \varphi_n \vdash \tau[\overline{L\ p}/\overline{L}] : \varphi \end{array}}{\dfrac{\Gamma, p : e \vdash \mathsf{let}\ \overline{L} = \overline{\pi[\overline{L\ p}/\overline{L}]}\ \mathsf{in}\ \tau[\overline{L\ p}/\overline{L}]\ \mathsf{end} : \varphi}{\Gamma \vdash \lambda p.\mathsf{let}\ \overline{L} = \overline{\pi[\overline{L\ p}/\overline{L}]}\ \mathsf{in}\ \tau[\overline{L\ p}/\overline{L}]\ \mathsf{end} : e \to \varphi}}$$

This is a derivation of the type for (A.20), which has the same type as derived in (A.19). This is what we needed to show.

- Let Π be a deduction of the form

$$\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\lambda x.\pi}\ \mathsf{in}\ \lambda x.\tau\ \mathsf{end} : \forall x.\varphi \tag{A.21}$$

where L only occurs in τ and the π_i applied to p. Using our **(specialize)** rule, we can get a derivation of the form

$$\lambda x.\mathsf{let}\ \overline{L} = \overline{\pi[\overline{L\ x}/\overline{L}]}\ \mathsf{in}\ \tau[\overline{L\ x}/\overline{L}]\ \mathsf{end} \tag{A.22}$$

We know the derivation of (A.21) must be of the form

$$\frac{\begin{array}{rcl}\Pi_1 & : & \Gamma \vdash \lambda x.\tau_1 : \forall x.\varphi_1 \\ \Pi_2 & : & \Gamma, L_1 : \forall x.\varphi_1 \vdash \lambda x.\tau_2 : \forall x.\varphi_2 \\ \ldots & & \\ \Pi_n & : & \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_{n-1} : \forall x.\varphi_{n-1} \vdash \lambda x.\tau_n : \forall x.\varphi_n \\ \Pi_\tau & : & \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_n : \forall x.\varphi_n \vdash \lambda x.\tau : \forall x.\varphi \end{array}}{\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\lambda x.\pi}\ \mathsf{in}\ \lambda x.\tau\ \mathsf{end} : \forall x.\varphi}$$

The derivations Π_i are of the form

$$\frac{\begin{array}{c}\Pi_i' \\ \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_{i-1} : \forall x.\varphi_{i-1} \vdash \tau_i : \varphi_i\end{array}}{\Gamma, L_1 : \forall x.\varphi_1, \ldots, L_{i-1} : \forall x.\varphi_{i-1} \vdash \lambda x.\tau_i : \forall x.\varphi_i}$$

Using Lemma A.5 repeatedly on our deductions $\Pi_2' \ldots \Pi_n'$ gives us deductions of the form

$$\Pi_i'' : \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_{i-1} : \forall x.\varphi_{i-1} \vdash \tau_i[\overline{L\ x}/\overline{L}] : \varphi_i$$

for $2 \leq i \leq n$. It is important to note that only $L_1, \ldots, L_{i-1}$ appear in the proof term τ_i. For τ_i, we apply Lemma A.2 $(i-1)$ times, once for each of the L_i that can appear in it.

The derivation Π_τ must be of the form

$$\frac{\begin{array}{c}\Pi_\tau' \\ \Gamma, L_1 : \forall x.\varphi_1, \ldots, L_n : \forall x.\varphi_n \vdash \tau : \varphi\end{array}}{\Gamma, L_1 : \forall x.\varphi_1, \ldots, L_n : \forall x.\varphi_n \vdash \lambda x.\tau : \forall x.\varphi}$$

We use Π_1', the Π_i'', and Π_τ' to get a derivation

$$\frac{\begin{array}{rcl}\Pi_1' & : & \Gamma \vdash \tau_1[\overline{L\ x}/\overline{L}] : \varphi_1 \\ \Pi_2'' & : & \Gamma, L_1 : \varphi_1 \vdash \tau_2[\overline{L\ x}/\overline{L}] : \varphi_2 \\ \ldots & & \\ \Pi_n'' & : & \Gamma, L_1 : \varphi_1, \ldots, L_{n-1} : \varphi_{n-1} \vdash \tau_n[\overline{L\ x}/\overline{L}] : \varphi_n \\ \Pi_\tau' & : & \Gamma, L_1 : \varphi_1, \ldots, L_n : \varphi_n \vdash \tau[\overline{L\ x}/\overline{L}] : \varphi\end{array}}{\dfrac{\Gamma \vdash \mathsf{let}\ \overline{L} = \overline{\pi[\overline{L\ x}/\overline{L}]}\ \mathsf{in}\ \tau[\overline{L\ x}/\overline{L}]\ \mathsf{end} : \varphi}{\Gamma \vdash \lambda x.\mathsf{let}\ \overline{L} = \overline{\pi[\overline{L\ x}/\overline{L}]}\ \mathsf{in}\ \tau[\overline{L\ x}/\overline{L}]\ \mathsf{end} : \forall x.\varphi}}$$

This is a derivation of the type for (A.22), which has the same type as derived in (A.21). This is what we needed to show.

- Let Π be a derivation of the form

$$\Gamma \vdash \lambda\alpha.\pi : \varphi \tag{A.23}$$

We can use our **(rename)** rule to get a proof term of the form

$$\lambda\beta.\pi[\alpha/\beta] \tag{A.24}$$

We can perform substitution on φ to get the type $\varphi[\alpha/\beta]$, which corresponds to the type of (A.24). The two types are equivalent, as we are simply performing α-renaming.

□

REFERENCES

[1] KAT-ML, 2003. `http://www.cs.cornell.edu/Projects/kat/`.

[2] Alf-Christian Achilles and Paul Ortyl. Distribution of publication dates, 2006.

[3] Teri Anderson, Theresa Drust, Dean Johnson, and Shelly R. Lesher. The growth of physics and mathematics. *Lost In Thought*, 2, 1999.

[4] Allegra Angus and Dexter Kozen. Kleene algebra with tests and program schematology. Technical Report 2001-1844, Computer Science Department, Cornell University, July 2001.

[5] Apple Computer, Inc. Sharing your library with your other computers. `http://www.apple.com/ilife/tutorials/itunes/it6-1.html`.

[6] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, Ferruccio Guidi, and Irene Schena. Mathematical knowledge management in HELM. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):27–46, 2003.

[7] David Aspinall, Thomas Kleymann, P. Courtieu, H. Goguen, D. Sequeira, and M. Wenz. Proof General. `http://proofgeneral.inf.ed.ac.uk`, April 2004.

[8] David Aspinall, Christoph Lüth, and Burkhart Wolff. Assisted proof document authoring. In Michael Kohlhase, editor, *MKM*, volume 3863 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2005.

[9] Jacek Chrząszcz. Implementing modules in the Coq system. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003.

[10] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stphane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical markup language (MathML) version 2.0 (second edition), 2003.

[11] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. 8th Int. SPIN Workshop on Model Checking of Software (SPIN 2001)*, volume 2057 of *Lect. Notes in Comput. Sci.*, pages 103–122. Springer-Verlag, May 2001.

[12] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.

[13] Clemens Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In Jonathan M. Borwein and William M. Farmer, editors, *Proc. 5th Conf. Mathematical Knowledge Management*, volume 4108 of *Lecture Notes in Artificial Intelligence*, pages 31–43. Springer, 2006.

[14] Naser S. Barghouti, John Mocenigo, and Wenke Lee. *Grappa:* a GRAPh PAckage in Java. In Giuseppe Di Battista, editor, *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 336–343. Springer, 1997.

[15] Adam Barth and Dexter Kozen. Equational verification of cache blocking in LU decomposition using Kleene algebra with tests. Technical Report 2002-1865, Computer Science Department, Cornell University, June 2002.

[16] Gertrud Bauer and Markus Wenzel. Calculational reasoning revisited—An Isabelle/Isar experience. In Richard J. Boulton and Paul B. Jackson, editors, *Proc. 14th Int. Conf. Theorem Proving in Higher Order Logics (TPHOLs'01)*, volume 2152 of *Lect. Notes in Comput. Sci.* Springer, 2001.

[17] Bernhard Beckert and Vladimir Klebanov. Proof reuse for deductive program verification. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference on (SEFM'04)*, pages 77–86, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, volume 789, pages 141–160, London, UK, 1994. Springer-Verlag.

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

[20] B. Buchberger. Mathematical Knowledge Management using THEOREMA. In B.Buchberger and O. Caprotti, editors, *First International Workshop on Mathematical Knowledge Management (MKM 2001)*, pages —-, RISC-Linz, A-4232 Schloss Hagenberg, September 24-26, 2001, 17 pages., 2001.

[21] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, pages –, 2005. To appear.

[22] Alan Bundy, Frank van Harmelen, Jane Hesketh, and Alan Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, 1991.

[23] Stephen Buswell, Olga Caprotti, David P. Carlisle, Michael C. Dewar, Marc Gaetano, and Michael Kohlhase. The open math standard, version 2.0. Technical report, The Open Math Society, 2004.

[24] Paul Cairns. Alcor: A user interface for Mizar. In *Mathematical User-Interfaces Workshop 2004*, 2004.

[25] J. G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach: Volume II*, pages 371–392. Kaufmann, Los Altos, CA, 1986.

[26] Ernie Cohen. Hypotheses in Kleene algebra. Technical Report TM-ARH-023814, Bellcore, 1993.

[27] Ernie Cohen. Lazy caching in Kleene algebra, 1994. `http://citeseer.nj.nec.com/22581.html`.

[28] Ernie Cohen. Using Kleene algebra to reason about concurrency control. Technical report, Telcordia, Morristown, N.J., 1994.

[29] Ernie Cohen, Dexter Kozen, and Frederick Smith. The complexity of Kleene algebra with tests. Technical Report 96-1598, Computer Science Department, Cornell University, July 1996.

[30] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[31] L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN: the constructive Coq repository at Nijmegen. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management, Third International Conference, MKM 2004*, volume 3119 of *LNCS*, pages 88–103. Springer–Verlag, 2004.

[32] David Delahaye. A tactic language for the system Coq. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[33] J. Desharnais, B. Moller, and G. Struth. Kleene algebra with domain. Technical Report 2003-07, Universität Augsburg, Institut für Informatik, June 2003.

[34] Francisco Durán and José Meseguer. An extensible module algebra for Maude. *Electr. Notes Theor. Comput. Sci.*, 15, 1998.

[35] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, 1993.

[36] Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, volume 822 of *LNAI*, pages 1–15, Kiev, Ukraine, 1994. Springer-Verlag.

[37] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[38] The Eclipse Foundation. Refactoring support. `http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jdt.doc.user/concepts/concepts-9.htm`, 2006.

[39] Herman Geuvers and Iris Loeb. Natural deduction via graphs: Formal definition and computation rules. *Mathematical Structures in Computer Science*, 2006.

[40] Herman Geuvers and Lionel Elie Mamane. A document-oriented Coq plugin for TEXMACS. In *Proceedings of Mathematical User-Interfaces Workshop 2006*, 2006.

[41] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[42] Fausto Giunchiglia and Paolo Traverso. A metatheory of a mechanized object theory. *Artif. Intell.*, 80(2):197–241, 1996.

[43] Fausto Giunchiglia and Paolo Traverso. Program tactics and logic tactics. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):235–259, 1996.

[44] Fausto Giunchiglia, Adolfo Villafiorita, and Toby Walsh. Theories of abstraction. *AI Commun.*, 10(3-4):167–176, 1997.

[45] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artif. Intell.*, 57(2-3):323–389, 1992.

[46] M. Gordon, R. Milner, and C. P. Wadswort. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[47] Chris Hardin and Dexter Kozen. On the complexity of the Horn theory of REL. Technical Report 2003-1896, Computer Science Department, Cornell University, May 2003.

[48] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL—A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proc. 16th Int. Conf. Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *LNCS*, pages 287–303. Springer-Verlag, 2003.

[49] Mateja Jamnik. Analogy and automated reasoning. Technical Report CSRP-99-14, University of Birmingham, June 1999.

[50] Peter Jipsen. PCP: Point and click proofs, 2001. `http://www1.chapman.edu/~jipsen/PCP/PCPhome.html`.

[51] Wolfram Kahl. Calculational relation-algebraic proofs in Isabelle/Isar. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *Proc. Int. Conf. Relational Methods in Computer Science (RelMiCS'03)*, volume 3051 of *Lecture Notes in Computer Science*, pages 178–190. Springer, 2003.

[52] Florian Kammüller. Modular reasoning in Isabelle. In David A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2000.

[53] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - a sectioning concept for Isabelle. In *TPHOLs '99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 149–166, London, UK, 1999. Springer-Verlag.

[54] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, N.J., 1956.

[55] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[56] Rob Kling. A paradigm for reasoning by analogy. In *IJCAI*, pages 568–585, 1971.

[57] Michael Kohlhase. *OMDoc – An Open Markup Format for Mathematical Documents*, volume 4180 of *Lecture Notes in Artificial Intelligence*. Springer Berlin/Heidelberg, 2006.

[58] T. Kolbe and C. Walther. Proof management and retrieval. In *IJCAI-14 Workshop on Formal Approaches to the Reuse of Plans, Proofs and Programs*, 1995.

[59] Thomas Kolbe and Christoph Walther. Reusing proofs. In *European Conference on Artificial Intelligence*, pages 80–84, 1994.

[60] Thomas Kolbe and Christoph Walther. Patching proofs for reuse (extended abstract). In *ECML '95: Proceedings of the 8th European Conference on Machine Learning*, pages 303–306, London, UK, 1995. Springer-Verlag.

[61] Thomas Kolbe and Christoph Walther. Second-order matching modulo evaluation: A technique for reusing proofs. In *IJCAI*, pages 190–195, 1995.

[62] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[63] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[64] Dexter Kozen. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic*, 1(1):60–76, July 2000.

[65] Dexter Kozen. On the complexity of reasoning in Kleene algebra. *Information and Computation*, 179:152–162, 2002.

[66] Dexter Kozen. Automata on guarded strings and applications. *Matématica Contemporânea*, 24:117–139, 2003.

[67] Dexter Kozen and Maria-Cristina Patron. Certification of compiler optimizations using Kleene algebra with tests. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proc. 1st Int. Conf. Computational Logic (CL2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 568–582, London, July 2000. Springer-Verlag.

[68] Dexter Kozen and Ganesh Ramanarayanan. Publication/citation: A proof-theoretic approach to mathematical knowledge management. Technical Report 2005-1985, Computer Science Department, Cornell University, March 2005.

[69] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.

[70] Dexter Kozen and Jerzy Tiuryn. Substructural logic and partial correctness. *Trans. Computational Logic*, 4(3):355–378, July 2003.

[71] Christoph Kreitz. *The Nuprl Proof Development System, Version 5: Reference Manual and User's Guide*. Department of Computer Science, Cornell University, December 2002.

[72] Lori Lorigo, Jon M. Kleinberg, Richard Eaton, and Robert L. Constable. A graph-based approach towards discerning inherent structures in a digital library of formal mathematics. In *MKM*, pages 220–235, 2004.

[73] Daniel W. Lozier. NIST digital library of mathematical functions. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):105–119, 2003.

[74] David MacQueen. Modules for standard ml. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM Press.

[75] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[76] A. P. Martin, P. H. B. Gardiner, and J.C.P Woodcock. A tactic calculus – abridged version. *Formal Aspects of Computing*, 8(4):479–489, 1996.

[77] Andrew Martin and Jeremy Gibbons. A monadic interpretation of tactics.

[78] Nikolay Mateev, Vijay Menon, and Keshav Pingali. Fractal symbolic analysis. In *Proc. 15th Int. Conf. Supercomputing (ICS'01)*, pages 38–49, New York, 2001. ACM Press.

[79] Erica Melis. A model of analogy-driven proof-plan construction. In *IJCAI*, pages 182–189, 1995.

[80] Erica Melis and Axel Schairer. Similarities and reuse of proofs in formal software verification. In *EWCBR*, pages 76–87, 1998.

[81] Erica Melis and Jon Whittle. Internal analogy in theorem proving. In *Conference on Automated Deduction*, pages 92–105, 1996.

[82] Erica Melis and Jon Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22(2):117–147, 1999.

[83] Bruce R. Miller and Abdou Youssef. Technical aspects of the digital library of mathematical functions. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):121–136, 2003.

[84] Robin Milner. Axioms for bigraphical structure. *Mathematical Structures in Comp. Sci.*, 15(6):1005–1032, 2005.

[85] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[86] James H. Morris. Lambda calculus models of programming languages. Technical report, Massachuseets Instititue of Technology, Laboratory for Computer Science, 1968.

[87] James Curie Munyer. *Analogy as a means of discovery in problem solving and learning*. PhD thesis, University of California, Santa Cruz, 1981.

[88] Tobias Nipkow. Structured proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer, 2003.

[89] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[90] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, March 2002.

[91] F. Piroi and B. Buchberger. An Environment for Building Mathematical Knowledge Libraries. In Wolfgang Windsteiger and Christoph Benzmueller, editors, *Proceedings of the Workshop on Computer-Supported Mathematical Theory Development, Second International Joint Conference (IJCAR)*, pages 19–29, Cork, Ireland, 4-8 July 2004.

[92] Florina Piroi. User interface features in *Theorema*: A summary. In *Mathematical User-Interfaces Workshop 2004*, 2004.

[93] David A. Plaisted. Abstraction mappings in mechanical theorem proving. In Wolfgang Bibel and Robert A. Kowalski, editors, *CADE*, volume 87 of *Lecture Notes in Computer Science*, pages 264–280. Springer, 1980.

[94] David A. Plaisted. Theorem proving with abstraction. *Artif. Intell.*, 16(1):47–108, 1981.

[95] David A. Plaisted. Abstraction using generalization functions. In Jörg H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 365–376. Springer, 1986.

[96] Olivier Pons. Proof generalization and proof reuse, 2000.

[97] Riccardo Pucella. On partially additive Kleene algebras. In *Proc. 8th Int. Conf. Relational Methods in Computer Science (RelMiCS 8)*, February 2005.

[98] Piotr Rudnicki and Andrez Trybulec. Mathematical Knowledge Management in MIZAR. In B. Buchberger and O. Caprotti, editors, *Proc. of First International Workshop on Mathematical Knowledge Management (MKM 2001)*, Linz, Austria, September 2001.

[99] Axel Schairer, Serge Autexier, and Dieter Hutter. A pragmatic approach to reuse in tactical theorem proving. *Electronic Notes in Theoretical Computer Science*, 58(2), 2001.

[100] Morten Heine Sørensen and Pawel Urzyczyn. Lectures on the Curry–Howard isomorphism. Available as DIKU Rapport 98/14, 1998.

[101] Access Science Math Squad. A schematic view of the various branches of mathematics. `http://s89940423.onlinehome.us/index.php?title=Image:123networkv2.png`, September 2006.

[102] Georg Struth. Isabelle specification and proofs of Church-Rosser theorems, 2001. `http://www.informatik.uni-augsburg.de/~struth/papers/isabelle`.

[103] Georg Struth. Calculating Church-Rosser proofs in Kleene algebra. In *Proc. 6th Int. Conf. Relational Methods in Computer Science (ReIMICS'01)*, pages 276–290, London, 2002. Springer-Verlag.

[104] Don Syme. Three tactic theorem proving. In *TPHOLs '99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 203–220, London, UK, 1999. Springer-Verlag.

[105] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.3*, May 2002. `http://coq.inria.fr`.

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

[107] Joakim von Wright. From Kleene algebra to refinement algebra. In Eerke A. Boiten and Bernhard Möller, editors, *Proc. Conf. Mathematics of Program Construction (MPC'02)*, volume 2386 of *Lect. Notes in Comput. Sci.*, pages 233–262. Springer, July 2002.

[108] Markus Wenzel and Stefan Berghofer. *The Isabelle System Manual*, May 2003.

[109] Markus M. Wenzel. *Isabelle/Isar: A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, TU München, 2002.

[110] J. Whittle. Analogy in CL^AM. Master's thesis, Edinburgh, 1995.

[111] Phillip J. Windley. Abstract theories in HOL. In *HOL'92: Proceedings of the IFIP TC10/WG10.2 Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 197–210. North-Holland/Elsevier, 1993.