December 1986

Using Extended Tactics to Do Proof Transformations

Amy P. Felty
University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
http://repository.upenn.edu/cis_reports/666


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/666
For more information, please contact repository@pobox.upenn.edu.
Using Extended Tactics to Do Proof Transformations

Abstract
In this thesis we develop a comprehensive human-oriented theorem proving system that integrates several different proof systems. The main theorem proving environment centers around a natural Gentzen first-order logic system. This allows construction of natural proofs, encourages user involvement in the search for proofs, and facilitates understanding of the resulting proofs. We integrate more abstract automatically generated proofs such as resolution refutations by transforming them to proofs in the Gentzen system. Expansion trees are another proof system used as an intermediate stage in transformations between the abstract and natural systems. They are a compact representation useful for transformations and other computations. We develop a programming language approach to theorem proving based on tactics and tacticals. Our extended tactics provide a method for doing proof transformations, as well as facilitate interactive theorem proving, allowing full integration of interactive and automatic theorem proving. In the system, we explicitly represent proofs in each proof system and view expansion tree proofs as types for Gentzen proof terms. This explicit proof representation allows proofs to be manipulated as meaningful data objects and used in various computations. For example, the proof terms in the natural Gentzen system can be used to obtain natural language explanations of proofs. We foresee several applications for this kind of theorem proving system, such as use as a logic tutor, a tool for doing mathematics, or an enhanced reasoner and explanation facility for existing AI systems.

Comments

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/666
Using Extended Tactics To Do Proof Transformations

MS-CIS-86-89

Amy P. Felty

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6339

1986
Using Extended Tactics To Do Proof Transformations

MS-CIS-86-89
LINC LAB 46

Amy P. Felty

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

December 1986

Acknowledgements:
This research was supported in part by DARPA grants N00014-85-K-0018, NSF-CER grant MCS-8219196 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027
USING EXTENDED TACTICS TO DO PROOF TRANSFORMATIONS

Amy P. Felty

Philadelphia, Pennsylvania
December, 1986

A thesis presented to the Faculty of Engineering and Applied Science of the University of Pennsylvania in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Computer and Information Science.

Dr. Dale Miller

Dr. O. Peter Buneman
Abstract

In this thesis we develop a comprehensive human-oriented theorem proving system that integrates several different proof systems. The main theorem proving environment centers around a natural Gentzen first-order logic system. This allows construction of natural proofs, encourages user involvement in the search for proofs, and facilitates understanding of the resulting proofs. We integrate more abstract automatically generated proofs such as resolution refutations by transforming them to proofs in the Gentzen system. Expansion trees are another proof system used as an intermediate stage in transformations between the abstract and natural systems. They are a compact representation useful for transformations and other computations. We develop a programming language approach to theorem proving based on tactics and tacticals. Our extended tactics provide a method for doing proof transformations, as well as facilitate interactive theorem proving, allowing full integration of interactive and automatic theorem proving. In the system, we explicitly represent proofs in each proof system and view expansion tree proofs as types for Gentzen proof terms. This explicit proof representation allows proofs to be manipulated as meaningful data objects and used in various computations. For example, the proof terms in the natural Gentzen system can be used to obtain natural language explanations of proofs. We foresee several applications for this kind of theorem proving system, such as use as a logic tutor, a tool for doing mathematics, or an enhanced reasoner and explanation facility for existing AI systems.
I am extremely grateful to my advisor Dale Miller for his guidance, support, and valuable instruction throughout this entire project. I owe much of my knowledge of logic and theorem proving to him.

I would also like to thank Greg Hager for his valuable comments on drafts of this thesis, for the insights I have gained in discussions with him, and for his extensive help in LaTeX hacking.

---

1This work has been supported by NSF grant MCS82-19196-CER and DARPA N00014-85-K-0018.
Contents

1 Introduction ........................................ 1
   1.1 Overview of the System Design .................... 2
   1.2 Overview of the Thesis ............................ 4

2 The Logic ........................................... 6
   2.1 The Logic Systems ................................ 6
      2.1.1 The LK+ Natural Deduction System .............. 6
      2.1.2 Resolution .................................. 12
      2.1.3 Expansion Tree Proofs and Matings ............ 13
   2.2 Proof Transformations ............................ 18
      2.2.1 Transforming Resolution Refutations to ET-Proofs .... 18
      2.2.2 Transforming ET-Proofs to LK+ Proofs .......... 18
      2.2.3 Transforming LK+ Proofs to ET-Proofs .......... 34

3 The \( \chi \) System ................................ 38
   3.1 The Programming Language Approach ................ 38
      3.1.1 The Data Structures .......................... 39
      3.1.2 The Primitive Tactics ......................... 41
      3.1.3 The Tacticals ............................... 54
      3.1.4 The Typing Mechanism ......................... 56
      3.1.5 Extending LCF .............................. 61
   3.2 The Interactive Proof Editor ....................... 63
   3.3 Integrating an Automatic Theorem Prover ........... 66
   3.4 Revising Proofs ................................ 70

4 Applications ...................................... 73
   4.1 Applications of Proofs as Objects ................. 73
   4.2 Applications of the System as a Whole ............ 75

A Soundness and Completeness of LK+ .................. 78
1 Introduction

Theorem proving has become an increasingly important subdiscipline of artificial intelligence arising out of a growing need for formal reasoning. One area in which theorem provers are widely used is in AI systems that need to make decisions or draw conclusions from a given database of information. The knowledge capacity of a system is greatly increased by the ability to deduce facts that are not explicitly represented from those that are. Using a formal logic proof system to deduce these new facts assures their validity and gives confidence in the accuracy of the system. Theorem provers are also interesting in themselves as tools for doing mathematics. Such a tool can be used by the novice as a tutor, or by the expert as an assistant, in each case, providing aid to the user in accomplishing a given task.

Theorem provers should be human-oriented since it is humans that must ultimately interact with them. At the very least, for automatic theorem provers, the user must input the statement of a theorem, and once proven, must be able to understand its output. At best, the user should be allowed to interact at any time during the theorem proving process to contribute any ideas s/he might have about how the proof should proceed. Understanding the output requires that the resulting proof be presented in a form that is natural and intelligible to the user. Participating in the proof process requires that the interactive environment be based on a natural formal proof system.

The goal of building human-oriented theorem provers directly competes with the goal of automating the theorem proving process. The main reason for this is that proof systems that are suitable for automation are not necessarily human-oriented and vice versa. Resolution is currently the paradigm in which most automatic theorem proving is done and much success has been achieved in this area. Resolution is suitable for automatic theorem proving because the search space, though very large, is very homogeneous, and the operations involved in building a proof—in this case, a resolution refutation—are very straightforward. Yet, in order to obtain this homogeneous search space the original theorem must be put through a severe normalization process. This normal form, and thus the search space, is very remote from the user's original input, making it difficult for the user to contribute to the construction of a proof. In addition, the end result is an abstract structure which gives very little insight into why a theorem is true. As a result, in general, a resolution prover simply gives a yes or no answer indicating whether or not it was successful.

Natural deduction systems, on the other hand, are just the opposite. They facilitate both interaction and understanding. In general, they contain inference rules that operate directly on subformulas of the main theorem, and thus each step of the proof is "natural" and easily understood—a property which can be used to
facilitate interaction. The end result of the natural deduction process is a proof which is readable and can be presented to the user to give some insight into the overall contents of the proof—facilitating understanding. Thus, natural deduction systems have several human-oriented features. On the other hand, they are difficult to automate because the choice that must be made at each step of the proof is quite complex. Automatically constructing good (i.e. readable and natural) proofs is even more difficult.

Most theorem provers opt for one or the other of these competing goals, and hence are either human-oriented and interactive or machine-oriented and automatic. We shall show how to take advantage of certain characteristics of each of these kinds of proof systems with the ultimate goal being the construction of natural proofs. The foundation of the system we develop to achieve this goal is an interactive environment within a natural deduction setting. The user has complete control over the construction of a proof, and has access to partial automation within the natural deduction system, or full automation through the use of a resolution style theorem prover. To integrate resolution and natural deduction proofs we provide the capability to translate between them. Thus, when a user requests an automatically generated proof from the resolution prover, the result is transformed to a natural deduction proof and integrated into the environment in which the user is working. Integrating interactive and automatic theorem proving in this way gives the user full access to an automatic theorem prover, as well as an interactive environment which provides all the human-oriented advantages of a natural deduction system.

Another theme that has emerged in recent theorem proving literature is the desirability to store proofs as first-class values, give them types, and have the ability to manipulate them in many ways. In order to capture our goals, we have taken this approach and extended it in various ways. We have developed an explicit representation of proofs in each proof system—one that will facilitate various manipulations. Many human-oriented manipulations will require natural proofs. This emphasizes the importance of our goal to provide means for constructing such natural proofs.

1.1 Overview of the System Design

Figure 1 shows the design of the $\chi$ system which is currently being developed. The Greek letter $\chi$ is spelled "chi" and is an acronym for the "Curry-Howard Isomorphism" in our context. The Curry-Howard Isomorphism (i.e. formulas-as-types) which appears in recent theorem proving literature provides sophistication and clarity to constructive logic proof systems. We adopt this name because it is symbolic of our goal to extend these ideas to more traditional theorem proving systems.

In this thesis, we discuss two of the three components of the $\chi$ system—the proof construction component and the proof revision component, with emphasis
1.1 Overview of the System Design

The theorem proving process begins in the proof construction component with a statement of a theorem input to the interactive environment. The user guides the construction of the proof within the natural deduction setting by applying inference rules. We have developed a programming language in which the user can write proof procedures or heuristics to apply some combination of several inference rules, allowing partial automation of proof construction. The user has the option of entering the name of such a procedure to the interactive editor at any point in the theorem proving process. Also, at any time, the user can call the automatic theorem prover which will produce a proof in an abstract system such as resolution, and then translate the proof or subproof into a natural deduction proof, and incorporate it into the larger proof that the user is working on.

When a proof is completed, it is passed to the proof revision component. Proof revision involves translating the natural deduction proof to an abstract proof representation, performing some logical analysis on this structure, and then translating it back to natural deduction. The basic idea is that the abstract structure removes some of the unimportant details of the natural deduction proof. It provides a very compact form of the proof and gives a base from which to perform logical analysis, and then build a new natural deduction proof using a well-designed transformation.
1.2 Overview of the Thesis

In Chapter 2 we present the logical basis for the theorem proving system just described. The first half of the chapter contains three proof systems that are incorporated into the theorem prover. The first system presented is LK+, the natural deduction system used in the interactive environment. This is followed by the resolution system which is used in the automatic component. Finally, we present another proof system called expansion tree proofs (ET-proofs) [Miller 83]. They are used as an intermediate form between resolution and LK+. Like resolution, the ET-proof system is an abstract proof representation, but has several advantages over resolution. The main advantage is that it is very straightforward to translate an ET-proof to an LK+ proof. Thus, in the system, an automatically generated resolution refutation is first translated to an ET-proof, and then to an LK+ proof.

The second half of Chapter 2 describes the transformation algorithms among the different proof systems. The first two are the transformations from resolution to ET-proof [Pfenning 84] and ET-proof to LK+ proof [Miller 85, Pfenning 84] as mentioned above. They form the two step process in transforming an automatically generated resolution refutation to a natural deduction proof. The final algorithm is the reverse transformation from LK+ to ET-proof [Miller 83]. It is used in the proof revision component to obtain the abstract structure which is the base structure used in proof revision.

Chapter 3 describes in detail the first two components of the system shown in Figure 1. Section 3.1 explains the programming language approach which we have developed to facilitate the integration of interactive and automatic theorem proving. It is a language based on tactics and tacticals as in LCF [Gorden, Milner, & Wadsworth 79]. The data structures in this programming language are the explicit representations of proofs in each of the proof systems. LK+ proofs are values in the system, and ET-proofs specify their types. This typing mechanism is based on the notion of formulas-as-types found in [Howard 80]. Section 3.2 shows how
1.2 Overview of the Thesis

the programming language primitives are used to create an interactive proof editor i.e. the natural proof environment of Figure 1. Section 3.3 describes how we integrate an automatic theorem prover into the proof construction component. Finally Section 3.4 describes the design of the proof revision component.

In the first part of Chapter 4, the concluding section, we discuss applications of the different types of proof objects found in the system. Finally, we discuss ways in which the comprehensive theorem proving system can be used as a tool in several AI applications.
2 The Logic

The logical basis for the $\chi$ theorem proving system is divided into two parts. The first contains the logic systems which are used in the theorem prover. They are described in the first half of this chapter. The second is the transformations among these proof systems which allow us to integrate the different kinds of proofs. These are presented in the second half.

2.1 The Logic Systems

The first system presented in this section is LK+, the natural deduction system used in the interactive environment. It is based on the Gentzen sequential system LK without the cut rule (LK-{cut}) and the related Gentzen natural deduction system NK [Gentzen 35]. NK is a slightly more natural system, but LK is better suited for our implementation. LK+ was developed by starting with LK and modifying and adding inference rules that facilitate the construction of more natural proofs, and so is technically a sequential system. Since sequential and natural deduction systems have many similarities, and "natural" proofs can be built in either, we use the term natural deduction loosely to include sequential systems.

We then present the resolution system which is used in the automatic component. This is followed by a presentation of expansion tree proofs (ET-proofs) [Miller 83] which are used as an intermediate form between resolution and LK+. In this section we also describe another proof structure called matings. Matings provide additional information that guide the transformation from ET-proof to LK+ proof. They are obtained from the resolution refutations in the transformation from resolution to ET-proof.

2.1.1 The LK+ Natural Deduction System

In the $\chi$ theorem proving system, the interactive environment for constructing proofs is based on the LK+ natural deduction system. We present this system and describe how it evolved from the Gentzen LK-{cut} system as described in [Gallier 86].

The basic structure of this system is the sequent, written $\Gamma \rightarrow \Theta$ where $\Gamma$ and $\Theta$ are lists of formulas. The meaning of this sequent is $[\wedge \Gamma] \supset [\vee \Theta]$, or more informally "from the formulas in $\Gamma$, some formula in $\Theta$ can be proved." Axioms in this system are of the form $A \rightarrow A$ where $A$ is any arbitrary first-order formula. The inference rules of the LK+ system are the following:
2.1.1 The LK+ Natural Deduction System

Introduction Rules

\[
\begin{align*}
\Gamma \rightarrow \Delta, A, \Theta & \quad \Gamma \rightarrow \Delta, C, \Theta \\
\hline
\Gamma \rightarrow \Delta, A \land C, \Theta & \quad \Gamma, A, C, \Delta \rightarrow \Theta
\end{align*}
\]
\[
\begin{align*}
\Gamma, A, \Delta \rightarrow \Theta & \quad \Gamma, C, \Delta \rightarrow \Theta \\
\hline
\Gamma, A \lor C, \Delta \rightarrow \Theta & \quad \Gamma \rightarrow \Delta, A \lor C, \Theta
\end{align*}
\]
\[
\begin{align*}
\Gamma, \Delta \rightarrow A, \Theta & \quad C, \Gamma, \Delta \rightarrow \Theta \\
\hline
\Gamma, A \supset C, \Delta \rightarrow \Theta & \quad A, \Gamma \rightarrow C, \Delta, \Theta
\end{align*}
\]
\[
\begin{align*}
\Gamma, \Delta \rightarrow A, \Theta & \quad A, \Gamma \rightarrow \Delta, \Theta \\
\hline
\Gamma, \neg A, \Delta \rightarrow \Theta & \quad \Gamma \rightarrow \Delta, \neg A, \Theta
\end{align*}
\]
\[
\begin{align*}
\Gamma, [x/t]P, \Delta \rightarrow \Theta & \quad \Gamma \rightarrow \Delta, [x/t]P, \Theta \\
\hline
\Gamma, \forall x P, \Delta \rightarrow \Theta & \quad \Gamma \rightarrow \Delta, \exists x P, \Theta
\end{align*}
\]
\[
\begin{align*}
\Gamma, [x/y]P, \Delta \rightarrow \Theta & \quad \Gamma \rightarrow \Delta, [x/y]P, \Theta \\
\hline
\Gamma, \exists x P, \Delta \rightarrow \Theta & \quad \Gamma \rightarrow \Delta, \forall x P, \Theta
\end{align*}
\]

Structural Rules

\[
\begin{align*}
\Gamma, \Delta \rightarrow \Theta & \quad \text{thin-L} \\
\hline
\Gamma, A, \Delta \rightarrow \Theta & \quad \text{thin-R}
\end{align*}
\]
\[
\begin{align*}
\Gamma, A, A, \Delta \rightarrow \Theta & \quad \text{contract-L} \\
\hline
\Gamma, A, \Delta \rightarrow \Theta & \quad \text{contract-R}
\end{align*}
\]

Additional Rules

\[
\begin{align*}
\Gamma, \exists x \neg P, \Delta \rightarrow \Theta & \quad \neg \forall-L \\
\hline
\Gamma, \neg \forall x P, \Delta \rightarrow \Theta & \quad \neg \forall-R
\end{align*}
\]
\[
\begin{align*}
\Gamma, \forall x \neg P, \Delta \rightarrow \Theta & \quad \neg \exists-L \\
\hline
\Gamma, \neg \exists x P, \Delta \rightarrow \Theta & \quad \neg \exists-R
\end{align*}
\]
\[
\begin{align*}
\Gamma, A, \Delta \rightarrow \Theta & \quad \neg \neg-L \\
\hline
\Gamma, \neg \neg A, \Delta \rightarrow \Theta & \quad \neg \neg-R
\end{align*}
\]
\[
\begin{align*}
\neg [\Delta], \Gamma \rightarrow \Delta & \quad \text{indirect} \\
\hline
\Gamma' \rightarrow \Delta' & \quad \text{thin*}
\end{align*}
\]
\[
\begin{align*}
\Gamma', \Delta' \rightarrow A, \Theta' & \quad C, \Gamma'', \Delta'' \rightarrow \Theta'' \\
\hline
\Gamma, A \supset C, \Delta \rightarrow \Theta & \quad \supset \text{-L*}
\end{align*}
\]
2.1.1 The LK+ Natural Deduction System

\[
\frac{\Gamma, \Delta \rightarrow A \quad C, \Gamma, \Delta \rightarrow \Theta}{\Gamma, \Delta \supset C, \Delta \rightarrow \Theta} \quad \text{positive}
\]

\[
\frac{\neg A, \Gamma, \Delta \rightarrow \Theta \quad C, \Gamma, \Delta \rightarrow \Theta}{\Gamma, \Delta \supset C, \Delta \rightarrow \Theta} \quad \text{contrapos}
\]

\[
\frac{\Gamma, \Delta \rightarrow A, \Theta, \Lambda \quad C \rightarrow C}{\Gamma, \Delta \supset C, \Delta \rightarrow \Theta, C, \Lambda} \quad \text{backchain}
\]

\[
\frac{A \rightarrow A \quad C, \Gamma, \Delta, \Theta \rightarrow \Lambda}{\Gamma, \Delta \supset C, \Delta, A, \Theta \rightarrow \Lambda} \quad \text{forwardchain}
\]

\[
\frac{A \rightarrow A \quad C, \Gamma, \Delta, \Theta \rightarrow \Lambda}{\Gamma, \Delta, A \supset C, \Theta \rightarrow \Lambda} \quad \text{forwardchain}
\]

The proviso that the variable y is not free in any formula of the lower sequent is placed on the \(\exists\)-L and \(\forall\)-R rules. In the thin* and \(\supset\)-L* rules \(\Gamma', \Gamma'' \subseteq \Gamma\), \(\Delta', \Delta'' \subseteq \Delta\), and \(\Theta', \Theta'' \subseteq \Theta\). In the indirect rule \(\neg[\Delta]\) denotes the list of negations of the formulas in \(\Delta\).

A proof of a formula \(A\) in this system is a finite tree constructed using a series of inference rules with the sequent \(\rightarrow A\) at the root and axioms at all the leaves. Soundness and completeness of LK+ is obtained from the following two propositions and corollary. See Appendix A for the proofs.

**Proposition 1** (Relative Completeness of LK+) *If a sequent \(\Gamma \rightarrow \Delta\) has an LK-\{cut\} proof, then it has an LK+ proof.*

**Proposition 2** (Soundness of LK+) *If there is an LK+ proof tree for a sequent \(\Gamma \rightarrow \Delta\), then the sequent is valid, i.e. \([\Lambda\Gamma] \supset [\forall\Delta]\) is valid.*

**Corollary 3** The \(\land\)-R, \(\land\)-L, \(\lor\)-R, \(\lor\)-L, \(\supset\)-L, \(\supset\)-R, \(\neg\)-L, \(\neg\)-R, \(\forall\)-L, \(\forall\)-R, \(\exists\)-L, \(\exists\)-R, contract-L, contract-R, and thin* rules form a sound and complete subset of LK+.

**Building Natural Proofs in LK+** The proof system LK+ was designed to facilitate the implementation of a theorem proving system with a proof environment that is as natural as possible in order to aid the user in constructing proofs. This involved adapting LK-\{cut\} according to two criteria: the individual inference rules must be natural (i.e. the conclusions should follow naturally from the premises), and natural proofs should be easily constructed using these rules. With this in mind we discuss the individual inference rules of LK+ as well as some general criteria and strategies for building natural proofs.
2.1.1 The LK+ Natural Deduction System

For reference, the complete LK-{cut} system is given in Appendix A. In LK-{cut}, rules can only be applied to the leftmost formulas on the left side of the sequent arrow and to the rightmost formulas on the right side of the sequent arrow. In LK+, we increase flexibility by allowing inference rules to be applied to a formula anywhere in the sequent. We do not require that formulas be moved to the beginning or end of a list, and thus do not need the following LK-{cut} interchange rules.

\[
\frac{\Gamma, A, C, \Delta \rightarrow \Lambda}{\Gamma, C, A, \Delta \rightarrow \Lambda} \quad \text{interchange-L}
\]
\[
\frac{\Gamma \rightarrow \Delta, A, C, \Lambda}{\Gamma \rightarrow \Delta, C, A, \Lambda} \quad \text{interchange-R}
\]

The individual inference rules of LK+ fall into two categories: those that are modified forms of LK-{cut} rules, and those that were added to the system for flexibility in building natural proofs. We begin by discussing those that closely resemble LK-{cut} rules. First of all, several rules were taken almost directly from LK-{cut}. The only difference is that they can be applied to a formula anywhere in the sequent. These include \(\wedge\)-R, \(\vee\)-L, \(\supset\)-R, \(\neg\)-L, \(\neg\)-R, \(\forall\)-L, \(\forall\)-R, \(\exists\)-L, \(\exists\)-R, \(\text{thin-L}\), \(\text{thin-R}\), \(\text{contract-L}\), and \(\text{contract-R}\).

The \(\wedge\)-L and \(\vee\)-R rules have a further modification. The LK-{cut} \(\wedge\)-L rule is defined as follows:

\[
\frac{A, \Gamma \rightarrow \Delta}{A \wedge C, \Gamma \rightarrow \Delta} \quad \text{\(\wedge\)-L}
\]

and the \(\vee\)-R rule is similar. These rules introduce one of the conjuncts or disjuncts "from nowhere." It is more natural to form a conjunct on the left or a disjunct on the right from two formulas already existing in the sequent, which is how the corresponding LK+ rules have been defined.

The \(\supset\)-L rule is also modified. All formulas in the conclusion on both sides of the arrow appear in both premises in the LK+ rule, whereas in the LK-{cut} rule, each formula appears in only one premise as follows:

\[
\frac{\Gamma \rightarrow \Delta, A \quad C, \Lambda \rightarrow \Theta}{A \supset C, \Gamma, \Lambda \rightarrow \Delta, \Theta} \quad \text{\(\supset\)-L}
\]

Each formula may not be needed in each branch of the proof, but when a user is interactively constructing a proof tree it may not be apparent which formulas are needed in each branch. Thus all are passed to both subtrees and the user may thin out appropriate formulas as s/he sees fit (using \(\text{thin-R}\) and \(\text{thin-L}\)). When an ET-proof is present and is being transformed to an LK+ proof, it will be possible to determine which formulas are needed to complete the proof of each branch. This is the reason for including the \(\supset\)-L* rule whose premises contain any subset of the formulas in the conclusion. (See Section 2.2.2.)
2.1.1 The LK+ Natural Deduction System

The remaining LK+ rules are those that were added in addition to the basic rules. The backchain, forwardchain, positive, and contrapos rules are all variations of the \( \vdash \text{-L}^* \) which aid the user in constructing more natural proofs. Backchain and forwardchain are used when it is possible to directly obtain an axiom from one branch. Notice that forwardchain is simply an application of modus ponens on the left side of the sequent arrow. Backchain has the standard interpretation—if we are trying to prove \( C \), and we know \( A \vdash C \), we want to try to prove \( A \). The positive rule uses the implication in the positive direction. If we can prove \( A \), and assuming \( C \) allows us to prove \( \Theta \), then we can prove \( \Theta \) from \( A \vdash C \). This is also similar to modus ponens. The contrapos rule uses the implication negatively. It is similar to the positive rule applied with the formula \( A \vdash C \) replaced by the equivalent form \( \neg C \vdash \neg A \) as follows:

\[
\begin{align*}
\frac{C, \Gamma, \Delta \rightarrow \Theta}{\neg A, \Gamma, \Delta \rightarrow \Theta} & \quad \text{\( \neg \text{-L} \)} \\
\frac{\Gamma, \Delta \rightarrow \neg C}{\Gamma, \neg C \vdash \neg A, \Delta \rightarrow \Theta} & \quad \text{positive}
\end{align*}
\]

If we can prove \( \Theta \) from \( \neg A \), and assuming \( C \) gives us a contradiction, then we can prove \( \Theta \) from \( A \vdash C \).

The four rules that push negation past a quantifier simply give the user a choice in how to operate on the expression. The user can use one of these rules instead of the \( \neg \text{-L} \) or \( \neg \text{-R} \) rules. This choice is useful for satisfying certain criteria for constructing natural proofs. For instance, building a natural proof involves proving a conclusion from a set of hypotheses. But, recall that the meaning of a sequent \( \Gamma \rightarrow \Theta \) is more general, “from the formulas (hypotheses) in \( \Gamma \), we can prove some formula in \( \Theta \).” This disjunctive nature of the conclusion is less natural. Suppose that \( A \) is a formula in \( \Theta \) that can be proved. Instead of proving \( A \) by building a proof tree for \( \Gamma \rightarrow \Theta \), we can build a proof tree of \( \Gamma, \neg C \rightarrow \neg A, \Delta \rightarrow \Theta \). In general, when building proofs, it is a good strategy to try to keep exactly one formula on the right and avoid rules such as \( \neg \text{-L} \) and \( \neg \text{-R} \) that violate this by moving formulas back and forth. Pushing a negation past a quantifier allows a formula to be replaced by an equivalent one, thereby avoiding undesirable movement of formulas within the sequent.

Both the contrapos and indirect rules could introduce a double negation. The purpose of the \( \neg \neg \text{-L} \) and \( \neg \neg \text{-R} \) rules is to eliminate such double negations. Notice that they can be simulated by an application of \( \neg \text{-L} \) followed by \( \neg \text{-R} \) and vice versa, respectively. Like the rules that push negation past a quantifier, these rules are included because it is more natural to simply replace a formula by an equivalent one, rather than move formulas from one side of the sequent arrow to the other.

Though, it is desirable to have exactly one formula on the right of the sequent.
2.1.1 The LK+ Natural Deduction System

arrow, another possibility is to have no formulas on the right. In this case, we are building a proof by contradiction. There are several rules that may cause this to occur, but the indirect rule was included specifically to increase flexibility by allowing such proofs. It can be applied at any time, though it is most natural to apply it when there is only one formula on the right.

Like the $\supset$-L$^*$ rule, the thin$^*$ rule is used when transforming an ET-proof to an LK+ proof. It uses the extra knowledge provided to determine which formulas are no longer needed, and obtains the premise of the rule by removing these formulas. It is included specifically for this transformation algorithm, and is really nothing more than one or more applications of the thin-L and thin-R rules. All the thinning rules contribute to building natural proofs, and it is desirable to apply them whenever possible. This enhances readability by allowing formulas that may clutter the proof to be removed, and avoids extra complexity that could be caused by applying inference rules to formulas that are not needed for the proof.

The cut rule in the LK proof system is very intuitive and when used can often create much more natural and less complex proofs. Yet, this rule is different from the other rules in that it does not “obey the subformula principle.” The other rules operate on a sequent such that the premises contain only subformulas of formulas in the conclusion. Proof transformations become much more difficult when the cut rule is used. In a sense, proofs with cuts are not “reduced” in the same way that ET-proofs, resolution refutations, and LK-{cut} proofs are. Systems that have this property have been called analytic proof systems [Smullyan 68, Miller 83, Pfenning 84]. We restrict our theorem proving environment to this kind of proof system, and do not consider proof systems which are non-analytic. Thus we exclude the cut rule from LK+.

While it is essential that the inference rules of a proof system be natural in order to build natural proofs, it is also important to consider some general strategies for constructing such proofs. Several such strategies, such as attempting to keep only one formula on the right or removing unnecessary formulas whenever possible, have emerged in the discussion of the individual inference rules. Another strategy is to delay rules that cause branching in the tree. If these rules are applied early, it may be the case that certain subproofs will have to be repeated in both branches. It is best to avoid such unnecessary repetition.

In general, the criteria for determining what makes a proof natural can be quite subjective. We have described several here that we feel are important. In the $\chi$ system that will be described in Chapter 3 we allow a great deal of flexibility in the construction of proofs, so that a user may customize proof construction to fit his or her own needs and ideas.
2.1.2 Resolution

Resolution is a common paradigm for building automatic theorem provers, and so we provide the capability to integrate automatically generated resolution refutations into the natural deduction theorem proving environment. The resolution method is described in this section. The following definition of resolution is taken from [Pfenning 84].

**Definition 1** If $B$ is a formula, let $B^*$ denote its skolem normal form, i.e. essentially existential quantifiers are instantiated with Skolem terms and all essentially universal variables are deleted. We shall use $\text{cnf}(B)$ to denote the set of sets of literals which comprises the conjunctive normal form of $B$. Let $\mathcal{H}_B$ denote the set of first-order terms which are composed only of functions and constants of $B$, plus an additional constant added to ensure that $\mathcal{H}_B$ is non-empty. A resolution refutation of $B$ is a list of clauses (which are sets of literals) $C_1, \ldots, C_m$, such that $C_m$ is the empty clause and for each $i = 1, \ldots, m$, one of the following is true:

(a) $C_i \in \text{cnf}((\neg B)^*)$, ($C_i$ is called a clause), or

(b) there are positive integers $j, k$ less than $i$ and sets of literals $S_1$ and $S_2$ such that $C_i = S_1 \cup S_2$, $C_j = S_1 \cup \{A\}$, and $C_k = S_2 \cup \{-A\}$, for some atomic formula $A$, ($A$ is called the resolvent of $C_j$ and $C_k$), or

(c) there is a substitution $\varphi$ built using only terms in $\mathcal{H}_B$ and a positive integer $j < i$ such that $C_i = \varphi C_j$.

**Example 1** A refutation of the formula:

$$[p(a) \lor q(b)] \land \forall x \ [p(x) \supset q(x)] \supset \exists x \ q(x)$$

is shown below. The formula is first negated and put into normal form:

$$\forall x \forall y \ [(p(a) \lor q(b)) \land [\neg p(x) \lor q(x)] \land \neg q(y)]$$

from which we obtain the clauses in steps (1)–(3).

(1) $p(a), q(b)$ by (a)
(2) $\neg p(x), q(x)$ by (a)
(3) $\neg q(y)$ by (a)
(4) $\neg p(a), q(a)$ by (c) from 2
(5) $q(a), q(b)$ by (b) from 1 and 4
(6) $\neg q(b)$ by (c) from 3
(7) $q(a)$ by (b) from 5 and 6
(8) $\neg q(a)$ by (c) from 3
(9) by (b) from 7 and 8
2.1.3 Expansion Tree Proofs and Matings

Expansion tree proofs (ET-proofs), like resolution refutations are an abstract proof structure. Computationally, ET-proofs have several advantages over resolution refutations which will become apparent and will be discussed later. We use them as an intermediate structure in the transformation from resolution to LK+. They are also the base structure for proof revision. In this section we define the expansion tree proof system. ET-proofs were first defined and developed in [Miller 83]. The following definition for expansion trees is a combination of those found in [Miller 85,Miller & Felty 86].

Definition 2 Expansion trees, dual expansion trees, selected variables and expansion terms are defined as follows:

1. Let $A$ be a formula. Then $A$ is both an expansion tree and a dual expansion tree for $A$.

2. If $y$ is a variable and $Q$ is an expansion tree for $[x/y]A$, then $(\forall x A, (y, Q))$ is an expansion tree for $\forall x A$. If $Q$ is a dual expansion tree for $[x/y]A$, then $(\exists x A, (y, Q))$ is a dual expansion tree for $\exists x A$. We call $y$ a selected variable.

3. If $t_1, \ldots, t_n$ are first-order terms and for $i = 1, \ldots, n$, $Q_i$ is an expansion tree for $[x/t_i]A$, and shares no selected variables with any other $Q_j$, for $j = 1, \ldots, n$ (except itself), then

   $$(\exists x A, (t_1, Q_1), \ldots, (t_n, Q_n))$$

is an expansion tree for $\exists x A$. If $Q_i$ is a dual expansion tree for $[x/t_i]A$, and shares no selected variables with any other $Q_j$, for $j = 1, \ldots, n$ (except itself), then

   $$(\forall x A, (t_1, Q_1), \ldots, (t_n, Q_n))$$

is a dual expansion tree for $\forall x A$. We call $t_1, \ldots, t_n$ expansion terms.

4. If $Q$ is an expansion tree for $A$, then $\neg Q$ is a dual expansion tree for $\neg A$. If $Q$ is a dual expansion tree for $A$, then $\neg Q$ is an expansion tree for $\neg A$.

5. If $Q_1$ and $Q_2$ are expansion trees for $A_1$ and $A_2$ respectively, that do not share any selected variables, then $Q_1 \land Q_2$ and $Q_1 \lor Q_2$ are expansion trees for $A_1 \land A_2$ and $A_1 \lor A_2$ respectively. If $Q_1$ and $Q_2$ are dual expansion trees for $A_1$ and $A_2$ respectively, that do not share any selected variables, then $Q_1 \land Q_2$ and $Q_1 \lor Q_2$ are dual expansion trees for $A_1 \land A_2$ and $A_1 \lor A_2$ respectively.

6. If $Q_1$ is a dual expansion tree for $A_1$ and $Q_2$ is an expansion tree for $A_2$, and $Q_1$ and $Q_2$ do not share any selected variables, then $Q_1 \supset Q_2$ is an expansion tree for $A_1 \supset A_2$ and $Q_2 \supset Q_1$ is a dual expansion tree for $A_2 \supset A_1$. 


Several more definitions are needed in order to develop the notion of a proof using expansion trees.

**Definition 3** Let $Q$ be an expansion tree for $A$. $Q$ is **sound** if none of the free variables in $A$ are selected in $Q$.

We now define the functions $Sh$ and $Dp$, which extract the shallow and deep formulas from an expansion tree or dual expansion tree.

**Definition 4** Let $Q$ be an expansion tree or a dual expansion tree for $A$. The functions $Sh$ and $Dp$ which map expansion trees and dual expansion trees to formulas are defined as follows:

1. If $Q$ is a one node tree then $Sh(Q) := A$ and $Dp(Q) := A$.
2. If $Q = \neg Q_1$ then $Sh(Q) := \neg Sh(Q_1)$ and $Dp(Q) := \neg Dp(Q_1)$.
3. If $Q = Q_1 \lor Q_2$, $Q = Q_1 \land Q_2$, or $Q = Q_1 \supset Q_2$, then $Sh(Q) := Sh(Q_1) \lor Sh(Q_2)$, $Sh(Q) := Sh(Q_1) \land Sh(Q_2)$, or $Sh(Q) := Sh(Q_1) \supset Sh(Q_2)$, respectively. $Dp(Q) := Dp(Q_1) \lor Dp(Q_2)$, $Dp(Q) := Dp(Q_1) \land Dp(Q_2)$, or $Dp(Q) := Dp(Q_1) \supset Dp(Q_2)$, respectively.
4. If $Q$ is an expansion tree and $Q = (\forall x B, (y, Q_1))$ or $Q$ is a dual expansion tree and $Q = (\exists x B, (y, Q_1))$ then $Sh(Q) := \forall x B$ or $Sh(Q) := \exists x B$ respectively. In both cases, $Dp(Q) := Dp(Q_1)$.
5. If $Q$ is an expansion tree and $Q = (\exists x B, (t_1, Q_1), \ldots, (t_n, Q_n))$ or $Q$ is a dual expansion tree and $Q = (\forall x B, (t_1, Q_1), \ldots, (t_n, Q_n))$ then $Sh(Q) := \exists x B$ or $Sh(Q) := \forall x B$ respectively. $Dp(Q) := Dp(Q_1) \lor \ldots \lor Dp(Q_n)$ or $Dp(Q) := Dp(Q_1) \land \ldots \land Dp(Q_n)$ respectively.

Note that if $Q$ is an expansion tree for $A$, then $Sh(Q) = A$.

**Definition 5** Let $Q$ be an expansion tree. Let $S_Q$ be the set of all selected variables in $Q$. Let $\Theta_Q$ be the set of occurrences of expansion terms in $Q$. We define $\prec_Q$ to be a binary relation on $S_Q$ such that $z \prec_Q y$ if there exists an expansion term occurrence $t \in \Theta_Q$ such that $z$ is free in $t$, and the expansion tree or dual expansion tree, $Q'$, that is paired with $t$ (i.e. $(t, Q')$ appears in $Q$) contains an expansion tree or dual expansion tree, $Q''$, that is paired with $y$ (i.e. $(y, Q'')$ appears in $Q'$). Let $\prec_Q$ denote the transitive closure of $\prec_Q$. $\prec_Q$ is called the imbedding relation for $Q$.

**Definition 6** Let $Q$ be an expansion tree for $A$. $Q$ is an expansion tree proof (ET-proof) for $A$ if
2.1.3 Expansion Tree Proofs and Matings

1. $Q$ is sound.
2. $Dp(Q)$ is a tautology.
3. $\neg_Q$ is acyclic.

For soundness and completeness of ET-proofs see [Miller 83]. We present one more definition related to ET-proofs that will be needed for proof transformations.

**Definition 7** A term $t$ is *admissible* in an expansion tree $Q$ if no variable free in $t$ is contained in $S_Q$.

**Example 2** The following are expansion trees for

\[ [p(a) \lor q(b)] \land \forall x \ [p(x) \supset q(x)] \supset \exists x \ q(x). \]

1. \[ [p(a) \lor q(b)] \land \forall x \ [p(x) \supset q(x)] \supset \exists x \ q(x) \]
2. \[ [p(a) \lor q(b)] \land \forall x \ [p(x) \supset q(x)] \supset (\exists x \ q(x), (a, q(a))) \]
3. \[ [p(a) \lor q(b)] \land (\forall x \ [p(x) \supset q(x)], (a, p(a) \supset q(a))) \supset (\exists x \ q(x), (a, q(a)), (b, q(b))) \]

The last one is an ET-proof.

Informally, the tree notation for an expansion tree is obtained by putting the main connective at the root and recursively constructing the expansion trees of the subformulas as the subtrees, with atoms or quantified formulas at the leaves. At quantified nodes, each child arc is labeled with a selected variable or expansion term, and its corresponding subtree is constructed below. The tree notation for the ET-proof above is shown in Figure 2.

One advantage of ET-proofs is that they are a very compact proof representation, and thus computationally oriented. They also store substitution information locally. The terms that must be substituted in quantified formulas are found at the labels of the arcs immediately below these formulas. Both of these characteristics facilitate the transformation to LK+. The choice of which LK+ inference rule to apply is based on the root node, which will be either a connective or a quantified formula with substitution information readily available.

We now introduce another proof structure called *matings*. They are also part of the intermediate representation. They are obtained in the transformation from resolution refutation to ET-proof, and are then used to help guide the transformation from ET-proof to LK+ proof. Matings often provide information that can contribute to building a natural LK+ proof. We begin with a few preliminary definitions.
Definition 8 We call any formula that contains no logical connectives an atom. An atom $A$ occurs positively, (negatively) in a formula $C$ if it is in the scope of an even (odd) number of occurrences of $\neg$ in the negation normal form of $B$.

In the following definition, when $A_1$ and $A_2$ are sets, then $A_1 \uplus A_2 := \{e_1 \cup e_2 | e_1 \in A_1, e_2 \in A_2\}$.

Definition 9 Let $D$ be a formula. The set $C_D$ of clauses in $D$ and the set $V_D$ of dual clauses in $D$ are defined as follows.

1. If $D$ is an atom, then $C_D := \{\{D\}\}$ and $V_D := \{\{D\}\}$.
2. If $D = \neg D_1$ then $C_D := V_{D_1}$ and $V_D := C_{D_1}$.
3. If $D = D_1 \lor D_2$ then $C_D := C_{D_1} \uplus C_{D_2}$ and $V_D := V_{D_1} \lor V_{D_2}$.
4. If $D = D_1 \land D_2$ then $C_D := C_{D_1} \cup C_{D_2}$ and $V_D := V_{D_1} \uplus V_{D_2}$.
5. If $D = D_1 \supset D_2$ then $C_D := V_{D_1} \uplus C_{D_2}$ and $V_D := C_{D_1} \lor V_{D_2}$.

Definition 10 Let $D$ be a formula. Let $M$ be a set of unordered pairs of atom occurrences of $D$, such that if $\{H, K\} \in M$, then $H$ and $K$ are different occurrences of the same atom, and either $H$ occurs positively and $K$ occurs negatively in $D$, or vice versa. $M$ is called a mating for $D$. $M$ is a clause-spanning mating (cs-mating)
for $D$ if for every $\varepsilon \in C_D$ there is an $\{H, K\} \in M$ such that $\{H, K\} \subseteq \varepsilon$. We also say $M$ spans $D$, or $D$ is spanned by $M$. If $D$ is a set of formulas, $M$ is a mating (cs-mating) for $D$ if $M$ is a mating (cs-mating) for $\forall D$.

**Proposition 4** Let $D$ be a first-order formula. $D$ is tautologous if and only if $D$ has a cs-mating.

**Proof:** See [Miller 83].

**Example 3** Given the formula:

$$(p(a)_1 \lor q(b)_1) \land (p(a)_2 \lor q(a)_1) \supset (q(a)_2 \lor q(b)_2)$$

a clause spanning mating is as follows:

$$\{\{p(a)_1, p(a)_2\}, \{q(a)_1, q(a)_2\}, \{q(b)_1, q(b)_2\}\}.$$ 

Note that this formula is the deep formula for the third ET-proof given in Example 2. Here, the atom occurrences which appear at the leaves of the tree have been subscripted. This is to distinguish different occurrences of the same atom, and to indicate corresponding atoms in the formula and mating. Since the mating is clause spanning, this formula is a tautology.

We now generalize the definition of a sequent and ET-proof.

**Definition 11** $\Gamma \rightarrow \Delta; M$ is a generalized sequent if $\Gamma$ is a set of dual expansion trees and $\Delta$ is a set of expansion trees, and $M$ is a (possibly empty) mating for $[\land \Gamma] \supset [\lor \Delta]$. $\Gamma \rightarrow \Delta; M$ is an ET-proof if it is a generalized sequent and $[\land \Gamma] \supset [\lor \Delta]$ is an ET-proof.

Note that since for any formula $A$, $A$ is an expansion tree and a dual expansion tree for itself, we can think of the set of formulas on the left of the arrow in the usual notion of sequent to be a set of dual expansion trees, and the set of formulas on the right to be a set of expansion trees. Thus any sequent is also a generalized sequent (with an empty mating). From this point on we will call a sequent with only sets of formulas on each side of the sequent arrow a simple sequent. Generalized sequents will be very useful for transforming proofs between the ET and LK+ proof systems.
2.2 Proof Transformations

This section presents the transformation algorithms among the different proof systems. The first algorithm we discuss is the transformation from resolution to ET-proof and mating [Pfenning 84]. In the \(\chi\) system, this transformation takes place within the automatic theorem prover in the proof construction component (see Figure 1). The second transformation algorithm is from ET-proof and mating to LK+ proof. This is the transformation represented by the arrow from the automatic theorem prover to the natural proof environment in the proof construction component. This algorithm is also used in the proof revision component, when translating from abstract proof to revised natural proof. The final algorithm is the transformation from LK+ to ET-proof [Miller 83]. It is the complement to the ET-proof to LK+ algorithm and thus completes a cycle of transformations. It is used in the proof revision component, and operates on a completed LK+ proof obtained during proof construction. This completed LK+ proof is transformed to an ET-proof which is the base structure used in proof revision.

2.2.1 Transforming Resolution Refutations to ET-Proofs

An algorithm to transform a resolution refutation to an expansion tree proof and mating can be found in [Pfenning 84]. In the \(\chi\) system that will be described in Chapter 3, any automatically generated resolution refutation will immediately be transformed to an ET-proof, and all further computations and manipulations will be done on the expansion tree structure. In a sense, this transformation can be thought of as a black box. The algorithm will not be presented here. The interested reader is referred to [Pfenning 84].

2.2.2 Transforming ET-Proofs to LK+ Proofs

The algorithm for transforming an ET-proof to an LK+ proof, based on algorithms found in [Miller 85,Pfenning 84], is presented in this section. It is generalized to operate on generalized sequents of the form \(\Gamma \rightarrow \Delta; \mathcal{M}\), where the expansion tree \(\sqcup \Gamma \supset \sqcup \Delta\) must be an expansion tree proof (see Definition 11). It builds a proof tree in a bottom up fashion starting with the formula to be proven and working upwards until all leaves are axioms. The algorithm is built from a series of functions, each representing an inference rule of LK+. We call these functions LK+ transformation functions. Each function takes a generalized sequent as input, and has some criteria it must check before it knows whether or not the LK+ rule can be applied. If it is possible to apply the rule, the function will operate on the relevant expansion tree(s) in the sequent, and produce the generalized sequents that correspond to the premise(s) of the rule. If it is not possible to apply the rule (if
any of the if conditions fail), the function as a whole will fail. For readability, from this point on, we will leave out the mating $M$ in a generalized sequent whenever its contents are not relevant to the discussion.

**Breaking Propositional Connectives** For any propositional rule to be applicable, the only criteria that must be satisfied is that the sequent must contain a formula with the appropriate connective. The functions for these rules are very straightforward. The following and-r function for the $\&$-R rule illustrates these propositional transformation functions.

**and-r**

If the generalized sequent is of the form $\Gamma \rightarrow \Delta, A \& C, \Theta$ then build the partial proof according to the $\&$-R rule as follows:

\[
\frac{\Gamma \rightarrow \Delta, A, \Theta \quad \Gamma \rightarrow \Delta, C, \Theta}{\Gamma \rightarrow \Delta, A \& C, \Theta}
\]

Return the generalized sequents representing the premises of the rule.

$$
\Gamma \rightarrow \Delta, A, \Theta \\
\Gamma \rightarrow \Delta, C, \Theta.
$$

The functions or-1, and-1, or-r, neg-1, neg-r, implies-r, and implies-1 are defined similarly for the $\lor$-L, $\lor$-R, $\neg$-L, $\neg$-R, $\supset$-R, and $\supset$-L inference rules, respectively. Though the $\neg\neg$-L and $\neg\neg$-R rules do not break propositional connectives, they fit into this category since they also only require a formula of a certain form in the sequent in order to be applicable. Thus the neg2-1 and neg2-r functions are defined corresponding to $\neg\neg$-L and $\neg\neg$-R respectively.

**Handling Quantifiers** The following rules make use of the substitution information obtained from the resolution refutation, and stored locally in the ET-proof. The functions for these rules illustrate the advantages of the structure and properties of ET-proofs.

**contract-r**

If the generalized sequent is of the form

$$
\Gamma \rightarrow \Delta, (\exists x P, (t_1, Q_1), \ldots, (t_n, Q_n)), \Theta
$$

and $n > 1$, and there is some $t_i$ that is admissible, then build the partial proof as follows:

\[
\frac{\Gamma \rightarrow \Delta, \exists x P, \exists x P, \Theta}{\Gamma \rightarrow \Delta, \exists x P, \Theta}
\]

**contract-R**
Return the generalized sequent representing the premise of the rule with expansion trees modified as follows:

\[ \Gamma \rightarrow \Delta, (\exists x \ P, (t_1, Q_1)), (\exists x \ P, (t_1, Q_1), \ldots, (t_{i-1}, Q_{i-1}), (t_{i+1}, Q_{i+1}), \ldots, (t_n, Q_n)), \Theta. \]

Note that \( t_1, \ldots, t_n \) are first-order terms, \((\exists x \ P, (t_1, Q_1), \ldots, (t_n, Q_n))\) is an expansion tree for \( \exists x \ P \), and for each \( i, i = 1, \ldots, n \), \( Q_i \) is an expansion tree for \( [x/t_i]P \).

\textbf{contract-l}

This function is similar to \textbf{contract-r} except that the sequent must contain a dual expansion tree of the form \((\forall x \ P, (t_1, Q_1), \ldots, (t_n, Q_n))\) on the left of the sequent arrow.

\textbf{exists-r}

If the generalized sequent is of the form \( \Gamma \rightarrow \Delta, (\exists x \ P, (t, Q)), \Theta \) and \( t \) is admissible, then build the partial proof as follows:

\[
\frac{\Gamma \rightarrow \Delta, [x/t]P, \Theta}{\Gamma \rightarrow \Delta, \exists x \ P, \Theta} \quad \exists\text{-R}
\]

Return the generalized sequent \( \Gamma \rightarrow \Delta, Q, \Theta \).

\textbf{forall-l}

Similar to \textbf{exists-l} for sequents of the form \( \Gamma, (\forall x \ P, (t, Q)), \Delta \rightarrow \Theta \).

\textbf{forall-r}

If the generalized sequent is of the form \( \Gamma \rightarrow \Delta, (\forall x \ P, (y, Q)), \Theta \) then build the partial proof as follows:

\[
\frac{\Gamma \rightarrow \Delta, [x/y]P, \Theta}{\Gamma \rightarrow \Delta, \forall x \ P, \Theta} \quad \exists\text{-R}
\]

Return the generalized sequent \( \Gamma \rightarrow \Delta, Q, \Theta \). Note that in the \textbf{exists-r} and \textbf{forall-l} functions we always check that a term is admissible before applying the rule. This, in conjunction with the fact that we have a sound expansion tree, prevents \( y \) from becoming free anywhere in the proof tree built so far. Thus we do not need to check the proviso to this rule in the function.

\textbf{exists-l}

Similar to \textbf{forall-r} for sequents of the form \( \Gamma, (\exists x \ P, (y, Q)), \Delta \rightarrow \Theta \).

\textbf{pushneg-r}
2.2.2 Transforming ET-Proofs to LK+ Proofs

If the generalized sequent is of the form

\[ \Gamma \rightarrow \Delta, \neg(\forall x\ P, (t_1, Q_1), \ldots, (t_n, Q_n)), \Theta \]

then build the partial proof as follows:

\[
\begin{align*}
\Gamma & \rightarrow \Delta, \exists x\ 
eg P, \Theta \\
\Gamma & \rightarrow \Delta, \neg \forall x\ P, \Theta
\end{align*}
\]

Return the generalized sequent \( \Gamma \rightarrow \Delta, (\exists x\neg P, (t_1, \neg Q_1), \ldots, (t_n, \neg Q_n)), \Theta \).

Otherwise, if the generalized sequent is of the form \( \Gamma \rightarrow \Delta, \neg(\exists x\ P, (y, Q)), \Theta \) then build the partial proof as follows:

\[
\begin{align*}
\Gamma & \rightarrow \Delta, \forall x\ 
eg P, \Theta \\
\Gamma & \rightarrow \Delta, \neg \exists x\ P, \Theta
\end{align*}
\]

Return the generalized sequent \( \Gamma \rightarrow \Delta, (\forall x\neg P, (y, \neg Q)), \Theta \).

pushneg-1

Similar to \texttt{exists-1} for sequents of the form \( \Gamma, \neg(\forall x\ P, (y, Q)), \Delta \rightarrow \Theta \) or \( \Gamma, \neg(\exists x\ P, (t_1, Q_1), \ldots, (t_n, Q_n)), \Delta \rightarrow \Theta \).

indirect*

If the generalized sequent is of the form \( \Gamma \rightarrow (\exists x\ P, (t_1, Q_1), \ldots, (t_n, Q_n)) \) where \( n > 1 \) (i.e. there is more than one expansion term) then build the partial proof as follows:

\[
\frac{\neg \exists x P, \Gamma \rightarrow \exists x P}{\Gamma \rightarrow \exists x P \text{ indirect}*}
\]

Return the generalized sequent \( \neg(\exists x\ P, (t_1, Q_1), \ldots, (t_n, Q_n)), \Gamma \rightarrow . \)

When there is more than one expansion term, the formula on the right has to be contracted before substitution is possible. This results in more than one formula on the right—an unnatural construct. The \texttt{indirect} rule provides an alternative for this case. It allows the construction of an indirect proof by negating the formula and moving it to the left. Then the \( \neg \exists - \text{L} \) rule can be applied to push the negation past the quantifier.

\textbf{The Role of Matings} Thus far, the rules for the ET to LK+ transformation have been very straightforward. Information is obtained directly from the structure of the expansion tree proof and the appropriate LK+ rule is applied. The rules described
in this section use the information contained in a mating to focus the construction of the natural deduction proof. Recall that the premise(s) of some LK+ rules such as \( \supset \text{-}L^* \), and some of its variations such as positive and contrapos contain a subset of the formulas of the conclusion of the rule. Determining which formulas to keep and which to "thin out," as well as which rule to apply, requires performing certain operations on the mating. It is possible to complete the transformation algorithm without the rules that use matings (as long as we consider any sequent that contains occurrences of the same formula on the left and right of the sequent arrow to be an axiom). But by keeping all formulas in every branch of the proof, the proof may become much more complex than necessary. Matings allow us to determine exactly which formulas in a sequent are needed to complete various subproofs of a given theorem. By removing appropriate formulas, we can avoid unnecessary steps and generally obtain simpler and more readable proofs.

To introduce the operations on matings, we first need several definitions and propositions in addition to those presented in Section 2.2.1.

**Definition 12** Let \( \mathcal{D} \) be a finite, nonempty set of formulas, and \( \mathcal{M} \) a mating for \( \mathcal{D} \). With respect to \( \mathcal{D} \) and \( \mathcal{M} \), define \( \approx^0 \) to be the binary relation on the formulas in \( \mathcal{D} \) such that when \( D_1, D_2 \in \mathcal{D}, D_1 \approx^0 D_2 \) if \( D_1 \) contains an atom subformula occurrence \( H \) and \( D_2 \) contains an atom subformula occurrence \( K \) such that \( \{H, K\} \in \mathcal{M} \). Let \( \approx \) be the reflexive, transitive closure of \( \approx^0 \). If \( D \in \mathcal{D} \) we write \([D]_{\approx}\) to denote the equivalence class of \( D \) which contains \( D \).

**Proposition 5** Let \( \mathcal{D} \) be a finite, nonempty set of formulas. If \( \mathcal{M} \) is a cs-mating for \( \mathcal{D} \) then \( \mathcal{M} \) spans at least one of the equivalence classes of the \( \approx \) relation on \( \mathcal{D} \).

**Proof:** See [Miller 83].

**Example 4** The block diagram in Figure 3 illustrates several definitions and propositions. Let \( \mathcal{D} = \{D_1, D_2, D_3, D_4, D_5\} \) be a set of formulas. Each formula is represented by a block in the figure. First of all, we illustrate a clause of \( D_i \), which is a set of atom occurrences in \( D_i \) (see Definition 9), by drawing a line through the \( D_i \) block. Note that by the definition of \( C_{\vee \mathcal{D}} \) (the set of clauses of \( \vee \mathcal{D} \)), a clause in \( \vee \mathcal{D} \) is composed of exactly one clause from each \( D_i \), for \( i = 1, \ldots, 5 \). Hence, each clause in \( \vee \mathcal{D} \) can be illustrated by a vertical path through the block diagram. Second, we illustrate mated pairs (see Definition 10) contained in a clause by indicating where they occur on the vertical path. \( \{H, K\} \) is an example of a mated pair in the clause shown in Figure 3. Third, note that if a mating \( \mathcal{M} \) for \( \mathcal{D} \) is a clause spanning mating, every vertical path contains a mated pair of atoms, and so by Proposition 4, \( \vee \mathcal{D} \) is tautologous. Fourth, we illustrate the equivalence classes of the \( \approx \) relation on the formulas in \( \mathcal{D} \) with bold face boxes. In the figure, \([D_1]_{\approx} = \{D_1, D_2\}, [D_3]_{\approx} = \{D_3\}, \)
and \([D_4]_\approx = \{D_4, D_5\}\). Finally, if \(M\) is a cs-mating, by Proposition 5 one of the equivalence classes is spanned by \(M\). Thus, for example, if \([D_1]_\approx\) is spanned by \(M\), the formula \(D_1 \lor D_2\) is a tautology.

Proposition 5 illustrates one way to determine which formulas to remove from a sequent when applying rules such as thin* during construction of an LK+ proof tree. The transformation functions corresponding to these rules are described at the end of this section. Informally, the formulas in a sequent to which we want to apply the rule construct the set \(\mathcal{D}\). We simply determine the equivalence classes for the \(\approx\) relation on this set of formulas and choose one equivalence class that is spanned by \(M\). Then for thin*, for example, we obtain the premise of the rule by retaining all formulas in the chosen class and removing all others. The remaining definitions and propositions provide us with an even better algorithm for removing unnecessary formulas from a sequent.

**Definition 13** Let \(D\) be a formula. We can write \(D = \wedge_{i=1}^{n} F_i\) where \(n \geq 1\), and no \(F_i\) is itself a conjunct. For every subset \(\mathcal{F}\) of \(\{F_1, \ldots, F_n\}\), \(\wedge \mathcal{F}\) is a cluster of \(D\).

**Definition 14** Let \(D\) be a set of formulas. A clustering \(\mathcal{C}\) of \(D\) is a set of clusters of the elements of \(D\) such that for every \(D \in \mathcal{D}\) where \(D = \wedge_{i=1}^{n} F_i\) and \(n \geq 1\), for \(i = 1, \ldots, n\), \(F_i\) is a conjunct of exactly one cluster of \(\mathcal{C}\).

**Definition 15** Let \(D\) be a finite, nonempty set of formulas, \(M\) a mating for \(D\), and \(\mathcal{C}\) a clustering of \(D\). With respect to \(D\), \(M\), and \(\mathcal{C}\), define \(\sim_0\) to be the binary relation on \(\mathcal{C}\) such that when \(C_1, C_2 \in \mathcal{C}\), \(C_1 \sim_0 C_2\) if \(C_1\) contains an atom subformula occurrence \(H\) and \(C_2\) contains an atom subformula occurrence \(K\) such that \(\{H, K\} \in M\). Let \(\sim\) be the reflexive, transitive closure of \(\sim_0\). If \(C \in \mathcal{C}\) we write \([C]_\sim\) to denote the equivalence class of \(\mathcal{C}\) which contains \(C\).
Example 5 We use brick diagrams to illustrate clusterings and the equivalence classes formed by the $\simeq$ relation. Both brick and block diagrams are similar to the two-dimensional format for displaying formulas found in [Andrews 81]. In brick and block diagrams, disjunctions are displayed vertically and conjunctions horizontally—the dual of Andrews’ format.

In Figure 4, $D$ is a set of formulas, and $C$ a clustering on $D$ as follows:

$$D = \{F_1, F_2 \land F_3 \land F_4 \land F_5, F_6 \land F_7, F_8 \land F_9\}$$
$$C = \{F_1, F_2, F_3 \land F_4, F_5, F_6, F_7, F_8 \land F_9\}$$

Each brick represents a cluster. Suppose a mating $M$ gives us the following equivalence classes for the $\simeq$ relation on the clusters of $C$:

$$[F_1]_{\simeq} = \{F_1, F_6\}$$
$$[F_2]_{\simeq} = \{F_2, F_3 \land F_4, F_8 \land F_9\}$$
$$[F_3]_{\simeq} = \{F_5\}$$
$$[F_7]_{\simeq} = \{F_7\}$$

They are shown in the figure by bold face boxes. Note that the equivalence classes of the $\approx$ relation are, in a sense, “composed” of one or more equivalence classes of the $\simeq$ relation. In this example, the four classes of the $\simeq$ relation form two for the $\approx$ relation as follows:

$$[F_1]_{\approx} = \{F_1, F_6 \land F_7\}$$
$$[F_8 \land F_9]_{\approx} = \{F_2 \land F_3 \land F_4 \land F_5, F_8 \land F_9\}$$
2.2.2 Transforming ET-Proofs to LK+ Proofs

From this new relation, it would seem that we could strengthen Proposition 5 to say that if $\mathcal{M}$ is a cs-mating of $\mathcal{D}$, then $\mathcal{M}$ spans one of the equivalence classes of the $\simeq$ relation on $\mathcal{C}$, which is often a "smaller" set than that obtained by the equivalence classes of the $\approx$ relation on $\mathcal{D}$. The following two definitions and lemma are needed to get this result.

Definition 16 Let $\mathcal{D}$ be a set of formulas, and $\mathcal{C}$ a clustering of $\mathcal{D}$. Let $P \subseteq \mathcal{C}$, and $D \in \mathcal{D}$. With respect to $P$ and $D$ we define the formula $C_P^D$.

$$C_P^D = \bigwedge \{ C \in P | C \text{ is a cluster of } D \}.$$ 

With respect to $P$, we define the set of formulas $\mathcal{D}_P$.

$$\mathcal{D}_P = \{ C_P^D | D \in \mathcal{D} \}.$$ 

Definition 17 Let $\mathcal{D}$ be a set of formulas, and $\mathcal{C}$ a clustering of $\mathcal{D}$. Let $P \subseteq \mathcal{C}$. We define $\mathcal{D}_{P+}$ as follows.

$$\mathcal{D}_{P+} = \{ D \in \mathcal{D} | \text{there is a cluster } C \text{ of } D \text{ s.t. } C \subseteq P \}$$ 

$\mathcal{D}_{P+}$ is the set of all formulas that have subformulas in $P$.

Example 6 From Figure 4 let $P = [F_2]_\simeq$ and $D = F_2 \wedge F_3 \wedge F_4 \wedge F_5$.

$$C_P^D = F_2 \wedge F_3 \wedge F_4$$

$$\mathcal{D}_P = \{ F_2 \wedge F_3 \wedge F_4, F_8 \wedge F_9 \}$$

$$\mathcal{D}_{P+} = \{ F_2 \wedge F_3 \wedge F_4, F_8 \wedge F_9 \}$$

Lemma 1 Let $\mathcal{D}$ be a finite, nonempty set of formulas, $\mathcal{M}$ a mating for $\mathcal{D}$, and $\mathcal{C}$ a clustering of $\mathcal{D}$. Let $P_1, \ldots, P_n$ be the equivalence classes of the $\simeq$ relation on $\mathcal{C}$. Let $\varepsilon_i$ be a clause in $\lor \mathcal{D}_{P_i}$. There is a clause $\varepsilon$ of $\lor \mathcal{D}$ such that $\varepsilon \subseteq \varepsilon_1 \cup \ldots \cup \varepsilon_n$.

Proof: (See Figure 5 for an illustration of this proof.) We construct such a clause $\varepsilon$. Let $\mathcal{D} = \{ D_1, \ldots, D_m \}$. We want to construct $\varepsilon$ by taking one clause of each formula $D_j$, for $j = 1, \ldots, m$. For each $j$, $j = 1, \ldots, m$, choose $C_j$, a cluster of $D_j$. $[C]_\approx \in \mathcal{C}$, so $[C]_\approx = P_i$ for some $i$, $1 \leq i \leq n$. Then $\varepsilon_i$ must contain a subset, $\varepsilon'_j$, which is a clause of some conjunct in $C_{P_i}^{D_j}$ (see definition 16). Hence, $\varepsilon'_j$ is a clause of $D_j$. $\varepsilon'_j \subseteq \varepsilon_i$, so $\varepsilon'_j \subseteq \varepsilon_1 \cup \ldots \cup \varepsilon_n$. Since $\varepsilon'_j \subseteq \varepsilon_1 \cup \ldots \cup \varepsilon_n$ for $j = 1, \ldots, n$, $\varepsilon'_1 \cup \ldots \cup \varepsilon'_m \subseteq \varepsilon_1 \cup \ldots \cup \varepsilon_n$. Since $\varepsilon = \varepsilon'_1 \cup \ldots \cup \varepsilon'_m$ is a clause of $\lor \mathcal{D}$, we have our result.

In Figure 5, for $i = 1, \ldots, 4$, $\varepsilon_i$ is a clause of $\lor \mathcal{D}_{P_i}$, for $j = 1, \ldots, 5$, $\varepsilon'_j$ is a clause of $D_j$, $\varepsilon = \varepsilon'_1 \cup \varepsilon'_2 \cup \varepsilon'_3 \cup \varepsilon'_4 \cup \varepsilon'_5$ is a clause of $\lor \mathcal{D}$, and $\varepsilon \subseteq \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 \cup \varepsilon_4$. 
Proposition 6 Let $D$ be a finite, nonempty set of formulas, $M$ a cs-mating for $D$, and $C$ a clustering of $D$. Then there is an equivalence class $P$ of the $\simeq$ relation on $C$ such that $M$ spans $D_P$.

**Proof:** Assume that while $M$ is a cs-mating for $D$, for every equivalence class $P$ of the $\simeq$ relation, $M$ does not span $D_P$. Let $P_1, \ldots, P_n$ be the equivalence classes of the $\simeq$ relation on $C$. Hence, for each $i$, $i = 1, \ldots, n$, there is a clause $\varepsilon_i$ in $\forall D_{P_i}$ which does not contain a mated pair. Let $\varepsilon_0 = \varepsilon_1 \cup \ldots \cup \varepsilon_n$. Then by Lemma 1 there exists an $\varepsilon \subseteq \varepsilon_0$ which is a clause of $D$ and hence must have a mated pair $H$ and $K$, and these are such that there are distinct integers $i, j$ such that $1 \leq i, j \leq n$ and $H \in \varepsilon_i$ and $K \in \varepsilon_j$. This implies that there is some cluster $C' \in P_i$, and some cluster $C'' \in P_j$ such that $C' \simeq C''$. This contradicts the fact that $P_i$ and $P_j$ are distinct equivalence classes. Hence, there must be an equivalence class $P$ of the $\simeq$ relation on $C$ such that $M$ spans $D_P$.

Figure 6 illustrates this proof. The contradiction arises from the fact that $P_1$ and $P_4$ are distinct equivalence classes, yet if $\{H, K\}$ is a mated pair then it must be the case that $P_1 = P_4$.

It would appear that we can use this proposition to create a “smarter” thin* function that would be able to remove more formulas than was possible using Proposition 5 because the equivalence classes of $\simeq$ on $C$ are “smaller” than those of $\simeq$ on $D$. But, this is not the case. Suppose we have a set of formulas corresponding to some sequent where $P$ is an equivalence class such that $D_P$ is spanned by the mating. Recall that each block roughly corresponds to one formula in a sequent. Since one block may contain more than one brick (one formula may contain more than one cluster), there may be a formula that has a brick in $P$, and another brick that is not in $P$. We’d have to break the formula in some way in order to remove everything except the chosen equivalence class $P$. The thin* inference rule cannot do this. Instead we turn to the following related proposition which does provide us...
2.2.2 Transforming ET-Proofs to LK+ Proofs

with a method for creating a "smarter" thin* function.

**Proposition 7** Let \( \mathcal{D} \) be a finite, nonempty set of formulas, \( \mathcal{M} \) a cs-mating for \( \mathcal{D} \), and \( \mathcal{C} \) a clustering of \( \mathcal{D} \) such that there is an equivalence class \( P \) of the \( \simeq \) relation on \( \mathcal{C} \) such that \( \mathcal{D}_P \) is not spanned by \( \mathcal{M} \). Then \( \mathcal{D} - \mathcal{D}_{p+} \) is spanned by \( \mathcal{M} \).

**Proof:** (See Figure 7 for an illustration of this proof.) Let \( \varepsilon' \) be a clause in \( \mathcal{V}\mathcal{D}_P \) that does not contain a mated pair. \( \varepsilon' \) is also a clause of \( \mathcal{V}\mathcal{D}_{P+} \). Assume \( \mathcal{D} - \mathcal{D}_{P+} \) is not spanned by \( \mathcal{M} \). Let \( \varepsilon'' \) be a clause of \( \mathcal{V}(\mathcal{D} - \mathcal{D}_{P+}) \) that does not contain a mated pair. \( \varepsilon = \varepsilon' \cup \varepsilon'' \) is a clause of \( \mathcal{V}\mathcal{D} \). Thus there is a mated pair \( H \) and \( K \), such that \( H \in \varepsilon' \) and \( K \in \varepsilon'' \). This implies that there is some cluster \( C' \in P \), and some cluster \( C'' \notin P \) such that \( C' \simeq C'' \), a contradiction. Hence, \( \mathcal{M} \) spans \( \mathcal{D} - \mathcal{D}_{p+} \).

**Corollary 8** Let \( \mathcal{D} \) be a finite, nonempty set of formulas, \( \mathcal{M} \) a cs-mating for \( \mathcal{D} \), and \( \mathcal{C} \) a clustering of \( \mathcal{D} \). Let \( P_1, \ldots, P_n \) be \( n \) equivalence classes of the \( \simeq \) relation on \( \mathcal{C} \) such that for \( i = 1, \ldots, n \), \( \mathcal{D}_{P_i} \) is not spanned by \( \mathcal{M} \). Then \( \mathcal{D} - \bigcup_{i=1}^n \mathcal{D}_{P_i+} \) is spanned by \( \mathcal{M} \).
2.2.2 Transforming ET-Proofs to LK+ Proofs

**Proof:** This follows from Proposition 7 and a simple induction on \( n \), the number of equivalence classes that are not spanned by \( \mathcal{M} \).

This corollary gives us a more thorough algorithm for removing formulas from a sequent when applying LK+ rules such as \textit{thin} than the one provided by Proposition 5. In this new algorithm, we simply remove all formulas that have subformulas in an equivalence class \( P \) where \( \mathcal{D}_{P+} \) is not spanned by the mating. Notice that this may often be more expensive than the algorithm provided by Proposition 5. In that algorithm, we only need to find one equivalence class that is spanned by the mating. In the new algorithm, we have to find all equivalence classes that are not spanned by the mating.

**Transformation Functions That Use Matings** From the above definitions and theorems, we construct the functions for the LK+ rules that use such information. First, we present two algorithms for the \textit{thin} rule. The second may find more formulas that can be removed than the first but at a much greater computational expense. The first is often sufficient, so we include it also. These algorithms involve extracting the deep formula of an expansion tree and converting it to negation normal form (i.e. \( \text{nff}(D_p(Q)) \) for some expansion tree \( Q \)). For clarity, we adopt the notation \( D_p^N(Q) \) to represent formulas in this normal form.

**thin**

1. Construct a set of formulas \( \mathcal{D} \) from the input sequent \( \Gamma \rightarrow \Delta; \mathcal{M} \) as follows:

\[
\mathcal{D} = \{ F | F \text{ is a disjunct of } D_p^N(\neg Q), Q \in \Gamma \} \\
\cup \{ F | F \text{ is a disjunct of } D_p^N(Q), Q \in \Delta \}
\]

2. Using the mating, \( \mathcal{M} \), construct the equivalence classes of the \( \approx \) relation on \( \mathcal{D} \).

3. Choose an equivalence class \( P \) that is spanned by \( \mathcal{M} \). (Note that this involves constructing the set of clauses for each equivalence class that is tested.)

4. Construct a new sequent \( \Gamma' \rightarrow \Delta' \) where \( \Gamma' \subseteq \Gamma \) and \( \Delta' \subseteq \Delta \), by retaining all formulas that have common atom occurrences with the formulas in \( P \). Construct the expansion trees for this sequent by retaining the expansion trees of the retained formulas. For quantified formulas, retain only those subtrees that have common atom occurrences with the formulas in \( P \).

5. If \( \Gamma' \neq \Gamma \) or \( \Delta' \neq \Delta \) then build the partial proof as follows.

\[
\frac{\Gamma' \rightarrow \Delta'}{\Gamma \rightarrow \Delta} \text{ thin}^*
\]

Return the generalized sequent \( \Gamma' \rightarrow \Delta' \) constructed in 4.
2.2.2 Transforming ET-Proofs to LK+ Proofs

\textbf{thin**}

1. Construct a set of formulas \(\mathcal{D}\) from the input sequent as in step 1 of the thin* function.
2. Construct the most refined clustering \(\mathcal{C}\) of \(\mathcal{D}\), the clustering in which every conjunct of every formula in \(\mathcal{D}\) is a cluster by itself.
3. Using the mating, determine the equivalence classes of the \(\simeq\) relation on \(\mathcal{C}\).
4. For each equivalence class \(P\), construct the set of clauses of \(\mathcal{D}_P\).
5. Let \(P_1, ..., P_n\) be the equivalence classes such that for \(i = 1, ..., n\), \(\mathcal{D}_{P_i}\) is not spanned by \(\mathcal{M}\), where \(\mathcal{M}\) is the mating for the input sequent \(\Gamma \rightarrow \Delta\).
6. Construct a new set of formulas \(\mathcal{D}' = \mathcal{D} - \bigcup_{i=1}^{n} \mathcal{D}_{P_i}\). We know from Corollary 8 that \(\mathcal{D}'\) is spanned by \(\mathcal{M}\).
7. Determine the equivalence classes of the \(\approx\) relation on \(\mathcal{D}'\).
8. Choose an equivalence class and construct the new sequent as in steps 2–4 in thin*.

\textbf{implies-1*}

If the generalized sequent is of the form \(\Gamma, A \supset C, \Delta \rightarrow \Theta; \mathcal{M}\) then

1. Construct a set of formulas \(\mathcal{D}\) from the input sequent as in step 1 of the thin* function. Note that \(\mathcal{D}\) will contain \(Dp^N(\neg(A \supset C))\) which is equivalent to \(Dp^N(A) \land Dp^N(\neg C)\).
2. Construct a clustering \(\mathcal{C}\) such that \(Dp^N(A)\) is a cluster, \(Dp^N(\neg C)\) is a cluster, and every conjunct of every formula in \(\mathcal{D} - \{Dp^N(\neg(A \supset C))\}\) is a cluster by itself.
3. Using the mating, determine the equivalence classes \(P_1 := [Dp^N(A)]_{\approx}\) and \(P_2 := [Dp^N(\neg C)]_{\approx}\).
4. Construct the sets of clauses for \(\mathcal{D}_{P_1}\) and \(\mathcal{D}_{P_2}\).
5. If \(\mathcal{D}_{P_1}\) and \(\mathcal{D}_{P_2}\) are both spanned by \(\mathcal{M}\) then construct the premises of the rule,

\[
\Gamma', \beta' \rightarrow A, \Theta'
\]
\[
\Delta', \beta'' \rightarrow \Theta''
\]

where \(\Gamma', \Gamma'' \subseteq \Gamma\), \(\Delta', \Delta'' \subseteq \Delta\), and \(\Theta', \Theta'' \subseteq \Theta\) such that \(\Gamma', \Delta', \Theta'\) contain all formulas that have common atom occurrences with the clusters in \(P_1\), and \(\Gamma'', \Delta'', \Theta''\) contain all formulas that have common atom occurrences with the clusters in \(P_2\). Construct the expansion trees for these sequents by retaining the expansion trees of the formulas in the constructed premises. For quantified formulas, retain only those subtrees that have common atom occurrences with the clusters in the appropriate equivalence class. Build the partial proof as follows.
2.2.2 Transforming ET-Proofs to LK+ Proofs

\[
\frac{\Gamma', \Delta' \rightarrow A, \Theta' \quad C, \Gamma'', \Delta'' \rightarrow \Theta''}{\Gamma, A \supset C, \Delta \rightarrow \Theta \quad \therefore \text{-L}^*}
\]

Return the generalized sequents constructed above representing the premises of the rule.

As stated in step 1, the formula \( A \supset C \) on the left of the arrow (the formula to which we are applying the rule) becomes \( Dp^N(\neg(A \supset C)) \) or equivalently \( Dp^N(A) \land Dp^N(\neg C) \) in the set of formulas \( \mathcal{D} \). Then, in step 2 we construct the clustering such that \( Dp^N(A), Dp^N(\neg C) \in \mathcal{C} \). If either \([Dp^N(A)]_\approx \) or \([Dp^N(\neg C)]_\approx \) is not spanned by \( \mathcal{M} \) (step 5), we know by Proposition 7 that the formula \( A \supset C \) is not needed to complete the proof and should be thinned out. Thus we do not want to apply this rule.

**positive**

If the generalized sequent is of the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta; \mathcal{M} \) then

1. Perform steps 1–4 as in the \text{implies-l}\* function.
2. If \( \mathcal{D}_{P_1} \) and \( \mathcal{D}_{P_2} \) are both spanned by \( \mathcal{M} \) and for every \( C' \in \mathcal{C} \) such that \( C' \) is a cluster of one of the formulas in \( \Theta \), \([Dp^N(A)]_\approx \neq [C']_\approx \) then build the partial proof as follows.

\[
\frac{\Gamma, \Delta \rightarrow A \quad C, \Gamma, \Delta \rightarrow \Theta}{\Gamma, A \supset C, \Delta \rightarrow \Theta \quad \text{positive}}
\]

Return the generalized sequents representing the premises of the rule.

\[
\begin{align*}
\Gamma, \Delta & \rightarrow A \\
C, \Gamma, \Delta & \rightarrow \Theta
\end{align*}
\]

**contrapos**

If the generalized sequent is of the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta; \mathcal{M} \) then

1. Perform steps 1–4 as in the \text{implies-l}\* function.
2. If \( \mathcal{D}_{P_1} \) and \( \mathcal{D}_{P_2} \) are both spanned by \( \mathcal{M} \) and for every \( C' \in \mathcal{C} \) such that \( C' \) is a cluster of one of the formulas in \( \Theta \), \([Dp^N(\neg C)]_\approx \neq [C']_\approx \) then build the partial proof as follows.

\[
\frac{\neg A, \Gamma, \Delta \rightarrow \Theta \quad C, \Gamma, \Delta \rightarrow \Theta}{\Gamma, A \supset C, \Delta \rightarrow \Theta \quad \text{contrapos}}
\]
2.2.2 Transforming ET-Proofs to LK+ Proofs

Return the generalized sequents representing the premises of the rule.

\[ \neg A, \Gamma, \Delta \rightarrow \Theta \]
\[ C, \Gamma, \Delta \rightarrow \]

The positive* and contrapos* functions require some explanation. The clustering \( C \) constructed in step 2 is roughly illustrated in Figure 8. If for every \( C' \in C \)

\[
\begin{array}{|c|c|}
\hline
\neg \Gamma & \\
\hline
\neg \Delta & \\
\hline
A & \neg C \\
\hline
\Theta & \\
\hline
\end{array}
\]

Figure 8: The clustering for the positive* and contrapos* functions

that is a cluster of one of the formulas in \( \Theta \), \([C']_\approx \neq [Dp^N(A)]_\approx\), we do not need any of the formulas in \( \Theta \) in the premise that contains \( A \). Thus we prove \( A \) from the hypothesis sets \( \Gamma \) and \( \Delta \). We then add \( C \) to the hypothesis list and try to prove one of the formulas in \( \Theta \) for the other premise \( C, \Gamma, \Delta \rightarrow \Theta \). This is similar to an application of modus ponens.

The contrapos rule applies when we have just the opposite i.e. when for every \( C' \in C \) that is a cluster of one of the formulas in \( \Theta \), \([C']_\approx \neq [Dp^N(\neg C)]_\approx\). This corresponds to an application of modus ponens on the contrapositive form \( \neg C \supset \neg A \) of \( A \supset C \). We prove \( \neg C \) by showing that assuming \( C \) gives a contradiction. This corresponds to the premise \( C, \Gamma, \Delta \rightarrow . \) The other premise is \( \neg A, \Gamma, \Delta \rightarrow \Theta \) where we assume \( \neg A \) to prove one of the formulas in \( \Theta \).

backchain*

If the generalized sequent is of the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta, C, \Delta; M \) then

1. Construct a set of formulas \( D \) from the input sequent as in step 1 of the thin* function.
2. Construct a clustering \( C \) such that \( Dp^N(A) \) is a cluster, \( Dp^N(\neg C) \) is a cluster, \( Dp^N(C) \) is a cluster, and every conjunct of every formula in \( D - \{Dp^N(\neg (A \supset C)), Dp^N(C)\} \) is a cluster by itself. (\( Dp^N(C) \) corresponds to the formula \( C \) occurring on the right of the sequent arrow.)
3. Using the mating, determine the equivalence classes \( P_1 := [Dp^N(A)]_\approx \) and \( P_2 := [Dp^N(\neg C)]_\approx \).
4. Construct the sets of clauses for \( D_{P_1} \) and \( D_{P_2} \).
5. If $\mathcal{D}_P$ and $\mathcal{D}_{P_1}$ are both spanned by $\mathcal{M}$ and for every $C' \in C$ such that $C'$ is a cluster of one of the formulas in $\Gamma, \Delta, \Theta, \Lambda$, or $A$, $[D_{pN}(C)_{\sim}] \neq [C'_{\sim}]$ then build the partial proof as follows.

\[
\begin{array}{c}
\Gamma, \Delta \rightarrow A, \Theta, \Lambda \\
\Gamma, A \rightarrow C, \Delta \rightarrow \Theta, C, \Lambda
\end{array}
\]

**backchain**

Return the generalized sequents representing the premises of the rule.

\[
\begin{array}{c}
\Gamma, \Delta \rightarrow A, \Theta, \Lambda \\
C \rightarrow C
\end{array}
\]

**forwardchain**

If the generalized sequent is of the form $\Gamma, A \supset C, \Delta, A, \Theta \rightarrow \Lambda; \mathcal{M}$ or $\Gamma, A, \Delta, A \supset C, \Theta \rightarrow \Lambda; \mathcal{M}$ then

1. Construct a set of formulas $\mathcal{D}$ from the input sequent as in step 1 of the thin* function.
2. Construct a clustering $\mathcal{C}$ such that $D_{pN}(A)$ is a cluster, $D_{pN}(\neg C)$ is a cluster, $D_{pN}(\neg A)$ is a cluster, and every conjunct of every formula in $\mathcal{D} - \{D_{pN}(\neg(A \supset C)), D_{pN}(\neg A)\}$ is a cluster by itself. ($D_{pN}(\neg A)$ corresponds to the formula $A$ occurring on the left of the sequent arrow.)
3. Using the mating, determine the equivalence classes $P_1 := [D_{pN}(A)]_{\sim}$ and $P_2 := [D_{pN}(\neg C)]_{\sim}$.
4. Construct the sets of clauses for $\mathcal{D}_P$ and $\mathcal{D}_{P_1}$.
5. If $\mathcal{D}_P$ and $\mathcal{D}_{P_1}$ are both spanned by $\mathcal{M}$ and for every $C' \in C$ such that $C'$ is a cluster of one of the formulas in $\Gamma, \Delta, \Theta, \Lambda$, or $C$, $[D_{pN}(\neg A)]_{\sim} \neq [C'_{\sim}]$ then build the partial proof corresponding to the appropriate form of the LK+ forwardchain rule.

Return the generalized sequents representing the premises of the rule.

\[
\begin{array}{c}
A \rightarrow A \\
C, \Gamma, \Delta, \Theta \rightarrow \Lambda
\end{array}
\]

The **backchain** and **forwardchain** functions also require some explanation. We want to apply the backchain rule when the formula $C$ occurs somewhere on the right side of the sequent as long as $C$ contains no atom occurrences that are mated with atom occurrences in any of $\Gamma, \Delta, \Theta, \Lambda$, or $A$. Then we can backchain on $C$ and do not have to include $C$ in the branch of the tree that attempts to prove $A$. The **forwardchain** function is similar with respect to the formula $A$ occurring on the left of the sequent.
2.2.2 Transforming ET-Proofs to LK+ Proofs

We could combine implies-1*, positive*, contrapos*, backchain*, and forwardchain* into one transformation function and apply one of them depending on how the \( \simeq \) relation partitions the set of formulas, but we separate them for flexibility in building natural proofs. Since positive*, contrapos*, backchain*, and forwardchain* are fairly natural, it is more desirable to apply one of them than implies-1* if possible. If all five were combined in one function, when none of positive*, contrapos*, backchain*, or forwardchain* are applicable we'd be restricted to applying implies-1* by default. Since implies-1* is less natural, we may want to delay its application until other more natural rules are attempted.

The proof of the correctness of an algorithm built from these functions involves several things. First notice that each function performs one step in building an LK+ proof tree by applying a valid inference rule. Second, it must be the case that for each function, if an ET-proof is input, the generalized sequents that the function returns are also ET-proofs. This is true since (1) the \( \prec \) relation of the resulting generalized sequents is a restriction of the \( \prec \) relation of the input sequents, so remains acyclic, (2) it is straightforward to show that the deep formula of a resulting sequent remains a tautology (in the case of \( \supset \)-L* and thin* exactly those formulas that are not needed to preserve the tautology of the deep form of the output sequent(s) are removed), and (3) the check that the substitution term is admissible before applying the \( \exists \)-R and \( \forall \)-L rules guarantees that the expansion trees remain sound. Finally, to guarantee correctness of an algorithm, we must show that it terminates, and when it terminates all the leaves of the LK+ proof tree are axioms. This is specific to the algorithm. One example is to repeatedly attempt each of the and-r, or-1, and-1, or-r, implies-r, implies-1, neg-r, neg-1, forall-r, exists-1, contract-r, contract-1, forall-1, and exists-r functions until no more can be done, and then apply thin* to each of the branches to get axioms. By Corollary 3 this list of rules forms a complete subset of LK+.

To show that it terminates, we can use an inductive argument to show that the number of connectives in the sequent must decrease as rules are applied. To show that all the leaves are axioms when it terminates, we can show that at least one rule is applicable to any sequent that does not contain only atoms. The propositional rules, structural rules, and forall-r, and exists-1 are always applicable when a formula of the correct form exists in the sequent. For forall-1 and exists-r, it must always be the case that at least one of the terms in a generalized sequent that is an ET-proof is admissible. This proof can be found in [Miller 84]. Thus if no more rules from this list can be applied, then the sequent contains only atoms. Since every sequent obtained during the proof process is an ET-proof, a sequent containing only atoms is either already an axiom, or can be thinned to an axiom by thin*. Thus, this simple algorithm is correct. For more on correctness, see [Miller 83]. This algorithm does not make much use of the additional rules. These
allow the construction of more natural proofs. For example we could use thin* more strategically during the proof process to remove unnecessary information and create more readable proofs. More examples of the use of these rules will be described later. The proof of termination of an algorithm using the additional rules becomes more complex. For example, in an algorithm that uses both the indirect* and neg-1 functions, it must be shown that it is not possible to repeatedly apply these rules one after the other causing an infinite loop (and building an infinite tree).

2.2.3 Transforming LK+ Proofs to ET-Proofs

The algorithm to transform an LK+ proof to an ET-proof is presented in this section. It is based on the algorithm found in [Pfenning 84]. It makes use of a merge algorithm for merging two expansion trees. The version presented here is based on the merge algorithm found in [Pfenning 84] and is a slightly improved version of that found in [Miller 83]. Like the converse transformation, this algorithm is built from a series of functions, again one for each LK+ inference rule. We start with the leaves of the LK+ proof tree (the axioms) and build the ET-proof from the top down. For each function, we will assume that one inference in the LK+ proof tree and the ET-proofs corresponding to the premises of the rule are passed as arguments to the function. The function then constructs the ET-proof corresponding to the conclusion. In this transformation algorithm, in addition to obtaining an ET-proof, we also obtain a cs-mating. We use a global variable, M, for the mating which is initialized to be empty. The axiom function will add pairs to the mating, and the merge algorithm will update the mating by performing the necessary substitutions whenever two atoms are merged. We also start with another global variable, S, which is an initially empty list used to keep track of substitutions done on expansion trees during the merge algorithm. We adopt the convention that whenever Γ is a set of formulas, Γ', and Γ'' are sets of expansion trees or dual expansion trees for the formulas in Γ.

axiom

For input A → A, return the ET-proof A' → A''. Here A' and A'' are both the same as A, but we make a distinction to distinguish between occurrences of the same atom. Add the pair \{A', A''\} to the mating, M.

and_r

\[
\frac{\Gamma \rightarrow \Delta, A, \Theta}{\Gamma \rightarrow \Delta, A \land C, \Theta} \quad \wedge R
\]

For ET-proofs \( \Gamma' \rightarrow \Delta', A', \Theta' \) and \( \Gamma'' \rightarrow \Delta'', C', \Theta'' \) of the premises, con-
2.2.3 Transforming LK+ Proofs to ET-Proofs

struct the ET-proof

\[ \text{merge}(\Gamma', \Gamma'') \rightarrow \text{merge}(\Delta', \Delta''), A' \land C', \text{merge}(\Theta', \Theta''). \]

The functions \( \text{or}_r \) and \( \text{implies}_r \) are similar since they merge all formulas except those that the LK+ rule was applied to. The functions \( \text{implies}_r^*, \text{positive}, \text{contapos}, \text{backchain}, \) and \( \text{forwardchain} \) are also similar except that there may be formulas that are not common to both premises. Expansion trees for all formulas in both premises are included in the ET-proof representing the conclusion, but those that are common to both are merged.

**or.r**

\[
\Gamma \rightarrow \Delta, A, C, \Theta \\
\Gamma \rightarrow \Delta, A \lor C, \Theta \quad \lor-R
\]

For ET-proof \( \Gamma' \rightarrow \Delta', A', C', \Theta' \) of the premise, construct the ET-proof \( \Gamma' \rightarrow \Delta', A' \lor C', \Theta'. \)

The functions \( \text{and}_r, \text{implies}_r, \text{neg}_r, \text{neg}_r, \text{neg}_1, \text{neg}_2, \text{neg}_2 \) are similar.

**contract.r**

\[
\Gamma \rightarrow \Delta, A, A, \Theta \\
\Gamma \rightarrow \Delta, A, \Theta \quad \text{contract-R}
\]

For ET-proof \( \Gamma' \rightarrow \Delta', A', A'', \Theta' \) construct the ET-proof \( \Gamma' \rightarrow \Delta', \text{merge}(A', A''), \Theta'. \)

The \( \text{contract}_r \) function is defined similarly.

**exists.r**

\[
\Gamma \rightarrow \Delta, [x/t]P, \Theta \\
\Gamma \rightarrow \Delta, \exists x P, \Theta \quad \exists-R
\]

For ET-proof \( \Gamma' \rightarrow \Delta', Q, \Theta' \) where \( Q \) is an expansion tree for \([x/t]P\), construct the ET-proof \( \Gamma' \rightarrow \Delta', (\exists x P, (t, Q)), \Theta'. \)

The functions for \( \text{forall}_r, \text{exists}_r, \) and \( \text{forall}_r \) are defined similarly.

**pushneg.r**

\[
\Gamma \rightarrow \Delta, \exists x \neg P, \Theta \quad \rightarrow \forall-R \\
\Gamma \rightarrow \Delta, \exists x P, \Theta \quad \rightarrow \exists-R
\]
For an ET-proof of the form $\Gamma' \rightarrow \Delta', (\exists x \neg P, (t_1, \neg Q_1), \ldots, (t_n, \neg Q_n)), \Theta'$ where, for $i = 1, \ldots, n$, $\neg Q_i$ is an expansion tree for $[x/t_i] \neg P$, construct the ET-proof $\Gamma' \rightarrow \Delta', (\forall x P, (t_1, Q_1), \ldots, (t_n, Q_n)), \Theta'$.

Otherwise, for an ET-proof of the form $\Gamma' \rightarrow \Delta', (\forall x \neg P, (y, \neg Q)), \Theta'$ where $\neg Q$ is an expansion tree for $[x/y] \neg P$, construct the ET-proof $\Gamma' \rightarrow \Delta', (\exists x P, (y, Q)), \Theta'$.

The pushneg-1 function is defined similarly.

\[
\frac{\lnot[\Delta], \Gamma \rightarrow \Delta}{\Gamma \rightarrow \lnot[\Delta], \Gamma} \quad \text{indirect}
\]

For ET-proof $\lnot[\Delta'], \Gamma' \rightarrow \Delta'$ construct the ET-proof $\Gamma' \rightarrow \Delta'$.

\[
\frac{\Gamma, \Delta \rightarrow \Theta}{\Gamma, A, \Delta \rightarrow \Theta} \quad \text{thin-}1
\]

For ET-proof $\Gamma', \Delta' \rightarrow \Theta'$ construct the ET-proof $\Gamma', A, \Delta' \rightarrow \Theta'$.

Note that in the ET-proof for the conclusion, the expansion tree for $A$ is the formula $A$ itself. The thin-x and thin* functions are defined similarly.

**The Merge Algorithm**  In order to merge two trees $Q$ and $Q'$, it must be the case that $Sh(Q) = Sh(Q')$ (they are both expansion trees or dual expansion trees for the same formula). Note that when $Q$ and $Q'$ are expansion trees (dual expansion trees) for $A$ then merge($Q, Q'$) is an expansion tree (dual expansion tree) for $A$. The merge function is defined recursively as follows.

1. If $Q$ is a one node tree then so is $Q'$, merge($Q, Q'$) := $Q$.

   When two atoms are merged, all occurrences of the atom $Q'$ must be replaced with the atom $Q$ in the mating, $M$.

2. merge($\neg Q, \neg Q'$) := $\neg$merge($Q, Q'$).

3. merge($Q_1 \land Q_2, Q'_1 \land Q'_2$) := merge($Q_1, Q'_1$) $\land$ merge($Q_2, Q'_2$).

4. merge($Q_1 \lor Q_2, Q'_1 \lor Q'_2$) := merge($Q_1, Q'_1$) $\lor$ merge($Q_2, Q'_2$).

5. merge($Q_1 \supset Q_2, Q'_1 \supset Q'_2$) := merge($Q_1, Q'_1$) $\supset$ merge($Q_2, Q'_2$).
6. If $Z_1 = (\forall x P, (y, Q))$ and $Z_2 = (\forall x P, (y', Q'))$ are expansion trees for $\forall x P$ then $\text{merge}(Z_1, Z_2) := (\forall x P, (y, \text{merge}(Q, [y'/y]Q'))).$ Here $[y'/y]Q'$ is the result of replacing every occurrence of $y'$ in the expansion tree $Q'$ by $y$.

If $Z_1 = (\exists x P, (y, Q))$ and $Z_2 = (\exists x P, (y', Q'))$ are dual expansion trees for $\exists x P$ then $\text{merge}(Z_1, Z_2) := (\exists x P, (y, \text{merge}(Q, [y'/y]Q'))).$

Add $[y'/y]$ to the end of the substitution list, $S$.

7. If $Z_1 = (\exists x P, (t_1, Q_1), \ldots, (t_n, Q_n))$ and $Z_2 = (\exists x P, (t'_1, Q'_1), \ldots, (t'_m, Q'_m))$ are expansion trees for $\exists x P$, then

$$\text{merge}(Z_1, Z_2) := (\exists x P, (\hat{t}_1, \hat{Q}_1), \ldots, (\hat{t}_k, \hat{Q}_k),$$

$$(\hat{t}_{k+1}, \text{merge}(Q_{k+1}, Q'_{k+1})), \ldots, (\hat{t}_l, \text{merge}(Q_l, Q'_l))).$$

Here, for $i = 1, \ldots, k$ ($0 \leq k \leq m + n$), $\hat{t}_i$ is an expansion term that occurs in only one of $t_1, \ldots, t_n$ or $t'_1, \ldots, t'_m$, and $\hat{Q}_i$ is its corresponding expansion tree. For $i = k + 1, \ldots, l$ ($k \leq l \leq (m + n + k)/2$), $\hat{t}_i$ appears in both, $\hat{Q}_i$ is its corresponding expansion tree from $Q_1, \ldots, Q_n$, and $\hat{Q}'_i$ is its corresponding expansion tree from $Q'_1, \ldots, Q'_m$.

If $Z_1 = (\forall x P, (t_1, Q_1), \ldots, (t_n, Q_n))$ and $Z_2 = (\forall x P, (t'_1, Q'_1), \ldots, (t'_m, Q'_m))$ are dual expansion trees for $\forall x P$, then

$$\text{merge}(Z_1, Z_2) := (\forall x P, (\hat{t}_1, \hat{Q}_1), \ldots, (\hat{t}_k, \hat{Q}_k),$$

$$(\hat{t}_{k+1}, \text{merge}(Q_{k+1}, Q'_{k+1})), \ldots, (\hat{t}_l, \text{merge}(Q_l, Q'_l))).$$

After the transformation is completed, all substitutions in $S$ must be applied to the expansion tree as a whole (in the order they appear in $S$) to get the final result. The reader is referred to [Pfenning 84] and [Miller 83] for correctness of the algorithm.
3 The $\chi$ System

In this chapter we describe the comprehensive proof construction system (the $\chi$ system) that is shown in Figure 1. Section 3.1 explains the programming language approach which is a part of the proof construction component, and was developed to facilitate user interaction in the proof process, and provide a means for integrating automatically generated proofs into the natural deduction environment.

The next section, Section 3.2, describes the design of the interactive theorem prover within the natural deduction environment. As mentioned, the user has access to an automatic theorem prover at any time, and Section 3.3 illustrates how this theorem prover is integrated into the proof construction component.

The second component of the $\chi$ system is the proof revision component. It is responsible for taking a completed LK+ proof and attempting to make it more "readable" or "natural." Section 3.4 describes this component and discusses some revisions that are feasible as a result of the design of this component.

The third component will be the explanation facility which takes a revised LK+ proof and produces a natural language explanation. Though this component has not yet been developed, it is important to include it. Producing explanations from proofs has been a goal that has influenced the overall design of the system.

The $\chi$ system is written in Common Lisp, and the current version is running under VMS and on Symbolics Lisp Machines.

3.1 The Programming Language Approach

In this section we describe many facets of the programming language developed to enhance the theorem proving environment: the data structures for proofs, the primitive tactics, the tacticals, the typing mechanism, and the ways in which our language extends the ideas found in LCF. The data structures are explicit representations of proofs which we have designed to facilitate manipulation of proofs as objects. The tactics are the primitives of the programming language. Each one corresponds to an LK+ inference rule. The tacticals are the control structures which allow us to write proof procedures (i.e. compound tactics) to apply some combination of several primitive tactics. As mentioned, the typing mechanism is based on the notion of formulas-as-types found in [Howard 80]. In $\chi$, the type of an LK+ proof is specified by a generalized sequent which may often be an ET-proof. We also discuss the typing of the primitive tactics. In addition to adopting the language of tactics and tacticals of LCF, we also extend some ideas in LCF. These extensions are described in the last part of this section.
3.1.1 The Data Structures

We want to represent both LK+ proofs and ET-proofs manageably and efficiently in order to use them as computational objects. The fact that expansion trees can be represented very compactly is one of their main advantages, and is one of the reasons we immediately transform an automatically generated resolution refutation to an ET-proof. The expansion trees defined in Section 2.1.3 are suitable for computations as they stand and are very close to the actual representation used in $\chi$.

LK+ proof trees are represented as term structures. These term structures are recursive data structures. A proof tree of height 1 is a sequent of the form $\Delta \rightarrow A$. It is represented as a term with one argument $\text{axiom}(A)$. Each inference rule is represented as a function symbol of 1 or 2 arguments, where the arguments are proof trees for the premise(s) of the rule. For example, if $T_1$ and $T_2$ are proof trees for $\Gamma \rightarrow \Delta, A, \Theta$ and $\Gamma \rightarrow \Delta, C, \Theta$ respectively, then and-$r(T'_1, T'_2)$ is the term representing the proof tree

$$\frac{T_1}{\frac{T_2}{\Gamma \rightarrow \Delta, A \land C, \Theta}} \land-R$$

where $T'_1$ and $T'_2$ are the terms representing the proofs of $T_1$ and $T_2$, respectively.

These terms must store enough information to reconstruct the complete LK+ proof tree. Each inference rule actually requires more information than the subproofs in order to apply the rule and put these subproofs together into a larger proof. For example, every rule needs to know the positions in the premises and conclusion of the formulas to which the rule must be applied. Such extra information is added as additional arguments to the functions. For readability we leave these arguments out in the examples in this and the following sections, but discuss it in more detail when we discuss the individual inference rules and the extra information specific to each (Section 3.1.2).

Example 7 The following is an example of an LK+ proof tree for

$$[p(a) \lor q(b)] \land \forall x [p(x) \supset q(x)] \supset \exists x q(x).$$

\begin{align*}
p(a) \rightarrow p(a) & \frac{q(a) \rightarrow q(a)}{\exists-R} \\
& \frac{q(a) \rightarrow \exists x q(x)}{\lor-L} \\
& \frac{p(a), p(a) \supset q(a) \rightarrow \exists x q(x)}{\forall-L} \\
& \frac{q(b) \rightarrow q(b)}{\exists-R} \\
& \frac{q(b) \rightarrow \exists x q(x)}{\forall-L} \\
& \frac{q(b), \forall x [p(x) \supset q(x)] \rightarrow \exists x q(x)}{\land-L} \\
& \frac{p(a) \lor q(b), \forall x [p(x) \supset q(x)] \rightarrow \exists x q(x)}{\lor-R} \\
& \frac{[p(a) \lor q(b)] \land \forall x [p(x) \supset q(x)] \rightarrow \exists x q(x)}{\supset-R}
\end{align*}
Its corresponding term representation is:

\[
\text{implies}_r(\text{and}_l(\text{or}_l(\text{forall}_l(\text{implies}_l(axiom(p(a))), \exists_r(axiom(q(a)))), \text{thin}_l(\exists_r(axiom(q(b)))))))
\]

We also represent partial proofs as proof terms. We introduce free variables that serve as place holders for the incomplete branches or subproofs. These variables are abstracted with \(\lambda\)-bindings. Thus a partial proof is represented as a function from subproofs to a completed proof, and \(\lambda\)-conversion represents the operation of supplying this partial proof with its subproofs.

**Example 8** The following is a partial proof of the sequent in the previous example.

\[
\begin{align*}
p(a), \forall x [p(x) \supset q(x)] & \quad \rightarrow \quad \exists x q(x) \\
q(b) & \quad \rightarrow \quad \exists x q(x) \\
\text{thin-L}&&& \\
\text{v-L}&&& \\
\text{\&-L}&&& \\
\text{\&-R}&&& \\
\rightarrow [p(a) \lor q(b)] \land \forall x [p(x) \supset q(x)] & \quad \rightarrow \quad \exists x q(x)
\end{align*}
\]

The term representing this proof is:

\[
\lambda X \lambda Y. \text{implies}_r(\text{and}_l(\text{or}_l(X, \text{thin}_l(Y))))
\]

where \(X\) and \(Y\) are place holders for proofs of the sequents \(p(a), \forall x [p(x) \supset q(x)] \rightarrow \exists x q(x)\) and \(q(b) \rightarrow \exists x q(x)\) respectively.

This representation of partial proofs facilitates interactive theorem proving in several ways. It allows us to store incomplete proofs as objects. This is useful when a user does not complete a proof in one session and wishes to come back to it later. It also provides a mechanism for combining subproofs to obtain more refined proofs. This combining is necessary as inference rules are applied during the theorem proving process, and when a subproof is completed independently and must be integrated into the main proof.
3. and-l-tac (sequent)
   If sequent has the form \( \Gamma, A \land C, \Delta \rightarrow \Theta \) then
   return
   \( [\Gamma, A, C, \Delta \rightarrow \Theta], \lambda T. \text{and}\_l(T, I) \)
   else fail

4. or-r-tac (sequent)
   If sequent has the form \( \Gamma \rightarrow \Delta, A \lor C, \Theta \) then
   return
   \( [\Gamma \rightarrow \Delta, A, C, \Theta], \lambda T. \text{or}\_r(T, I) \)
   else fail

5. implies-l-tac (sequent)
   If sequent has the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta \) then
   return
   \( [\Gamma, \Delta \rightarrow A, \Theta ; C, \Gamma, \Delta \rightarrow \Theta], \lambda T_1 \lambda T_2. \text{implies}\_l(T_1, T_2, I) \)
   else fail

6. implies-r-tac (sequent)
   If sequent has the form \( \Gamma \rightarrow \Delta, A \supset C, \Theta \) then
   return
   \( [A, \Gamma \rightarrow C, \Delta, \Theta], \lambda T. \text{implies}\_r(T, I) \)
   else fail

7. neg-l-tac (sequent)
   If sequent has the form \( \Gamma, \neg A, \Delta \rightarrow \Theta \) then
   return
   \( [\Gamma, \Delta \rightarrow A, \Theta], \lambda T. \text{neg}\_l(T, I) \)
   else fail

8. neg-r-tac (sequent)
   If sequent has the form \( \Gamma \rightarrow \Delta, \neg A, \Theta \) then
   return
   \( [A, \Gamma \rightarrow \Delta, \Theta], \lambda T. \text{neg}\_r(T, I) \)
   else fail

9. exists-l-tac (sequent)
   If sequent has the form \( \Gamma, (\exists x P, (y, Q)), \Delta \rightarrow \Theta \) and
   \( y \) is not free in \( \Gamma, \Delta, \Theta \), or \( P \) then
   return
3.1.2 The Primitive Tactics

The primitives of the programming language are the simple tactics. A tactic takes a generalized sequent as input, and returns a list of generalized sequents and a $\lambda$-term. The majority of the primitive tactics correspond to an LK+ inference rule. The rule is applied to the input generalized sequent, and the list of generalized sequents that is returned contains the premises of the rule i.e. the sequents that still must be proven. The $\lambda$-term is the term representation (described in the previous section) of the partial proof created by applying a given inference rule. The subproofs of the premises of the rule are always arguments to the term representing that rule. Some rules require additional information in order to build larger proofs from the subproofs. We represent this by including a final argument $I$ to the $\lambda$-term returned by each tactic where $I$ contains all the necessary extra information. The specific contents of $I$ for each inference rule will be discussed following the presentation of the tactics.

In general, the input sequent to a tactic will either be a simple sequent or an ET-proof. When a user is interactively constructing a proof, it will be a simple sequent. It will be ET-proof when a resolution refutation was automatically generated and then transformed to an ET-proof, and is then being transformed by tactics to an LK+ proof. Note that when the sequent is an ET-proof, the tactics behave like the LK+ transformation functions described in Section 2.2.2. It is also possible to have a generalized sequent that is not an ET-proof but contains some “expansion tree information” such as a partial mating, or some substitution information. Our tactics will be general enough to handle this. In general, if the information necessary to apply the rule is present in the generalized sequent, it will be used by the tactic.

We now present all of the primitive tactics. Many require additional explanation which follows.

1. and-r-tac (sequent)
   If sequent has the form $\Gamma \rightarrow \Delta, A \wedge C, \Theta$ then
   return $[\Gamma \rightarrow \Delta, A, \Theta ; \Gamma \rightarrow \Delta, C, \Theta], \lambda T_1 \lambda T_2.\text{and}-r(T_1, T_2, I)$
   else fail

2. or-l-tac (sequent)
   If sequent has the form $\Gamma, A \vee C, \Delta \rightarrow \Theta$ then
   return $[\Gamma, A, \Delta \rightarrow \Theta ; \Gamma, C, \Delta \rightarrow \Theta], \lambda T_1 \lambda T_2.\text{or}-l(T_1, T_2, I)$
   else fail
3.1.2 The Primitive Tactics

\[ [\Gamma, Q, \Delta \rightarrow \Theta], \lambda T.\text{exists}_r(T, I) \]

else if sequent has the form \( \Gamma, \exists x P, \Delta \rightarrow \Theta \) then

let \( y \) be a new variable not free in \( \Gamma, \Delta, \Theta, \) or \( P \)

return

\[ [\Gamma, [x/y]P, \Delta \rightarrow \Theta], \lambda T.\text{exists}_r(T, I) \]

else fail

10. forall-r-tac (sequent)

If sequent has the form \( \Gamma \rightarrow \Delta, (\forall x P, (y, Q)), \Theta \) and

y is not free in \( \Gamma, \Delta, \Theta, \) or \( P \) then

return

\[ [\Gamma \rightarrow \Delta, Q, \Theta], \lambda T.\text{forall}_r(T, I) \]

else if sequent has the form \( \Gamma \rightarrow \Delta, \forall x P, \Theta \) then

let \( y \) be a new variable not free in \( \Gamma, \Delta, \Theta, \) or \( P \)

return

\[ [\Gamma \rightarrow \Delta, [x/y]P, \Theta], \lambda T.\text{forall}_r(T, I) \]

else fail

11. forall-l-tac (sequent)

If sequent has the form \( \Gamma, (\forall x P, (t, Q)), \Delta \rightarrow \Theta \) and

t is admissible then

return

\[ [\Gamma, Q, \Delta \rightarrow \Theta], \lambda T.\text{forall}_l(T, I) \]

else fail

12. forall-l-user-tac (sequent)

If sequent has the form \( \Gamma, \forall x P, \Delta \rightarrow \Theta \) then

obtain the substitution term \( t \) from the user

if \( t \) is admissible then

return

\[ [\Gamma, [x/t]P, \Delta \rightarrow \Theta], \lambda T.\text{forall}_l(T, I) \]

else fail

else fail

13. exists-r-tac (sequent)

If sequent has the form \( \Gamma \rightarrow \Delta, (\exists x P, (t, Q)), \Theta \) and

t is admissible then

return

\[ [\Gamma \rightarrow \Delta, Q, \Theta], \lambda T.\text{exists}_r(T, I) \]

else fail
14. **exists-r-user-tac (sequent)**
   If sequent has the form $\Gamma \rightarrow \Delta, \exists x P, \Theta$ then
   obtain the substitution term $t$ from the user
   if $t$ is admissible then
   return
   $[\Gamma \rightarrow \Delta, [x/t]P, \Theta], \lambda T.\text{exists}_r(T,I)$
   else fail
   else fail

15. **thin-l-tac (sequent)**
   Obtain the position of the formula to be removed from $\Gamma$ in
   the sequent $\Gamma \rightarrow \Delta$ from the user
   If there is a formula in the specified position then
   return
   $[\Gamma' \rightarrow \Delta], \lambda T.\text{thin}_l(T,I)$
   where $\Gamma'$ is $\Gamma$ with the specified formula removed
   else fail

16. **thin-r-tac (sequent)**
   Obtain the position of the formula to be removed from $\Delta$ in
   the sequent $\Gamma \rightarrow \Delta$ from the user
   If there is a formula in the specified position then
   return
   $[\Gamma \rightarrow \Delta'], \lambda T.\text{thin}_r(T,I)$
   where $\Delta'$ is $\Delta$ with the specified formula removed
   else fail

17. **contract-l-tac (sequent)**
   If sequent has the form $\Gamma,(\forall x P,(t_1,Q_1),\ldots,(t_n,Q_n)),\Delta \rightarrow \Theta$, and $n > 1$, and there is some $t_i$ that is admissible then
   return
   $[\Gamma,(\forall x P,(t_i,Q_i)),(\forall x P,(t_1,Q_1),\ldots,(t_{i-1},Q_{i-1}),$
   $(t_{i+1},Q_{i+1}),\ldots,(t_n,Q_n)),\Delta \rightarrow \Theta], \lambda T.\text{contract}_l(T,I)$
   else if sequent has the form $\Gamma,\forall x P,\Delta \rightarrow \Theta$ then
   return
   $[\Gamma,\forall x P,\forall x P,\Delta \rightarrow \Theta], \lambda T.\text{contract}_l(T,I)$
   else fail

18. **contract-r-tac (sequent)**
   If sequent has the form $\Gamma \rightarrow \Delta, (\exists x P,(t_1,Q_1),\ldots,(t_n,Q_n)), \Theta,$
3.1.2 The Primitive Tactics

and $n > 1$, and there is some $t_i$ that is admissible then
return
$$\left[ \Gamma \rightarrow \Delta, (\exists x P, (t_i, Q_i)), (\exists x P, (t_1, Q_1), \ldots, (t_{i-1}, Q_{i-1}), (t_{i+1}, Q_{i+1}), \ldots, (t_n, Q_n)), \Theta \right],$$
$\lambda T.\text{contract}_r(T, I)$
else if sequent has the form $\Gamma \rightarrow \Delta, \exists x P, \Theta$ then
return
$$\left[ \Gamma \rightarrow \Delta, \exists x P, \exists x P, \Theta \right], \lambda T.\text{contract}_r(T, I)$$
else fail

19. pusneg-1-tac (sequent)
If sequent has the form
$$\Gamma, \neg(\exists x P, (t_1, Q_1), \ldots, (t_n, Q_n)), \Delta \rightarrow \Theta$$
then return
$$\left[ \Gamma, (\forall x P, (t_1, \neg Q_1), \ldots, (t_n, \neg Q_n)), \Delta \rightarrow \Theta \right], \lambda T.\text{pushneg}_1(T, I)$$
else if sequent has the form $\Gamma, \neg(\forall x P, (y, Q)), \Delta \rightarrow \Theta$ then
return
$$\left[ \Gamma, (\exists x P, (y, \neg Q)), \Delta \rightarrow \Theta \right], \lambda T.\text{pushneg}_1(T, I)$$
else if sequent has the form $\Gamma, \neg\exists x P, \Delta \rightarrow \Theta$ then
return
$$\left[ \Gamma, \forall x P, \Delta \rightarrow \Theta \right], \lambda T.\text{pushneg}_1(T, I)$$
else if sequent has the form $\Gamma, \forall x P, \Delta \rightarrow \Theta$ then
return
$$\left[ \Gamma, \exists x P, \Delta \rightarrow \Theta \right], \lambda T.\text{pushneg}_1(T, I)$$
else fail

20. pusneg-r-tac (sequent)
If sequent has the form
$$\Gamma \rightarrow \Delta, \neg(\forall x P, (t_1, Q_1), \ldots, (t_n, Q_n)), \Theta$$
then return
$$\left[ \Gamma \rightarrow \Delta, (\exists x P, (t_1, \neg Q_1), \ldots, (t_n, \neg Q_n)), \Theta \right],$$
$\lambda T.\text{pushneg}_r(T, I)$
else if sequent has the form $\Gamma \rightarrow \Delta, \neg(\exists x P, (y, Q)), \Theta$ then
return
$$\left[ \Gamma \rightarrow \Delta, (\forall x P, (y, \neg Q)), \Theta \right], \lambda T.\text{pushneg}_r(T, I)$$
else if sequent has the form $\Gamma \rightarrow \Delta, \forall x P, \Theta$ then
return
$$\left[ \Gamma \rightarrow \Delta, \exists x P, \Theta \right], \lambda T.\text{pushneg}_r(T, I)$$
else if sequent has the form $\Gamma \rightarrow \Delta, \forall x P, \Theta$ then
return
$$\left[ \Gamma \rightarrow \Delta, \forall x P, \Theta \right], \lambda T.\text{pushneg}_r(T, I)$$
else fail

21. \textit{indirect*-tac (sequent)}
   If sequent has the form $\Gamma \rightarrow (\exists x \ P, (t_1, Q_1), \ldots, (t_n, Q_n))$ and $n > 1$ then
   return
   $\neg(\exists x \ P, (t_1, Q_1), \ldots, (t_n, Q_n)), \Gamma \rightarrow ]$, $\lambda T.\text{indirect}(T, I)$
else fail

22. \textit{indirect-tac (sequent)}
   If $\Delta \neq \emptyset$ in the input sequent $\Gamma \rightarrow \Delta$ then
   return
   $\neg[\Delta], \Gamma \rightarrow ]$, $\lambda T.\text{indirect}(T, I)$
else fail

23. \textit{thin*-tac (sequent)}
   For input sequent $\Gamma \rightarrow \Delta; \mathcal{M}$ construct the set of formulas $\mathcal{D}$ and the equivalence classes of the $\approx$ relation as in steps 1-2 of the thin* transformation function.
   If there is an equivalence class $P$ that is spanned by $\mathcal{M}$ then
   using $P$, construct the sequent $\Gamma' \rightarrow \Delta'$ as in step 4 of the transformation function
   If $\Gamma' \neq \Gamma$ or $\Delta' \neq \Delta$ then
   return
   $[\Gamma' \rightarrow \Delta'], \lambda T.\text{thin*}(T, I)$
else fail
else fail

24. \textit{thin**-tac (sequent)}
   Perform steps 1-7 of the thin** transformation function.
   If there is an equivalence class $P$ of the $\approx$ relation on $\mathcal{D}'$ that is spanned by $\mathcal{M}$ then
   using $P$, construct the sequent $\Gamma' \rightarrow \Delta'$ as in step 4 of the thin* transformation function
   If $\Gamma' \neq \Gamma$ or $\Delta' \neq \Delta$ then
   return
   $[\Gamma' \rightarrow \Delta'], \lambda T.\text{thin*}(T, I)$
else fail
else fail
3.1.2 The Primitive Tactics

25. **neg2-l-tac** (sequent)
   If sequent has the form \( \Gamma, \neg \neg A, \Delta \rightarrow \Theta \) then
   return
   \[ [\Gamma, A, \Delta \rightarrow \Theta, \lambda T.\text{neg2-l}(T, I)] \]
   else fail

26. **neg2-r-tac** (sequent)
   If sequent has the form \( \Gamma \rightarrow \Delta, \neg \neg A, \Theta \) then
   return
   \[ [\Gamma \rightarrow \Delta, A, \Theta, \lambda T.\text{neg2-r}(T, I)] \]
   else fail

27. **implies-1*-tac** (sequent)
   If sequent has the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta; M \) then
   determine the equivalence classes \( P_1 \) and \( P_2 \) and
   construct the sets of clauses for \( D_{P_1} \) and \( D_{P_2} \) as in
   steps 1-4 of the implies-1* transformation function.
   If \( D_{P_1} \) and \( D_{P_2} \) are both spanned by the mating \( M \) then
   return
   \[ [\Gamma', \Delta' \rightarrow A, \Theta'; C, \Gamma'', \Delta'' \rightarrow \Theta''], \lambda T_1 \lambda T_2.\text{implies-1*}(T_1, T_2, I) \]
   else fail
   else fail

28. **positive*-tac** (sequent)
   If sequent has the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta; M \) then
   determine the equivalence classes \( P_1 \) and \( P_2 \) and
   construct the sets of clauses for \( D_{P_1} \) and \( D_{P_2} \) as in step 1 of the positive* transformation function.
   If \( D_{P_1} \) and \( D_{P_2} \) are both spanned by the mating \( M \)
   and for every \( C' \in C \) such that \( C' \) is a cluster of one of
   the formulas in \( \Theta \), \([Dp^N(A)]_\sim \neq [C']_\sim \) then
   return
   \[ [\Gamma, \Delta \rightarrow A ; C, \Gamma, \Delta \rightarrow \Theta], \lambda T_1 \lambda T_2.\text{positive}(T_1, T_2, I) \]
   else fail
   else fail

29. **positive-tac** (sequent)
   If sequent has the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta \) then
   return
3.1.2 The Primitive Tactics

\[ [\Gamma, \Delta \rightarrow A ; C, \Gamma, \Delta \rightarrow \Theta], \lambda T_1 \lambda T_2 . \text{positive}(T_1, T_2, I) \]

else fail

30. contrapos*-tac (sequent)
If sequent has the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta \); \( M \) then
determine the equivalence classes \( P_1 \) and \( P_2 \)
and construct the sets of clauses for \( D_{P_1} \) and \( D_{P_2} \)
as in step 1 of the contrapos* transformation function.
If \( D_{P_1} \) and \( D_{P_2} \) are both spanned by the mating \( M \)
and for every \( C' \in C \) such that \( C' \) is a cluster of one of
the formulas in \( \Theta \), \( [D_{P_1}(\neg C)] \not= [C'] \) then
return
\[ [-A, \Gamma, \Delta \rightarrow \Theta ; C, \Gamma, \Delta \rightarrow \nu], \lambda T_1 \lambda T_2 . \text{contrapos}(T_1, T_2, I) \]
else fail
else fail

31. contrapos-tac (sequent)
If sequent has the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta \) then
return
\[ [\neg A, \Gamma, \Delta \rightarrow \Theta ; C, \Gamma, \Delta \rightarrow \nu], \lambda T_1 \lambda T_2 . \text{contrapos}(T_1, T_2, I) \]
else fail

32. backchain*-tac (sequent)
If sequent has the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta, C, \Lambda \) then
determine the equivalence classes \( P_1 \) and \( P_2 \)
and construct the sets of clauses for \( D_{P_1} \) and \( D_{P_2} \)
as in step 1 of the backchain* transformation function.
If \( D_{P_1} \) and \( D_{P_2} \) are both spanned by the mating \( M \)
and for every \( C' \in C \) such that \( C' \) is a cluster of one of
the formulas in \( \Gamma, \Delta, \Theta, \Lambda \), or \( A \), \( [D_{P_1}(C)] \not= [C'] \) then
return
\[ [\Gamma, \Delta \rightarrow A, \Theta, \Lambda ; C \rightarrow C], \lambda T_1 \lambda T_2 . \text{backchain}(T_1, T_2, I) \]
else fail
else fail

33. backchain-tac (sequent)
If sequent has the form \( \Gamma, A \supset C, \Delta \rightarrow \Theta, C, \Lambda \) then
3.1.2 The Primitive Tactics

return
\[ [\Gamma, \Delta \rightarrow A, \Theta, \Lambda ; C \rightarrow C] , \]
\[ \lambda T_1 \lambda T_2 . \text{backchain}(T_1, T_2, I) \]
else fail

34. forwardchain*-tac (sequent)
If sequent has the form \[ \Gamma, A \supset C, \Delta, A, \Theta \rightarrow \Lambda \]
\[ \Gamma, A, \Delta, A \supset C, \Theta \rightarrow \Lambda \]
then determine the equivalence classes \( P_1 \) and \( P_2 \)
and construct the sets of clauses for \( D_{P_1} \) and \( D_{P_2} \)
as in step 1 of the forwardchain* transformation function.
If \( D_{P_1} \) and \( D_{P_2} \) are both spanned by the mating \( \mathcal{M} \)
and for every \( C' \in C \) such that \( C' \) is a cluster of one of
the formulas in \( \Gamma, \Delta, \Theta, \Lambda \), or \( C \), \[ D_{P_1}(\neg A) \neq [C'] \]
then return
\[ [A \rightarrow A ; C, \Gamma, \Delta, \Theta \rightarrow \Lambda] , \]
\[ \lambda T_1 \lambda T_2 . \text{forwardchain}(T_1, T_2, I) \]
else fail
else fail

35. forwardchain-tac (sequent)
If sequent has the form \[ \Gamma, A \supset C, \Delta, A, \Theta \rightarrow \Lambda \]
\[ \Gamma, A, \Delta, A \supset C, \Theta \rightarrow \Lambda \]
then return
\[ [A \rightarrow A ; C, \Gamma, \Delta, \Theta \rightarrow \Lambda] , \]
\[ \lambda T_1 \lambda T_2 . \text{forwardchain}(T_1, T_2, I) \]
else fail

36. thin-to-axiom (sequent)
If sequent has the form \[ \Gamma, A, \Delta \rightarrow \Theta, A, \Lambda \]
then return
\[ [A \rightarrow A], \lambda T. \text{thin}^*(T, I) \]
else fail

37. axiomatize (sequent)
If sequent has the form \[ A \rightarrow A \]
then return
\[ [\ ], \text{axiom}(A) \]
else fail

38. query-tac (sequent)
3.1.2 The Primitive Tactics

Ask the user to input the name of a tactic or a tactical expression
Let tac:= the user's input
Let tac-result:= tac (sequent)
Return tac-result

39. auto-tac (sequent)
Let ET:= the result of calling the automatic theorem prover to obtain an ET-proof of sequent
If ET is an ET-proof then
return
[ ET ], λT.T
else fail

The first tactics in the list are those which apply a propositional rule to the generalized sequent. These include the tactics that correspond to the ∧-R, ∨-L, ∧-L, ∨-R, ⊃ -L, ⊃ -R, ¬-L, and ¬-R rules. In order to apply any of these tactics, there must be a formula with the appropriate connective on the left or right of the sequent arrow. The tactics for the ∼-L and ∼-R rules are similar since they only require the existence of a formula of a certain form in the sequent. For all of these rules the necessary information to apply the rule is present in the generalized sequent whether it is a simple sequent, an ET-proof, or anything in between. Note that any expansion trees present in the input sequent will be passed to the resulting sequent(s). In the proof term that is returned from these tactics, the only additional information needed to construct a proof from the subproofs is the position of the formula in the sequent to which the rule must be applied. This is included in the argument I.

In addition to the existence of a connective, other information is needed to apply any of the remaining rules. For the quantifier introduction rules, the substitution terms or variables must be determined. In the exists-1-tac and forall-r-tac tactics, if there is a variable y in the expansion tree, we check to see if it is free in the sequent according to the proviso on these rules. If it is not free, then either we did not start with a sound expansion tree, or y was introduced as a free variable by an application of forall-1-user-tac or exists-r-user-tac, so the tactic fails. In the proof term that is returned from exists-1-tac, λT.exists₁(T, I), T is a place holder for a proof of a sequent of the form Γ, [x/y]P, Δ → Θ. In addition to the position of the formula to which the rule is applied, I must contain the variable x so that reverse substitution can be applied to [x/y]P to obtain the formula ∀x P in the conclusion. The same applies to the proof term for forall-r-tac tactic.

To apply the ∀-L and ∃-R rules, determining the substitution term is much more complicated, since we do not have a proviso that restricts what is allowed. If a term
is not present in the generalized sequent the forall-1-tac and exists-r-tac tactics will fail. To apply the rule when this information is not present, a tactic that obtains this information from the user must be called. Both forall-1-tac and forall-1-user-tac return a proof term of the form $\lambda T.\text{forall-1}(T, I)$ where $T$ is a placeholder for a proof of a sequent of the form $\Gamma, [x/t]P, \Delta \rightarrow \Theta$. As in the proof terms returned from the exists-1-tac and forall-r-tac tactics, $I$ must contain information to obtain $\forall x P$ in the conclusion which replaces $[x/t]P$ in the premise. In this case, $I$ contains a copy of $P$ and the variable $x$ from which we can trivially obtain $\forall x P$. We cannot simply store $x$ alone and apply reverse substitution to $[x/t]P$ because we may not want to replace every occurrence of $t$ with $x$. This is in contrast to the exists-1-tac and forall-r-tac tactics where the proviso requires that every occurrence of $y$ be replaced with $x$, otherwise $y$ would be free in the conclusion.

Like many of the other tactics, the tactics for the LK+ rules that require the mating analysis described in Section 2.2.3 are similar to their corresponding transformation functions. The thin*-tac and thin**-tac tactics are similar to the thin* and thin** functions, respectively, except that they include a check for an equivalence class that is spanned by the mating. The mating must be clause spanning in order to determine which formulas will not be needed to complete the proof. When we do not have a cs-mating there will be no equivalence class that is spanned, and so the tactic will fail. For this rule $I$ must contain all the formulas that were removed from the original sequent and their corresponding positions. The thin-r-tac and thin-1-tac tactics are provided to allow a user to interactively specify which formula to remove from a sequent. This allows the user to selectively remove formulas in order to construct more readable proofs. This is useful when there is no mating and the user is aware that certain formulas are not needed to complete the proof. Again $I$ includes the formula that is removed and its position in the original sequent.

The implies-1*-tac, contrapos*-tac, positive*-tac, backchain*-tac, and forwardchain*-tac tactics are all similar to their corresponding transformation functions. If the mating is not clause spanning, $D_P$ and $D_P$ will not be spanned, so the tactic will fail. Thus, these rules will not be applied to any sequent that is not an ET-proof with a cs-mating.

It may be desirable to apply one of these rules even though a clause spanning mating is not present. For example, a user may want to attempt the contrapos, positive, backchain, or forwardchain rule when attempting to interactively build a natural proof. The contrapos-tac, positive-tac, backchain-tac, and forwardchain-tac tactics are provided for this reason. The interactive version of implies-1*-tac is implies-1-tac which is one of the propositional rules.

The indirect*-tac tactic is similar to its corresponding LK+ transformation
function which looks for an expansion tree of a specific form on the right of the sequent to determine if the rule is applicable. The indirect-tac tactic applies the more general indirect rule which negates all formulas on the right and moves them to the left. It gives the user the flexibility to construct an indirect proof at any point.

To make things a bit simpler conceptually, we can divide the rules into two groups. The first are those that work in “automatic mode” or perhaps more appropriately “transformation mode.” The system runs in this mode when we start with a generalized sequent that is an ET-proof with a cs-mating. In general this ET-proof and mating are obtained from an automatically generated resolution refutation and the tactics are actually performing the transformation from ET-proof to LK+ proof. The other group are the tactics that operate in “interactive mode” which is generally used when the input sequent is a simple sequent and a user is interacting with the system to direct the search for a proof. It is also possible that a user may be building a proof of a sequent that contains expansion trees with some substitution information (but is not an ET-proof) or a partial mating. This information is used to help direct the search. In this case the individual tactics operate in interactive or automatic mode depending on the information contained in the mating or the expansion tree of the formula to which the rule is being applied. Table 1 contains a list of all the primitive tactics and the modes under which they operate. Many operate in interactive or automatic mode exclusively. For those that operate in both modes, a further distinction can be made. Some will operate differently on the input sequent depending on the information contained in the sequent, while others (mainly the propositional rules) will perform the same operation on any input sequent.
### 3.1.2 The Primitive Tactics

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Interactive</th>
<th>Automatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>and-r-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>or-l-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>and-l-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>or-r-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>implies-l-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>implies-r-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>neg-l-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>neg-r-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>exists-l-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>forall-r-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>forall-l-tac</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>forall-l-user-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>exists-r-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>exists-r-user-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>thin-l-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>thin-r-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>contract-l-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>contract-r-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>pushneg-l-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>pushneg-r-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>indirect*-tac</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>indirect-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>thin*-tac</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>thin**-tac</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>neg2-l-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>neg2-r-tac</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>implies-1*-tac</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>positive*-tac</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>positive-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>contrapos*-tac</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>contrapos-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>backchain*-tac</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>backchain-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>forwardchain*-tac</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>forwardchain-tac</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>thin-to-axiom</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>axiomatize</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>query</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>auto-tac</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The Primitive Tactics
3.1.3 The Tacticals

We have taken a functional programming approach to theorem proving by using tactics which are defined to be functions that perform individual proof steps. They each take a generalized sequent as input and return a generalized sequent list and a partial proof as output. Yet these values are returned only when a tactic is successful. A tactic may fail when a particular condition within the tactic is not satisfied. In this case, the function fails and a failure token is returned.

At a higher level these failure tokens are used to control the search for a proof. When a failure token is returned from a tactic, we know that the tactic is not applicable to the sequent, and another must be attempted. Thus, we need some kind of control algorithm to specify the order in which tactics are attempted. The tacticals, which will be described in this section, are the control procedures in our programming language. Controlling search in this way gives our paradigm aspects of a logic programming approach. Yet when using these tacticals as control structures, the user has more control over the search process than that provided by the simple backtracking algorithm used by logic programming languages such as Prolog. The tactic and tactical approach attempts to combine functional and logic programming, using aspects of each where appropriate to increase flexibility and efficiency.

The tacticals are described below. They are nearly identical to those found in LCF [Gorden, Milner, & Wadsworth 79].

1. idtac (sequent)
   Return [sequent] ; \lambda T. T

2. then (tactic1 tactic2 sequent)
   Apply tactic1 to sequent.
   If this fails then
   fail
   else
   apply tactic2 to the each sequent in the resulting
   sequent list.
   If this fails on any sequent then
   fail
   else
   return the list of sequents returned from the calls
   to tactic2, and the \lambda-term representing
   the 2-step proof combining the results of applying
   tactic1 followed by tactic2.

3. then* (tactic1 tactic2 sequent)
3.1.3 The Tacticals

Apply tactic1 to sequent.
If this fails then
fail
else
apply tactic2 to the each sequent in the resulting sequent list.
If this fails on any sequent then
return the result of applying tactic1
else
return the list of sequents returned from the calls to tactic2, and the λ-term representing the 2-step proof combining the results of applying tactic1 followed by tactic2.

4. orelse (tactic1 ... tacticN sequent)
If N = 0 then
fail
else
apply tactic1 to sequent.
If this fails then
recursively call orelse with arguments tactic2 ... tacticN and sequent
else
return the result of applying tactic1.

5. repeat (tactic sequent)
(orelse (then tactic (repeat tactic)) idtac sequent)

Note that all combining of partial proofs is done within the then (and then*) tacticals. The orelse tactical only applies one tactic so it not necessary to do combining. The repeat tactical is defined in terms of the others.

Tacticals and tactics make up the programming language for writing proof procedures (compound tactics). Programs written in this language have syntax very close to Common Lisp. To define a tactic, we provide a define-tac macro which looks very much like Lisp’s defun. It takes a tactic name, a list of arguments, and a body which is a tactical expression. This will be illustrated later when we write tactics for the interactive theorem prover, and for transforming automatically generated proofs to LK+ proofs. The define-tac macro expands the tactic to Common Lisp which is what actually gets applied when the tactic is called. Compound tactics are applied as a unit and will succeed or fail in the same way that the primitive tactics do.
3.1.4 The Typing Mechanism

In $\chi$, generalized sequents are viewed as types, and LK+ proofs as values over these types. A simple sequent is the most general kind of type and can be refined by adding information. Informally, a type $T_1$ is more refined than $T_2$ if the generalized sequents in $T_2$ contain all of the same information as those in $T_1$, and also include additional information in the form of substitution information at quantifier nodes, and/or additional mated pairs in the mating.

Example 1 The following generalized sequents specify types for LK+ proofs of the sequent $p(a) \lor q(b), \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)$.

1. $p(a) \lor q(b), \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)$
2. $p(a) \lor q(b), \forall x (p(x) \supset q(x)) \rightarrow (\exists x q(x), (a, q(a)))$
3. $p(a) \lor q(b), (\forall x (p(x) \supset q(x)), (a, p(a) \supset q(a)))$
   $\rightarrow (\exists x q(x), (a, q(a)), (b, q(b)))$
4. $p(a)_1 \lor q(b)_1, (\forall x (p(x) \supset q(x)), (a, p(a)_2 \supset q(a)_1))$
   $\rightarrow (\exists x q(x), (a, q(a)_2), (b, q(b)_2));$
   $\{\{p(a)_1, p(a)_2\}, \{q(a)_1, q(a)_2\}, \{q(b)_1, q(b)_2\}\}$

Each one is a more refined type than the previous ones. The last two generalized sequents are ET-proofs. Whenever we have an ET-proof we know that elements (LK+ proofs) of this type exist. In the last one, the atom occurrences are subscripted to indicate corresponding atoms in the ET-proof and mating.

If a sequent is provable, it generally has many LK+ proofs. An ET-proof always has one and generally has many corresponding LK+ proofs, depending on the order that transformation functions are applied. Thus an ET-proof underspecifies an LK+ proof. Hence there is a (possibly infinite) set of LK+ proof terms (values) corresponding to a given generalized sequent (type) when it is provable (i.e. is a theorem). When it is not provable, the generalized sequent is equivalent to the type void. In this paradigm the theorem proving process can be viewed as the search for the existence of a value of a given type. Because we explicitly represent proofs as $\lambda$-terms, at the end of the theorem proving process for a provable sequent we not only know that a value of that type exists, but we have an example of such a value.

A value is of a given type when, in addition to using particular inference rules corresponding to the connectives in the generalized sequent, we also use the substitution information in the expansion trees (when it is present) when applying the
quantifier rules. Also when mating information is present in the type specification, the axioms of the LK+ proof correspond to the mated pairs.

Partial proofs are also given types. This assures that the λ-conversion mechanism correctly represents the operation of supplying a partial proof with a subproof. Partial proofs are functions from generalized sequents to a generalized sequent. For example if \( \lambda x \lambda y T(x, y) \) represents a partial LK+ proof of generalized sequent \( \sigma \), where \( x \) and \( y \) are place holders for LK+ proofs of the generalized sequents \( \sigma_1 \) and \( \sigma_2 \) respectively, then this \( \lambda \)-term has type \( \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma \). This provides a type checking mechanism for filling in subproofs. For example, we can apply \( \lambda x \lambda y T(x, y) \) to an actual proof \( P \) as long as the generalized sequent specifying the type of \( P \) is the same as the type of the variable \( x \).

For tactics, we need to extend our typing mechanism to include dependent types as in [Constable et. al. 86]. In our case, the type of the output of the tactics will depend on the value of the input. Each tactic is a mapping from a generalized sequent to a generalized sequent list and partial proof as follows:

\[
\sigma \mapsto ([\sigma_1; \ldots; \sigma_n] \# T : (\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma))
\]

where the type of the partial proof, \( T \), depends on the input sequent \( \sigma \), and the list of generalized sequents which are the subproofs that still must be completed. \( T \) is a function from the subproofs to a proof of type \( \sigma \) (the input sequent) and thus itself has type \( \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma \).

For each of the inference rules, \( n = 1 \) or \( n = 2 \) depending on the number of premises in the rule. For example, the mapping for the and-r-tac tactic is

\[
(\Gamma \rightarrow \Delta, A \land C, \Theta) \mapsto \\
([\Gamma \rightarrow \Delta, A, \Theta; \Gamma \rightarrow \Delta, C, \Theta] \# \\
\text{and.r}:(\Gamma \rightarrow \Delta, A, \Theta) \rightarrow (\Gamma \rightarrow \Delta, C, \Theta) \rightarrow (\Gamma \rightarrow \Delta, A \land C, \Theta))).
\]

In the \( \lambda \)-terms representing partial proofs that are returned from the tactics, the abstracted variables have types given by the list of generalized sequents. For example, \( T_1 \) and \( T_2 \) in the term \( \lambda T_1 \lambda T_2 \text{and.r}(T_1, T_2, I) \), returned by \text{and-r-tac}, have types \( \Gamma \rightarrow \Delta, A, \Theta \) and \( \Gamma \rightarrow \Delta, C, \Theta \) respectively.

Note that in order for \text{forall-1-user-tac} to conform to the general type above it has the following mapping:

\[
(\Gamma, \forall x P, \Delta \rightarrow \Theta) \mapsto \\
([\Gamma, [x/t] P, \Delta \rightarrow \Theta] \# \\
\text{forall.1}:(\Gamma, [x/t] P, \Delta \rightarrow \Theta) \rightarrow (\Gamma, \forall x P, \Delta \rightarrow \Theta)).
\]

This can also be specified as:
In the second expression, the `forall1` term has a more refined type than in the first because the tactic is called with a sequent that has no substitution information, but since one is supplied by the user, this can be included in the type of the λ-term that is returned. In more general terms, the mapping of the tactic is of the form

\[ \sigma \mapsto ([\sigma_1; \ldots; \sigma_n]#T : \sigma_1 \to \ldots \to \sigma_n \to \sigma') \]

where \( \sigma' \) is a more refined type for (or subtype of) \( \sigma \). We would like our tactics to return a function that produces a proof of the same type \( \sigma \) that was input to the tactic. But since any LK+ proof of type \( \sigma' \) is also an LK+ proof of \( \sigma \), this refined specification for the type of the partial proof returned from the `forall1-user-tac` tactic is also acceptable. The `exists-r-tac` tactic is similar. The `exists1-tac` and `forall-r-tac` tactics also return a partial proof with a more refined type, but only when they must obtain the substitution term by finding a new variable (i.e. when it is not already present in the expansion tree).

The `auto-tac` tactic can be viewed as taking a type and, if possible, producing a more refined type—one that we know is not void. This tactic has the mapping

\[ \sigma \mapsto ([\sigma']#I\beta : (\sigma' \to \sigma')) \]

where \( I\beta = \lambda T.T \). But since \( \sigma' \) is a subtype of \( \sigma \), this is acceptable within our typing paradigm.

Finally, note that `axiomatize` has mapping \( \sigma \mapsto ([]#axiom : \sigma) \). It simply produces a proof term which is a constant of type \( \sigma \).

In the process of building an LK+ proof, we need to be able to combine partial proofs as a proof becomes more refined. As in LCF, the then tactical is responsible for combining partial proofs. If we have a partial proof of type \( \sigma_1 \to \ldots \to \sigma_n \to \sigma \), we need to find proofs of types specified by the generalized sequents \( \sigma_1, \ldots, \sigma_n \) in order to obtain a complete proof of \( \sigma \). In the search for a proof of type \( \sigma \), some tactic or combination of tactics may return a partial proof of the type \( \tau_1 \to \cdots \to \tau_m \to \sigma_i \).

This is a partial proof with \( m \) missing subproofs. These two partial proofs are combined by matching and replacing \( \sigma_i \), into a single partial proof of type

\[ \sigma_1 \to \cdots \to \sigma_{i-1} \to \tau_1 \to \cdots \to \tau_m \to \sigma_{i+1} \to \cdots \to \sigma_n \to \sigma \]

which is a more refined partial proof for a proof of type \( \sigma \).

**Example 10** Suppose that some combination of tactics returns the following partial proof:
3.1.4 The Typing Mechanism

\[
\frac{p(a), \forall x \ [p(x) \supset q(x)] \quad \exists x \ q(x)}{q(b) \quad \exists x \ q(x) \quad \text{thin-L}}
\]

\[
\frac{p(a) \lor q(b), \forall x \ [p(x) \supset q(x)] \quad \exists x \ q(x)}{\forall-l \quad \text{V-L}}
\]

\[
\frac{[p(a) \lor q(b)] \land \forall x \ [p(x) \supset q(x)] \quad \exists x \ q(x)}{\land-L \quad \supset -R}
\]

where the term representing this proof is:

\[\text{lambda X lambda Y. } \text{implies_r(and_l(or_l(X,thin_l(Y))))}\].

In this example, for readability, we drop the I argument that specifies additional information needed to fully specify the LK+ partial proofs. For example, the thin_l function needs an argument specifying which formula was removed. This term has type:

\[
(p(a), \forall x \ [p(x) \supset q(x)] \quad \exists x \ q(x) \quad (q(b) \quad \exists x \ q(x)) \quad \rightarrow \quad \rightarrow [p(a) \lor q(b)] \land \forall x \ [p(x) \supset q(x)] \supset \exists x \ q(x)).
\]

A partial proof of the left branch is:

\[
\frac{p(a) \quad p(a) \quad q(a) \quad \exists x \ q(x)}{\supset -L}
\]

\[
\frac{p(a), p(a) \supset q(a) \quad \exists x \ q(x)}{\forall-l \quad \text{V-L}}
\]

\[
\frac{p(a), \forall x \ [p(x) \supset q(x)] \quad \exists x \ q(x)}{\forall-l \quad \text{V-L}}
\]

where the term representing this partial proof is:

\[\text{lambda Z. forall_l(implies_l(axiom(p(a)),Z))}\]

of type:

\[
(q(a) \quad \exists x \ q(x)) \quad (p(a), \forall x \ [p(x) \supset q(x)] \quad \exists x \ q(x)).
\]

When these two partial proofs are combined, we get the partial proof:

\[\text{lambda Z lambda Y. } \text{implies_r(and_l(or_l(forall_l(implies_l(axiom(p(a)),Z)), thin_l(Y))))}\]

of type:

\[
(q(a) \quad \exists x \ p(x)) \quad (q(b) \quad \exists x \ q(x)) \quad \rightarrow \quad \rightarrow [p(a) \lor q(b)] \land \forall x \ [p(x) \supset q(x)] \supset \exists x \ q(x)).
\]
This paradigm for combining partial proofs depends on the fact that tactics return a function that produces a proof whose type is specified by the input sequent. A partial proof for input sequent $\sigma_i$ will always have type of the form $\tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \sigma_i$ where $m \geq 0$, so that when combining with a larger proof, matching can take place on $\sigma_i$.

This can be viewed slightly differently for tactics such as the auto-tac tactic that produce a partial proof of more refined type than their input. When we have a partial proof of type $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma$ as before, we can first call the auto-tac tactic with input $\sigma_i$ producing refined type $\sigma'_i$, then build a partial proof of type $\tau'_1 \rightarrow \ldots \rightarrow \tau'_m \rightarrow \sigma'_i$. We combine the partial proofs as described above, this time obtaining a partial proof of type

$$\sigma_1 \rightarrow \ldots \rightarrow \sigma_{i-1} \rightarrow \tau'_1 \rightarrow \ldots \rightarrow \tau'_m \rightarrow \sigma_{i+1} \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma'.$$

In this case, we can match $\sigma'_i$ with $\sigma_i$, since $\sigma'_i$ is a subtype of $\sigma_i$. Also, $\sigma'$ is a refined type of $\sigma$ containing the same extra information as $\sigma'_i$. The following example illustrates this.

**Example 11** If we have the first partial proof in Example 10 of type:

$$(p(a), \forall x [p(x) \supset q(x)] \rightarrow \exists x q(x)) \rightarrow$$

$$(q(b) \rightarrow \exists x q(x)) \rightarrow$$

$$(\rightarrow [p(a) \lor q(b)] \land \forall x [p(x) \supset q(x)] \supset \exists x q(x))$$

and we call the auto-tac tactic with input:

$$q(b) \rightarrow \exists x q(x)$$

it will return:

$$[q(b) \rightarrow (\exists x q(x), (b,q(b)))] \text{, } \lambda T.T$$

where the $\lambda$-term has type:

$$(q(b) \rightarrow (\exists x q(x), (b,q(b)))) \rightarrow (q(b) \rightarrow (\exists x q(x), (b,q(b)))).$$  

The combined partial proof then has type:

$$(p(a), \forall x [p(x) \supset q(x)] \rightarrow \exists x q(x)) \rightarrow$$

$$(q(b) \rightarrow (\exists x q(x), (b,q(b)))) \rightarrow$$

$$(\rightarrow [p(a) \lor q(b)] \land \forall x [p(x) \supset q(x)] \supset (\exists x q(x), (b,q(b))))).$$

This is simply a subtype of the previous type, so both specify types of the partial proof.
3.1.5 Extending LCF

We have described in detail how we have adapted the LCF tactic and tactical approach to theorem proving [Gorden, Milner, & Wadsworth 79] to work within our setting to create a high level programming language that allows the user to become involved in the theorem proving process. We have also extended the LCF approach in several ways. First, we have replaced the LCF notion of validation with explicit term representation of proofs. In LCF, a tactic takes an object of type goal as input. It returns a list of subgoals that must still be achieved, and a validation which is a function that can be applied to the subgoals once they are achieved, to infer the achievement of the goal. When a validation is applied, a new object of type thm is created. This is a special type reserved only for those items which can be deduced from a series of LCF inference rules. The contents of these validation functions are hidden from the user, and once applied are discarded since they are no longer needed.

Our tactics are similar in that they take in a sequent and return a list of sequents that still must be proved, but instead of a validation, they return a partial proof. Note the similarity to LCF if we view the proof terms as validation functions. For example, the term \( \lambda T_1 \lambda T_2. \text{and-r}(T_1, T_2, I) \) which is the partial proof returned from the and-r-tac can be thought of as a function that will "validate" a proof of \( \Gamma \rightarrow \Delta, A \land C, \Theta \) from proofs of sequents \( \Gamma \rightarrow \Delta, A, \Theta \) and \( \Gamma \rightarrow \Delta, C, \Theta \) by applying the \( \land \)-R LK+ inference rule. (Here \( T_1 \) and \( T_2 \) are place holders for the premises of the \( \land \)-R rule.) But, in \( \chi \), we explicitly store the entire proof as it is filled in and do not discard it along the way. We know we have a complete proof, and that the sequent we set out to prove is a theorem, when all of the subproofs are filled in. But we do not give this sequent a special type. In fact, rather than having a type of its own, it specifies the type of the complete proof term. This data object is an explicit representation of the proof, which can be manipulated in many ways. For example, we can present the complete proof to a user when requested. Several other applications are described in Chapter 4.

We also extend the typing system of LCF. In LCF, every tactic has the same type: \( \text{goal} \rightarrow (\text{goal list } \# \text{ validation}) \). The token "goal" covers every item that can be input to a tactic. In the \( \chi \) system, each generalized sequent specifies a type. Each tactic takes in a generalized sequent that must match a specific "type template." Thus, in \( \chi \), much of the test for whether or not a tactic applies to a generalized sequent involves some kind of type checking i.e. testing to see if the argument specifies the appropriate type. Hence, it is possible for a tactic to fail before it begins execution. Also, the list of generalized sequents that is returned can be thought of as a list of type specifications.

In LCF, the token "validation" (which represents \( \text{thm list } \rightarrow \text{thm} \)) describes every validation function. Our tactics, on the other hand, return a partial proof
which is a function mapping arguments of the types in the sequent list to a result of the type specified by the input sequent. This mapping is specific to each tactic. For example, the \texttt{and-r-tac} tactic returns a term of type

\[(\Gamma \rightarrow \Delta, A, \Theta) \rightarrow (\Gamma \rightarrow \Delta, C, \Theta) \rightarrow (\Gamma \rightarrow \Delta, A \land C, \Theta).\]

By developing this richer type structure (much of which was described in Section 3.1.4), we are attempting to give a formalization to these proof objects and the operations on them. There is still much work to be done in this area.
3.2 The Interactive Proof Editor

The interactive proof editor is simply a program written in our programming language of tactics and tacticals. It is defined as follows.

\[
\text{(define-tac interactive (sequent)}
\]
\[
\quad \text{(repeat)}
\]
\[
\quad \text{(orelse)}
\]
\[
\quad \text{axiomatize}
\]
\[
\quad \text{query}} \text{sequent)}
\]

It is basically repeated calls to the query tactic and its power lies in what can be entered by the user as input to this tactic.

The basic input to the query tactic is the name of another primitive tactic. Any of those that operate in “interactive mode” are acceptable. (Note that it is not illegal to enter the name of an “automatic” tactic, but when this is done, the tactic will immediately fail because the information necessary for the tactic to succeed is not present.) Thus the user can direct the proof by applying one LK+ inference rule at a time. Some of these rules will make further inquiries to the user for substitution or thinning information. The theorem proving process proceeds in a depth first fashion. When a rule succeeds, the list of sequents that it returns (the premises of the rule) are the sequents for which subproofs must be completed. They are presented to the user in the order they appear in the list. The user must complete or stop the proof of one branch before the next is presented.

The user may also enter compound tactics as input to the query tactic. This can be done in one of two ways. The first is to enter the tactical expression directly. For example, if the user wants to repeatedly apply the \(\wedge\)-L and \(\vee\)-R rules to replace all top level \(\wedge\)'s on the left and \(\vee\)'s on the right by commas, s/he can enter

\[
\text{(repeat (orelse and-1-tac or-r-tac) query)}
\]

The user can also enter the name of an existing compound tactic that has been written by him/herself (or any other user) in the programming language of tactics and tacticals. The following examples will illustrate this.

Example 12 Suppose we want to prove a set of theorems that all have the form

\[
\rightarrow (A_1 \wedge A_2 \wedge \ldots \wedge A_n) \supset B.
\]

In other words, we want to prove \(B\) from a set of hypotheses \(A_1, \ldots, A_n\). We may want to automate the part of the proof tree that breaks these connectives to get \(A_1, A_2, \ldots, A_n \rightarrow B\), then apply all non-branching propositional rules. A procedure to do this can be written as follows:

\[
\text{(define-tac start-proof (sequent)}
\]
\[
\quad \text{(then (then imp-r-tac)}
\]
\[
\quad \quad \text{(repeat and-1-tac))}
\]

\[
\text{3.2 The Interactive Proof Editor}
\]

The interactive proof editor is simply a program written in our programming language of tactics and tacticals. It is defined as follows.

\[
\text{(define-tac interactive (sequent)}
\]
\[
\quad \text{(repeat)}
\]
\[
\quad \text{(orelse)}
\]
\[
\quad \text{axiomatize}
\]
\[
\quad \text{query}} \text{sequent)}
\]

It is basically repeated calls to the query tactic and its power lies in what can be entered by the user as input to this tactic.

The basic input to the query tactic is the name of another primitive tactic. Any of those that operate in “interactive mode” are acceptable. (Note that it is not illegal to enter the name of an “automatic” tactic, but when this is done, the tactic will immediately fail because the information necessary for the tactic to succeed is not present.) Thus the user can direct the proof by applying one LK+ inference rule at a time. Some of these rules will make further inquiries to the user for substitution or thinning information. The theorem proving process proceeds in a depth first fashion. When a rule succeeds, the list of sequents that it returns (the premises of the rule) are the sequents for which subproofs must be completed. They are presented to the user in the order they appear in the list. The user must complete or stop the proof of one branch before the next is presented.

The user may also enter compound tactics as input to the query tactic. This can be done in one of two ways. The first is to enter the tactical expression directly. For example, if the user wants to repeatedly apply the \(\wedge\)-L and \(\vee\)-R rules to replace all top level \(\wedge\)'s on the left and \(\vee\)'s on the right by commas, s/he can enter

\[
\text{(repeat (orelse and-1-tac or-r-tac) query)}
\]

The user can also enter the name of an existing compound tactic that has been written by him/herself (or any other user) in the programming language of tactics and tacticals. The following examples will illustrate this.

Example 12 Suppose we want to prove a set of theorems that all have the form

\[
\rightarrow (A_1 \wedge A_2 \wedge \ldots \wedge A_n) \supset B.
\]

In other words, we want to prove \(B\) from a set of hypotheses \(A_1, \ldots, A_n\). We may want to automate the part of the proof tree that breaks these connectives to get \(A_1, A_2, \ldots, A_n \rightarrow B\), then apply all non-branching propositional rules. A procedure to do this can be written as follows:

\[
\text{(define-tac start-proof (sequent)}
\]
\[
\quad \text{(then (then imp-r-tac)}
\]
\[
\quad \quad \text{(repeat and-1-tac))}
\]
Example 13 Since applying a propositional rule requires only the existence of a formula with a particular connective, we can write a procedure to apply all possible propositional rules until only quantifiers are left at the top level. The following procedure will do this.

```
(define-tac propositional (sequent)
  (repeat (orelse axiomatize
           thin-to-axiom
           implies-r-tac
           and-1-tac
           or-1-tac
           and-r-tac
           implies-1-tac
           or-r-tac
           neg-r-tac
           neg-1-tac
           neg-l-tac)) sequent)
```

Note that any ordering of the tactics in the above procedure will give us a complete theorem prover for propositional logic. This procedure tries to incorporate two strategies. One is to minimize branching in the proof tree, and the other is to minimize the movement of formulas to or from the right side of the sequent arrow. The rules are ordered so that "non-movement, non-branching" rules are attempted first, followed by "non-movement, branching" rules, and finally "movement" rules. These two examples illustrate that by writing such proof procedures we can automate the "uninteresting" details of a proof, as well as develop quite sophisticated strategies. This gives us great flexibility in customizing proof search and building proof heuristics.

It may be the case that a particular inference rule is applicable to more than one formula in a given sequent. If the user enters the name of the tactic for that rule, the rule will be applied to the first possible formula. As well as choosing which tactic to apply, the user needs the flexibility to choose which formula it should be applied to. For every primitive tactic that corresponds to an inference rule there is a pretactic whose name is the same with the suffix "-tac" replaced by "-pretac". Pretactics take an extra argument which is an integer corresponding to the position.
of the formula on the left or right of the sequent arrow. For example, to apply the \(\land\)-L rule as follows,

\[
\frac{A_1 \land B_1, A_2, B_2, A_3 \land B_3}{A_1 \land B_1, A_2 \land B_2, A_3 \land B_3} \rightarrow \Theta \land\-L
\]

the user must enter \(\text{and}\-1\-pretac 2\).

When proving a theorem interactively, it may be desirable to stop the process at some point, and later come back to it to fill in missing subproofs. To stop proving a branch of the proof tree, the user simply enters \texttt{stop} as input to the \texttt{query} tactic. When all branches have been completed or stopped, the partial proof is a \(\lambda\)-term with abstracted variables representing the incomplete subproofs as described in Section 3.1.1. When we have completed a partial proof of one of the missing subproofs, we simply call the procedure \texttt{combine-proofs} which does type checking and \(\lambda\)-conversion and creates a more refined partial proof very much like the \texttt{then} tactical. This allows the user to work on subproofs independently of the main proof. Thus the user is not limited to depth first construction of proofs, nor to completing a proof in one session.
3.3 Integrating an Automatic Theorem Prover

One of the goals of the $\chi$ theorem proving system is to bring automatically generated proofs into a natural deduction setting, so that they are readable and can be manipulated as objects. Any resolution or automatic expansion tree prover can be integrated into the system. For resolution, a proof is first converted from a resolution refutation to an ET-proof and mating (see Section 2.2.1). The process begins with a call to the auto-tac tactic with a simple sequent as input. This tactic calls the automatic theorem prover and returns a generalized sequent which is an ET-proof of the input sequent, refining the type of LK+ proof term originally specified by the simple sequent. An ET-proof with a mating is a more refined type than an ET-proof without, but in either case, we know there exist values of this type. To obtain such a value, $\chi$ provides a compound tactic, called complete-transform-1 which is a proof procedure written using tacticals and primitive tactics that operate in "automatic mode" and will produce a complete LK+ proof from an ET-proof. The auto-tac and complete-transform-1 tactics are combined to form the atp tactic which given any provable sequent will automatically produce an LK+ proof. It is simply defined as

\[
\text{(define-tac atp (sequent) (then auto-tac complete-transform-1) sequent).}
\]

The complete-transform-1 tactic is defined in Figure 9. It was written with the purpose of constructing LK+ proofs that are as natural and readable as our primitive tactics in Section 3.1.2 permit. Its design will be discussed in detail here. It basically tries to follow the criteria discussed in Section 2.1.1. It is doubtful that there is one proof procedure that, for any ET-proof, will build its corresponding "most natural" LK+ proof. The programming language of tactics and tacticals gives the user the flexibility of developing his/her own complete transformation algorithms customized to specific needs and ideas that can be used as alternatives to the one presented here. (Note that in order for a transformation function to be complete, a user must follow the criteria discussed in Section 2.2.2.)

The main structure of this function is the orelse construct enclosed by a repeat. It simply goes down the list until one of the tactics succeeds and then repeats. Most of the strategy of this procedure is in the ordering of the tactics within this orelse statement.

First of all, if a sequent is an axiom or can be thinned to an axiom, we want to complete the branch of the proof tree, so axiomatize and thin-to-axiom appear first in the list.

\[
\text{(define-tac atp (sequent) (then auto-tac complete-transform-1) sequent).}
\]
(define-tac complete-transform-1 (sequent)
  (repeat
    (orelse axiomatize
      thin-to-axiom
      (then* (repeat and-l-tac thin*)
        implies-r-tac
        exists-l-tac
        forall-r-tac
        exists-r-tac
        forall-l-tac
      (then contract-l-tac forall-l-tac)
      (then* or-l-tac thin*)
      (then* and-r-tac thin*)
      (orelse (then thin*
        (orelse backchain*-tac
          forwardchain*-tac
          positive*-tac
          contrapos*-tac
          implies-l*-tac)))
    (orelse backchain*-tac
      forwardchain*-tac
      positive*-tac
      contrapos*-tac
      implies-l*-tac)))
  neg2-l-tac
  neg2-r-tac
  pushneg-l-tac
  pushneg-r-tac
  (then indirect*-tac pushneg-l-tac)
  neg-r-tac
  (then* or-r-tac thinning)
  neg-l-tac)) sequent)

Figure 9: A Complete Transformation Tactic
Any sequent of the form $\Gamma \rightarrow \Delta, A_1 \land \ldots \land A_n \supset B, \Theta$ is conceptually the same as $A_1, \ldots, A_n, \Gamma \rightarrow B, \Delta, \Theta$ obtained by applications of the $\land$-$L$ and $\supset$-$R$ rules. The and-$1$-tac and implies-$r$-tac tactics that apply these rules appear early in the list, and so will be applied whenever possible. The thin* tactic is applied after all possible applications of the and-$1$-tac tactic removing all unnecessary hypotheses so that they do not clutter the proof and possibly cause it to become more complex than necessary.

Formulas of the form $\forall x \ P$ on the right of the sequent arrow and $\exists x \ P$ on the left have at most one selected variable in their corresponding expansion trees. These substitutions can be applied whenever formulas of this form appear in the sequent. Since these rules remove a selected variable from the generalized sequent, it is possible that some expansion terms now become admissible, so it is desirable to apply them early. It is also desirable to apply the $\forall$-$L$ rule if there is an admissible expansion term. This rule, like the other quantifier rules, causes no branching and performs a conceptually simple operation. Thus the forall-$1$-tac and contract-$1$-tac tactics are attempted next. The exists-$r$-tac tactic is attempted also at this point, to apply the appropriate substitution if there is only one expansion term, but the contract-$r$-tac tactic is delayed since this would cause more than one formula to appear on the right.

The next tactics in the list are those that cause branching. The first such tactic to appear is or-$1$-tac. This rule corresponds to a cases argument, another natural construct. When there are two conclusions (connected by an $\land$), the and-$r$-tac tactic breaks them up so that each can be proven separately. The thin*-tac tactic is attempted after each of these rules, because it is likely that some formulas are needed only in one branch and so can be removed from the other.

The $\supset$-$L*$ rule is quite unnatural and that is the reason for including so many more natural versions of it. The backchain*-tac and forwardchain*-tac tactics are tried first because they can be considered non-branching rules, since one branch immediately becomes an axiom. The positive*-tac and contrapos*-tac tactics are then attempted before resorting to the implies-$1$*-tac tactic. The thin*-tac tactic is attempted before any of these tactics to prevent applying these rules to a formula that can be removed from the sequent. This avoids creating an unnecessary step in the proof. We could use thin**-tac instead to do a more thorough job of removing formulas, though at a higher cost.

The neg2-$1$-tac, neg2-$r$-tac, pushneg-$1$-tac, and pushneg-$r$-tac tactics appear next in the list. They are one last attempt to avoid violating the constraint of keeping one formula on the right of the sequent arrow. Rather than moving a negated formula from the right to the left, or vice versa, we may be able to replace a formula in its place with an equivalent one. If it is a negated quantified formula, it may then be possible to do a substitution after moving the negation past the
3.3 Integrating an Automatic Theorem Prover

quantifier.

If none of the rules discussed so far have succeeded, at this point the choice is to add or remove a formula from the right. We first attempt to remove one, creating a proof by contradiction. The indirect*-tac tactic is attempted and will add a negated formula on the left. We then apply pushneg-1-tac to push the negation past the quantifier. The neg-r-tac rule is also a form of proof by contradiction, so this is attempted next. It is dual to indirect*-tac since it removes a negation.

Finally, we are left with no choice but to create more than one formula on the right. The or-r-tac tactic is attempted, followed by thin*-tac which may possibly remove a formula from the right. Finally, if no other rule is applicable the neg-1-tac tactic is attempted.

The proof of correctness of this algorithm is fairly straightforward. (See [Miller 83] and the discussion in Section 2.2.2.) The basic idea is that the procedure contains at least one inference rule for every connective, and is structured so that at least one of them will apply to any generalized sequent that is not an axiom.

Fully Integrated Interactive and Automatic Theorem Proving It was illustrated in Section 3.2 that partial automation to varying degrees is possible by writing proof procedures. This degree depends on the contents and degree of sophistication of the proof procedure, as well as the amount of information in the generalized sequent. For example, a more refined generalized sequent may contain expansion trees with expansion terms. This allows the quantifier rules that need this substitution information to be applied automatically.

The atp tactic provides another kind of integration of interactive and automatic theorem proving. At any point in the theorem proving process, a user may call this tactic on a particular subproof. The sequent will first be passed to an automatic resolution or expansion tree prover. A resolution proof will be transformed to an ET-proof and mating. Then the complete-transform-1 tactic will transform the ET-proof to an LK+ proof. This proof term representing the subproof will then be integrated into the larger proof that the user is working on.

A user also has the option of calling the auto-tac tactic, obtaining an ET-proof and mating for the sequent, but instead of calling the complete-transform-1 tactic immediately at this point, s/he may choose to interactively apply inference rules (i.e. interactively apply rules that operate in "automatic" or "transformation" mode). In this case, the user is actually guiding the transformation process, rather than proving a theorem. This gives the user complete control in building the LK+ proof tree.
3.4 Revising Proofs

Once a proof has been completed either interactively, automatically, or some combination of both, the LK+ proof term can be passed to the proof revision component of $\chi$. The top level algorithm of this component includes transforming the LK+ proof to an ET-proof and mating, performing some logical analysis on these structures, then transforming the revised ET-proof and mating back to an LK+ proof. The current version of the revision algorithm transforms an LK+ proof term to an ET-proof and mating using the transformation algorithm described in Section 2.2.3. Notice that by considering the LK+ proof term to be a function, we can simply evaluate it to get the ET-proof. For example, a term of the form $\text{and}\cdot r(T_1, T_2)$ would be a call to the $\text{and}\cdot r$ transformation function described in Section 2.2.3. The terms $T_1$ and $T_2$ (the arguments to the function) would then be evaluated, resulting in ET-proofs which would be passed as input to the $\text{and}\cdot r$ function. The completed ET-proof and mating form the generalized sequent which is used as input to a call to the $\text{complete}\cdot \text{transform}\cdot 1$ tactic. The resulting LK+ proof term is the revised proof.

We can also view this revision within the typing paradigm. The LK+ to ET-proof transformation can be viewed as type inferencing i.e. figuring out the type (generalized sequent) of the value (LK+ proof term). We know in advance that the generalized sequent specifying the type will be an ET-proof since we are starting with a complete LK+ proof term and using a transformation algorithm that transforms a proof in one system to a proof in another. The converse transformation, from ET-proof to LK+ proof can be viewed as the process of searching for a value of a given type. The value we end up with will most likely be different from the value we started with, but this is to be expected because we are looking for a “better” value (since the purpose is to revise).

This algorithm does not include any analysis on the ET-proof and mating. Further investigation is necessary to determine what kinds of analysis will be useful for proof revision, but we will illustrate by example that the transformations alone can result in substantial revision. This is because the LK+ proof term stores more details about a proof than an expansion tree. For example, the order of application of inference rules is explicit in the term. Expansion trees are a much more compact representation and by transforming the LK+ proof, we get rid of many of the unnecessary details. Then a well designed $\text{complete}\cdot \text{transform}\cdot 1$ tactic can build an LK+ proof tree from this ET-proof that is as natural and compact as our tactics and tacticals will allow.

The transformations preserve the substitution and the mating in the original proof. For example, any axiom in the revised LK+ proof will have one member of a mated pair on each side of the sequent arrow. Since each mated pair is formed by mating atoms occurring as axioms in the original proof, the mating information
in the original proof is preserved. In revising proofs, it is possible that altering the mating during logical analysis may result in better proofs in some cases. Again, more investigation is needed.

**Example 14** Suppose a user constructs the following proof of

\[
\text{i} \rightarrow p(a) \land \forall x \ [P(x) \supset Q(x)] \supset \exists x \ Q(x).
\]

\[
\begin{array}{c}
\frac{q(a) \rightarrow q(a)}{p(a) \rightarrow p(a)} \quad \text{thin*} \\
\frac{q(a), p(a) \rightarrow p(b), q(a)}{q(a), p(a) \rightarrow p(b), \exists x \ q(x)} \quad \exists \text{-R} \\
\frac{p(a), p(a) \supset q(a) \rightarrow p(b), \exists x \ q(x)}{p(a), p(a) \supset q(a) \rightarrow p(b), q(x)} \quad \forall \text{-L} \\
\frac{q(b) \rightarrow q(b)}{q(b), p(a) \supset q(a) \rightarrow q(b)} \quad \exists \text{-R}
\end{array}
\]

The corresponding LK+ proof term is:

\[
\text{implies}_r\text{(and}_1\text{(contract}_1\text{(forall}_1\text{(implies}_1\text{(forall}_1\text{(implies}_1\text{(axiom}(p(a)),} \\
\text{exists}_r\text{(thin*}(\text{axiom}(q(a))))),} \\
\text{implies}_1\text{(exists}_r\text{(thin*}(q(a))))))\]

The LK+ to ET transformation algorithm transforms this term into the following ET-proof:

\[
p(a)_1 \land (\forall x \ [p(x) \supset q(x)], (a, p(a)_2 \supset q(a)_1), (b, p(b)_1 \supset q(b)_1)) \\
\supset (\exists x \ q(x), (a, q(a)_2), (b, q(b)_2))
\]

and mating:

\[
\{\{p(a)_1, p(a)_2\}, \{q(a)_1, q(a)_2\}, \{q(b)_1, q(b)_2\}\}.
\]

The complete-transform-1 tactic will produce the following proof which is a much shorter and more readable proof of the sequent.
3.4 Revising Proofs

The pair \{q(b)_1, q(b)_2\} is a mated pair, but does not show up as an axiom in the revised proof. By examining how the complete-transform-1 tactic operates on the ET-proof, we see why this occurs. The LK+ proof term constructed is:

\[
\begin{align*}
P(a) & \implies P(a) & \quad Q(a) & \implies Q(a) & \quad \exists\text{-}L \\
P(a), P(a) \supset Q(a) & \implies Q(a) & \quad \forall\text{-}L \\
P(a), \forall x [P(x) \supset Q(x)] & \implies Q(a) & \quad \exists\text{-}R \\
P(a), \forall x [P(x) \supset Q(x)] & \implies \exists x Q(x) & \quad \wedge\text{-}L \\
P(a) & \land \forall x [P(x) \supset Q(x)] & \implies \exists x Q(x) & \quad \exists\text{-}R \\
\rightarrow & \quad P(a) & \land \forall x [P(x) \supset Q(x)] & \supset \exists x Q(x) & \quad \supset\text{-}R
\end{align*}
\]

Notice that it includes thin*, though no formulas are removed from the LK+ proof. This occurs in the call to \(\text{then* (repeat and-1-tac) thin*}\). The generalized sequent returned from this call to and-1-tac is:

\[
\begin{align*}
\text{implies}_r & (\text{and}_1)(\text{thin*})(\text{exists}_r)(\text{forall}_1)(\text{implies}_l(\text{axiom}(p(a))), \\
& \quad \text{axiom}(q(a)))))))
\end{align*}
\]

At this point thin* is called. It finds that the equivalence class corresponding to the formulas \(p(b)_1 \supset q(b)_1\) and \(q(b)_2\) is not spanned by the mating, thus these formulas are removed from the ET-proof resulting in:

\[
\begin{align*}
p(a)_1, (\forall x [p(x) \supset q(x)]), (a, p(a)_2 \supset q(a)_1), (b, p(b)_1 \supset q(b)_1) & \rightarrow \\
& \quad (\exists x q(x), (a, q(a)_2), (b, q(b)_2)).
\end{align*}
\]

By eliminating these branches from the ET-proof, we greatly simplify the remainder of the LK+ proof.

Another common revision is the elimination of the repetition of a particular subproof in several branches of the tree. Repetition of a subproof occurs when the application of a branching LK+ rule is applied too early in the proof process. This may cause a particular series of rules to have to be applied in both branches (or possibly more than twice when several branching rules are applied too early). This can often be eliminated by applying this particular series of inference rules before the branching rules. The revision algorithm accomplishes this by, first, during the transformation from LK+ proof to ET-proof, merging the copies of the expansion trees for the repeated series of rules, and second, ordering the application of LK+ rules in the complete-transform-1 tactic so that non-branching rules are attempted before branching rules. Strategies such as this were discussed in detail in Section 3.3.
4 Applications

We have been stressing the goals of building natural proofs and storing them as explicit objects so that they can be manipulated in many significant ways. To meet these goals we have developed a theorem proving system with several distinctive characteristics. It is a system in which we explicitly represent proofs and partial proofs in several different proof systems, we integrate these different proofs by performing transformations among them, and we provide a high level programming language approach which facilitates the integration of interactive and automatic theorem proving. As a result it is a system with many characteristics and applications not generally found in other theorem proving systems. We discuss some of the applications here.

The applications are divided into two categories. The first section discusses the proof objects and manipulations that can be performed on them. The second describes some applications of the system as a whole.

4.1 Applications of Proofs as Objects

The most direct advantage of storing proofs is the capability of organizing completed proofs into proof libraries. In this type of library, instead of simply storing the statement of each theorem and something to indicate that it was proven, we store the theorem and its entire proof. Since our proof representation includes partial proofs as well as completed proofs, we have the option of including entries in the library for proofs that are not yet finished. When a user does not complete a proof, s/he can avoid losing the work that has been done, by storing the incomplete proof as an entry in the library. S/he can then continue to work on it at any time by accessing and updating this entry.

Storing proofs and partial proofs explicitly also allows us to extend proof libraries by combining proofs. For example, one small proof may be a subproof of several larger proofs. This small proof can be constructed once and combined with larger proofs when needed, avoiding repeated construction of proofs of the same subtheorem (or lemma). Here the type information makes it clear when such a smaller proof can be used in constructing larger proofs. The type of the smaller proof is matched with the type of the missing subproof in the larger proof before combining will take place.

Since proof terms represent LK+ proof trees, one very straightforward manipulation is to reconstruct these trees from the information contained in the term. To do this we use the same technique found in Section 3.4. In that section, to obtain an ET-proof from an LK+ proof term, we interpreted the term as a function and obtained an ET-proof by evaluating it. We apply the same principle here. In this
4.1 Applications of Proofs as Objects

case, a term of the form \texttt{and}\_r(T_1, T_2), for example, is a call to an \texttt{and}\_r function which will, first, by evaluating its arguments, present the subproofs \(T_1\) and \(T_2\), and second, construct and present the conclusion of the \&-R rule joining these subtrees. The ability to evaluate a proof term in this way gives the user access to a readable form of a (complete or partial) proof tree at any time.

Both the LK+ proof to ET-proof transformation algorithm and the proof presentation algorithm illustrate a general type of manipulation that may be exploited in other ways. For each LK+ inference rule, we can define a corresponding function that will perform a certain operation depending on the particular application. Then by regarding the term as a function, it can be evaluated in an environment where the individual functions are appropriately defined. In this type of manipulation the arguments to each rule (function) will be evaluated first, recursively traversing the tree from the axioms to the root.

Representing ET-proofs as objects (in the form of generalized sequents) gives us the capability to “browse” through proofs. This was alluded to in Section 3.3, where it was mentioned that a user can interactively guide the construction of an LK+ proof tree for a generalized sequent that contains both an ET-proof and mating (i.e. interactively guide the transformation from ET-proof to LK+ proof). “Browsing” through a proof falls somewhere between the extremes of interactively constructing a proof of a generalized sequent that is not already an ET-proof, and calling an automatic transformation tactic to completely transform an ET-proof to an LK+ proof term. The user is given control in making some decisions about the order in which rules are applied, but is restricted by the information in the ET-proof and mating. For example, only substitution terms present in the tree can be used, and only when they are admissible. Also, all axioms must result from mated pairs. By browsing through a proof, the user is given the opportunity to explore the proof at will to gain a better understanding of its contents.

Building explanations from proofs is yet another manipulation on LK+ proof terms. We have mentioned the explanation component of \(\chi\) and though it has not been developed, it has had significant influence on the overall design of the system. In particular, this influence falls under the general goal of building natural and understandable proofs. Thus far, to achieve this goal we have concentrated on building natural proofs within a formal natural deduction system, in this case, LK+. We’d like to take this one step further and build readable natural language explanations from natural deduction proofs. Some early work in this area can be found in [Chester 76]. These proof explanations should be understandable to those not familiar with such formal proof systems. We hope the design of LK+ will facilitate this task. We saw in Section 2.1.1 in the discussion of the individual inference rules, that most had a simple natural language explanation for why the conclusion followed from the premise(s). Exploiting this property to automatically
generate explanations from LK+ proof terms is currently under investigation.

There are often many proofs for a given sequent, and thus there will be a corresponding explanation for each one. The programming language approach allows us to easily obtain several proofs by making use of different proof tactics written in the language of tactics and tacticals. This allows us to enhance understanding by giving several explanations of the same theorem. It also allows us to tailor our explanations to the needs of a specific users by designing tactics to handle different rhetorical aspects.

Another proof manipulation involves the automatic conversion of proofs of a certain (constructive) kind to executable programs, as was done in the PRL and NuPRL systems in [Bates & Constable 85] and [Constable et. al. 86]. This has not yet been examined in our context.

In this discussion, we have described many applications that result from the explicit representation of proofs, some that have been implemented, and some that have not. This list is not complete but illustrates the wide range of applicability of proofs-as-objects.

4.2 Applications of the System as a Whole

In this section we present some applications in which we foresee the $\chi$ system as a whole to be used. These applications illustrate that $\chi$ can be a useful tool in several different contexts.

One possibility is to use the $\chi$ system as a logic tutor, to aid students in learning formal proof systems. When using the system, the logic student will work within the LK+ system, using the interactive component to learn to construct LK+ proof trees. Access to the automatic theorem prover through the use of the atp tactic provides the student with an extensive help facility. When s/he is working on a theorem and does not see how to proceed with a particular subproof, the atp tactic can be called to automatically generate this proof. Then, after it is transformed to an LK+ proof, it can be presented to the student to illustrate how s/he could have proceeded. After working with the system for awhile, the student may attain a certain level of knowledge that allows him/her to make use of the programming language to write compound tactics that will automate the construction of certain parts of the proof trees. The instructor may also use this language to write tactics that provide the student with additional tools for learning proof techniques. As another teaching aid, the instructor may provide, in addition to complete-transform-1, other “complete” compound tactics that generate a complete LK+ proof tree from an ET-proof. Using these procedures, several different proofs of the same theorem can be built to enhance a user’s understanding of a theorem, or to customize the presentation of a theorem and its proof for possibly very different users.
Another quite different application of the \( \chi \) system is to employ it as an explanation and enhanced reasoning facility for existing AI systems that have a deductive reasoning component. Often such systems use a resolution theorem prover to fill this role. When a proof is needed, the resolution prover is called upon to automatically generate a resolution refutation. It will then return a yes or no answer indicating whether or not it was successful. In this setting the user is offered little assistance in understanding why a particular theorem or statement is true. It is also very difficult for a user to become involved in the reasoning process. The \( \chi \) system could greatly enhance the reasoning environment. First of all, it can provide the user with some insight into the contents of such an automatically generated resolution refutation by transforming it to an ET-proof and mating, then to an LK+ proof. This LK+ proof could be presented to the user upon request providing some explanation of the contents of the proof. Hopefully, these explanations will eventually be some sort of natural language description of the proofs. Secondly, \( \chi \) provides a way for the user to become involved in the reasoning process. Instead of immediately calling the resolution prover, a user can call the interactive theorem prover and guide the construction of a proof, making use of the automatic theorem prover if desired, through the \texttt{atp} tactic. The user will have the full power of the programming language of tactics and tacticals, and thus can write and call upon compound tactics to perform various tasks in the theorem proving process. A developer of such an AI system also has full access to the programming language, and can replace the \texttt{complete-transform-1} tactic with one that is customized to meet the needs of the system and its users. A third enhancement is the ability to keep a record of how new facts are deduced. It is often desirable to store the chain of reasoning that led to a particular conclusion which resulted in the assertion of a new fact. This can be achieved by augmenting the new facts in the database with their corresponding LK+ proof terms, providing additional information which the system may make use of when needed.

The LK+ proof system was developed to facilitate the construction of natural proofs, yet there is very little about it that is central to the construction of the \( \chi \) theorem proving system as a whole. Many other equally or less formal natural deduction systems could also be supported in many of the same ways we have discussed here. In order to do this, the transformations between systems must be extended to incorporate the new system by, for example, customizing the tactics to correspond to the inference rules of the new system, or possibly adding another layer of transformation from LK+ to the alternate system. In addition to different first order proof systems, the \( \chi \) system could also be extended to other logics. For example, ET-proofs can be extended to higher-order logic [Miller 83] or modal logic [Hager 85], and the transformation algorithms appropriately modified. These extensions allow us to build, for example, a tutor for the Gentzen NK natural deduction system, or an explanation facility and enhanced reasoner for an AI system.
that uses modal logic. The ideas developed here in designing the \( \chi \) system are sufficiently general to allow us to incorporate a wider range of applications with some fairly simple modifications.
A Soundness and Completeness of LK+

We present the Gentzen LK−{cut} system as described in Gallier 86, and prove soundness and completeness of LK+ with respect to this system.

The Gentzen LK−{cut} System As in LK+, the basic structure of LK−{cut} is the sequent, written $\Gamma \rightarrow \Theta$ where $\Gamma$ and $\Theta$ are lists of formulas. Also, as in LK+, axioms in LK−{cut} are of the form $A \rightarrow A$ where $A$ is any arbitrary first-order formula. The inference rules of the LK−{cut} system are the following:

Logical Rules

\[
\frac{\Gamma \rightarrow \Delta, A \quad \Gamma \rightarrow \Delta, C}{\Gamma \rightarrow \Delta, A \land C} \quad \land\text{-R}
\]

\[
\frac{A, \Gamma \rightarrow \Delta \quad C, \Gamma \rightarrow \Delta}{A \lor C, \Gamma \rightarrow \Delta} \quad \lor\text{-L}
\]

\[
\frac{A, \Gamma \rightarrow \Delta}{A \land C, \Gamma \rightarrow \Delta} \quad \land\text{-L}
\]

\[
\frac{\Gamma \rightarrow \Delta, A}{\Gamma \rightarrow \Delta, A \lor C} \quad \lor\text{-R}
\]

\[
\frac{C, \Gamma \rightarrow \Delta}{A \land C, \Gamma \rightarrow \Delta} \quad \land\text{-R}
\]

\[
\frac{\Gamma \rightarrow \Delta, A \quad C, \Lambda \rightarrow \Theta}{A \supset C, \Gamma, \Lambda \rightarrow \Delta, \Theta} \quad \supset\text{-L}
\]

\[
\frac{A, \Gamma \rightarrow \Delta, C}{\Gamma \rightarrow \Delta, A \supset C} \quad \supset\text{-R}
\]

\[
\frac{\Gamma \rightarrow \Delta, A \quad \neg A, \Gamma \rightarrow \Delta}{\neg\text{-L}}
\]

\[
\frac{A, \Gamma \rightarrow \Delta}{\Gamma \rightarrow \Delta, \neg A} \quad \neg\text{-R}
\]

\[
\frac{[x/t]P, \Gamma \rightarrow \Delta}{\forall x \ P, \Gamma \rightarrow \Delta} \quad \forall\text{-L}
\]

\[
\frac{\forall x \ P, \Gamma \rightarrow \Delta}{\Gamma \rightarrow \Delta, \exists x \ P} \quad \exists\text{-R}
\]

\[
\frac{[x/y]P, \Gamma \rightarrow \Delta}{\exists x \ P, \Gamma \rightarrow \Delta} \quad \exists\text{-L}
\]

\[
\frac{\Gamma \rightarrow \Delta, [x/y]P}{\Gamma \rightarrow \Delta, \forall x \ P} \quad \forall\text{-R}
\]

Structural Rules

\[
\frac{\Gamma \rightarrow \Delta}{A, \Gamma \rightarrow \Delta} \quad \text{thin-L}
\]

\[
\frac{\Gamma \rightarrow \Delta}{\Delta, \Gamma \rightarrow \Delta} \quad \text{thin-R}
\]

\[
\frac{A, A, \Gamma \rightarrow \Delta}{A, \Gamma \rightarrow \Delta} \quad \text{contract-L}
\]

\[
\frac{\Gamma \rightarrow \Delta, A, A}{\Gamma \rightarrow \Delta, A} \quad \text{contract-R}
\]
A Soundness and Completeness of LK+

\[ \frac{\Gamma, A, C, \Delta \rightarrow \Lambda}{\Gamma, C, A, \Delta \rightarrow \Lambda} \text{ interchange-L} \]

\[ \frac{\Gamma \rightarrow \Delta, A, C, \Lambda}{\Gamma \rightarrow \Delta, C, A, \Lambda} \text{ interchange-R} \]

The proviso that the variable \( y \) is not free in any formula of the lower sequent is placed on the \( \exists \)-L and \( \forall \)-R rules.

A proof of a formula \( A \) in this system is a finite tree constructed using a series of inference rules with the sequent \( \rightarrow A \) at the root and axioms at all the leaves, (as in LK+).

**Proposition 1** (Relative Completeness of LK+) If a sequent \( \Gamma \rightarrow \Delta \) has an LK-\{cut\} proof, then it has an LK+ proof.

**Proof:** The proof is by induction on the height of the LK-\{cut\} proof tree.

Base Case: A proof tree of height 1 must be an axiom. An axiom in LK-\{cut\} is also an axiom in LK+.

Induction Case: Assume that if \( \Gamma \rightarrow \Delta \) has a proof tree of height less than \( n \), then there is an LK+ proof of \( \Gamma' \rightarrow \Delta' \) where \( \Gamma' \) is any permutation of the formulas in \( \Gamma \), and \( \Delta' \) is any permutation of the formulas in \( \Delta \). For proof trees of height \( n \), we consider the last rule of inference. Each premise of the last rule is the root of a proof tree of height less than \( n \), so by the induction hypothesis, any permutation of it can be replaced by an LK+ tree. We need to show that we can simulate each LK-\{cut\} rule in LK+ for every permutation of the formulas in the root sequent. We adopt the convention that whenever \( \Gamma \) is a set of formulas, \( \Gamma' \) is any arbitrary permutation of the formulas in \( \Gamma \).

**Case:** \( \land \)-R

We have an LK-\{cut\} proof tree as follows where \( T_1 \) is a proof tree for \( \Gamma \rightarrow \Delta, A \) and \( T_2 \) is a proof tree for \( \Gamma \rightarrow \Delta, C \).

\[ \frac{T_1}{\Gamma \rightarrow \Delta, A} \]

\[ \frac{T_2}{\Gamma \rightarrow \Delta, C} \]

\[ \frac{T_1 \quad T_2}{\Gamma \rightarrow \Delta, A \land C} \land\text{-R} \]

By the induction hypothesis, we have LK+ proof trees \( T_1' \) and \( T_2' \) for \( \Gamma' \rightarrow \Delta_1', A, \Delta_2' \) and \( \Gamma' \rightarrow \Delta_1', C, \Delta_2' \) respectively. (Here \( \Delta_1', \Delta_2' \) is any permutation of \( \Delta \).) Then by an application of the LK+ \( \land \)-R rule, we have the following LK+ proof tree.

\[ \frac{T_1' \quad T_2'}{\Gamma' \rightarrow \Delta_1', A \land C, \Delta_2'} \land\text{-R} \]

**Case:** \( \supset \)-R
A Soundness and Completeness of LK+

\[
\Gamma \rightarrow_T \Delta, A \supset C \supset -R
\]

\(T\) is an LK-\{cut\} proof tree for \(A, \Gamma \rightarrow \Delta, C\). By the induction hypothesis, we have an LK+ proof tree \(T'\) for \(A, \Gamma' \rightarrow C, \Delta'_1, \Delta'_2\) (\(\Delta'_1, \Delta'_2\) defined as before). Then by an application of the LK+ \(\supset -R\) rule, we have the following LK+ proof tree.

\[
\Gamma' \rightarrow \Delta'_1, A \supset C, \Delta'_2 \supset -R
\]

Case: \(\supset -L\)

\[
\begin{array}{c}
\Gamma \rightarrow_T \\
\Delta, A \supset C \supset -R
\end{array}
\]

\(T\) is a proof tree for \(\Delta, A\). By the induction hypothesis we have an LK+ proof tree \(T'\) for \(\Delta', A\). We can construct the following LK+ proof tree.

\[
\begin{array}{c}
\Gamma' \rightarrow_T \\
\Delta'_1, A \supset C, \Delta'_2 \supset -R
\end{array}
\]

Case: \(\supset -L\)

This case is similar to \(\supset -L\).

Case: \(\supset -L\)

\[
\begin{array}{c}
\Delta, A \supset C, \Gamma, \Lambda \rightarrow_T \\
\Theta \supset -L
\end{array}
\]

\(T_1\) is a proof tree for \(\Gamma \rightarrow \Delta, A\), and \(T_2\) is a proof tree for \(C, \Lambda \rightarrow \Theta\). By the induction hypothesis we have proof trees \(T'_1\) and \(T'_2\) for \(\Gamma' \rightarrow \Delta', A\) and \(C, \Lambda' \rightarrow \Theta'\). We can construct the following LK+ proof tree.

\[
\begin{array}{c}
\Sigma' \rightarrow_T A, \Phi' \supset \text{thin*} \\
\Sigma_1, A \supset C, \Sigma'_2 \rightarrow \Phi' \supset \text{thin*}
\end{array}
\]

\[
\begin{array}{c}
\Sigma', C, \Sigma' \rightarrow_T \Phi' \supset \text{thin*} \\
\Phi' \supset -L
\end{array}
\]
A Soundness and Completeness of LK+  

Here $\Sigma'$ is any arbitrary join of $\Gamma'$ and $\Lambda'$ where the ordering of formulas within $\Gamma'$ and $\Lambda'$ remain the same. Similarly $\Phi'$ is a join of $\Delta'$ and $\Theta'$. Also, $\Sigma'_1, \Sigma'_2$ is equal to $\Sigma'$.

Case: interchange-L and interchange-R

$$\begin{align*}
\Delta, A, C, \Delta & \rightarrow \Theta \quad \text{interchange-L} \\
\Gamma, C, A, \Delta & \rightarrow \Theta \quad \text{interchange-R}
\end{align*}$$

This case follows directly from the induction hypothesis.

**Proposition 2 (Soundness of LK+)** If there is an LK+ proof tree for a sequent $\Gamma \rightarrow \Delta$, then the sequent is valid, i.e. $\forall \Gamma \exists \forall \Delta$ is valid.

**Proof:** The proof is by induction on the height of the LK+ proof tree.

Base Case: A proof tree of height 1 must be an axiom of the form $A \rightarrow A$ which is valid.

Induction Case: Assume that if $\Gamma \rightarrow \Delta$ has an LK+ proof tree of height less than $n$, then it is valid. Again we consider the last rule of inference in a proof tree of height $n$. For the introduction, structural, and some of the additional rules, we show that the LK+ rule can be simulated in LK-{cut}. For the remaining rules, we show that if the premises are valid, then the conclusion must be valid. We adopt the convention that an "*" after the name of a rule in a proof tree means 0 or more applications of that rule.

Case: $\wedge$-R

We have a proof tree in LK+ as follows where $T_1$ and $T_2$ are a proof trees for $\Gamma \rightarrow \Delta, A, \Theta$ and $\Gamma \rightarrow \Delta, C, \Theta$ respectively.

$$\begin{array}{c}
\Gamma \rightarrow \Delta, A \wedge C, \Theta \\
\hline
T_1 \quad T_2
\end{array} \quad \wedge-R$$

By the induction hypothesis, $\Gamma \rightarrow \Delta, A, \Theta$ and $\Gamma \rightarrow \Delta, C, \Theta$ are valid, and thus have LK-{cut} proof trees $T_1'$ and $T_2'$, respectively. We can simulate the $\wedge$-R LK+ rule in LK-{cut}.

$$\begin{align*}
\Delta, A, \Theta & \rightarrow \Delta, C, \Theta \\
\Delta, A \wedge C, \Theta & \rightarrow \Delta, A \wedge C, \Theta
\end{align*} \quad \text{interchange*}$$

By soundness of LK-{cut}, $\Gamma \rightarrow \Delta, A \wedge C, \Theta$ is valid.

These cases are similar to $\wedge$-R since the LK+ rules can be simulated in LK-{cut} with a series of interchanges before and after the rule application.

Case: $\forall$-R

$$\frac{T}{\Gamma \rightarrow \Delta, A \lor C, \Theta} \text{ } \forall$-R$$

$T$ is an LK+ proof tree for $\Gamma \rightarrow \Delta, A, C, \Theta$. Let $T'$ be an LK-\{cut\} proof tree for this sequent. Then the following is an LK-\{cut\} proof for $\Gamma \rightarrow \Delta, A \lor C, \Theta$.

$$\frac{T'}{\Gamma \rightarrow \Delta, C, \Theta, A} \text{ } \forall$-R$$
$$\frac{\Gamma \rightarrow \Delta, C, \Theta, A \lor C}{\Gamma \rightarrow \Delta, C, \Theta, A \lor C} \text{ } \forall$-R$$
$$\frac{\Gamma \rightarrow \Delta, \Theta, A \lor C, C}{\Gamma \rightarrow \Delta, \Theta, A \lor C, C} \text{ } \forall$-R$$
$$\frac{\Gamma \rightarrow \Delta, \Theta, A \lor C, A \lor C}{\Gamma \rightarrow \Delta, \Theta, A \lor C, C} \text{ } \text{contract-R}$$
$$\frac{\Gamma \rightarrow \Delta, \Theta, A \lor C}{\Gamma \rightarrow \Delta, A \lor C, \Theta} \text{ } \text{interchange}^*$$

Case: $\wedge$-L This case is similar to $\forall$-R.

Case: $\exists$-L

$$\frac{T_1 \quad T_2}{\Gamma, A \supset C, \Delta \rightarrow \Theta} \exists$-L$$

$T_1$ and $T_2$ are LK+ proof trees for $\Gamma, \Delta \rightarrow A, \Theta$ and $C, \Gamma, \Delta \rightarrow \Theta$ respectively. Let $T'_1$ and $T'_2$ be LK-\{cut\} proof trees for the same sequents respectively. The following is an LK-\{cut\} proof for $\Gamma, A \supset C, \Delta \rightarrow \Theta$.

$$\frac{T_1'}{\Gamma, \Delta \rightarrow \Theta, A} \text{ } \text{interchange}^*$$
$$\frac{T_1'}{\Gamma, \Delta \rightarrow \Theta} \exists$-L$$
$$\frac{T_2'}{A \supset C, \Gamma, \Delta, \Gamma, \Delta \rightarrow \Theta, \Theta} \text{ } \text{interchange}^*, \text{contract}^*$$

Case: $\supset$-L*, backchain, forwardchain, positive, contrapos

These cases vary from $\supset$-L only in which formulas must be contracted in the LK-\{cut\} simulation of the LK+ rule.
A Soundness and Completeness of $LK^+$

Case: $\neg\forall$-L

\[ \frac{T}{\Gamma, \forall x P, \Delta \rightarrow \Theta} \neg\forall$-L \]

$T$ is an $LK^+$ proof tree for $\Gamma, \exists x \neg P, \Delta \rightarrow \Theta$. By the induction hypothesis this sequent is valid. In applying this rule we simply replace the formula $\exists x \neg P$ in the premise with the logically equivalent formula $\neg \forall x P$. Hence, the conclusion is valid also.

Case: $\neg\exists$-L, $\neg\forall$-R, $\neg\exists$-R, $\neg\neg$-L, $\neg\neg$-R These cases are similar to $\neg\forall$-L.

Case: indirect

\[ \frac{T}{\Gamma \rightarrow A \text{ indirect}} \]
\[ \frac{T'}{\Gamma \rightarrow \neg \neg A \neg$-R \]

$T$ is an $LK^+$ proof tree for $\neg A, \Gamma \rightarrow$ which is valid by the induction hypothesis. $T'$ is an $LK-\{\text{cut}\}$ proof tree for this sequent. By an application of the $\neg$-R rule, we know that $\Gamma \rightarrow \neg \neg A$ is valid. Since this is logically equivalent to $\Gamma \rightarrow A$ this sequent is also valid.

Case: thin*

This rule is simply several applications of the thin-L and thin-R rules which were shown to be sound.


Proof: Completeness follows from the fact that these were the only $LK^+$ rules used to build $LK^+$ subtrees equivalent to $LK-\{\text{cut}\}$ inference rules in Proposition 1. Soundness follows directly from Proposition 2.
References


REFERENCES


