PROOF EXPLANATION AND REVISION

Amy Felty and Dale Miller
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

March 1987

Abstract

Proof structures in traditional automatic theorem proving systems are generally designed for efficiently supporting certain search strategies. They are not meant as a useful representation or presentation of complete proofs: usually only the experts who designed such systems can read them. As a result, complete proofs are of little value and are generally discarded. The failure of such systems to manipulate proofs as values of their own right is one reason why theorem proving systems have not found more use in artificial intelligence and mathematical software. In this paper, we present the design of the *χ-proof system* which attempts to correct this failure of theorem proving systems. Proofs in χ are represented by natural and flexible tree-structured deductions. These deductions make very good presentations of proofs, and χ has a very simple mechanism for *lexicalizing* them into readable natural language text. Other proof structures, such as resolution refutations, are used to provide the information necessary for building such proof trees. A programming language based on an extension of LCF tactics and tacticals is available for writing programs which manipulate proof trees. Such programs include interactive proof editors and fully automatic theorem provers. Finally, χ is capable of making substantial changes in the presentation of proofs: proofs can be revised or restructured in order to present their deductions in different styles. For example, proofs which contain uses of the indirect proof method can occasionally be automatically restructured into direct proofs.

Available as University of Pennsylvania Technical Report MS-CIS-88-17, LINC LAB 104. This work has been supported by NSF AI Center grants NSF-MCS-83-05221, US Army Research office grant ARO-DAA29-84-9-0027, and DARPA N000-14-85-K-0018.

1 Introduction

The CHI or χ-proof system is a framework for developing and testing procedures which manipulate proofs in classical logic. CHI is an acronym for the Curry-Howard Isomorphism, a proof-theoretic concept, also know as *formulas-as-type* [8], which provides a very simple and elegant approach to the understanding of proofs as first-class, typed data structures. This concept inspired several aspects of this proof system. The current implementation of χ was built by the first author to test the ideas of her Master's thesis [2]. This system is currently implemented in Common Lisp and all the examples described in this paper were generated using this implementation. This paper is not, however, a description of this specific implementation. We intend to present some very general issues about the design of proof systems — at least for various forms of classical logic.

Although we shall limit ourselves to proofs of classical first-order logic in this paper, it is known that many of the technical devices at the heart of χ (expansion trees and matings, for example) can be extended to several other logics. For example, the papers [4, 7, 9] provide the theoretical foundations necessary to extend χ to first-order equational logic, modal logic, and higher-order logic, respectively.

Numerous proof systems have recently been developed and used in the area of program synthesis. These systems generally formalize higher-order, intuitionistic logic and manipulate proofs in this logic to obtain their *constructive content.* Such systems, for example Nuprl [1], treat proofs as objects and contain a wide range of procedures for computing with them. One such operation is that of extracting programs from proofs. While χ shares some similarities with such proof systems, the aspects of χ described in this paper deal with proofs in classical logic: such proofs are not generally specifications of programs. Instead, we are concerned with the *presentation* of proofs; particularly, their presentation to human readers.

2 Sequential Trees

Most traditional theorem proving systems represent proofs using such structures as resolution refutations, connection graphs, or general matings. These structures were designed to support particular kinds of automatic search paradigms efficiently. They are often constructed using such technical devices as Skolem normal and disjunctive normal forms and contain unnatural inference rules which attempt to prove contradictions. As a result, generally only the experts who constructed the theorem prover are able to understand the actual structure of such proofs. Such proofs are generally discarded and the only explanation of the theorem proving process is simply a yes or no answer indicating whether or not the theorem prover found a proof. Occasionally, "answer substitutions" accumulated in building a proof are extracted and presented to a user.

The χ system employs a *sequential* variant of *natural deduction* as one representation of proofs. Proofs in a sequential system are tree structure arrangements of deductions and can be designed to represent rich forms of formal deductions. Nodes in such trees are *sequents* and these are connected by *inference rules.* A sequent is a pair, written as $\Gamma \longrightarrow A$, where Γ is a list of formulas and A is single formula. Sequents are intended to represent the proposition "from the formulas in Γ, A can be proved." Since the proposition intended by the sequent $A \longrightarrow A$, where A is any arbitrary formula, is obviously true, this sequent is taken as an *axiom*. There are, in general, many other obvious sequents which could also be accepted as axioms. The actual set of axioms used would depend on criteria such as the domain in which proofs are being constructed or the degree of expertise of the user. Trees of sequents are constructed by applying inference rules: the conclusion sequent of a particular rule is the parent of one or more premise sequents. A proof of a formula A is a finite tree constructed in this way with the sequent $\longrightarrow A$ at the root and axioms at all the leaves.

Sequential proof systems exist for many logics and have well understood metatheories. Gentzen in [5] presented such a system, called LK, and proved it to be sound and complete for first-order classical logic (see also [3]). In this paper, we assume that most of Gentzen's LK inference rules are available as well as several additional derived inference rules which are described in [2]. In principle, sequential systems allow the flexibility to include many different kinds of inference rules. Some of these rules will be general in the sense that for any reader (human or machine), the conclusion will follow immediately from the premise(s). Other rules may be intended for experts and have interpretations that are not as straightforward. Our examples in this paper will use only general inference rules although the actual collection of inference rules depends on the logic being formalized and the kind of presentation of proofs desired. Below we illustrate four of these general inference rules, namely conjunction, case analysis, modus ponens, and indirect proof.

Consider the conjunction rule, here called `and_r`, for $\wedge$ introduction on the right.

$$\frac{\Gamma \longrightarrow B \qquad \Gamma \longrightarrow C}{\Gamma \longrightarrow B \wedge C}\,\texttt{and\_r}$$

Given the interpretation of sequents as propositions, this inference rule has the interpretation: if from Γ we can prove B, and from Γ we can prove C, then from Γ we have proved $B \wedge C$. Note that each rule in a sequential system is named and this name is attached to the line between the premise(s) and conclusion.

Case analysis is often used in proofs when it has been assumed or is known that for some formulas B and C, either B or C must be true. This inference rule requires two subproofs: B is assumed in one and C is assumed in the other. This is represented by the following `or_l` rule:

$$\frac{B, \Gamma \longrightarrow A \qquad C, \Gamma \longrightarrow A}{B \vee C, \Gamma \longrightarrow A}\,\texttt{or\_l}$$

This rule is interpreted as: if we can prove A from B and Γ, and we can prove A from C and Γ, then we have proved A from $B \vee C$ and Γ.

Modus ponens is involved in the following inference rule:

$$\frac{\Gamma \longrightarrow B \qquad C, \Gamma \longrightarrow A}{B \supset C, \Gamma \longrightarrow A}\,\texttt{positive}$$

The interpretation of `positive` is: if we can prove B from Γ and A from C and Γ, then we have proved A from $B \supset C$ and Γ. Modus ponens is employed to conclude C from B and

$B \supset C$.

The indirect proof method is represented by the inference rule:

$$\frac{\neg A, \Gamma \longrightarrow \bot}{\Gamma \longrightarrow A}\,\texttt{indirect}$$

Hence, if assuming $\neg A$ along with Γ leads to a contradiction (denoted by the special formula $\bot$), we have proved A from Γ.

χ represents these sequential proof trees by term structures. The axiom sequent, $A \longrightarrow A$, for example, is represented by the simple term `axiom(A)`. Each inference rule is represented by a function symbol with the same name. This function symbol has one argument position for each subproof proving its premises. Occasionally, additional arguments are needed to retain information such as the substitution terms needed for correctly representing quantifier rules. For the sake of compactness, such extra arguments will be suppressed in this paper. The inference rule `and_r`, for example, is represented by a binary function symbol `and_r` such that if T_1 and T_2 are the term representations of proof trees for $\Gamma \longrightarrow B$ and $\Gamma \longrightarrow C$ respectively, then `and_r`(T_1, T_2) represents a proof tree for $\Gamma \longrightarrow B \wedge C$.

Example 1 Consider the theorem which states that if R is a symmetric and transitive relation, it is reflexive on its domain. This can be expressed by the first order formula $\forall x \forall y \forall z\ (R(x,y) \wedge R(y,z) \supset R(x,z)) \wedge \forall x \forall y\ (R(x,y) \supset R(y,x)) \supset \forall x (\exists y\ R(x,y) \supset R(x,x))$. The following term represents a sequential proof of this formula.

```
implies_r(and_l(forall_r(implies_r(exists_l(forall_l(forall_l(forall_l
    (forall_l(forall_l(positive(and_r(positive(axiom(R(a,b)),
                                                  thin(axiom(R(b,a)))),
                                    thin(axiom(R(a,b)))),
                              thin(axiom(R(a,a))))))))))))))
```

Although such a sequential proof is a faithful representation of the tree structured deduction needed to establish this theorem, terms such as these are certainly not easy to read. As we now show, however, these proof terms can be lexicalized by a very simple mechanism to yield readable natural language text.

3 Generation of Natural Language from Sequential Trees

Along with each inference rule presented in the preceding section, we gave its corresponding natural language interpretation. For example, `or_l` reads: if we can prove A from Γ and B, and we can prove A from Γ and C, then we have proved A from Γ and $B \vee C$. In this section we present a simple algorithm which can string these kinds of interpretations for individual inference rules into readable text for an entire proof. We will only be concerned with "lexicalizing" inference rules. Formulas themselves will be left as formulas. Of course, if formulas were also lexicalized, the following presentations of proofs would be at times very readable.

A proof term can be viewed functionally: that is, each inference rule in the term can be thought of as being a function from text to text. Under this interpretation, a proof term for a sequent would be interpreted as a textual argument for the proposition represented by that sequent. For example, a term of the form `or_l`(T_1, T_2) represents a proof using case analysis. Assume that T_1 is interpreted as $text_1$ which argues that A follows from B and Γ, and that T_2 is interpreted as $text_2$ which argues that A follows from C and Γ. The interpretation of `or_l`(T_1, T_2) would then need to be an argument that A follows from $B \vee C$ and Γ. This is easily done if we make the interpretation of `or_l` be the function which takes $text_1$ and $text_2$ into the following text:

> We have two cases. Case 1: Assume B. $text_1$ Case 2: Assume C. $text_2$ Thus, in either case, we have A.

Other rules are not as "wordy" as `or_l`: there are inference rules such as `and_l` that do not contribute anything to an explanation. Formally this rule is written as:

$$\frac{B, C, \Gamma \longrightarrow A}{B \wedge C, \Gamma \longrightarrow A}\ \mathtt{and\_l}$$

These two sequents are conceptually the same because the comma and the $\wedge$ connective have the same meaning on the left of the sequent. The rule `and_l` is, therefore, interperted as the identity function on text. All inference rules can be given such functional interpretations. To handle the base case in this interpretation scheme, we need to attach to axioms some text. This is very simple: no words are needed to argue that A follows from A. The interpretation of the term `axiom(A)` is simply the empty text string.

Example 2 Putting all these elements together, the large proof term in Example 1 could be directly lexicalized into the following English language proof.

> Assume $\forall x \forall y \forall z (R(x,y) \wedge R(y,z) \supset R(x,z)) \wedge \forall x \forall y (R(x,y) \supset R(y,x))$. Assume $\exists y\ R(a,y)$. Choose b such that $R(a,b)$. By modus ponens, we have $R(b,a)$. Hence, $R(a,b) \wedge R(b,a)$. By modus ponens, we have $R(a,a)$. Since a was arbitrary, we have $\forall x\ (\exists y\ R(x,y) \supset R(x,x))$.

Clearly, if we choose to lexicalize formulas by associating strings with formulas, for example, associate "R is transitive" with the first assumption, we could obtain very readable proofs. While explanations are obtained directly from sequential proof trees and have the same general structure, they are much easier to read. For this reason, we will present only the English explanation of example proofs in the rest of this paper. The relevant parts of the underlying sequential proof trees should be easily discernible from the text.

One task which can be attempted in χ is that of generating "good" natural language text for different readers and different purposes. Since the mechanism for translating a proof tree into text is so simple, much of the challenge in constructing natural text can be transferred to constructing proof trees: to first generate good text, generate good proof terms. There are several criteria for judging the quality of explanations of proofs. One such criterion is how focused and coherent the flow of the explanation is maintained: we have not examined this aspect of constructing proof terms. Another criterion is whether or not the proof contains natural uses of interesting inference rules. It is these aspects of the construction of proofs with which we will now be concerned.

4 Construction of Sequential Trees

Proofs trees get constructed in χ from the root (*i.e.* the theorem) backwards. The process of constructing proofs in this manner means trying to find an inference rule which can prove a given sequent and then repeating this process on that inference rule's premise(s). In constructing proofs in this way, there are at least two problems which are encountered: (1) Generally there are many inference rules which could be applied to yield a given sequent. Controlling the selection of inference rules must be done carefully. (2) Certain steps in proof construction, such as the instantiation of quantifiers, require information which can not be obtained through the local analysis of sequents. Our solution to the first problem involves using *tactics* and *tacticals* to write programs which explicitly determine the order in which inference rules are attempted. Our solution to the second problem is either to get the user involved in supplying the missing information or to invoke an automatic theorem prover.

At any point in building a proof there are generally numerous ways to proceed. For example, we may choose to manipulate one of the hypotheses (*i.e.* introduce a connective on the left of a sequent) or simplify the conclusion (introduce a connective on the right). There may be as many applicable inference rules as there are formulas in the sequent. Furthermore, there may be multiple choices for manipulating any one formula in the sequent. χ provides a programming language for specifying how these choices should be organized and executed. This programming language is an extension to the LCF tactic and tactical approach to theorem proving [6].

A primitive tactic is a (partial) function that takes a sequent, and checks if it can be the conclusion of a particular inference rule. If the sequent can be such a conclusion, the premise(s) of that instance of the rule are returned. If that particular inference rule can not be used, the tactic fails. Compound tactics can be constructed by using tacticals; these being high level operations which are used to coordinate and control the application of tactics. Such compound tactics allow some combination of several inference rules to be applied as a unit. For example, the following is a compound tactic for a very simple interactive prover in χ:

```
(define-tac interactive1
   (repeat
      (orelse
          axiomatize and_r_tac or_l_tac query)))
```

The `and_r_tac` and `or_l_tac` are primitive tactics which attempt to justify a sequent with the `and_r` and `or_l` inference rules, respectively. `axiomatize` attempts to justify the sequent as an axiom. If this tactic succeeds, that sequent is removed from further consideration. The `query` primitive tactic is a special tactic which queries the user for what tactic to apply next. The tacticals `repeat` and `orelse` are similar to those found in LCF. This compound tactic, therefore, automatically applies three tactics, and when these are no longer applicable, asks the user for assistance. A wide variety of interactive theorem provers can be built in this fashion. See [2] and [10] for more examples of interesting compound tactics.

Although tactics provide a great deal of flexibility in specifying how proofs should be built, there are certain issues in building proofs which can not be answered by simply examining such local facts as whether or not a given inference rule can be applied. For example, consider the following two inference rules which make use of an implicational hypothesis.

$$\frac{\Gamma \longrightarrow B \qquad C, \Gamma \longrightarrow A}{B \supset C, \Gamma \longrightarrow A}\,\texttt{positive}$$

$$\frac{\neg B, \Gamma \longrightarrow A \qquad C, \Gamma \longrightarrow}{B \supset C, \Gamma \longrightarrow A}\,\texttt{contrapos}$$

Although these rules might be applicable, they might not lead to a proof even if a proof exists. The simplest example of this situation is the sequent $p \supset q \longrightarrow \neg p \vee q$. Unless there is additional information available, rules like these could not be used to construct proofs: they would have to be replaced by more "complete" but more unnatural rules for implication. This would be very unfortunate since both `positive` and `contrapos` correspond to natural and frequently used methods of presenting proofs.

Extra information is also needed for the instantiation of quantifiers. In particular, consider the following two quantifier related inference rules.

$$\frac{\Gamma \longrightarrow [x/t]P}{\Gamma \longrightarrow \exists x\ P}\,\texttt{exists\_r} \qquad\qquad \frac{[x/t]P, \Gamma \longrightarrow A}{\forall x\ P, \Gamma \longrightarrow A}\,\texttt{forall\_l}$$

The appropriate selection of the term t in these inference rules is not available from a simple analysis of the lower sequent.

This kind of additional information can be obtained from two sources: the user or an automatic theorem prover. The user can be asked by the `query` tactic to make these difficult choices. More interestingly, however, would be to use an automatic theorem prover to discover these choices. Such an integration was the subject of [10]. We briefly outline the approach taken there since an extension of it is required in the rest of this paper.

Resolution refutations form a deduction system very different from those of sequential proof trees. Such refutations, however, do contain certain information — such as substitutions and connections among literals — which could be used in building sequential proof trees. This information is distilled into two data structures, called *expansion trees* and *matings*: an expansion tree conveniently stores substitution information while a mating specifies which literals in the expansion tree are to be connected. The "essence" of a resolution refutation can then be captured in a pair containing an expansion tree and a mating. Such a pair, called an *expansion proof*, captures a proof without referring to any deduction system. As a result, it makes a valuable tool for relating two such different deduction systems as resolution refutations and sequential proofs.

Example 3 An expansion proof for the theorem in Example 1 is given below.

$$\begin{array}{l}(\forall x \forall y \forall z\ (R(x,y) \wedge R(y,z) \supset R(x,z)), (a, b, a, (R(a,b)_1 \wedge R(b,a)_1 \supset R(a,a)_1)))\wedge \\ \quad (\forall x \forall y\ R(x,y) \supset R(y,x)), (a, b, R(a,b)_2 \supset R(b,a)_2)) \supset \\ \qquad (\forall x (\exists y\ R(x,y) \supset R(x,x)), (a, b, R(a,b)_3 \supset R(a,a)_2)); \\ \{(R(a,a)_1, R(a,a)_2), (R(b,a)_1, R(b,a)_2), (R(a,b)_1, R(a,b)_3), (R(a,b)_2, R(a,b)_3)\}\end{array}$$

The first item above is an expansion tree: quantified subformulas are followed by substitution terms and the corresponding instantiated formula. The last structure, a set of pairs of atom occurrences, is the corresponding mating. An algorithm for transforming resolution refutations to expansion proofs can be found in [11].

If we now add to a sequent an expansion proof for it, the above mentioned tactics are able to work locally to find the correct substitution term or decide if either the `positive` or `contrapos` inference rule will yield a complete proof. This additional information is also useful to deciding numerous other sophisticated ways to build a proof (see [2]).

When used in this setting, tactics can be viewed as proof transformers: they can take expansion proofs and automatically generate sequential proofs. Coupled with the transformation of resolution refutations into expansion proofs, automatically generated natural proofs can be made from the results of resolution theorem provers. We now outline how χ can be used to orchestrate such proof transformers into a useful and powerful logic tutoring system.

5 χ as an Intelligent Logic Tutor

The fact that proofs and their presentations are central to χ suggests that χ can be used to build tutors to help students in learning formal proof techniques. There are several different modes in which χ's facilities can be organized into such useful tutors. We describe three such modes in this section.

First of all, flexible interactive proof editors can be built using compound tactics. The simplest tactic, which does nothing but ask the user to specify all primitive tactics, could be used by beginning students. Since each primitive tactic is sound, only correct proofs can be constructed. If the student attempts an illegal proof construction, the failure of the tactic immediately signals a problem. As a student becomes more familiar with proofs and their construction, s/he can build compound tactics which can automate part of the proof construction method. The organization and complexity of such compound tactics would supply evidence of how well the student has managed to understand the "mechanical" aspects of proof building. Finally, the student can easily see the natural language text generated for his/her proof. The relationship between the tree structured skeleton of a proof which the student directly manipulates and the natural language form of it should make it possible for him/her to understand how mathematical proofs, which are written in natural language, are organized. The pairing of natural language explanations with the more formal sequential proof trees should also help students learn how to write proofs themselves when they are not using the tutoring system. For example, it should become clear to the student in what contexts phrases such as "choose b such that ..." and "let n be an arbitrary number such that ..." are to be used.

Another mode for using χ assumes that an automatic theorem prover is present which satisfies the following three properties: (1) that it succeeds for a high percentage of the theorems asked of it, (2) that it terminates if no proof is found using "reasonable" resources, and (3) that any proof constructed by this prover can be converted to an expansion proof. If the tutor's domain is elementary logic, then traditional resolution theorem provers can be used to satisfy all three of these constraints. In other, more difficult domains, the tutor would only be as good as the automatic theorem prover under it. Such a theorem prover can be used in the following fashion. If a student reaches a sequent with which s/he doesn't know how to proceed, the sequent could be given to the theorem prover to prove. If no proof is found after a reasonable amount of time, the student will be given an indication that a proof of that sequent is unlikely and that s/he should backtrack to an earlier stage in the proof. If a proof is found, it would be converted to an expansion proof. With this expansion proof, χ could do several things. It could employ a compound tactic which would automatically transform it into a complete proof of the troublesome sequent. Also, a tactic could be employed which only revealed a hint into how the proof should proceed, leaving the rest of its completion up to the student. Such a help facility would not have to rely on "canned scripts" or be restricted to providing help only if the user has proceeded in one predetermined way. Furthermore, no extra effort on the part of the instructor would be necessary to "train " this kind of tutoring system.

The student can also take advantage of the flexibility of the programming language of tactics and tacticals to write and experiment with various tactics to construct different sequential proofs from a given expansion proof. For example, direct and indirect proofs of the same theorem could be built and compared. Furthermore, it is possible to take an interactively built proof and generate from it an expansion proof. Hence, a student's proof could be reduced to its "essence" and then rewritten using these same kinds of tactics. A student could have his/her proof rewritten by a tactic which exemplifies a "good" style of proof. The rewriting could be done automatically and quickly. We illustrate this revision mechanism in the next section.

6 Proof Revision

The first step in revising a proof is to remove much of the detail of the given proof. This, of course, could be done if sequential proof trees could be converted into expansion proofs. From an expansion proof, a new proof tree could be constructed under the guidance of compound tactics identical to those described in Section 4. Such tactics can be designed to stress good styles of proof.

The transformation from sequential proof tree to an expansion proof can be done by recursion on the structure of proof trees. This transformation can thus be accomplished by a functional style interpretation of inference rules, in much the same way that text was generated from sequential trees in Section 3. The expansion proof for an axiom are trivial: the simple formula $A_1 \supset A_2$ and the set $\{(A_1, A_2)\}$ are the expansion tree and mating (resp.) for the axiom $A \longrightarrow A$. Each inference rule is interpreted as a function which performs one step in the transformation by taking the translations for its premise(s) and yielding an expansion proof for its conclusion. For example, the `or_l` function takes expansion proofs for $B, \Gamma \longrightarrow A$ and $C, \Gamma \longrightarrow A$ and produces an expansion proof for $B \vee C, \Gamma \longrightarrow A$. All the inference rules used in χ can be given similar interpretations and as a result, all proof trees can be reduced to expansion proofs in this fashion. For example, the sequential proof tree in Example 1 is reduced to the expansion proof in Example 3.

Once an expansion proof is distilled from a proof tree, various tactics could be applied to it in order to transform it into a proof tree again. With the following example, we will illustrate a compound tactic which tries to avoid repeating subproofs along different branches of the proof as well as favoring direct proofs over indirect proofs.

To illustrate proof revision, consider the following abstract form of the theorem which states that there exists no largest prime number.

$$\begin{aligned}&(\forall x\ (f(x) > x) \wedge \forall x \forall y\ (\mathrm{div}(x, f(y)) \supset (x > y)) \wedge \\ &\quad \forall x\ (\neg \mathrm{prime}(x) \supset \exists y\ (\mathrm{prime}(y) \wedge \mathrm{div}(y, x)))) \supset \\ &\qquad\qquad\qquad\qquad \forall n\ (\exists x\ ((x > n) \wedge \mathrm{prime}(x))).\end{aligned}$$

Although the intended domain here is the set of positive integers, all the facts necessary to reason about the integers are contained in the statement of the theorem. The theorem assumes the existence of a function f on positive integers such that for any x, $f(x) > x$ and for any x and y, if x divides $f(y)$, then $x > y$. In the domain of arithmetic, such a function does in fact exists: $f(x) = x! + 1$ would be such a function. The remaining assumption simply states that if a number is not prime, it has a prime divisor. From these hypotheses, it can be shown that there is a prime larger than any given number.

Suppose a student successfully proves this theorem, creating a proof term with the following natural language rendering.

> Assume $\forall x\ (f(x) > x) \wedge \forall x\ \forall y\ (\mathrm{div}(x, f(y)) \supset (x > y)) \wedge \forall x\ (\neg\mathrm{prime}(x) \supset \exists y\ (\mathrm{prime}(y) \wedge \mathrm{div}(y, x))$. Assume $\neg\exists x((x > a) \wedge \mathrm{prime}(x))$. Hence $\forall x(\neg(x > a) \vee \neg\mathrm{prime}(x))$. We have two cases. Case 1: Assume $\neg(f(a) > a)$. We have a contradiction. Case 2: Assume $\neg\mathrm{prime}(f(a))$. By modus ponens, we have $\exists y(\mathrm{prime}(y) \wedge \mathrm{div}(y, f(a))$. Choose b such that $\mathrm{prime}(b) \wedge \mathrm{div}(b, f(a))$. By modus ponens, we have $(b > a)$. We have two cases. Case 2.1: Assume $\neg(b > a)$. We have a contradiction. Case 2.2: Assume $\neg\mathrm{prime}(b)$. We have a contradiction. Thus, in either case, we have a contradiction. Thus, in either case, we have a contradiction. Hence, by indirect proof, we have $\exists x((x > a) \wedge \mathrm{prime}(x))$. Since a was arbitrary, we have $\forall n(\exists x((x > n) \wedge \mathrm{prime}(x)))$.

In constructing this proof, the student applied the `indirect` tactic fairly early, causing the negation of the conclusion $\neg\exists x\ ((x > a) \wedge \mathrm{prime}(x))$ to become an assumption. That negation was equivalent to $\forall x\ (\neg(x > a) \vee \neg\mathrm{prime}(x))$. This quantified disjunction allowed the proof to be broken up into cases twice. First, it was instantiated with $f(a)$. The first case resulted in an immediate contradiction, while the second broke into two further cases, since this disjunction was instantiated with b. These two subcases immediately result in contradictions, completing the indirect proof. Such indirect proofs are not at all uncommon among students who are learning formal proofs for the first time.

It should be clear that this proof could be written more clearly. By applying our above mentioned tactic to the expansion proof of this proof tree, the following proof was constructed.

> Assume $\forall x\ (f(x) > x) \wedge \forall x\ \forall y\ (\mathrm{div}(x, f(y)) \supset (x > y)) \wedge \forall x\ (\neg\mathrm{prime}(x) \supset \exists y\ (\mathrm{prime}(y) \wedge \mathrm{div}(y, x))$. We have two cases. Case 1: Assume $\neg\mathrm{prime}(f(a))$. By modus ponens, we have $\exists y\ (\mathrm{prime}(y) \wedge \mathrm{div}(y, f(a)))$. Choose b such that $\mathrm{prime}(b) \wedge \mathrm{div}(b, f(a))$. By modus ponens, we have $(b > a)$. Hence, $(b > a) \wedge \mathrm{prime}(b)$. Thus, $\exists x\ ((x > a) \wedge \mathrm{prime}(x))$. Case 2: Assume $\mathrm{prime}(f(a))$. Hence, $(f(a) > a) \wedge \mathrm{prime}(f(a))$. Thus, $\exists x\ ((x > a) \wedge \mathrm{prime}(x))$. Thus, in either case, we have $\exists x\ ((x > a) \wedge \mathrm{prime}(x))$. Since a was arbitrary, we have $\forall n\ (\exists x\ ((x > n) \wedge \mathrm{prime}(x)))$.

Although these proofs are significantly different, much of the student original proof is retained in the expansion proof and thus reappears in the revised proof. For example, all of the substitution terms (*i.e.* a, b, and $f(a)$) used in the first proof appear exactly the same in the second.

Notice that this proof also breaks up into cases, this time given by the formula $\neg\text{prime}(f(a)) \vee \text{prime}(f(a))$. Although this formula is not a member of the original list of assumptions, it is trivially true and thus could be taken as an assumption. Discovering that this additional assumption was important to introduce in order to make the presentation of this proof more natural requires rather sophisticated manipulations of expansion proofs. In χ this processing step is represented by a tactical which attempts to build the proof using the following inference rule:

$$\frac{\neg B \vee B, \Gamma \longrightarrow A}{\Gamma \longrightarrow A}\ \texttt{excluded\_middle}$$

This tactic is very different from others we have considered so far for two reasons. First, it can always be applied to any sequent, so it must be applied only when it genuinely improves the presentation of the proof. Second, an appropriate formula B must be chosen. This selection is solved by an algorithm called *symmetric simplification* which is described by Pfenning in [12]. Symmetric simplification is a powerful technique which, when used in proof revision, often has the effect of transforming an indirect proof to a direct proof.

7 Conclusion

We have presented the high-level details of the χ-proof system. This system makes use of two proof structures not commonly represented explicitly in traditional theorem proving systems, namely, sequential proof trees and expansion proofs. Proof trees and their lexicalizations are used for presenting deductions to human readers. Expansion proofs are used to represent the essence of proofs in a deduction-free setting. This system also contains numerous proof transformation mechanisms. Some of these transformers — the two that convert resolution refutations and sequential proof trees into expansion proofs — do not require any choices to be made, since they involve deleting details form those proofs. The transformation of expansion proofs to sequential proof trees, however, is a process of adding details to a proof. The choice and organization of these details is, of course, crucial for the proofs to be readable. These latter transformations are, therefore, written in the tactic programming language which is capable of specifying these choices.

The χ-proof system can be seen as an enrichment of more traditional theorem proving systems. χ can permits flexible interfaces to an automatic theorem prover as well as several tools for manipulating any proofs which are discovered. This system could also be used to integrate several different kinds of automatic theorem proving systems: a special tactic could decide which prover would be appropriate for proving a given sequent. Finally, χ could also be viewed as a more generic way to manipulate proofs: the kinds of inference rules contained in sequential proofs as well as the actual number and kind of logical constants present in the logic need not be exactly those presented in this paper. Our current implementation of χ is now, for example, being modified to handle modal and higher-order logics.

Acknowledgements The authors would like to thank Frank Pfenning and Greg Hager for technical discussions concerning the ideas in this paper and the construction of the χ-proof system. We would also like to thank Bonnie Webber and Aravind Joshi for suggesting that this work was worth pursuing and for providing an environment in which to nurture it. This work has been supported by NSF AI Center grants NSF-MCS-83-05221, US Army Research office grant ARO-DAA29-84-9-0027, and DARPA N000-14-85-K-0018.

References

[1] R. L. Constable et. al., *Implementing Mathematics with the Nuprl Proof Development System*, Prentice–Hall, 1986.

[2] Amy Felty, "Using Extended Tactics to Do Proof Transformations," Master's thesis, December 1986, University of Pennsylvania. Technical report MS-CIS-86-89.

[3] Jean H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper & Row, 1986.

[4] Jean H. Gallier, Stan Raatz, and Wayne Snyder, "Theorem Proving Using Rigid E-Unification: Equational Matings," to appear in the 1987 IEEE Symposium on Logic in Computer Science.

[5] Gerhard Gentzen, *Investigations into Logical Deductions* in *The Collected Papers of Gerhard Gentzen* edited by M. E. Szabo, North-Holland Publishing Co., Amsterdam, 1969, 68 – 131.

[6] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth, "Edinburgh LCF," Lecture Notes in Computer Science, No. 78, Springer-Verlag, 1979.

[7] Gregory D. Hager, "Computational Aspects of Proofs in Modal Logic," Master's thesis, University of Pennsylvania, 1985.

[8] W. A. Howard, "The formulae-as-type notion of construction," 1969. Published in J. P. Seldin and R. Hindley, ed. *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, 479 – 490, Academic Press, New York, 1980.

[9] Dale A. Miller, "Expansion Trees and Their Conversion to Natural Deduction Proofs," *7th Conference on Automated Deduction, Napa CA*, edited by R. E. Shostak, Lecture Notes in Computer Science, No. 170, Springer-Verlag, 1984, 375 – 393.

[10] Dale Miller and Amy Felty, "An Integration of Resolution and Natural Deduction Theorem Proving," 1986 National Conference on Artificial Intelligence, 198 – 202.

[11] Frank Pfenning, "Analytic and Non-analytic Proofs," *7th Conference on Automated Deduction, Napa CA*, edited by R. E. Shostak, Lecture Notes in Computer Science, No. 170, Springer-Verlag, 1984, 394 – 413.

[12] Frank Pfenning, "Proof Transformations in Higher-Order logic," Ph. D. Dissertation, Carnegie-Mellon University, January 1987.