November 1987

# Implementing Theorem Provers in Logic Programming

Amy Felty
*University of Pennsylvania*

## Recommended Citation

Amy Felty, "Implementing Theorem Provers in Logic Programming", . November 1987.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-87-109.

# Implementing Theorem Provers in Logic Programming

## Abstract

Logic programming languages have many characteristics that indicate that they should serve as good implementation languages for theorem provers. For example, they are based on search and unification which are also fundamental to theorem proving. We show how an extended logic programming language can be used to implement theorem provers and other aspects of proof systems for a variety of logics. In this language first-order terms are replaced with simply-typed λ-terms, and thus unification becomes higher-order unification. Also, implication and universal quantification are allowed in goals. We illustrate that inference rules can be very naturally specified, and that the primitive search operations of this language correspond to those needed for searching for proofs. We argue on several levels that this extended logic programming language provides a very suitable environment for implementing tactic style theorem provers. Such theorem provers provide extensive capabilities for integrating techniques for automated theorem proving into an interactive proof environment. We are also concerned with representing proofs as objects. We illustrate how such objects can be constructed and manipulated in the logic programming setting. Finally, we propose extensions to tactic style theorem provers in working toward the goal of developing an interactive theorem proving environment that provides a user with many tools and techniques for building and manipulating proofs, and that integrates sophisticated capabilities for automated proof discovery. Many of the theorem provers we present have been implemented in the higher-order logic programming language λProlog.

## Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-87-109.

# IMPLEMENTING THEOREM PROVERS IN LOGIC PROGRAMMING

**Amy Felty**

**MS-CIS-87-109**
**LINC LAB 87**

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

**December 1987**

**Acknowledgements:** This research was supported in part by DARPA grants NOOO14-85-K-0018, NSF grants CCR-87-05596,MCS-8219196-CER and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

# Implementing Theorem Provers in Logic Programming

Dissertation Proposal

Amy Felty
University of Pennsylvania

November 6, 1987

**Abstract**

Logic programming languages have many characteristics that indicate that they should serve as good implementation languages for theorem provers. For example, they are based on search and unification which are also fundamental to theorem proving. We show how an extended logic programming language can be used to implement theorem provers and other aspects of proof systems for a variety of logics. In this language first-order terms are replaced with simply-typed $\lambda$-terms, and thus unification becomes higher-order unification. Also, implication and universal quantification are allowed in goals. We illustrate that inference rules can be very naturally specified, and that the primitive search operations of this language correspond to those needed for searching for proofs. We argue on several levels that this extended logic programming language provides a very suitable environment for implementing tactic style theorem provers. Such theorem provers provide extensive capabilities for integrating techniques for automated theorem proving into an interactive proof environment. We are also concerned with representing proofs as objects. We illustrate how such objects can be constructed and manipulated in the logic programming setting. Finally, we propose extensions to tactic style theorem provers in working toward the goal of developing an interactive theorem proving environment that provides a user with many tools and techniques for building and manipulating proofs, and that integrates sophisticated capabilities for automated proof discovery. Many of the theorem provers we present have been implemented in the higher-order logic programming language $\lambda$Prolog.

Advisor: Dale Miller
Committee: Val Breazu-Tannen
Robert Constable
Jean Gallier (Chair)
Andre Scedrov

# Contents

# 1 Introduction

Logic programming languages have many characteristics that indicate that they should serve as good implementation languages for theorem provers. First, at the foundation of computation in logic programming is search. While logic programs are specified declaratively, the execution of logic programming programs is based on an underlying search algorithm. Search is also fundamental to theorem proving. The process of discovering a proof involves traversing an often very large search space in some controlled manner. Second, unification is an important mechanism in logic programming which is used to solve equations between various objects. This mechanism is crucial to the theorem proving process, in particular for the proper treatment of formulas and proofs. Also, the fact that programs have a declarative reading is an important characteristic of logic programming languages: one can often write programs that represent natural specifications for a given task. The capability to both write and understand programs easily is especially valuable in theorem proving because the tasks involved are often complex and because soundness and completeness results must often be proved.

Traditional logic programming languages such as Prolog are not sufficient for handling certain aspects of implementing proof systems. One deficiency, as argued in [Miller & Nadathur 87], is that first-order terms are quite inadequate for a clean representation of formulas. For instance, first-order terms provide no mechanism for representing variable abstraction required for quantification in first-order formulas. Quantification must be specially encoded. For example, in Prolog, we can represent abstractions in formulas by including the bound variables as arguments in the terms. The formula $\forall x \exists y P(x, y)$ could be written as the first-order term `forall(x,exists(y,p(x,y)))`. The logic variables of Prolog cannot be used to represent variables in the formula because we need to distinguish between variables within the scope of a quantifier, and those outside it. Thus the substitution and unification that is available on logic variables is not available for these terms. The programmer would have to write new procedures that accomplish these tasks for the encoded representation. In addition, by manipulating such an encoding, we lose much of the declarative nature that should be present in logic programming programs.

In this paper, we will introduce a higher-order logic programming language that extends the first-order Horn clause theory on which Prolog is based. The logical foundation of this language is a collection of formulas called *higher-order hereditary Harrop* formulas [Miller, Nadathur, & Scedrov 87]. This language replaces first-order terms with simply typed $\lambda$-terms. The abstractions built into $\lambda$-terms can thus be used to represent quantification. Our extended language also permits goal formulas to be both implications and universally quantified and we shall show how such goal formulas are, in fact, necessary for implementing various kinds of theorem provers. Many of the programs that we present in this paper have been tested using a logic programming language called $\lambda$Prolog which is based on these higher-order hereditary Harrop formulas. Various aspects of this language have been discussed in [Miller & Nadathur 86a, Miller & Nadathur 86b, Miller & Nadathur 87, Nadathur 86].

Our main claim in this paper is that such a language is a very suitable environment for implementing theorem provers. We will show that search and unification accommodate the tasks involved in theorem proving very naturally. In this case search is based on our extended set of goals and unification is over higher-order terms. Most of our theorem provers will have a clean declarative reading which provides them with implementation independent semantics.

In particular, one of our main goals is to use such a language to build a theorem proving environment in which a user can become involved in the search for proofs. Such an environment should provide the user with many tools and techniques for searching for and constructing proofs, and should contain somewhat sophisticated capabilities for automated proof discovery. In working toward this goal, we will illustrate how to implement a theorem prover based on tactics and tacticals as in the LCF theorem prover, [Gordon, Milner & Wadsworth 79] and the Nuprl proof system [Constable *et al.* 86].

We are also interested in representing and storing proofs as they are discovered, so that they can be used in computations. We discuss the formulas-as-types paradigm [Howard 80] within our setting, where we consider formulas to be types and proofs to be the objects that inhabit these types. We will demonstrate how such proof objects can be constructed and manipulated in our environment. Some examples include constructing programs from proofs, building natural language explanations from proofs, and using proof objects to do proof by analogy.

Another goal is to show that this logic programming language can be used to specify proof systems for a wide class of logics. In this respect, we share a common goal with the Edinburgh Logical Framework (LF) [Harper, Honsell & Plotkin 87]. LF is a logic developed to provide a general theory of inference systems that captures many uniformities across different logics. We have been able to show that all the example signatures specified in LF in the paper [Avron, Honsell & Mason 87] can be specified as logic programming programs. In addition to natural specifications, these programs represent non-deterministic theorem provers for these logics.

In the next section, we will present the subset of the class of higher-order hereditary Harrop formulas on which we base our logic programming language, and describe an interpreter for this language. In Section 3 we present some simple programs to manipulate formulas in this language. It will become evident that simple operations on formulas are handled quite differently in this setting than by traditional methods. This is followed, in Section 4, by a discussion of how to specify inference rules. Each rule will correspond to a definite clause. Theorem provers are implemented using collections of such clauses. We discuss both the declarative reading and the operational meaning (under the interpreter of Section 2) of these clauses. We will also consider the specification of proof objects.

In Section 5 we begin to consider some issues in controlling the search for proofs. We present an automatic theorem prover for a variant of the Gentzen LK proof system for classical first-order logic. In Section 6 we discuss the implementation of tactic style theorem provers which allow greater control in searching for proofs and provide means for user participation in the theorem proving process. We first present the tacticals and other general definite clauses that will be used in any tactic prover. As an example, we build a theorem prover for natural deduction where the basic tactics are the inference rules of this proof system, and then present ways in which we can add to the tactic database to enhance the basic theorem proving environment.

In Section 7, we discuss some algorithms that use proof objects for different purposes, including those mentioned above. It will become apparent that there are many options for representing proof objects and that choosing a representation should depend on how the proofs will be used.

In Section 8, we give an informal presentation of the algorithm used to translate the example LF signatures in [Harper, Honsell & Plotkin 87] and [Avron, Honsell & Mason 87] to specifications for theorem provers in our extended logic programming language. In Section 9, we discuss related work, and finally in Section 10, we discuss future research.

# 2 Extended Logic Programs

We shall need to extend the logic of first-order positive Horn clauses in two essential ways. The first extensions provide for the interpretation of queries (goals) which can be implications, disjunctions, universally and existentially quantified, as well as the usual conjunction which is permitted in Horn clause theorems. Although the addition of disjunctive and existential goals does not depart much from the usual presentation of Horn clauses, the addition of implicational and universal goals makes a significant departure. The second extension makes this language *higher-order* in the sense that it is possible to quantify over function symbols. In order to represent constructions which can be functions, first-order terms are replaced with simply typed $\lambda$-terms. To implement the application of function terms, $\lambda$-conversion is also required. Finally, to perform unification on $\lambda$-terms, higher-order unification is required.

The logic programming language presented in this section is a subset of the class of higher-order hereditary Harrop formulas which was presented in [Miller, Nadathur & Scedrov 87]. The richer language permits quantification over predicate variables and permits $\lambda$-terms to contain logical connectives. Neither of these features are needed in this paper so we shall simplify our presentation by ignoring these possibilities.

## 2.1 A Logic Programming Language and Interpreter

In the logic programming language used in the remainder of this paper, we will assume that a certain set of non-functional types is provided, which contains at least one type, namely $o$, which denotes the type of logic programming propositions. The full set of types is then all the non-functional types along with all functional types, that is, types of the form $\alpha \to \beta$ where $\alpha$ and $\beta$ are (non-functional or functional) types. Simply typed $\lambda$-terms are then built in the usual fashion. $\lambda$-terms which are propositions, *i.e.* of type $o$, will be called *atomic* formulas. In this section we shall let $A$ be a syntactic variable for atomic formulas.

We now define two new classes of formulas, called *goal formulas* and *definite clauses.* Let $G$ be a syntactic variable for goal formulas and let $D$ be a syntactic variable for definite clauses. These two classes are defined by the following mutual recursion.

$$G := A \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid D \supset G \mid \exists v\, G \mid \forall v\, G$$

$$D := A \mid G \supset A \mid \forall v\, D$$

A *logic program* or just simply a *program* is a finite set, say $\mathcal{P}$, of closed definite formulas. The following theorem on definite clauses follows immediately from theorems in [Miller, Nadathur, & Scedrov 87].

**Theorem 1** *Let $\vdash_I$ denote intuitionistic provability for this higher-order logic and assume that the formulas $G, G_1, G_2, A$ and $D$ are closed formulas. Then all the following hold.*

1. *$\mathcal{P} \vdash_I G_1 \wedge G_2$ if and only if $\mathcal{P} \vdash_I G_1$ and $\mathcal{P} \vdash_I G_2$.*
2. *$\mathcal{P} \vdash_I G_1 \vee G_2$ if and only if $\mathcal{P} \vdash_I G_1$ or $\mathcal{P} \vdash_I G_2$.*
3. *$\mathcal{P} \vdash_I D \supset G$ if and only if $\mathcal{P}, D \vdash_I G$. Here $\mathcal{P}, D$ denotes the set $\mathcal{P} \cup \{D\}$.*
4. *$\mathcal{P} \vdash_I \forall x\, G$ if and only if for any parameter $c$ which does not occur in $\mathcal{P}$ or $G$, $\mathcal{P} \vdash_I [x/c]G$.*
5. *$\mathcal{P} \vdash_I \exists x\, G$ if and only if there exists a closed $\lambda$-term $t$ of the same type as $x$ such that $\mathcal{P} \vdash_I [x/t]G$.*
6. *If $A$ is atomic, then $\mathcal{P} \vdash_I A$ if and only if either some universal instantiation of a $D \in \mathcal{P}$ is either $A$ or is of the form $G \supset A$ and $\mathcal{P} \vdash_I G$.*

Notice that these properties of $\vdash_I$ can be used to describe a very high-level interpreter which can determine if a given goal is provable from a given program. This interpreter will be called the *non-deterministic interpreter.* Assume that a program $\mathcal{P}$ and a closed goal formula $G$ are given and we wish to determine if $G$ is intuitionistically provable from $\mathcal{P}$. This interpreter can be described non-deterministically as being composed of the following six basic *search operations.*

AND If $G$ is $G_1 \wedge G_2$ then try to show that both $G_1$ and $G_2$ follow from $\mathcal{P}$.

OR If $G$ is $G_1 \vee G_2$ then try to show that either $G_1$ or $G_2$ follows from $\mathcal{P}$.

AUGMENT If $G$ is $D \supset G'$ then add $D$ to the current program and try to prove the goal $G'$.

GENERIC If $G$ is $\forall x\, G'$ then pick a new parameter $c$ and try to prove the goal $[x/c]G'$.

INSTANCE If $G$ is $\exists x\, G'$ then pick some closed $\lambda$-term $t$ and try to prove $[x/t]G'$.

BACKCHAIN If $G$ is atomic, we must now consider the current program. If there is a universal instance of a definite clause which is equal to $G$ then we have found a proof. If there is a definite clause with a universal instance of the form $G' \supset G$ then try to prove $G'$ from $\mathcal{P}$. If neither case holds then there is no proof of $G$ from $\mathcal{P}$.

There are actually many ways to implement the GENERIC search operation. The description above indicates one possibility. It is not necessarily the best for implementation purposes, yet is a good conceptual description. We are not concerned with the implementation of the interpreter in this paper, but will consider it to be implemented in this way for the purpose of discussion. Everything that is said will still hold for any other implementation.

In actual operation of this interpreter, the INSTANCE operation will introduce a variable which will become instantiated as necessary through unification. Thus, it will not be forced to commit to a particular closed $\lambda$-term at the time it is invoked. This corresponds to the notion of "logic variables" in Prolog.

This logic programming language properly contains first-order positive Horn clauses. That is, if $A_0, A_1, \ldots, A_n$ are all atomic formulas then the formula $\forall x_1 \ldots \forall x_n\ (A_1 \wedge \ldots \wedge A_n \supset A_0)$ is both a positive Horn clause and a definite clause (in our sense). Hence, any pure Prolog program is also a program in our setting. Richer programs are, of course, possible. We give an example in the next subsection.

## 2.2 An Example

The following definite clauses are motivated by an example of John McCarthy.

$$\begin{array}{l}
\forall y(\forall x((\mathit{bug}\ x) \supset ((\mathit{in}\ x\ y) \supset (\mathit{dead}\ x))) \supset (\mathit{sterile}\ y)) \\
\forall x \forall y((\mathit{heated}\ y) \wedge (\mathit{in}\ x\ y) \wedge (\mathit{bug}\ x) \supset (\mathit{dead}\ x)) \\
(\mathit{heated}\ j)
\end{array}$$

Given two types, *jar* and *insect*, the predicates in the above formula could be given the "functional" types

$$\begin{array}{rl}
\mathit{sterile}\colon & \mathit{jar} \to o \\
\mathit{bug}\colon & \mathit{insect} \to o \\
\mathit{dead}\colon & \mathit{insect} \to o \\
\mathit{heated}\colon & \mathit{jar} \to o \\
\mathit{in}\colon & \mathit{insect} \to \mathit{jar} \to o \\
j\colon & \mathit{jar}.
\end{array}$$

$\lambda$-terms of functional types which map to propositions can be thought of as predicates. Notice that while the last two formulas above are Horn clauses, the first is not. All three, however, are definite clauses. Let $\mathcal{P}_0$ denote the set of these three definite formulas. Given the above description of provability, it is easy to show that $\mathcal{P}_0 \vdash_I (\mathit{sterile}\ j)$. The following represents the search strategy the above interpreter could follow to find a proof.

$$\begin{array}{rl}
\mathcal{P}_0 \vdash_I & (\mathit{sterile}\ j) \\
\mathcal{P}_0 \vdash_I & \forall x((\mathit{bug}\ x) \supset ((\mathit{in}\ x\ j) \supset (\mathit{dead}\ x))) \\
\mathcal{P}_0 \vdash_I & (\mathit{bug}\ r) \supset ((\mathit{in}\ r\ j) \supset (\mathit{dead}\ r)) \\
\mathcal{P}_0, (\mathit{bug}\ r) \vdash_I & (\mathit{in}\ r\ j) \supset (\mathit{dead}\ r) \\
\mathcal{P}_0, (\mathit{bug}\ r), (\mathit{in}\ r\ j) \vdash_I & (\mathit{dead}\ r) \\
\mathcal{P}_0, (\mathit{bug}\ r), (\mathit{in}\ r\ j) \vdash_I & (\mathit{heated}\ j) \wedge (\mathit{in}\ r\ j) \wedge (\mathit{bug}\ r) \\
\mathcal{P}_0, (\mathit{bug}\ r), (\mathit{in}\ r\ j) \vdash_I & (\mathit{heated}\ j) \\
\mathcal{P}_0, (\mathit{bug}\ r), (\mathit{in}\ r\ j) \vdash_I & (\mathit{in}\ r\ j)
\end{array}$$

$$\mathcal{P}_0, (bug\ r), (in\ r\ j) \vdash_I (bug\ r)$$

The last three lines are proved immediately by the BACKCHAIN rule. Here, it was necessary to argue generically about an arbitrary bug, called $r$ of type *insect*.

It is important to realize that the metatheory of definite clauses is intuitionistic logic, a logic which is weaker than classical logic. (See [Miller 86, Miller, Nadathur & Scedrov 87] for a discussion on the role of intuitionistic logic in this extended notion of logic programming.) Hence, there are inferences from formulas in classical logic which can not be inferred by the kind of interpreter described above. For example, the goal formula $\exists x\ (bug\ x) \vee \forall y\ (sterile\ y)$ can be classically inferred from $\mathcal{P}_0$. Our interpreter would try either to prove that there exists an insect which is a bug or to prove that all jars are sterile. Neither attempt to find a proof will succeed. Because classical logic contains the axiom scheme of excluded middle (absent from intuitionistic logic), classical derivations do not obey all the above six search rules: in particular, the OR and INSTANCE rules are seldom valid. For example, although $\exists x\ (bug\ x) \vee \forall y\ (sterile\ y)$ is classically provable, neither of its disjuncts are provable separately. The classical proof of this formula would start by noting that

$$\exists x\ (bug\ x) \vee \forall x\ \neg(bug\ x).$$

If we assume that there does exist an insect which is a bug, the formula follows immediately. On the other hand, if we assume that all insects are not bugs, then it is easy to show that all jars must be sterile.

We will present further examples of definite clauses, in particular higher-order examples, in later sections.

Our main claim in this paper is that the above six search operations along with higher-order unification, $\lambda$-convertibility, and the notion of a "logic variable" provide a very valuable environment for the design of theorem proving programs.

## 2.3 Notation

For readability, we will use some abbreviations when writing definite clauses. For the most part, these abbreviations define the syntax of the logic programming language $\lambda$Prolog (which adopts much of the syntax of Prolog).

**$\lambda$-Terms** Variables are represented by tokens with an upper case initial letter, and constants are represented by tokens with a lower case initial letter. Function application will be represented using curried notation *i.e.* juxtaposition of terms represents application and this application associates to the left. $\lambda$-abstraction is represented using the infix symbol \. A term of the form $\lambda x\ T$ is written as `X\T`. Terms are most accurately thought of as being representatives of $\alpha\beta\eta$-conversion equivalence classes of terms. For example, the terms `X\(f X)`, `Y\(f Y)`, `(F\Y\(F Y) f)`, and `f` all represent the same class of terms. Since bound variables have no distinct "name," the programmer will not have to deal with renaming bound variables. Also, substitutions are handled directly because of the availability of $\lambda$-conversion.

As in Prolog, we allow infix operators, and will use infix notation when appropriate to enhance readability. For example, we could have considered the `in` predicate in the above example as an infix operator, and written `(X in Y)` instead of `(in X Y)`.

**Search Connectives** The symbols `,` and `;` represent $\wedge$ and $\vee$ respectively, and `,` binds tighter than `;`. The symbol `:-` represents the top-level implication in a definite clause. Clauses of this form are written backwards *i.e.* a clause $G \supset A$ is written `A :- G`. Implications at all other levels are represented using the symbol `=>` in the forward direction. We omit the outermost universal quantifiers in a definite clause, and existential quantifiers in a goal. Thus, free variables in a definite clause are assumed to be universally quantified, while free variables in a goal are assumed to be existentially quantified. Internal universal quantification is represented with the `pi` constant and the $\lambda$ operator (`\`). A formula of the form $\forall x\ P$ is represented by `(pi X\ P)`, where `X` is a capital letter and all occurrences of $x$ in $P$ occur as `X` in `P`.

The program in the previous subsection can be abbreviated:

```
sterile Y :- pi X\ ((bug X) => ((in X Y) => (dead X))).
dead X :- heated Y, in X Y, bug X.
heated j.
```

**Modules** We sometimes want to think of a set of definite clauses as a module that can be imported by a goal formula so that the clauses in this module will be available when attempting to satisfy the goal formula. For more on the theory of adding modules to logic programming see [Miller 86]. Here, we simply allow a name (*e.g.* `Mod`) to be associated with a set of definite clauses and allow abbreviations of the form `(Mod => G)` where `G` is a goal formula. The AUGMENT search operation must then be extended to allow sets of definite clauses to be added to the current program.

**Types** As we saw in the example in the last subsection, a set of type declarations will be associated with each logic program. In addition to functional types, other type constructors are allowed. In many of our programs we will use list structures, and so `list` will be the only type constructor needed in this paper. It takes one type as an argument. In the previous example, `(list insect)` and `(list jar)` are also types. The syntax for lists will be the same as in Prolog.

Sometimes, we will use capital letters to represent variables in type declarations. Such a declaration represents an infinite number of declarations, each of which is obtained by substituting closed types for the variables that occur in the type. For example, most of the list manipulation functions can operate on lists of any type. Some of predicates used in this paper and their declarations are as follows:

```
         member : A -> (list A) -> o.
         append : (list A) -> (list A) -> (list A) -> o.
member_and_rest : A -> (list A) -> (list A) -> o.
```

These three predicates have the following definitions.

```
member X [X | L].
member X [Y | L] :- member X L.

append [] L L.
append [X | L1] L2 [X | L3] :- append L1 L2 L3.

member_and_rest X [X | L] L.
member_and_rest X [Y | L1] [Y | L2] :- member_and_rest X L1 L2.
```

The type variable A must be instantiated before the interpreter can use these definite clauses.

# 3 Manipulation of Formulas

As was argued earlier, first-order terms of traditional logic programming languages are not adequate for representing formulas. In particular it is awkward to represent quantification using such terms. Our higher-order language replaces these terms with simply typed $\lambda$-terms. Using these terms, we can still easily represent first-order terms. Data structures that can be built using first-order terms such as lists, trees, and even propositional formulas are represented in essentially the same way in this setting. The additional ability to represent abstractions directly within terms makes it possible to very directly represent quantified formulas.

We use $\lambda$-terms to represent formulas as introduced by Church in his formulation of the simple theory of types [Church 40], and adopted by many others (*e.g.* [Paulson 86, Miller & Nadathur 87, Harper, Honsell & Plotkin 87]). We introduce a new type `bool`, and specify the logical connectives of the object language by introducing new constants and giving them types. In Section 2 we said that we would not permit logical constants to appear in $\lambda$-terms. That restriction was on logical constants which involved the special type `o` of search propositions. This special type was reserved for the interpreter of hereditary Harrop formulas, and was needed since the behavior of formulas of type `o`, as described by Theorem 1, is very specialized. Thus our hereditary Harrop formula language will presuppose nothing about our newly introduced constants. Another way to look at this is that at the program- or meta-level, the logical constants have a very set meaning, *i.e.* provided by a higher-order intuitionistic logic. At the term- or object-level, logical connectives have only the meaning which is attributed to them by the programs which use them.

For classical first-order formulas, which we present as an example, in addition to the new type `bool`, let us introduce the type `i` to represent the domain of first-order individuals. We then declare the following constants and types.

```
   and : bool -> bool -> bool
    or : bool -> bool -> bool
   imp : bool -> bool -> bool
   neg : bool -> bool
forall : (i -> bool) -> bool
exists : (i -> bool) -> bool
```

We will demonstrate in this and the next sections that the extended logic programming language presented in Section 2 gives us a language in which we can write programs to manipulate these $\lambda$-terms in sophisticated ways. We begin here with some basic formula manipulation programs. These programs will illustrate that certain simple operations on formulas are handled quite differently in this setting than by traditional methods. The following is a small program which can be used to instantiate a universally quantified formula. The program produces an "instance" of a formula by replacing the outermost variables bound by universal quantification with new logic variables which can later become instantiated with specific terms. The `instantiate` predicate has type `bool -> bool -> o` where the first argument is the (possibly) universally quantified formula, and the second is its instantiated form.

```
instantiate (forall A) B :- instantiate (A T) B.
instantiate A A.
```

Note that each time the first definite clause is used, it introduces a new logic variable `T`. For example, it might instantiate the formula `(forall X\ (forall Y\ ((p X) imp (p Y))))` to `((p T1) imp (p T2))`. These logic variables can later be instantiated through unification.

As another example, the following program illustrates how to construct the negation normal form of a formula. There is one predicate, called `nnf` which has type `bool -> bool -> o`. The first argument is the input formula and the second is the output formula in negation normal form. The behavior of the program is fairly straightforward for propositional formulas. Since first-order unification is all that is needed to break a propositional formula into its subformulas, the algorithm proceeds by recursively descending the structure of the formula, building the normal form from the normal forms of the subformulas. The main departure from traditional algorithms occurs when determining the negation normal form of quantified formulas. For example, traditionally, finding the negation normal form of the formula $\neg(\forall x\ A)$ entails changing it to $\exists x\ \neg A$ and then finding the negation normal form of $\neg A$. The quantifier and bound variable are stripped off and $x$ becomes a free variable in $A$ during the rest of the procedure. Using our representation for formulas, the formula `(neg (forall A))` becomes `(exists X\ (neg (A X)))`. In order to obtain a subformula for which we can take the negation normal form, we must first apply the $\lambda$-term `(X\ (neg (A X)))` to something. The universal goal can be used here to put in a generic constant for `X` as in the following definite clause:

```
nnf (neg (forall A)) (exists B) :- pi X\ (nnf (neg (A X)) (B X)).
```

The GENERIC search operation will pick a constant `c`, and then the interpreter will search for the normal form `(B c)` of `(neg (A c))`. The use of the universal goal insures that the new constant used for `X` will not appear in `B`, and thus `B` will be an abstraction over this constant. The final negation normal form is then `(exists B)`. The complete algorithm is described by the following set of definite clauses.

```
nnf (A and B) (C and D) :- nnf A C, nnf B D.
nnf (A or B) (C or D) :- nnf A C, nnf B D.
nnf (A imp B) (C or D) :- nnf (neg A) C, nnf B D.
nnf (forall A) (forall B) :- pi X\ (nnf (A X) (B X)).
nnf (exists A) (exists B) :- pi X\ (nnf (A X) (B X)).
nnf (neg (neg A)) B :- nnf A B.
nnf (neg (A and B)) (C or D) :- nnf (neg A) C, nnf (neg B) D.
nnf (neg (A or B)) (C and D) :- nnf (neg A) C, nnf (neg B) D.
nnf (neg (A imp B)) (C and D) :- nnf A C, nnf (neg B) D.
nnf (neg (forall A)) (exists B) :- pi X\ (nnf (neg (A X)) (B X)).
nnf (neg (exists A)) (forall B) :- pi X\ (nnf (neg (A X)) (B X)).

nnf A A.
```

Other programs for normal form algorithms such as conjunctive, disjunctive, and prenex normal forms can be written similarly. For prenex normal form, as for negation normal form, the universal goal is used to handle quantified formulas.

Formula manipulation plays a large role in the task of implementing theorem provers. For example, inference rules are generally applied to specific formulas, modifying them as the search for a proof proceeds. Thus, the ways in which formulas are manipulated will continue to be important in the next sections as we discuss the construction of theorem provers in our logic programming setting. We begin, in the next section, by discussing how to specify inference rules for various proof systems.

# 4 Specifying Inference Rules

In this section we discuss how to specify inference rules of proof systems as definite clauses and illustrate the role of the six search operations of our extended logic programming language in using these rules to search for proofs.

In the examples below, each inference rule can be very naturally understood as combining a unification step and a search step, and thus has a natural rendering as a logic programming definite clause. We obtain complete non-deterministic theorem provers under the interpreter described in Section 2 from a set of definite clauses representing all of the inference rules for a proof system. This is the basis for our claim that logic programming is a suitable domain for specifying natural theorem provers.

In this section we will draw examples from a Gentzen sequent system and a natural deduction system. In addition to determining whether or not a sequent or formula is provable, the programs will also build and store the proofs as they are discovered. We use a particular representation for our proofs in the examples below. It is important to note that our main point is not to promote a particular representation as the "correct" one for sequent systems or natural deduction, but to illustrate that it is straightforward to represent proofs for many proof systems. The particular representation chosen should correspond to what the proof objects will be used for in a given proof system.

To place these inference rules in the context of the theorem provers in which they appear, we must consider the type declarations associated with the programs. The declarations for a theorem prover will consist of first, a set of declarations for the logical constants used in constructing formulas, such as those used in the negation normal form program in the previous section. Second, they will include declarations for specifying the terms used in constructing proof objects. Finally, they include the types given to the predicates used in search. For the theorem provers from which we draw our example inference rules, we will define one predicate called `proof` which has two arguments: a sequent or formula and its proof. Thus the type of the `proof` predicate in a sequential system and a natural deduction system are, respectively

```
sequent -> proof_object -> o     and     bool -> proof_object -> o.
```

We will give examples of objects of type `proof_object` as we present the inference rules.

It is important to note that we do not always need to construct proof objects. We may define the `proof` predicate to take only a formula as an argument. In this case, when a formula is provable, the result will simply be a "yes" answer. In all of our examples, we include the second argument for the proof objects to illustrate how they can be constructed. Then later, we demonstrate some possibilities for using them in computations.

The type declarations, as above, for the program predicates and terms are specified at the meta-level and are used by the interpreter to insure proper types. On another level more important to theorem proving, we would like to represent formulas as types and proofs as objects inhabiting these types as in [Howard 80, Martin-Löf 82, Harper, Honsell & Plotkin 87], so that theorem proving corresponds to type checking and type inference as in systems based on these formalisms (*e.g.* [Constable *et al.* 86, deBruijn 80]). On this level, formulas and sequents will represent the types for their corresponding proofs (*i.e.* the first argument to the `proof` predicate will represent the type of the second).

In Section 4.1, we begin with some examples of inference rules for the Gentzen LJ system for intuitionistic logic. Then, in Section 4.2, we consider the specification of natural deduction inference rules, which requires investigating some additional issues. The inference rules presented in each section will be discussed on two levels. First, they will have a declarative meaning, defining what a proof of the conclusion of a rule will be, based on the proofs of its premises. On this level, these clauses will give meaning to the objects used to represent proofs. Additionally, these clauses will have operational meaning. On this level, there will be a discussion of how each definite clause will be used by the interpreter of Section 2 during the search for proofs. In this discussion, we will exhibit the correspondence between the logical connectives of the object-language and the search operations of the meta-language.

## 4.1 Definite Clauses for Sequential Proof Systems

For the LJ proof system, in addition to the logical constants defined in Section 3, we need an additional constant to represent sequents. A sequent, in this system, has the form $\Gamma \longrightarrow A$ where $\Gamma$ is a list of formulas and $A$ is a formula. We add `sequent` to our set of primitive types (`bool` and `i`) and define `-->` as an infix operator as follows.

```
--> : (list bool) -> bool -> sequent
```

We begin with the following inference rule for a conjunction on the right side of the sequent (the LJ $\wedge$-R rule).

$$\frac{\Gamma \longrightarrow A \qquad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\ \wedge\text{-R}$$

It can be represented by the following definite clause.

```
proof (Gamma --> (A and B)) (and_r P1 P2) :- proof (Gamma --> A) P1,
                                              proof (Gamma --> B) P2.
```

The declarative meaning of such a clause may be stated: if `P1` is a proof of `(Gamma --> A)` and `P2` is a proof of `(Gamma --> B)`, then `(and_r P1 P2)` is a proof of `(Gamma --> (A and B))`.

The rule essentially gives meaning to and_r: it is a function from two proofs (the premises of the $\wedge$-R rule) to a new proof (its conclusion). In the context of logic programming declarations, the type of this term is specified as follows.

```
and_r : proof_object -> proof_object -> proof_object
```

In the formulas-as-types paradigm, as we stated, one way to view any of the inference rule definite clauses that we present is that they define the objects (proofs) that inhabit the types specified by a sequent or formula. In the case of the and_r rule, notice the correspondence to the constructive logic declaration (and_r P1 P2): $\Gamma \longrightarrow A \wedge B$ where (and_r P1 P2) is a proof of the judgement $\Gamma \longrightarrow A \wedge B$ when P1 and P2 are proofs of $\Gamma \longrightarrow A$ and $\Gamma \longrightarrow B$ respectively.

Operationally, this definite clause may be used by the interpreter when a goal matches a universal instance of the right side (a unification step). It is used, for example, in goal-directed search when attempting to find a proof of a sequent of the form (Gamma --> (A and B)). An AND search operation is then necessary to handle the conjunctive goal that results after backchaining. Thus one role of the AND search operation in theorem proving is to handle the subgoals generated by inference rules with more than one premise.

Next we consider the two LJ inference rules for proving disjunctions.

$$\frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B} \vee\text{-R} \qquad\qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B} \vee\text{-R}$$

These rules have a very natural rendering as the following logic programming clause.

```
proof (Gamma --> (A or B)) (or_r P) :- proof (Gamma --> A) P;
                                       proof (Gamma --> B) P.
```

Declaratively, this clause specifies the meaning of a proof of a disjunction. Here P is either a proof of (Gamma --> A) or (Gamma --> B). Operationally, the OR search operation is used here to arbitrate between choices of inference rules that can be applied to the same sequent. All propositional rules for Gentzen sequential systems can be very naturally understood as combining a first-order unification step, possibly followed by an AND or an OR search operation.

In the next examples, we will present some quantifier rules. Operationally, they will illustrate a use of the INSTANCE and GENERIC search operations and higher-order unification for theorem proving. Consider the following $\exists$-R inference rule

$$\frac{\Gamma \longrightarrow [x/t]A}{\Gamma \longrightarrow \exists x\, A} \exists\text{-R}$$

which can be written as the following definite clause.

```
proof (Gamma --> (exists A)) (exists_r P) :- proof (Gamma --> (A T)) P.
```

Note that T is a new logic variable introduced in the subgoal. This T is existentially quantified *i.e.* in its unabbreviated hereditary Harrop form the subgoal is equivalent to

$(\exists T\, (\mathit{proof}\ \Gamma\ \longrightarrow\ (A\ T)\ \ P)).$

Declaratively, the clause reads: if there exists a term T (of type i) such that P is a proof of (Gamma --> (A T)), then (exists_r P) is a proof of (Gamma --> (exists A)). Operationally, we rely on higher-order unification (in this case, second-order matching) to match the A of (exists A) to a function of type i -> bool. Then (exists A) is replaced by (A T) in the sequent to form the subgoal, where $\lambda$-application and normalization substitutes T for the bound variable in A. By making T a logic variable, we do not need to commit to a specific term for the substitution. It will later be assigned a value through unification if there is one that results in a proof. This introduction of a new logic variable is the role of the INSTANCE search operation.

Next consider the following $\forall$-R inference figure.

$$\frac{\Gamma \longrightarrow [x/y]A}{\Gamma \longrightarrow \forall x\ A}\ \forall\text{–R}$$

with the condition that $x$ is not free in $\Gamma$ or $A$. This restriction is handled by universally quantifying the subgoal to obtain:

```
proof (Gamma --> (forall A)) (forall_r P) :- pi T\ (proof (Gamma --> (A T)) (P T)).
```

Declaratively, it will read: if we have a function P that maps arbitrary terms T to proofs (P T) of the sequent (Gamma --> (A T)), then (forall_r P) is a proof of (Gamma --> (forall A)). In this case, the proof object (forall_r P) contains a functional argument, and thus forall_r is declared as follows.

```
forall_r : (i -> proof_object) -> proof_object
```

Operationally, the GENERIC goal is necessary to achieve the subgoal (pi T\ (proof (Gamma --> (A T)) (P T))). As stated in Section 2, one way to view the operational use of this goal is that the interpreter will pick a new constant c for the universally quantified variable, in this case for T. In the logic programming setting, in order for c to be a truly generic constant, we must insure that it will not appear in any of the logic variables in subsequent unifications. In this clause, in addition to not appearing in the sequent, c cannot appear in the proof. As a result we have a proof P which is a function from arbitrary terms (of type i) to a proof_object. The fact that P must be a function agrees with our declarative reading of this definite clause.

In our discussion of each of the inference rules, we have been considering the operational meaning of each of the definite clauses because we would like to use the `proof` program as a whole to do theorem proving. To use the interpreter of Section 2 for this task, we can present it with a goal formula of the form `(proof (Gamma --> Delta) P)`, where the sequent will be specified and the proof will not. (It will be a logic variable.) When we start with a goal of this form, at each step of the proof, the initial unification step needed to determine if a definite clause can be used involves only the first argument. This argument acts as the input while the second argument (the proof) is the output, which gets constructed as the subgoals are completed. Clearly, it is possible to give both arguments at the onset, and thus both would be important for the initial unification step. In this case, the program acts as a proof-checker. This dual role of theorem prover/proof-checker applies to all of the theorem provers we discuss in this paper.

## 4.2 Definite Clauses for Natural Deduction

The natural deduction inference rules presented in this section are from Gentzen's system as presented in [Prawitz 65]. Several of the introduction rules from this system resemble rules that apply to formulas on the right of the sequent in the LJ proof system. Those that correspond to the example inference rules given in the previous section are as follows.

$$\frac{A \qquad B}{A \wedge B}\wedge\text{-I} \qquad\qquad \frac{A}{A \vee B}\vee\text{-I} \qquad\qquad \frac{B}{A \vee B}\vee\text{-I}$$

$$\frac{[x/t]A}{\exists x\ A}\exists\text{-I} \qquad\qquad \frac{[x/y]A}{\forall x\ A}\forall\text{-I}$$

The ∀-I rule also has the proviso that $y$ cannot appear in $A$.

They can be translated to the following definite clauses which are all similar in appearance to their corresponding definite clauses presented in the previous section.

```
proof (A and B) (and_i P1 P2) :- proof A P1, proof B P2.

proof (A or B) (or_i P) :- proof A P; proof B P.

proof (exists A) (exists_i P) :- proof (A T) P.

proof (forall A) (forall_i P) :- pi T\ (proof (A T) (P T)).
```

Clearly, they have similar declarative readings, and proof terms are built in the same way as in their LJ counterparts. Operationally they also require the AND, OR, INSTANCE, and GENERIC search operations respectively.

In natural deduction, we have the additional task of specifying the operation of discharging assumptions. We illustrate how this can be accomplished using the $\supset$-I rule below.

$$\frac{\begin{array}{c}(A)\\ B\end{array}}{A \supset B} \supset\text{-I}$$

We can translate this rule to the following definite clause (which uses implication).

```
proof (A imp B) (imp_i P) :- pi PA\ ((proof A PA) => (proof B (P PA))).
```

This clause represents the fact that if P is a "proof function" which maps proofs of A to proofs of B such that given an arbitrary proof PA of A, (P PA) is a proof of B, then (imp_i P) is a proof of (A imp B). Here, the proof of an implication is represented by a function from proofs to proofs. The discharge of assumptions will always result in such proof functions. In this case, imp_i has the following type.

```
imp_i : (proof_object -> proof_object) -> proof_object
```

Operationally, the AUGMENT search operation plays a role in the discharge of assumptions. In this case, to solve the subgoal (pi PA\ ((proof A PA) => (proof B (P PA)))), the GENERIC goal is needed to pick a generic proof pa for the formula A. Then the AUGMENT goal is used to add the clause (proof A pa), a proof of the discharged assumption, to the current program. This clause is then available to use in the search for a proof of B. The proof of B will most likely contain instances of the proof of A (the term pa). The resulting function P is the abstraction over this term.

Representing proofs as functions, in addition to being a natural encoding of the operation of discharging assumptions, provides abstractions over proofs that can actually be applied to subproofs. For example, if we have a proof of (A imp B), and if A were a lemma that was later proved, we could apply P to this proof of A and obtain a proof of B directly. A definite clause to perform this operation might be as follows.

```
proof B (P PA) :- proof (A imp B) (imp_i P), proof A PA.
```

In the term (P PA), $\lambda$-application followed by normalization results in a new proof term with the actual proof term PA replacing all occurrences of the variable bound by $\lambda$-abstraction in P. When viewed in terms of proof trees of the style found in [Prawitz 65], this operation has the effect of substituting the proof of A above all occurrences of A that were discharged by this application of the $\supset$-I rule.

# 5 An Automatic Theorem Prover Using Depth-First Control

Complete non-deterministic theorem provers are obtained by translating all of the rules of a proof system to definite clauses using techniques such as those in the previous section. There, we demonstrated that such specification can be quite straightforward. If we want to go further and consider these specifications as executable programs for finding proofs, many more issues are raised. Controlling the search for a proof (*i.e.* forcing determinism) is a much more complicated task. For example, if we adopt the simple depth-first search algorithm of traditional logic programming, the order of the clauses (inference rules) becomes very important. It is unlikely that they can be ordered in such a way as to always avoid running into infinite branches in the search tree which cause the program to loop forever. Also, proofs of completeness for theorem provers using depth-first search might become quite complex. As a simple example, the left contraction rule of the Gentzen LK system and its corresponding definite clause are given below.

$$\frac{A, A, \Gamma \longrightarrow \Delta}{A, \Gamma \longrightarrow \Delta}\ \mathrm{contract} - \mathrm{L}$$

```
proof ([A | Gamma] --> Delta) (contract_l P) :-
        proof ([A,A | Gamma] --> Delta) P.
```

This rule always produces a subgoal with an extra copy of the first formula on the left of the sequent. Thus it will always be applicable when there is at least one formula on the left. This causes a problem since it could be applied repeatedly, making multiple copies of the same formula and preventing rules that appear after it from ever being attempted. Even if no other rules appear after it, its interactions with other rules must be considered. For example, both contraction rules have the same problem, so that when one is placed before the other, the second may never get applied. Also, since either contraction rule will always be applied to the first formula in a list, in order to insure completeness, we need a guarantee that each formula that needs to be doubled will appear at the front of the list at some point.

Although depth-first search is a naive and limited approach to the complex task of searching for proofs, at times it can be exploited successfully to build complete automatic theorem provers. As an example, in this section we will construct a theorem prover for a sequent-style calculus for first-order classical logic. In this case, by slightly modifying the LK inference rules, we are able to obtain a reasonable automatic theorem prover. We present the system, called LKC (C for Computational) and discuss the changes to the inference rules. The purpose of these changes is to minimize non-determinism in the search for proofs. The changes are similar to those made to LK to obtain the system G in [Gallier 86]. There, a search algorithm used to find proofs in the G system is described, and a completeness result for this algorithm established. The changes made to LK to obtain LKC are as follows.

(1) The interchange, contraction, thinning, and cut rules are removed.

We eliminate the cut rule because in goal-directed automatic proof, we would have to specify the "cut formula" that appears in the premises by introducing a new logical variable, and it is unlikely that unification could determine such cut formulas. By cut-elimination, this rule is not needed to obtain a complete theorem prover. We eliminate the structural rules since they can be applied to almost any sequent and to any formula in the sequent. We are then left with the rules that apply to a formula of a particular form (based on the main logical connective) in the conclusion, *i.e.* the introduction rules.

> (2) An inference rule can be applied to a formula at any position on the left or right of the sequent in the conclusion of a rule. In the premises, the subformulas will appear at the beginning of the lists of formulas on the left and right of the sequent. For example, the rule for implication on the right will have the following form.

$$\frac{A, \Gamma \longrightarrow B, \Delta_1, \Delta_2}{\Gamma \longrightarrow \Delta_1, A \supset B, \Delta_2} \supset\text{-R}$$

(3) Initial sequents are of the form $\Gamma_1, A, \Gamma_2 \longrightarrow \Delta_1, A, \Delta_2$.

(2) above eliminates the need for the interchange rules, and allowing initial sequents as in (3) eliminates the need for the thinning rules. The remaining changes to the inference rules are those that must be made to account for the removal of the contraction rules. For a proof that these changes preserve completeness, see [Miller 87].

(4) The $\supset$-L rule has the following form.

$$\frac{\Gamma_1, \Gamma_2 \longrightarrow A, \Delta \qquad B, \Gamma_1, \Gamma_2 \longrightarrow \Delta}{\Gamma_1, A \supset B, \Gamma_2 \longrightarrow \Delta} \supset\text{-L}$$

In the LK $\supset$-L rule, each formula in the conclusion appears in only one of the premises. The LKC $\supset$-L rule differs only in that all formulas (except the implication to which the rule is being applied) are duplicated in the premises. This is equivalent to repeatedly applying the contraction rule before applying the LK $\supset$-L rule. The LKC rule is better suited for goal-directed proof since, when applying the rule, it will not be known which formulas are needed in each premise to complete their respective proofs.

(5) The two rules for ∧-L and ∨-R are combined into one for each as follows.

$$\frac{A, B, \Gamma_1, \Gamma_2 \longrightarrow \Delta}{\Gamma_1, A \wedge B, \Gamma_2 \longrightarrow \Delta}\ \wedge\text{-L} \qquad\qquad \frac{\Gamma \longrightarrow A, B, \Delta_1, \Delta_2}{\Gamma \longrightarrow \Delta_1, A \vee B, \Delta_2}\ \vee\text{-R}$$

In the LK rules, only one of the two conjuncts on the left or disjuncts on the right appears in the premise. Again, the LKC ∧-L and ∨-R rules are a combination of contraction and the corresponding LK rules. At the time of application it will not be known which conjunct on the left or disjunct on the right will be needed to complete the proof.

(6) The ∀-L and ∃-R rules are of the following form.

$$\frac{[x/t]A, \Gamma_1, \Gamma_2, \forall x\ A \longrightarrow \Delta}{\Gamma_1, \forall x\ A, \Gamma_2 \longrightarrow \Delta}\ \forall\text{-L} \qquad\qquad \frac{\Gamma \longrightarrow [x/t]A, \Delta_1, \Delta_2, \exists x\ A}{\Gamma \longrightarrow \Delta_1, \exists x\ A, \Delta_2}\ \exists\text{-R}$$

The ∀-L and ∃-R rules also encompass a contraction, in this case, of the quantified formula. In addition, the quantified formula appears at the end of the list while the formula to which the substitution is applied appears at the beginning. The quantified formula is put at the end so that we can write a goal-directed theorem prover that applies rules to formulas in the order they appear in a list, so that all other formulas in the sequent will be examined before the quantifier rule is applied to the same formula again. Note that we will have to include a definite clause that applies both of these rules at once. Otherwise, in the case when both are applicable, placement of the definite clause for one before the clause for the other in any ordering would cause one to be applied repeatedly and prevent the other from ever being applied. These are the only rules that need this treatment because all other rules replace a formula in the conclusion with subformulas in the premises, and thus the sequent becomes "smaller" in the sense that the total number of logical connectives decreases. The program below represents a depth-first theorem prover for the LKC system. Based on the observations above, it should be straightforward to show that it is complete.

The sequents in the LK (and thus LKC) proof system are of the form $\Gamma \longrightarrow \Delta$ where $\Gamma$ and $\Delta$ are both lists of formulas. Thus, for this program the sequent arrow is an infix operator with the following declaration.

```
--> : (list bool) -> (list bool) -> sequent
```

This program also makes extensive use of the list predicates discussed in Section 2.

```
proof (Gamma --> Delta) (initial A) :- member A Gamma, member A Delta.

proof (Gamma --> Delta) (and_l P) :-
       member_and_rest (A and B) Gamma Gamma1,
       proof ([A,B | Gamma1] --> Delta) P.

proof (Gamma --> Delta) (and_r P1 P2) :-
       member_and_rest (A and B) Delta Delta1,
       proof (Gamma --> [A | Delta1]) P1,
       proof (Gamma --> [B | Delta1]) P2.

proof (Gamma --> Delta) (or_r P) :-
       member_and_rest (A or B) Delta Delta1,
       proof (Gamma --> [A,B | Delta1]) P.

proof (Gamma --> Delta) (or_l P1 P2) :-
       member_and_rest (A or B) Gamma Gamma1,
       proof ([A | Gamma1] --> Delta) P1,
       proof ([B | Gamma1] --> Delta) P2.

proof (Gamma --> Delta) (imp_r P) :-
       member_and_rest (A imp B) Delta Delta1,
       proof ([A | Gamma] --> [B | Delta1]) P.

proof (Gamma --> Delta) (imp_l P1 P2) :-
       member_and_rest (A imp B) Gamma Gamma1,
       proof (Gamma1 --> [A | Delta]) P1,
       proof ([B | Gamma1] --> Delta) P2.

proof (Gamma --> Delta) (neg_l P) :-
       member_and_rest (neg A) Gamma Gamma1,
       proof (Gamma1 --> [A | Delta]) P.

proof (Gamma --> Delta) (neg_r P) :-
       member_and_rest (neg A) Delta Delta1,
       proof ([A | Gamma] --> Delta1) P.

proof (Gamma --> Delta) (forall_r P) :-
       member_and_rest (forall A) Delta Delta1,
       pi T\ (proof (Gamma --> [(A T) | Delta1]) (P T)).

proof (Gamma --> Delta) (exists_l P) :-
       member_and_rest (exists A) Gamma Gamma1,
       pi T\ (proof ([(A T) | Gamma1] --> Delta) (P T)).

proof (Gamma --> Delta) (forall_r (exists_l P)) :-
       member_move_to_end (forall A1) Gamma Gamma1,
       member_move_to_end (exists A2) Delta Delta1,
       proof ([(A1 T1) | Gamma1] --> [(A2 T2) | Delta1]) P.

proof (Gamma --> Delta) (exists_r P) :-
```

```
        member_move_to_end (exists A) Delta Delta1,
        proof (Gamma --> [(A T) | Delta1]) P.

proof (Gamma --> Delta) (forall_l P) :-
        member_move_to_end (forall A) Gamma Gamma1,
        proof ([(A T) | Gamma1] --> Delta) P.
```

# 6 Construction of Tactic Theorem Provers

Tactics and tacticals provide a powerful and flexible device for building proof systems, and have been used with much success in recent years. As mentioned, examples of systems that have been developed using this style of theorem proving include the Edinburgh LCF theorem prover [Gordon, Milner & Wadsworth 79], and the Nuprl proof system at Cornell [Constable *et al.* 86]. Tactics and tacticals promote modular design of proof systems and provide flexibility in controlling the search for proofs. They allow an interactive proof environment to be enhanced with partial automation. Tactics provide the basic operations and inference rules for a particular proof system, while tacticals provide control mechanisms which can be used in combination with tactics to automate tedious details of building proofs, as well as to develop more complex proof strategies. As a result of the success of current tactic style proof systems in providing a mechanism for direct user involvement in the incremental construction of proofs, this method continues to receive much support in the theorem proving community [Milner 87].

Logic programming provides a very suitable environment for writing tactics and tacticals. They can be used to develop an interpreter that provides a much richer mechanism for controlling the search for proofs than depth-first search. In this section, we will illustrate how to construct tactic style theorem provers in our extended logic programming language.

In building tactic theorem provers, there will be certain definite clauses that will be common to all tactic provers while others will be specific to a particular logic. We will group sets of definite clauses together in program fragments or modules as described in Section 2. We will first present the modules that are general to all tactic theorem provers and then discuss the set of definite clauses that will be needed to specialize to a particular logic. We will illustrate with a theorem prover for natural deduction.

There will be one main predicate in any tactic prover which we call prove and give the following type.

```
        prove : tacticalexp -> goalexp  -> goalexp  -> o
```

This predicate has three arguments, the first of which is a tactical expression used in controlling search. The simplest kind of tactical expression is the name of a tactic (often corresponding to an inference rule). The second argument is the input goal which can be a complex structure containing formulas or sequents and their corresponding proofs. At the onset, the proofs will most likely be logic variables, which will become instantiated step by step as the formulas or sequents are proven. The third argument is the output goal which will contain the subgoals remaining after applying the tactical expression. There are three types of prove clauses. The first two are general to all tactic provers. They are the tacticals and the goal reduction clauses. Together they form the interpreter which controls the search for proofs. The tacticals specify control based on the first argument to the prove clause–the tactical expression–which indicates which inference rule(s) to apply. The goal reduction clauses are needed in the logic programming setting to handle complex goals that may be generated when applying inference rules. They specify control based on the structure of the second argument. We provide a goal structure corresponding to each of the search operations discussed in Section 2. The third type of prove clause includes all the definite clauses that are specific to a given proof system. These clauses are the tactics. The basic tactics encode the inference rules of a proof system. We translate each inference rule to a tactic, similar to the specification of inference rules presented in Section 4.

When gathered together, the modules presented in this section can be considered, on one hand, as a declarative specification for a tactic theorem prover. Under the interpreter described in Section 2 such programs specify non-deterministic theorem provers. On the other hand, we want to consider our theorem provers as deterministic executable programs. One purpose of adopting the tactic paradigm was to provide more control in the search for proofs. For example, we give "names" to the basic tactics. This allows search to be directed by specifying which rule to apply by calling it by name, so that only one (or a very small number) of tactics will be applicable at any one time. To obtain completely deterministic theorem provers, we must resolve the remaining non-determinism that results when more than one definite clause is applicable. We will resolve these cases as in Prolog where the clauses are attempted in the order they appear. Thus we still assume depth-first search control of the underlying logic programming language, but we write an interpreter on top of this language for tactics and tacticals.

## 6.1 Definite Clauses for Tactic Provers

In this section we will specify the modules Tacticals and GoalRed for the tacticals and goal reduction clauses, respectively. Together they form the meta-interpreter for tactic provers since they provide the mechanisms for controlling the application of the individual inference rules of proof systems. We first present the type declarations for both of these sets of definite clauses. The types tacticalexp and goalexp introduced earlier comprise the set of primitive types (along with the type o for logic programming propositions). The first group of declarations are the tacticals used in building tactical expressions. The second group is used to construct compound goals.

```
  then : tacticalexp -> tacticalexp -> tacticalexp
orelse : tacticalexp -> tacticalexp -> tacticalexp
 idtac : tacticalexp
```

```
      repeat : tacticalexp -> tacticalexp
         try : tacticalexp -> tacticalexp
    complete : tacticalexp -> tacticalexp

    truegoal : goalexp
     andgoal : goalexp -> goalexp -> goalexp
      orgoal : goalexp -> goalexp -> goalexp
     allgoal : (A -> goalexp) -> goalexp
  existsgoal : (A -> goalexp) -> goalexp
     impgoal : o -> goalexp -> goalexp

       prove : tacticalexp -> goalexp -> goalexp -> o
  goalreduce : goalexp -> goalexp -> o
```

Here, `andgoal` corresponds to the AND search operation, `orgoal` to OR, `allgoal` to GENERIC, `existsgoal` to INSTANCE, and `impgoal` to AUGMENT. The meaning of these goal structures as well as the tactical expressions will become apparent as we present the definite clauses in the next sections. The `prove` predicate is the main theorem proving predicate. `goalreduce` is an auxiliary predicate used in the `GoalRed` module to handle completed subgoals. Note the presence of the logic variable `A` in specifying the type of `allgoal` and `existsgoal`. We want the interpreter to be able to universally or existentially quantify over any type in any tactic prover. The type assigned to `A` in any given instance will depend on the particular tactic prover and the goal structures within that prover.

### 6.1.1 Definite Clauses for Tacticals

The definite clauses below provide control based on the structure of the tactical expression in the first argument. They correspond to the tacticals found in [Gordon, Milner & Wadsworth 79, Constable *et al.* 86]. and [Constable *et al.* 86]

```
(1) prove (then Tac1 Tac2) InGoal OutGoal :-
       prove Tac1 InGoal MidGoal, prove Tac2 MidGoal OutGoal.

(2) prove (orelse Tac1 Tac2) InGoal OutGoal :-
       prove Tac1 InGoal OutGoal; prove Tac2 InGoal OutGoal.

(3) prove idtac Goal Goal.

(4) prove (repeat Tac) InGoal OutGoal :-
       prove (orelse (then Tac (repeat Tac)) idtac) InGoal OutGoal.

(5) prove (try Tac) InGoal OutGoal :- prove (orelse Tac idtac) InGoal OutGoal.
```

```
(6) prove (complete Tac) InGoal truegoal :- prove Tac InGoal truegoal.
```

The then tactical allows composition of tactics. Tac1 is applied to the input goal, and then Tac2 is applied to the resulting goal. This tactical plays a fundamental role in combining the results of step-by-step proof construction. This role will become apparent in later examples. Here, we will simply note that MidGoal provides the sharing of logic variables across the two separate calls to tactics, so that the results from applying these tactics get combined. The orelse tactical simply uses the OR search operation so that Tac1 is attempted, and if it fails (in the sense that the logic programming interpreter cannot satisfy the meta-goal), then Tac2 is tried. The notion of success and failure of tactic application is defined here directly in terms of the success and failure of the interpreter of Section 2. The third tactical, idtac, returns an input goal unchanged. It is useful in constructing compound tactical expressions such as the one found in the repeat tactical. repeat is defined in terms of the other tacticals. It repeatedly applies a tactic until it is no longer applicable. The try tactical prevents failure by applying idtac when Tac does not succeed. It might be used, for example, in the second argument of an application of the then tactical. It prevents failure when the first argument tactic succeeds and the second does not. Finally the complete tactical tries to finish all goals. It will fail if there are any subgoals remaining after Tac is applied.

### 6.1.2 Definite Clauses for Goal Reduction

In contrast to the tacticals which break down tactical expressions, the following definite clauses direct control by examining goal expressions, in this case the input goal given by the second argument to the prove predicate. The definite clauses below represent the interpreter for the goal structures corresponding to the logic programming search operations.

```
(7)  prove Tac truegoal truegoal.

(8)  prove Tac (andgoal InGoal1 InGoal2) OutGoal :-
        prove Tac InGoal1 OutGoal1, prove Tac InGoal2 OutGoal2,
        goalreduce (andgoal OutGoal1 OutGoal2) OutGoal.

(9)  prove Tac (orgoal InGoal1 InGoal2) OutGoal :-
        prove Tac InGoal1 OutGoal; prove Tac InGoal2 OutGoal.

(10) prove Tac (allgoal InGoal) OutGoal :-
        pi T\ (prove Tac (InGoal T) (OutGoal1 T)),
        goalreduce (allgoal OutGoal1) OutGoal.

(11) prove Tac (existsgoal InGoal) OutGoal :-
        prove Tac (InGoal T) OutGoal.
```

```
(12) prove Tac (impgoal D InGoal) OutGoal :-
        (D => (prove Tac InGoal OutGoal1)),
        goalreduce (impgoal D OutGoal1) OutGoal.

goalreduce (andgoal truegoal Goal) OutGoal :- goalreduce Goal OutGoal.

goalreduce (andgoal Goal truegoal) OutGoal :- goalreduce Goal OutGoal.

goalreduce (orgoal truegoal Goal) truegoal.

goalreduce (orgoal Goal truegoal) truegoal.

goalreduce (allgoal T\ truegoal) truegoal.

goalreduce (impgoal D truegoal) truegoal.

goalreduce Goal Goal.
```

Note that clause (8) transfers the object level andgoal to an $\wedge$ (represented by ,) at the meta-level (the logic programming language). The other definite clauses behave similarly for the other corresponding search operations.

In writing programs for tactic theorem provers, we can actually eliminate the definite clause above for existsgoal from the interpreter because this goal can be handled directly. We can introduce a new logic variable directly into the program. This is achieved by replacing every goal of the form (existsgoal G) by (G T) for some new T. The remaining compound goals require the extra control of the interpreter given by the definite clauses above.

The goalreduce predicate is provided to handle the cases when a subgoal is achieved. Then the output goal (third argument) gets the value truegoal.

## 6.2 Specifying Tactics

The modules of the previous section will be included in any tactic theorem prover. In this section we illustrate how to specialize tactic provers to a particular proof system. We choose the Gentzen NK natural deduction system, and in the next subsections specify definite clauses for the inference rules and for a proof editor to interface to the user. Each new module will add some new declarations and new clauses for the prove predicate. In Section 6.3, we demonstrate how to put these modules together to obtain a complete tactic prover.

### 6.2.1 Inference Rules as Tactics

We will call the module containing the inference rules for natural deduction NDrules. The declarations for this set of clauses will include those for basic first-order logic as discussed in

Section 3. Again, we will construct proofs, so we also include the declarations for natural deduction proof objects. The exact form that these proof objects take will not be important to the presentation in this section. Finally, we must add declarations for the basic tactical expressions and goals. We begin with the tactics corresponding to the inference rules. Each inference rule tactic is given a name which is a token of type `tacticalexp`. Thus declarations of the form:

```
        and_i_tac : tacticalexp
```

are included for every tactic. The basic goal will contain the formula to be proved and its proof and is declared as follows.

```
        proofgoal : bool -> proof_object -> goalexp
```

We will call goals of the form `(proofgoal A P)` the *atomic* goals of the natural deduction theorem prover, in contrast to *compound* goals built from the goal constructors in Section 6.1.2. We also introduce a new predicate called `rule` which has the same arguments and function as the `prove` predicate, but allows the inference rules to be distinguished from the more general `prove` definite clauses. We then include the clause

```
prove Tac InGoal OutGoal :- rule Tac InGoal OutGoal.
```

In Section 4, we required the premises of an inference rule to have proofs in order to build a proof for the conclusion. Here, an inference rule tactic will only complete one step of the proof. The input goal specifies the formula to be proven and the output goal specifies the subgoals (premises of an inference rule) which still need to be proven after a rule is applied. The interpreter can then take control to direct the remaining search to achieve the incomplete subgoals. The basic form of a tactic is illustrated by the following example for $\wedge$-I.

```
rule and_i_tac (proofgoal (A and B) (and_i P1 P2))
               (andgoal (proofgoal A P1) (proofgoal B P2)).
```

It can be applied whenever the formula in the input goal is a conjunction. Since there are two subgoals which must both be completed, the output goal is specified using `andgoal`, which is later handled by clause `(8)` from the `GoalRed` module.

We will use another predicate, called `proof` (similar to the `proof` predicate in Section 5) to represent discharged assumptions. As assumptions are discharged, clauses of the form `(proof A P)` will be added to the goal structure and eventually to the program as in the following definite clause for the $\supset$-I rule.

```
rule imp_i_tac (proofgoal (A imp B) (imp_i P))
               (allgoal PA\ (impgoal (proof A PA) (proofgoal B (P PA)))).
```

These proof clauses, once they are added to the program by clause (12) from the GoalRed module are examined by many of the tactics for elimination rules which build upon the proofs contained in them, and add these larger proofs to the program in the form of new proof clauses. For example the tactic for the $\wedge$-E rule looks for a program clause of the form (proof (A and B) P).

```
rule and_e_tac (proofgoal C PC)
               (impgoal (proof A (and_e1 P))
                 (impgoal (proof B (and_e2 P)) (proofgoal C PC))) :-
  proof (A and B) P.
```

The two new clauses (proof A (and_e1 P)) and (proof B (and_e2 P)) become part of the output goal. They will then be added to the program also. The remaining inference rules are given below. We use the constant perp of type bool to represent the formula $\perp$.

```
rule or_i_tac (proofgoal (A or B) (or_i P))
              (orgoal (proofgoal A P) (proofgoal B P)).

rule forall_i_tac (proofgoal (forall A) (forall_i P))
                  (allgoal T\ (proofgoal (A T) (P T))).

rule exists_i_tac (proofgoal (exists A) (exists_i P))
                  (existsgoal T\ (proofgoal (A T) P)).

rule neg_i_tac (proofgoal (neg A) (neg_i P))
               (allgoal PA\ (impgoal (proof A PA) (proofgoal perp (P PA)))).

rule or_e_tac (proofgoal C (or_e P P1 P2))
              (andgoal (allgoal PA\ (impgoal (proof A PA) (proofgoal C (P1 PA))))
                       (allgoal PB\ (impgoal (proof B PB) (proofgoal C (P2 PB))))) :-
  proof (A or B) P.

rule imp_e_tac (proofgoal C PC)
               (impgoal (proof B (imp_e P PA)) (proofgoal C PC)) :-
  proof (A imp B) P, proof A PA.

rule forall_e_tac (proofgoal C PC)
                  (existsgoal T\ (impgoal (proof (A T) (forall_e P))
                                          (proofgoal C PC))) :-
  proof (forall A) P.

rule exists_e_tac (proofgoal C (exists_e P PC))
                  (allgoal T\ (allgoal PA\ (impgoal (proof (A T) PA)
                                                    (proofgoal C (PC T PA))))) :-
  proof (exists A) P.

rule neg_e_tac (proofgoal perp (neg_e P1 P2)) truegoal :-
  proof (neg A) P1, proof A P2.
```

```
rule perp_tac (proofgoal A (contra PA))
              (allgoal P\ (impgoal (proof (neg A) P) (proofgoal perp (PA P)))).

rule close_tac (proofgoal A P) truegoal :- proof A P.
```

### 6.2.2 A Proof Editor

Providing a means for accommodating user interaction is one of the strong points of tactic theorem provers. One way to provide an interface to the user in this paradigm is by writing tactics that request input. A very simple tactic for this purpose is as follows.

```
prove query (proofgoal A P) OutGoal :-
   write A, write "Enter tactic:", read Tac,
   prove Tac (proofgoal A P) OutGoal.
```

Here we have a tactic that, for any atomic input goal, will present the formula to be proved to the user, query the user for a tactic to apply to the input goal, then apply the input tactic. We restrict its application to atomic goals so that the GoalRed clauses will break down compound goals and present the subgoals one by one to the user. Note that the write and read predicates used here are outside the logic of hereditary Harrop formulas, yet they are necessary for a practical proof editor. As in Prolog, (write A) prints A to the screen and will always succeed and (read A) prompts the user for input and will succeed if A unifies with the input.

Using this tactic, the following tactic, named interactive, represents a proof editor for natural deduction for which the user must supply all steps of the proof.

```
prove interactive InGoal OutGoal :- prove (repeat query) InGoal OutGoal.
```

Note that the query tactic only operates on atomic goals. This means that any assumptions that have been discharged along the way (causing them to become part of an impgoal structure) must be added to the current program using definite clause (12) before the query tactic can be attempted. A practical interactive prover needs to present these assumptions to the user so that it is possible to work forward from the assumptions in addition to backward from the conclusion. One way to add this capability is to use the following goal reduction clause in place of (12) during interactive proof construction.

```
prove Tac (impgoal D InGoal) OutGoal :-
   write "Adding", write D,
   (D => (prove Tac InGoal OutGoal1)),
   write "Removing", write D,
   goalreduce (impgoal D OutGoal1) OutGoal;
   write "Removing", write D, fail.
```

This clause will inform the user of the clause that is being added to the program, which will then be available during the execution of the tactical expression Tac. If the tactical expression is completed successfully (*i.e.* at the meta-interpreter level the goal (prove Tac InGoal OutGoal) succeeds), the user is informed that D is no longer available. If OutGoal still contains subgoals to be completed, the task of completing them must be accomplished in the program environment that no longer contains D (unless it is added again). D also becomes unavailable in the case when the goal fails during the execution of the tactical expression. The user is notified and the definite clause as a whole fails.

These additions to the tactic prover will still not be sufficient, in general, for interactive theorem proving in a natural deduction setting. For example, if there is more than one conjunction among the discharged assumptions, the $\wedge$-I rule will be applicable in more than one way. The user needs the capability to specify which formula to apply the tactic to. One way to solve this problem is to extend the program with inference rule tactics that request input from the user. Since tactics are modular, this is easily accomplished by adding to NDrules or creating a new module with tactics such as:

```
rule and_e_query (proofgoal C PC)
     (impgoal (proof A (and_e1 P))
        (impgoal (proof B (and_e2 P)) (proofgoal C PC))) :-
   write "Enter conjunction:", read (A and B), proof (A and B) P.
```

The user must then enter enough information so that the input will unify with the desired conjunction.

All of the definite clauses above will be grouped in a module called ProofEd. In general, proof editors will be specific to the proof system that is being implemented. The above examples for natural deduction illustrate that the tactic paradigm provides flexibility in writing such proof editors.

## 6.3 A Tactic Theorem Prover

We can now define the top level module for the natural deduction theorem prover. This program will be named TacProver and contains only the following definite clause.

```
prove_top Tac InGoal OutGoal :-
   (Tactical => (GoalRed => (NDrules => (ProofEd => (prove Tac InGoal OutGoal))))).
```

As stated in Section 2, the use of a module name in a definite clause stands for the set of all of the definite clauses in that module.

Using this program, we can, for example, attempt to prove some formula `A` with the `interactive` tactic. This entails trying to establish

$$\texttt{TacProver} \vdash_I \texttt{prove\_top interactive (proofgoal A P) truegoal.}$$

The `P` of the goal formula is an existentially quantified variable *i.e.* it is a logic variable that will be instantiated to a proof (or partial proof) if the interpreter succeeds on this goal.

In the next subsections we will discuss other tactics that may be used to extend the natural deduction theorem prover specified by `TacProver`. To include them, we can either import them by including them as in the above `prove_top` definite clause, or we can access them during the proof process as will be discussed in Section 6.4.3.

## 6.4 Defining New Tactics and Tacticals

So far we have presented the core of a tactic style theorem prover for natural deduction including the inference rule tactics and a facility for interactive theorem proving. We will now describe some possibilities for adding new tactics and tacticals to enhance the proof environment.

### 6.4.1 Induction

There are many other inference rules that we could add to increase the capabilities of our natural deduction theorem prover. Like the main core of inference rules, new rules can be added as basic tactics. In this section, we discuss the addition of induction rules. We present inference rules for induction on different kinds of structures. We will also want to add information about the relations and operations on these structures to the theorem proving environment. Here we discuss only the induction inference rules, and illustrate the form that these new rules take in the tactic setting.

An inference rule for non-negative integer induction in the natural deduction proof system might look something like the following where the base case and the inductive case specify two separate subproofs.

$$\frac{[x/0]A \qquad \begin{matrix}([x/n]A) \\ [x/(n+1)]A\end{matrix}}{\forall x\ A}$$

We translate it to the following definite clause.

```
rule induction (proofgoal (forall A) (ind P1 P2))
               (andgoal (proofgoal (A 0) P1)
                        (allgoal N\ (allgoal P\ (impgoal (proof (A N) P)
                          (proofgoal (A (N + 1)) (P2 N P)))))).
```

In a similar manner, we can specify definite clauses for induction over other structures. For example a rule for induction on lists could be specified as follows.

```
rule list_induction (proofgoal (forall A) (list_ind P1 P2))
                    (andgoal (proofgoal (A nil) P1)
                          (allgoal L\ (allgoal PL\ (impgoal (proof (A L) PL)
                            (allgoal X\ (proofgoal (A (cons X L)) (P2 L PL X))))))).
```

### 6.4.2 Compound Tactics

Another way to add to the tactic database is to use existing tactics and tacticals to define compound tactics. Such tactics provide partial automation by applying some combination of inference rules. They range in complexity from automating simple details of proof construction to encoding more complex proof heuristics and strategies. The `interactive` tactic was a simple example of a compound tactic. Another example is the following tactic, named `intro` which does backward construction of a proof by applying some of the introduction rules to a formula before giving control to the user.

```
prove intro InGoal OutGoal :-
  prove (repeat (orelse imp_i_tac (orelse and_i_tac (orelse exists_i_tac
                (orelse forall_i_tac query))))) InGoal OutGoal.
```

It is also possible to write tactics which integrate other programs written in the logic programming language. For example, in a tactic theorem prover for the LKC system of Section 5, if the automatic theorem prover presented there were available, the `automatic` tactic below could be included, so that the tactic prover has access to completely automated proof construction.

```
prove automatic (proofgoal (Gamma --> Delta) P) truegoal :- proof (Gamma --> Delta) P.
```

This tactic works by "calling" the `proof` predicate of the automatic theorem prover to complete the proof of the sequent. Automatic construction of proofs in LKC also provides an example of a tactic that can be constructed from existing tactics. If there were tactics corresponding to each LKC rule, the following tactic would furnish the same automation.

```
prove automatic InGoal OutGoal :-
  prove (repeat (orelse initial_tac and_l_tac and_r_tac or_r_tac or_l_tac
                        imp_r_tac imp_l_tac neg_l_tac neg_r_tac forall_r_tac
                        exists_l_tac (then exists_r_tac forall_l_tac)
                        exists_r_tac forall_l_tac))
        InGoal OutGoal.
```

For readability, we use the abbreviation `(orelse Tac`$_1$ ... `Tac`$_n$`)` to represent `(orelse Tac`$_1$... `(orelse Tac`$_{n-1}$ `Tac`$_n$`)...)`. Note that the expression `(then exists_r_tac forall_l_tac)` has the effect of applying both the ∃-R and ∀-L rules. As was argued in Section 5 this combination is essential for insuring the completeness of the theorem prover and the termination of the program for provable sequents. In the tactic setting, we do not need a separate definite clause to handle this case as we did in the direct implementation.

### 6.4.3 Accessing Modules Dynamically

One can imagine that with a growing library of tactics, it might be desirable to organize tactics into modules containing sets of related tactics and be given the flexibility to access only those that are needed at different points during proof construction. This selective use of tactics can be achieved using the AUGMENT search operation which allows dynamic access of modules. To do this we first add a new tactical called **use_module** which takes a module name and a tactical expression, and adds the module to the current program so that the new definite clauses are available during the execution of the tactical expression. To add this capability we add the definite clause below to the `Tacticals` module.

```
prove (use_module Mod Tac) InGoal OutGoal :- (Mod => (prove Tac InGoal OutGoal)).
```

If this clause is used in a setting where the current program is the set of clauses `Prog`, then after a BACKCHAIN on this clause followed by an AUGMENT operation which adds the module `Mod` we are left with the following meta-level sequent to establish.

$$\texttt{Prog,Mod} \vdash_I \texttt{(prove Tac InGoal OutGoal).}$$

The effect here is similar to that described for discharging assumptions. In this case, all of the clauses in `Mod` will be available during the execution of the tactical expression `Tac`. Upon completion of `Tac` there may be subgoals remaining in OutGoal. Unless added again, the clauses of `Mod` will not be available during the completion of these subgoals.

Note the distinction between theorem proving in the meta-language ($\vdash_I$) and theorem proving in the object language (establishing `truegoal` under the rules for the NK system). Successful completion of `Tac` means that the above meta-sequent has been proven by the interpreter. If `OutGoal` still contains incomplete subgoals, then at the object level, the formula has not yet been proven. In this case, the object-level search for a natural deduction proof may continue in the previous environment that did not contain `Mod`.

# 7 Proof Manipulations

One important characteristic of all of the inference rules and theorem provers that we have been discussing in this paper is that they can construct proof terms as they prove theorems. There are many options for representing proof terms and example representations were given whenever inference rules were presented. In this section, we show that such proof objects can be useful by presenting some procedures that employ these objects for different purposes.

From the straightforward declarative interpretations, we saw that all of the logic programming renderings of the inference rules and theorem provers that have been presented are quite natural, and that the proof terms represent fairly natural encodings of the application of these inference rules. The first procedure we discuss below takes this idea a step further and produces English text, a more natural rendering of these proof terms. This procedure takes the form of a simple mapping from proof terms to strings of text. The second procedure below constructs Craig style interpolation formulas based on the structure of proof terms. This is followed by a discussion of some possibilities for constructing programs from proofs.

The remaining procedures involve transformations of proofs to different proofs. We discuss how one might go about specifying algorithms for proof normalization of natural deduction proofs and cut-elimination for sequential proofs. Finally, we discuss some possibilities for proof by analogy, in this case using one proof as a guiding proof to construct a structurally analogous proof of a related theorem.

All of these procedures proceed by recursively descending through the structure of proof terms. At many points in these algorithms, we need to know the formula or sequent associated with a proof or subproof. In the formulas-as-types paradigm, this information corresponds to the type of the object. On one hand, we may be given the formula or sequent as an additional argument along with its proof object, but we still might need to know the types of the subproofs. This is the situation in many of the programs presented in this section. In each case, we are able to get the type of subproofs from the type of the input proof by using the `proof` program in its proof-checking capacity. On the other hand, for the case when the type of the proof is not explicitly given, the question is whether or not it can be obtained from the proof. It might be contained explicitly in the proof term, or it might be possible to deduce it. We have discussed the `proof` program in terms of theorem proving and proof-checking. We can now ask if it is possible to use it for "type inference" where the proof object is specified at the onset and the formula or sequent is not (*i.e.* is a logic variable). Using the proof representation we have chosen, this is usually not possible. Proof terms, as we have specified them, store the complete structure of proof trees, but contain very little type information. For example, using the LKC prover we obtain the proof term

```
(and_l (or_r (forall_l (initial (q (f b))))))
```

for the sequent $p(a) \wedge \forall x\; q(x) \;\longrightarrow\; \forall y\; s(y) \vee q(f(b))$. But note that this also is a proof for $\forall x\; q(x) \wedge r \;\longrightarrow\; q(f(b)) \vee z$.

There is clearly a tradeoff between the amount of information stored in proof terms and the extra work that must be done to obtain type information. In this presentation, we emphasize the declarative reading of the programs and continue to opt for economy and readability in proof objects. Thus we include the type of proofs as an argument, and demonstrate how we use the `proof` predicate to associate subproofs with types. In terms of program execution, this extra computation is often inefficient, so it is important to be aware of other possibilities.

## 7.1 Building Explanations from Proofs

We will illustrate the construction of explanations using proof terms as generated by the LKC theorem prover of Section 5. These proof terms can be viewed functionally. In the case of proof explanation, each inference rule in a 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 or formula. For example, a term of the form `(or_l P1 P2)` generated by the definite clause for the ∨-L rule:

```
proof (Gamma --> Delta) (or_l P1 P2) :-
        member_and_rest (A or B) Gamma Gamma1,
        proof ([A | Gamma1] --> Delta) P1,
        proof ([B | Gamma1] --> Delta) P2.
```

represents a proof using case analysis. Assume that `P1` is interpreted as `Text1` which argues that `Delta` follows from `A` and `Gamma1`, and that `T2` is interpreted as `Text2` which argues that `Delta` follows from `B` and `Gamma1`. The interpretation of `(or_l P1 P2)` would then need to be an argument that `Delta` follows from `(A or B)` and `Gamma1`. This is easily done if we make the interpretation of `or_l` be the function which takes `Text1` and `Text2` into the following text.

> We have two cases. Case 1: Assume `A`. `Text1` Case 2: Assume
> `B`. `Text2` Thus, in either case, we have `Delta`.

We will have one logic programming definite clause corresponding to the lexical interpretation of each inference rule. Using the above interpretation, the following is the clause for the ∨-L rule.

```
explain (Gamma --> Delta) (or_l P1 P2) Text :-
        member_and_rest (A or B) Gamma Gamma1,
        explain ([A | Gamma1] --> Delta) P1 Text1,
        explain ([B | Gamma1] --> Delta) P2 Text2,
        append [(boolstr "Assume"),A | Text1]
               [(boolstr "Assume"),B | Text2] Text3,
        append Text3 [(boolstr "Thus in either case we have") | Delta] Text.
```

There are several technical details to note about this program. First of all we will only be concerned here with "lexicalizing" inference rules. Formulas themselves will be left as formulas. Also, we represent explanations as lists of formulas (items of type `bool`). We define a function `boolstr` which acts as a coercion function from strings to type `bool` (*i.e.* has type `string -> bool`) so that both formulas and strings may appear in the list. We can then write a program which takes an explanation and prints it out in a slightly more readable form without the list notation, string quotations, or coercion functions.

The above clause is actually the same as the clause specifying the ∨-L rule in the LKC prover, except that the `explain` predicate has an extra argument for the explanation. Operationally, the `explain` program will have both a theorem prover and proof-checker within it. If only the sequent is specified, the explanation and proof will be constructed simultaneously. If the sequent and proof are specified, the program will act as a proof-checker at the same time that it is building an explanation. This proof-checking is an example of how we obtain the "type" of the subproofs given a proof object and its type. In this clause, there may be more than one disjunction in `Gamma`. The `member_and_rest` predicate extracts the first one, and the next two subgoals attempt to verify that the subproofs `P1` and `P2` are proofs of the sequents with the ∨-L rule applied to the chosen disjunction. If these subgoals fail, backtracking takes place, and the `member_and_rest` predicate must find another disjunction, continuing until the correct one is found.

In sequent proof systems, recall that in proof terms of the form `(exists_l P)` and `(forall_r P)` corresponding to the ∃-L and ∀-R rules, the proof `P` of the premise is an abstraction from terms to proofs (has type `i -> proof_object`). The explanation for the corresponding premise, `Text`, will also be an abstraction over terms (will have type `i -> (list bool)`). In these cases, we explain the proof using an arbitrary term obtained by introducing a new logical variable, `Var`, and applying the explanation to it. Then, using the following explanations, the clauses for these rules would be as below.

Explanations:

∃-L: Choose `Var` such that `(A Var). (Text Var)`

∀-R: `(Text Var)` Since `Var` was arbitrary we have `Delta`.

Clauses:

```
explain (Gamma --> Delta) (exists_l P) Text :-
       member_and_rest (exists A) Gamma Gamma1,
       pi T\ (explain ([(A T) | Gamma1] --> Delta) (P T) (Text1 T)),
       append [(boolstr "Choose"),(var Var),(boolstr "such that"),(A Var)]
              (Text1 Var) Text.

explain (Gamma --> Delta) (forall_r P) Text :-
       member_and_rest (forall A) Delta Delta1,
pi T\ (explain (Gamma --> [(A T) | Delta1]) (P T) (Text1 T)),
append (Text1 Var) [(boolstr "Since"),(var Var),
                    (boolstr "was arbitrary we have") | Delta] Text.
```

Note that we have introduced the additional coercion function: `var : i -> bool` to allow terms of type `i` to appear in explanations. The following is a complete explanation obtained from a proof term for the formula which states that a reflexive transitive relation is symmetric on its domain: $\forall x \forall y \forall z\ (R(x,y) \wedge R(y,z) \supset R(x,z)) \wedge \forall x \forall y\ (R(x,y) \supset R(y,x)) \supset \forall x (\exists y\ R(x,y) \supset R(x,x))$.

```
Assume ((forall X\ (forall Y\ (forall Z\ (((r X Y) and (r Y Z)) imp (r X Z)))))
and (forall X\ (forall Y\ ((r X Y) imp (r Y X))))).
Assume (exists Y\ (r V1 Y)). Choose V2 such that (r V1 V2).
By modus ponens, we have (r V2 V1). Hence, ((r V1 V2) and (r V2 V1)).
By modus ponens, we have (r V1 V1). Since V1 was arbitrary, we have
(forall X\ (exists Y\ (r X Y)) imp (r X X)).
```

## 7.2 Finding Interpolants

In this section, we describe a program to construct an interpolation formula from proof terms representing proofs for *Craig-sequents* as defined in [Smullyan 68]. A formula $X$ is called an *interpolation* formula for a sequent $\Gamma \longrightarrow \Delta$ if all predicates and constants of $X$ occur in both $\Gamma$ and $\Delta$, and $\Gamma \longrightarrow X$ and $X \longrightarrow \Delta$ are both provable. Craig's Interpolation Lemma states that for any provable sequent $\Gamma \longrightarrow \Delta$, if $\Gamma$ and $\Delta$ have at least one predicate in common, then it has an interpolation formula. We introduce a new set of definite clauses that will specify a theorem prover for Craig's sequent style proof system. In addition, we add a third argument, as we did in the `explain` program, in this case for the interpolation formula.

Craig-sequents are similar to Gentzen sequents, except that they contain an interpolation formula. When a Craig-sequent $\Gamma \rightarrow X \rightarrow \Delta$ is provable in this system, this means that the ordinary sequent $\Gamma \longrightarrow \Delta$ is provable and $X$ is its interpolant. In the proof system presented in [Smullyan 68], the negation normal form of the formulas in the sequent is constructed "during" the proof process. For example, if a formula $\neg(A \vee B)$ appears on the left of the sequent, the $\wedge$-L rule is applied to the formula $\neg A \wedge \neg B$. To simplify matters, we will assume all formulas in the sequent are initially in negation normal form. The resulting program could easily be expanded to the more general system. The inference rules for this system, and thus their definite clause specifications are similar to those in the Gentzen LK system. As examples, the $\wedge$-R and $\forall$-R rules are given below. In the $\forall$-R rule, the proviso that $y$ cannot appear in the conclusion also applies to the interpolant $X$ since the interpolant can only contain constants that appear on both sides.

$$\frac{\Gamma \rightarrow X \rightarrow A, \Delta \qquad \Gamma \rightarrow Y \rightarrow B, \Delta}{\Gamma \rightarrow X \wedge Y \rightarrow A \wedge B, \Delta} \wedge\text{-R} \qquad \frac{\Gamma \rightarrow X \rightarrow [y/x]A, \Delta}{\Gamma \rightarrow X \rightarrow \forall x\ A, \Delta} \forall\text{-R}$$

They may be specified as follows with the interpolant as the third argument.

```
interpolate (Gamma --> Delta) (and_r P1 P2) (X and Y) :-
        member_and_rest (A and B) Delta Delta1,
        interpolate (Gamma --> [A | Delta1]) P1 X,
        interpolate (Gamma --> [B | Delta1]) P2 Y.

interpolate (Gamma --> Delta) (forall_r P) X :-
        member_and_rest (forall A) Delta Delta1,
        pi T\ (interpolate (Gamma --> [(A T) | Delta1]) (P T) X).
```

For the $\exists$-R and $\forall$-L rules, there is more than one possibility for the interpolant depending on whether or not the substitution term still appears on both sides of the conclusion after the rule is applied. The rules for $\exists$-R, for example, are as below where the first one applies when $t$ does not occur in $X$ at all, or if it does, it also occurs in $\Gamma$ and $\exists x\ A, \Delta$, *i.e.* if $t$ occurs in the interpolant of the premise, then after the rule is applied, $t$ must still occur on both sides of the conclusion. The second applies when $t$, which appears on the right of the premise, does not occur on the right of the conclusion. As a result $t$ cannot appear in the interpolation formula. These conditions represent a new kind of proviso that must be satisfied by the interpolation program.

$$\frac{\Gamma \rightarrow X \rightarrow [t/x]A, \Delta}{\Gamma \rightarrow X \rightarrow \exists x\ A, \Delta} \exists\text{-R} \qquad \frac{\Gamma \rightarrow [t/x]X \rightarrow [t/x]A, \Delta}{\Gamma \rightarrow \exists x\ X \rightarrow \exists x\ A, \Delta} \exists\text{-R}$$

We can represent these two possibilities using the following definite clauses.

```
interpolate (Gamma --> Delta) (exists_r P) X :-
        member_and_rest (exists A) Delta Delta1,
        interpolate (Gamma --> [(A T) | Delta1]) P X.

interpolate (Gamma --> Delta) (exists_r P) (exists X) :-
        member_and_rest (exists A) Delta Delta1,
        interpolate (Gamma --> [(A T) | Delta1]) P (X T).
```

To satisfy the conditions, we must insure that only the correct one will be applied. We will show that such a proviso can be satisfied by requiring queries of a particular form. We illustrate the form these queries take with an example (from [Gallier 86]). Suppose we have the proof

```
(and_l (or_r (forall_l (initial (q (f b))))))
```

for the sequent $p(a) \wedge \forall x\, q(x) \longrightarrow \forall y\, s(y) \vee q(f(b))$. To find the interpolant, we form a query by taking the universal closure (in the meta-language) over all of the predicates and constants in the sequent, and then specifying which ones can appear in the interpolant (*i.e.* those that appear in both sides of the sequent). In our example, only $q$ appears on both sides. We query the logic programming interpreter with the following goal.

```
(pi P\ (pi A\ (pi Q\ (pi S\ (pi F\ (pi B\
  (interpolate
    ([((P A) and (forall X\ (Q X)))] --> [((forall Y\ (S Y)) or (Q (F B)))])
    (and_l (or_r (forall_l (initial (Q (F B))))))
    (I Q))))))))
```

Upon successful completion of this goal, none of the variables of the universal closure will appear in `I` since `I` is within their scope, and only `Q` will appear at all in the interpolation formula `(I Q)`. As a result `I` will be an abstraction over the predicate `Q`. We can view this as "permitting" `Q` to appear in the interpolation formula, while preventing the appearance of any of the other constants or variables. Note that we can similarly view the subgoal

```
interpolate (Gamma --> [(A T) | Delta1]) P (X T)
```

in the second definite clause for the ∃-R rule as permitting an additional constant `T` to appear in the interpolation formula of the subtree above the conclusion of this rule.

## 7.3 Extracting Programs from Proofs

One way we have been viewing proofs of formulas is that the proof exhibits an element of the type specified by the formula. Certain formulas such as $A \supset B$ can be considered to have "functional type" and in certain cases, when the proof contains "constructive content," we can view their proofs as functions from elements of type $A$ into elements of type $B$. We would like to be able to use the proof terms to extract executable code for these functions. Such program extraction from proofs provides a method for verified programming [Bates & Constable 85, Martin-Löf 82, Manna & Waldinger 80]. We show how programs might be extracted in the logic programming setting. We should be able to apply the mechanisms for extracting programs from proofs in the logic programming setting to proof objects with "constructive content" from many different logics. In the `realize` program, each definite clause will specify the code fragment associated with an inference rule. This investigation is preliminary and we illustrate it on a simple propositional example.

Each logical connective corresponds to a type constructor. For example, we mentioned that $A \supset B$ corresponds to a function type *i.e.* $A \rightarrow B$. The definite clause below constructs the function corresponding to the proof of `(A imp B)`. Here, functions are represented by $\lambda$-abstractions in the $\lambda$-calculus of our logic programming language.

```
realize (A imp B) (imp_i P) Prog :-
       (pi Q\ (pi X\ ((realize A Q X) => (realize B (P Q) (Prog X))))).
```

Here, Prog is a function from A to B if for any program X that produces an element of type A, (Prog X) returns an element of type B. Note the equating of formulas and types implicit in this definite clause. Prog, a function from *type* A to *type* B, is constructed based on the structure of P, a function from proofs of *formula* A to proofs of *formula* B.

We view the formula (A and B) as the product type where the constructor pair builds objects of this type, and the destructors fst and snd extract elements of A and B, respectively. The formula (A or B) represents the disjoint union type where the constructors inl and inr construct elements of the disjoint union type from elements of A and B, respectively. The $\vee$-E rule generates conditional statements from disjoint union types (A or B), which allow branching in one of two ways depending on whether an object is from A or B. The construct is_left is the test used to determine if an object originates from the left or right of the disjunctive type. The use of these constructs is illustrated by the definite clauses of the following program.

```
realize (A and B) (and_i P1 P2) (pair X1 X2) :-
       realize A P1 X1,
       realize B P2 X2.

realize (A or B) (or_i1 P) (inl X) :- realize A P X.

realize (A or B) (or_i2 P) (inr X) :- realize B P X.

realize (A imp B) (imp_i P) Prog :-
       (pi Q\ (pi X\ ((realize A Q X) => (realize B (P Q) (Prog X))))).

realize A (and_e1 P) (fst X) :- realize (A and B) P X.

realize B (and_e2 P) (snd X) :- realize (A and B) P X.

realize C (or_e P1 P2 P3) (if (is_left X1) (X2 X1) (X3 X1)) :-
       realize (A or B) P1 X1,
       (pi Q\ (pi X\ ((realize A Q X) => (realize C (P2 Q) (X2 X))))),
       (pi Q\ (pi X\ ((realize B Q X) => (realize C (P3 Q) (X3 X))))).

realize B (imp_e P1 P2) (X1 X2) :-
       realize (A imp B) P1 X1,
       realize A P2 X2.
```

Given the formula $(x \vee (y \wedge z)) \supset ((x \vee y) \wedge (x \vee z))$ and its proof

```
(imp_i P\ (or_e P Q\ (and_i (or_i1 Q) (or_i1 Q))
                  Q\ (and_i (or_i2 (and_e1 Q)) (or_i2 (and_e2 Q)))))
```

we obtain the program

```
X\ (if (is_left X)
       (pair (inl X) (inl X))
       (pair (inr (fst X)) (inr (snd X)))).
```

## 7.4 Proof Normalization and Cut Elimination

Both proof normalization in natural deduction and cut-elimination in sequent systems are procedures which, given a proof, perform a transformation on the proof to obtain a new proof. Such proof transformations involve examining the structure of a proof and altering it as necessary. The transformations that are at the heart of these procedures can easily be specified in our logic programming setting by unifying over proof terms, breaking them up so that new proofs may be composed from the subproofs. We do not present the entire algorithms here—only some of the definite clauses that illustrate the central ideas.

Proof normalization, as presented in [Prawitz 65], is based on proof reductions that are performed when a formula occurrence is a conclusion of an introduction rule and the premise of an elimination rule. For example, the two below are reductions for $\wedge$ and $\supset$ with the initial proof on the left and the reduced proof on the right.

$$\dfrac{\dfrac{\dfrac{\Sigma_1}{A} \qquad \dfrac{\Sigma_2}{B}}{A \wedge B}}{A} \qquad\qquad \Rightarrow \qquad\qquad \dfrac{\Sigma_1}{A}$$

$$\dfrac{\dfrac{\Sigma_1}{A} \qquad \dfrac{\begin{matrix}(A)\\ \dfrac{\Sigma_2}{B}\end{matrix}}{A \supset B}}{B} \qquad\qquad \Rightarrow \qquad\qquad \begin{matrix}\dfrac{\Sigma_1}{(A)}\\ \dfrac{\Sigma_2}{B}\end{matrix}$$

We define a `reduce` predicate used to specify definite clauses to apply these reductions to proof terms. There are three arguments to this predicate: a formula and two proofs. The second argument is the initial proof of the specified formula, and the third argument is its reduced proof. The subgoals are calls to the `proof` predicate. Again, proof-checking is necessary to insure that the subproofs are proofs of the appropriate formulas in order for the reduction to take place. The definite clauses for the transformations above, and for all the other connectives are as follows.

```
reduce A (and_e (and_i P1 P2)) P1 :-
   proof (A and B) (and_i P1 P2).

reduce B (and_e (and_i P1 P2)) P2 :-
   proof (A and B) (and_i P1 P2).

reduce B (imp_e P1 (imp_i P2)) (P2 P1) :-
   proof A P1,
   proof (A imp B) (imp_i P2).

reduce C (or_e (or_i P1) P2 P3) (P2 P1) :-
   proof A P1,
   pi P\ ((proof A P) => (proof C (P2 P))).

reduce C (or_e (or_i P1) P2 P3) (P3 P1) :-
   proof B P1,
   pi P\ ((proof B P) => (proof C (P3 P))).

reduce (A T) (forall_e (forall_i P)) (P T) :-
   proof (forall A) (forall_i P).

reduce B (exists_e (exists_i P1) P2) (P2 T P1) :-
   proof (A T) P1,
   pi Y\ (pi P\ ((proof (A Y) P) => (proof B (P2 Y P)))).
```

This algorithm in an illustration of how "proof functions" may be used. For example, in $\supset$-reduction, the argument to imp_i is a proof function, in this case a function that takes proofs of A to proofs of B. But P1 is a proof of A, so a simpler proof (the "reduced proof" of B) is obtained by applying P2 to P1 directly. In $\forall$-reduction, P is an abstraction over terms which gets applied to the term T that appears in the formula (A T).

In proof normalization for intuitionistic logic, in addition to the above reductions, we also need clauses to remove occurrences of an application of the $\perp_I$ rule, followed by an application of an elimination rule, and clauses to reduce the length of maximum segments (series of repeated occurrences of a formula in a string of applications of $\vee$-E and $\exists$-E). The former are fairly straightforward. We will demonstrate the latter. These proof transformations have the following form.

$$\dfrac{\dfrac{\dfrac{\Sigma_1}{A \vee B} \quad \dfrac{\Sigma_2}{F} \quad \dfrac{\Sigma_3}{F}}{F} \qquad \Sigma_4}{C} \qquad \Rightarrow \qquad \dfrac{\dfrac{\Sigma_1}{A \vee B} \quad \dfrac{\dfrac{\Sigma_2}{F} \qquad \Sigma_4}{C} \quad \dfrac{\dfrac{\Sigma_3}{F} \qquad \Sigma_4}{C}}{C}$$

$$\dfrac{\dfrac{\dfrac{\Sigma_1}{\exists x\, A} \qquad \dfrac{\Sigma_2}{F}}{F} \qquad \Sigma_4}{D} \qquad \Rightarrow \qquad \dfrac{\dfrac{\Sigma_1}{\exists x\, A} \qquad \dfrac{\dfrac{\Sigma_2}{F} \qquad \Sigma_4}{D}}{D}$$

In both cases, there are several possibilities for the last inference rule. (If it does not branch then $\Sigma_4$ will be empty.) Below are the definite clauses for the case when this last rule is $\supset$-E.

```
reduce C (imp_e (or_e P1 P2 P3) P4)
         (or_e P1 (X\ (imp_e (P2 X) P4)) (X\ (imp_e (P3 X) P4))).

reduce C (imp_e P1 (or_e P2 P3 P4))
         (or_e P2 (X\ (imp_e P1 (P3 X))) (X\ (imp_e P1 (P4 X)))).

reduce D (imp_e (exists_e P1 P2) P3)
         (exists_e P1 (X\ (imp_e (P2 X) P3))).

reduce D (imp_e P1 (exists_e P2 P3))
         (exists_e P2 (X\ (imp_e P1 (P3 X)))).
```

Similar clauses are needed for each of the other possibilities. In these reductions the structure of the "proof functions" is altered. As a result, the scope of the variable X bound by $\lambda$-abstraction changes.

In implementing the complete normalization algorithm given by the proof of normalization for intuitionistic logic in [Prawitz 65], the order in which the above reductions get applied is very important. Although insuring the correct order is non-trivial, it should be possible to specify a complete algorithm that operates by repeatedly searching for possible reductions, checking to see whether these reductions can be applied without violating the required ordering, and applying only the appropriate ones, continuing until there are no more possible reductions.

In cut-elimination [Gentzen 35] (discussed here in the context of the LK system), instead of reductions, each instance of the cut rule is "pushed" up the tree as far as possible, until it "disappears" at the leaves. There are several cases to consider in order to push an application of a cut past other inference rules. The cases depend on whether the cut formula is involved in the inference rules immediately preceding the cut in both the left and right premises, just one of the premises, or neither of the premises. We first consider some cases when the cut formula is involved in both inference rules immediately preceding the cut. The transformations below are those that apply when the cut formula is of the form $A \wedge B$ and $\forall x\ A$ respectively. (We only show one possibility for the $\wedge$-L rule. The other is analogous.)

$$\cfrac{\cfrac{\cfrac{\Sigma_1}{\Gamma \longrightarrow \Delta, A} \qquad \cfrac{\Sigma_2}{\Gamma \longrightarrow \Delta, B}}{\Gamma \longrightarrow \Delta, A \wedge B}\,\wedge\text{-R} \qquad \cfrac{\cfrac{\Sigma_3}{A, \Theta \longrightarrow \Lambda}}{A \wedge B, \Theta \longrightarrow \Lambda}\,\wedge\text{-L}}{\Gamma, \Theta \longrightarrow \Delta, \Lambda}\,\text{cut} \quad \Rightarrow$$

$$\cfrac{\cfrac{\Sigma_1}{\Gamma \longrightarrow \Delta, A} \qquad \cfrac{\Sigma_3}{A, \Theta \longrightarrow \Lambda}}{\Gamma, \Theta \longrightarrow \Delta, \Lambda}\,\text{cut}$$

$$\cfrac{\cfrac{\cfrac{\Sigma_1}{\Gamma \longrightarrow \Delta, [x/y]A}}{\Gamma \longrightarrow \Delta, \forall x\, A}\,\forall\text{-R} \quad \cfrac{\cfrac{\Sigma_2}{[x/t]A, \Theta \longrightarrow \Lambda}}{\forall x\, A, \Theta \longrightarrow \Lambda}\,\forall\text{-L}}{\Gamma, \Theta \longrightarrow \Delta, \Lambda}\,\text{cut} \quad \Rightarrow \quad \cfrac{\cfrac{[y/t]\Sigma_1}{\Gamma \longrightarrow \Delta, [x/t]A} \qquad \cfrac{\Sigma_2}{[x/t]A, \Theta \longrightarrow \Lambda}}{\Gamma, \Theta \longrightarrow \Delta, \Lambda}\,\text{cut}$$

The cut-elimination program has one predicate called `cut_elim` which takes a sequent and two proofs. Whenever the first proof is a proof of the sequent, then the second proof will be its corresponding cut-free proof. This procedure operates by examining the structure of the last inference rule in a proof tree, and whenever there is an application of the cut rule, either reducing the size of the cut formula (as in the transformations above), or pushing the cut formula further up the tree (as will be demonstrated next), then repeating the same procedure until all applications of the cut rule are eliminated. The definite clauses for the transformations above, and for the others that occur when the cut formula is involved in the inference rules of both premises are given below. The third definite clause (for the ∀ transformation above) illustrates the instantiation of a proof term that is an abstraction over terms.

```
cut_elim (Sigma --> Phi) (cut (and_r P1 P2) (and_l P3)) P4 :-
        proof (Gamma --> [A | Delta]) P1,
        proof (Gamma --> [B | Delta]) P2,
        proof ([A | Theta] --> Lambda) P3,
        disjoint_union Gamma Theta Sigma,
        disjoint_union Delta Lambda Phi,
        cut_elim (cut P1 P3) P4.

cut_elim (Sigma --> Phi) (cut (and_r P1 P2) (and_l P3)) P4 :-
        proof (Gamma --> [A | Delta]) P1,
        proof (Gamma --> [B | Delta]) P2,
        proof ([B | Theta] --> Lambda) P3,
        disjoint_union Gamma Theta Sigma,
        disjoint_union Delta Lambda Phi,
        cut_elim (cut P2 P3) P4.

cut_elim (Sigma --> Phi) (cut (forall_r P1) (forall_l P2)) P3 :-
        pi Y\ (proof (Gamma --> [(A Y) | Delta]) (P1 Y)),
        proof ([(A T) | Theta] --> Lambda) P2,
        disjoint_union Gamma Theta Sigma,
        disjoint_union Delta Lambda Phi,
        cut_elim (Sigma --> Phi) (cut (P1 T) P2) P3.

cut_elim (Sigma --> Phi) (cut (or_r P1) (or_l P2 P3)) P4 :-
        proof (Gamma --> [A | Delta]) P1,
        proof ([A | Theta] --> Lambda) P2,
        proof ([B | Theta] --> Lambda) P3,
        disjoint_union Gamma Theta Sigma,
        disjoint_union Delta Lambda Phi,
        cut_elim (cut P1 P2) P4.

cut_elim (Sigma --> Phi) (cut (or_r P1) (or_l P2 P3)) P4 :-
        proof (Gamma --> [B | Delta]) P1,
        proof ([A | Theta] --> Lambda) P2,
        proof ([B | Theta] --> Lambda) P3,
        disjoint_union Gamma Theta Sigma,
        disjoint_union Delta Lambda Phi,
        cut_elim (cut P1 P3) P4.

cut_elim (Sigma --> Phi) (cut (imp_r P1) (imp_l P2 P3)) P4 :-
        proof ([A | Gamma] --> [B | Delta]) P1,
        proof (Theta1 --> [A | Lambda1]) P2,
        proof ([B | Theta2] --> Lambda2) P3,
        disjoint_union Theta1 Theta2 Theta, disjoint_union Gamma Theta Sigma,
        disjoint_union Lambda1 Lambda2 Lambda, disjoint_union Delta Lambda Phi,
        cut_elim (cut (cut P2 P1) P3) P4.

cut_elim (Sigma --> Phi) (cut (neg_r P1) (neg_l P2)) P3 :-
        proof ([A | Gamma] --> Delta) P1,
        proof (Theta --> [A | Lambda]) P2,
        disjoint_union Gamma Theta Sigma,
        disjoint_union Delta Lambda Phi,
        cut_elim (cut P2 P1) P3.

cut_elim (Sigma --> Phi) (cut (exists_r P1) (exists_l P2)) P3 :-
        proof (Gamma --> [(A T) | Delta]) P1,
        pi Y\ (proof ([(A Y) | Theta] --> Lambda) (P2 Y)),
        disjoint_union Gamma Theta Sigma,
        disjoint_union Delta Lambda Phi,
        cut_elim (cut P1 (P2 T)) P3.
```

In this program, we once again use the proof program for proof-checking when necessary. We must also check that the left and right of the conclusion sequent are composed of the disjoint union of the formulas in the left and right of the premises. (This is different than the LKC rules where all of the formulas in the conclusion appeared in the left and right of two-premise inference rules.) Note that there is more non-determinism in this program than in others. For example, Gamma, Delta, Theta, and Lambda are not instantiated when the proof subgoals are attempted. They must eventually match Sigma and Phi which insures that they are correct, but there will most likely be a lot of backtracking before a match occurs.

If none of the above transformations can be performed, this means that the cut formula is not involved in at least one of the inference rules in the premises of the cut rule. In this case we can push the application of the cut rule past this inference. We will need definite clauses to handle every inference rule on both the left and right premise of the cut rule. The following two inference figures illustrate this transformation for the ∧-R rule in the right premise and the ∃-L rule in the left premise. In pushing an application of cut past ∃-L, we choose a new variable $y'$ such that $y'$ does not appear in the conclusion. Then we replace $y$ with $y'$ in $\Sigma_1$.

$$\dfrac{\dfrac{\Sigma_1}{\Gamma \longrightarrow \Delta, A} \qquad \dfrac{\dfrac{\Sigma_2}{A,\Theta \longrightarrow \Lambda, B} \qquad \dfrac{\Sigma_3}{A,\Theta \longrightarrow \Lambda, C}}{A,\Theta \longrightarrow \Lambda, B \wedge C}\wedge\text{-R}}{\Gamma,\Theta \longrightarrow \Delta,\Lambda, B \wedge C}\text{cut} \quad \Rightarrow$$

$$\dfrac{\dfrac{\dfrac{\Sigma_1}{\Gamma \longrightarrow \Delta, A} \qquad \dfrac{\Sigma_2}{A,\Theta \longrightarrow \Lambda, B}}{\Gamma,\Theta \longrightarrow \Delta,\Lambda, B}\text{cut} \qquad \dfrac{\dfrac{\Sigma_1}{\Gamma \longrightarrow \Delta, A} \qquad \dfrac{\Sigma_3}{A,\Theta \longrightarrow \Lambda, C}}{\Gamma,\Theta \longrightarrow \Delta,\Lambda, C}\text{cut}}{\Gamma,\Theta \longrightarrow \Delta,\Lambda, B \wedge C}\wedge\text{-R}$$

$$\dfrac{\dfrac{\dfrac{\Sigma_1}{[x/y]B,\Gamma \longrightarrow \Delta, A}}{\exists x\, B,\Gamma \longrightarrow \Delta, A}\exists\text{-L} \qquad \dfrac{\Sigma_2}{A,\Theta \longrightarrow \Lambda}}{\exists x\, B,\Gamma,\Theta \longrightarrow \Delta,\Lambda}\text{cut} \quad \Rightarrow \quad \dfrac{\dfrac{\dfrac{[y/y']\Sigma_1}{[x/y']B,\Gamma \longrightarrow \Delta, A} \qquad \dfrac{\Sigma_2}{A,\Theta \longrightarrow \Lambda}}{[x/y']B,\Gamma,\Theta \longrightarrow \Delta,\Lambda}\text{cut}}{\exists x\, B,\Gamma,\Theta \longrightarrow \Delta,\Lambda}\exists\text{-L}$$

The definite clauses corresponding to these inference figures are given below. The clauses for the other inference rules in each premise may be specified similarly.

```
cut_elim (Sigma --> Phi) (cut P1 (and_r P2 P3)) P4 :-
        proof (Gamma --> [A | Delta]) P1,
        proof ([A | Theta] --> [B | Lambda]) P2,
        proof ([A | Theta] --> [C | Lambda]) P3,
        disjoint_union Gamma Theta Sigma,
        disjoint_union Delta Lambda Phi,
        cut_elim (Sigma --> Phi) (and_r (cut P1 P2) (cut P1 P3)) P4.

cut_elim (Sigma --> Phi) (cut (exists_l P1) P2) P3 :-
        pi Y\ (proof ([(B Y) | Gamma] --> [A | Delta]) (P1 Y)),
        proof ([A | Theta] --> Lambda) P2,
        disjoint_union Gamma Theta Sigma,
        disjoint_union Delta Lambda Phi,
        cut_elim (Sigma --> Phi) (exists_l (X\ (cut (P1 X) P2))) P3.
```

An application of the cut rule is eliminated once it gets pushed to the leaves. The following is an example of a final step in the transformation.

$$\dfrac{\dfrac{\Sigma}{\Gamma \longrightarrow \Delta, A} \qquad A \longrightarrow A}{\Gamma \longrightarrow \Delta, A}\,\text{cut} \qquad \Rightarrow \qquad \dfrac{\Sigma}{\Gamma \longrightarrow \Delta, A}$$

The definite clause specifications for these transformations are straightforward.

```
cut_elim (Gamma --> [A | Delta]) (cut P1 (initial A)) P2 :-
        cut_elim (Gamma --> [A | Delta]) P1 P2.

cut_elim ([A | Gamma] --> Delta) (cut (initial A) P1) P2 :-
        cut_elim ([A | Gamma] --> Delta) P1 P2.
```

To complete the algorithm, we need definite clauses that traverse past inference rules that are not applications of the cut rule. These clauses must simply call the `cut_elim` predicate with the proofs of the premises as arguments. The program terminates when all instances of the cut rule have been pushed beyond the leaves of the proof tree.

## 7.5 Constructing Proofs By Analogy

Proof by analogy has been recognized as a powerful tool used in human mathematical reasoning, one that is important yet difficult to incorporate in machine theorem provers [Bledsoe 86, Bledsoe 87]. Though not much has been done in this area, there have been some attempts to construct analogous proofs based on structural similarities. In [de la Tour & Caferra 87], abstractions over rules or series of rules are used to capture the generalities in the structure of a proof that may carry over to other proofs. This requires higher-order variables to represent "proof schemas," which require, at the very least, second-order matching to instantiate. Our extended logic programming language provides a medium to experiment with such techniques and algorithms. For example, the "transformation rules" of [de la Tour & Caferra 87] can easily be implemented in our setting, and the second-order matching that is required to instantiate them is already available through the higher-order unification of our language.

In [Brock, Cooper & Pierce 86] an analogous resolution proof of a calculus theorem about the limit of a product is constructed using a proof of a similar theorem about the limit of a sum as a guiding proof. The analogy is based on structural similarity between terms in the clauses. In the Nuprl proof system [Constable *et al.* 86], proof by analogy is considered in the tactic setting, using "transformation tactics." This technique also involves building proofs that are structurally similar, in this case by starting with one proof and constructing an analogous one step by step.

We will discuss some simple techniques for this kind of step by step proof analogy in our setting. We return to the tactic style theorem provers of Section 6. First we add a copy tactical to the Tactical module. Along with the other tacticals, it will be available to any tactic prover. It takes as arguments a predicate used in applying tactics (*e.g.* **rule** or **prove** from our natural deduction tactic prover), and a goal structure which contains the completed proof(s) or verifications to be used in building analogous proofs. Thus copy is declared with the following type.

```
copy : (tacticalexp -> goalexp -> goalexp -> o) -> goalexp -> tacticalexp.
```

A proof is just one kind of structure that can be used in building analogous structures. We use the more general term "verification" to emphasize that this tactical can be used with any tactic prover. As an example, we will discuss how it can be used to do proof by analogy in our natural deduction tactic prover. The copy tactical is defined by the definite clause below:

```
prove (copy TacPred CopyGoal) InGoal OutGoal :-
  copy_verification TacPred CopyGoal InGoal OutGoal.
```

where the copy_verification procedure is defined by the following definite clauses.

```
copy_verification TacPred (andgoal CopyGoal1 CopyGoal2)
                          (andgoal InGoal1 InGoal2) OutGoal :-
  copy_verification TacPred CopyGoal1 InGoal1 OutGoal1,
  copy_verification TacPred CopyGoal2 InGoal2 OutGoal2,
  goalreduce (andgoal OutGoal1 OutGoal2) OutGoal.

copy_verification TacPred (orgoal CopyGoal1 CopyGoal2)
                          (orgoal InGoal1 InGoal2) OutGoal :-
  copy_verification TacPred CopyGoal1 InGoal1 OutGoal;
  copy_verification TacPred CopyGoal2 InGoal2 OutGoal.

copy_verification TacPred (allgoal CopyGoal) (allgoal InGoal) OutGoal :-
  pi T\ (copy_verification TacPred (CopyGoal T) (InGoal T) (OutGoal1 T)),
  goalreduce (allgoal OutGoal1) OutGoal.

copy_verification TacPred CopyGoal InGoal OutGoal :-
  TacPred Tactic CopyGoal NewCopyGoal,
  TacPred Tactic InGoal MidGoal,
  copy_verification TacPred NewCopyGoal MidGoal OutGoal.

copy_verification TacPred CopyGoal Goal Goal.
```

Most of the copy_verification definite clauses are included to handle compound goals. In this program the goal structure of the new verification must imitate that of the guiding one. The heart of the program is the clause that makes calls to TacPred (the second to last above). The first subgoal has CopyGoal as its input goal. This subgoal finds a tactic that can be applied based on the structure of the verification in CopyGoal. This tactic is then attempted on the input goal InGoal. The copy_verification program will continue to apply tactics as long as they can be applied to both CopyGoal and InGoal. As soon as the verifications must differ, the last definite clause will be used and the output goal will contain the (possibly incomplete) verification for InGoal constructed up to this point. It must then be completed by other means.

In the natural deduction tactic prover, we can use the copy tactical to attempt to copy as much as possible of the structure of a given proof object to the proof of a new formula. If we specify TacPred to be rule, the new proof will be constructed step by step, applying the same inference rules to the new formula as were applied to obtain the guiding proof. This exact copying is achieved by the second to last copy_verification definite clause, which determines the last inference rule applied to the formula in CopyGoal, and then attempts to apply it to the formula in InGoal. It will copy as much of the structure of the guiding proof as possible and end with the (possibly incomplete) proof of the new formula.

This "verification by analogy" program is limited in that it only handles verifications that are analogous in the sense that there is a series of tactics that can be applied in both cases in exactly the same order. The additional definite clauses below are some other possibilities to include in the copy_verification program. For example, it is possible that two verifications are similar, yet they differ only in the application of a small number of tactics. The first definite clause allows for differences in the tactics that are applied as long as the verifications contain the same structure. For example, they must both branch in the same places. This new definite clause requires some search since Tactic2 is not bound when the second call to TacPred is made. In our natural deduction prover with rule used for TacPred, such a clause will handle the construction of a complete proof for $(\forall x\ q(x) \wedge p) \supset \forall x\ (q(x) \wedge p)$ given a proof for $(\forall x\ q(x) \vee p) \supset \forall x\ (q(x) \vee p)$. The second clause below encompasses the idea that two verifications are analogous if one has an application of a certain tactic, and the second has repeated applications of the same tactic.

```
copy_verification TacPred CopyGoal InGoal OutGoal :-
  TacPred Tactic1 CopyGoal NewCopyGoal,
  TacPred Tactic2 InGoal MidGoal,
  copy_verification NewCopyGoal MidGoal OutGoal.

copy_verification TacPred CopyGoal InGoal OutGoal :-
  TacPred Tactic CopyGoal NewCopyGoal,
  prove (repeat Tactic) InGoal MidGoal,
  copy_verification TacPred NewCopyGoal MidGoal OutGoal.
```

Although these algorithms are very preliminary, they do illustrate that we can guide the construction of at least small analogous fragments of proofs or verifications, and indicate how one might go about further experimenting with using analogy for theorem proving in a logic programming environment.

# 8 Translating LF Signatures to Logic Programming

One of our goals is to show that our extended logic programming language provides a language to specify proof systems for a wide variety of logics. In this respect, we share a common goal with the Edinburgh LF system [Harper, Honsell & Plotkin 87]. In this section, we compare the two methods and show that they are similar in ways that go beyond simply sharing common goals. We begin with some observations about natural deduction inference rule specifications in logic programming and their corresponding specification within an LF signature.

Recall the specification of the natural deduction $\wedge$-I rule using the proof predicate.

```
proof (A and B) (and_i P1 P2) :- proof A P1, proof B P2.
```

It can be considered as a declarative specification defining and_i. In [Avron, Honsell & Mason 87], first-order logic is one of many logics specified in LF. In this setting $\wedge$-I is an object whose type is specified by the judgement:

$$\prod_{A:bool}.\prod_{B:bool}.T(A) \rightarrow T(B) \rightarrow T(A \ and \ B)$$

In this judgement, $T$ is a function that takes a formula and produces a type ($T : bool \rightarrow Type$). This $T$ plays a role that is similar to the proof predicate in the logic programming setting. If we take $\rightarrow$ as $\supset$, $\prod$ as $\forall$, and $T$ as *proof* we get the hereditary Harrop formula $\forall A \, \forall B \, ((proof \ A) \supset ((proof \ B) \supset (proof \ (A \ and \ B))))$. This is equivalent to the formula $\forall A \, \forall B \, (((proof \ A) \wedge (proof \ B)) \supset (proof \ (A \ and \ B)))$ which can be abbreviated:

```
proof (A and B) :- proof A, proof B.
```

The arguments to the dependent ($\prod$-) types of the LF judgement, $A$ and $B$, correspond to the universally quantified logic variables in the definite clause. Note that this clause is the same as the one above, but without proof objects. Yet, the notion of proofs as objects inhabiting types is central to LF. For example, the judgement above specifies the type for the $\wedge$-I object. This object is functional and takes four arguments. If $A$ and $B$ have type *bool*, and if $P_1$ is a proof that $A$ is "true" (*i.e.* $P_1$ has type $T(A)$), and similarly $P_2$ is a proof of $B$, then we can apply $\wedge$-I to these four objects to obtain $(\wedge\text{–I} \ A \ B \ P_1 \ P_2)$ which is a proof of $(A \ and \ B)$ (*i.e.* has type $T(A \ and \ B)$). We can include this kind of proof object in the definite clause specification to obtain the following clause that corresponds even more closely to the LF definition of $\wedge$-I.

```
proof (A and B) (and_i A B P1 P2) :- bool A, bool B, proof A P1, proof B P2.
```

Here, as before, proof corresponds to the LF declaration $T : bool \rightarrow Type$. We also add the bool predicate which corresponds to the LF declaration $bool : Type$.

Both the LF declaration and the logic programming definite clause correspond to declarative specifications of the meaning of the $\wedge$-I (or **and_i**) constant. The definite clause version also has operational meaning and makes explicit the unification and search steps that are involved in finding a proof of the given formula (or judgement). Finding a universal instance of the clause involves finding unifiers for $A$ and $B$. Then finding a proof of $(A\ \mathit{and}\ B)$ involves verifying that $A$ and $B$ are formulas, and searching for subproofs for $A$ and $B$.

The LF notions of *schematic* and *hypothetical* judgements have corresponding concepts in the logic programming setting. Recall the definite clause specifying the $\forall$-I rule.

```
proof (forall A) (forall_r P) :- pi T\ (proof (A T) (P T)).
```

Here P is a function which maps arbitrary terms to proofs. The universally quantified goal of the logic programming language, which here plays a role in obtaining the function P, shares some similarities with an LF schematic judgement. A schematic judgement in LF is of the form $\bigwedge_{x:A} J(x)$ and is proved by a function mapping objects $x$ of type $A$ to proofs of $J(x)$.

We saw that the AUGMENT goal was important for generating functions from proofs to proofs as a result of discharging assumptions. For example, in the definite clause for the $\supset$-I rule:

```
proof (A imp B) (imp_i P) :- pi PA\ ((proof A PA) => (proof B (P PA)))
```

P is a function from proofs of A to proofs of B. This use of implication in logic programming corresponds to the LF hypothetical judgement. This judgement takes the form $J_1 \vdash J_2$ and represents the assertion that $J_2$ follows from $J_1$. Objects of this type are functions mapping proofs of $J_1$ to proofs of $J_2$.

We now examine the similarities in the two approaches more closely, and based on the observations made so far, informally describe an algorithm for translating LF signatures to logic programming programs. Using this algorithm, we have been able to translate all of the example signatures in [Harper, Honsell & Plotkin 87] and [Avron, Honsell & Mason 87] to logic programming programs. First, we take a closer look at LF.

The syntax of LF is given by the following classes of objects.

$$\begin{array}{rcl}
\Sigma & ::= & \langle\rangle \mid \Sigma, c : K \mid \Sigma, c : A \\
\Gamma & ::= & \langle\rangle \mid \Gamma, x : A \\
K & ::= & \mathit{Type} \mid \prod_{x:A}.K \\
A & ::= & c \mid \prod_{x:A}.B \mid \lambda x : A.B \mid AM \\
M & ::= & c \mid x \mid \lambda x : A.M \mid MN
\end{array}$$

$\Sigma$ is used to represent a signature, and $\Gamma$ a context (a set of variables and their types that can be used in constructing proofs). $M$ and $N$ range over expressions for objects, $A$ and $B$ over types and families of types, $K$ over kinds, $x$ and $y$ over variables, and $c$ over constants. Another notational convention is to write $A \rightarrow B$ for $\prod_{x:A}.B$ when $x$ does not occur free in $B$. Proof systems are specified by building signatures. The signatures in [Avron, Honsell & Mason 87] are divided into four categories: (1) syntactic categories, (2) operations, (3) judgements, and (4) axioms and rules. The syntactic categories and judgements correspond to kind declarations (declarations of the form $c : K$) and the operations, axioms and rules to type declarations (of the form $c : A$). Both LF types (syntax class $A$) and objects that inhabit these types (syntax class $M$) will be at the level of objects in logic programming. LF types cannot correspond to logic programming types, since our logic programming language does not have dependent types. As we have seen in the example above, we can represent all the necessary information as two arguments to the `proof` clause, and view the first argument as the type of the second.

Each item in an LF signature will have a type declaration in the logic programming program. The purpose of the logic programming type declarations is different from those in LF. The logic programming types are, in a sense, at another level that is used by the logic programming interpreter to insure proper typing of the terms it deals with. Each syntactic category and judgement (declaration of the form $c : K$) in LF will correspond to a predicate used in the search for proofs. These predicates are used by the program to prove that an object has the type given by a category or judgement, *i.e.* each type has its own procedure for proving that elements inhabit it. (In the above example, we had `proof` for $T$ and `bool` for *bool*). The definite clauses of the logic programming program are obtained from translations of the operations, and axioms and rules (declarations of the form $c : A$). These definite clauses are the logic programming embodiment of the type declarations of the LF signature.

First, we describe how to obtain the declarations of the logic programming predicates from the syntactic categories and judgements. Any declaration of the form $c : Type$ will have a logic programming declaration of the form `c : c_proof -> o`. Types of the form `c_proof` are generated for each constant $c$ and specify the types in the logic programming setting for LF objects (or proofs). In the example above, the declaration *bool* : *Type* gave rise to the `bool` predicate which is declared `bool : bool_proof -> o`. Judgements of the form $c : \prod_{x:A}.K$ will also correspond to a logic programming declaration for object `c` whose last argument has type `c_proof` and whose target type is `o`. In addition, there will be another argument to the predicate `c` for every argument (of the form $x : A$) in the judgement. The type of these arguments will depend on the term given by $A$. For example, if $A$ is *bool* as in $T : bool \rightarrow Type$ (which is an abbreviation for $\prod_{x:bool}.Type$), then the $x : bool$ will correspond to an argument of type `bool_proof`. Thus the predicate corresponding to $T$ would be declared `t : bool_proof -> t_proof -> o`. (Note that this is the same type as the `proof` predicate above.) If $A$ has more complicated structure as in $x : T(B)$ for example, then the corresponding argument type in the logic programming predicate depends only on the head of the type expression. In this case it will be `t_proof`. In summary, for an LF judgement that takes $n$ arguments, the corresponding predicate will have $n+1$ arguments: one for each LF argument type, plus one for the objects (proofs) that inhabit the types specified by the judgement.

The type declarations for the operations, axioms, and rules (declarations of the form $c : A$) are also obtained by examining the head of the type expressions for each argument. In this case, there will be no extra arguments such as those that were needed above for the proof objects, and the target type in the logic programming type declaration will correspond to the head of the LF target type. For example the LF declaration for $\wedge$-I:

$$\wedge\text{-I} : \textstyle\prod_{A:bool}.\prod_{B:bool}.T(A) \rightarrow T(B) \rightarrow T(A\ and\ B)$$

would yield the following logic programming declaration.

```
and_i : bool_proof -> bool_proof -> t_proof -> t_proof -> t_proof.
```

Finally, we need to specify definite clauses that construct objects of the types given by the operations, axioms, and rules, such as $\wedge$-I. We will describe this translation in terms of two rules that translate an LF declaration to a hereditary Harrop formula. It is then straightforward to obtain a definite clause from this formula. We use the notation $[\![ M : A ]\!]$ to represent the translation of an LF declaration to a higher-order hereditary Harrop formula. The translation rules are as follows.

$$\begin{aligned} [\![ M : \textstyle\prod_{x:A}.B ]\!] \quad &\Rightarrow \quad \forall X\ ([\![ X : A ]\!] \supset [\![ (M\ X) : B ]\!]) \\ [\![ M : c(t_1, \ldots, t_n) ]\!] \quad &\Rightarrow \quad (\mathtt{c}\ t_1\ \ldots\ t_n\ M) \end{aligned}$$

Proof objects are constructed by introducing new logic variables (*e.g.* $X$ in the first rule above) which act as place holders for the arguments to the LF signature constants. These place holders take on values when the definite clause is used in constructing proofs. The second rule is responsible for inserting the appropriate predicates based on the head of the type expression. We return to the example above for $\wedge$-I to illustrate the use of these rules. (We use the constant symbol *and_i* instead of $\wedge$-I and *and* instead of $\wedge$ in order to avoid conflict with symbols of the meta-language, *i.e.* the logic programming language.) Using the above rules, the LF declaration

$$and\_i : \textstyle\prod_{A:bool}.\prod_{B:bool}.T(A) \rightarrow T(B) \rightarrow T(A\ and\ B).$$

translates to the formula

$$\forall A\ ((\mathtt{bool}\ A) \supset \forall B\ ((\mathtt{bool}\ B) \supset \forall P_1\ ((\mathtt{t}\ A\ P_1) \supset \forall P_2\ ((\mathtt{t}\ B\ P_2) \supset (\mathtt{t}\ (A\ and\ B)\ (and\_i\ A\ B\ P_1\ P_2)))))).$$

This formula is equivalent to the definite clause

$$\forall A\, \forall B\, \forall P_1\, \forall P_2\, (((\mathtt{bool}\ A) \wedge (\mathtt{bool}\ B) \wedge (\mathtt{t}\ A\ P_1) \wedge (\mathtt{t}\ B\ P_2)) \supset (\mathtt{t}\ (A\ \mathit{and}\ B)\ (\mathit{and\_i}\ A\ B\ P_1\ P_2)))$$

which can be abbreviated

```
t (A and B) (and_i A B P1 P2) :- bool A, bool B, t A P1, t B P2.
```

It is exactly the same as the definite clause given above with proof replaced with t.

We illustrate this translation process with the signature for a natural deduction style modal S4 as in [Avron, Honsell & Mason 87]. We present the logic programming declarations and program obtained by translating the LF declarations of this signature. First, the logic programming declarations are the following.

```
   bool : bool_proof -> o.
   taut : bool_proof -> taut_proof -> o.
  valid : bool_proof -> valid_proof -> o.

   perp : bool_proof.
    imp : bool_proof -> bool_proof -> bool_proof.
    box : bool_proof -> bool_proof.
      c : bool_proof -> taut_proof -> valid_proof.
      r : bool_proof -> bool_proof -> (taut_proof -> valid_proof) ->
              valid_proof -> valid_proof.
imp_i_v : bool_proof -> bool_proof -> (valid_proof -> valid_proof) ->
              valid_proof.
 perp_e : bool_proof -> taut_proof -> taut_proof.
 2neg_e : bool_proof -> taut_proof -> taut_proof.
imp_i_t : bool_proof -> bool_proof -> (taut_proof -> taut_proof) -> taut_proof.
imp_e_t : bool_proof -> bool_proof -> taut_proof -> taut_proof -> taut_proof.
imp_e_v : bool_proof -> bool_proof -> valid_proof -> valid_proof ->
              valid_proof.
  box_i : bool_proof -> valid_proof -> valid_proof.
  box_e : bool_proof -> valid_proof -> valid_proof.
```

The basic types generated from the syntactic categories and judgements are bool_proof, taut_proof, and valid_proof. The syntactic categories and judgements give us three predicates (bool, taut, and valid) for proving that an object is a formula, for proving that a formula is a tautology, and for proving that a formula is valid, respectively. The latter two represent two different predicates for "proving" formulas. The fact that there are two is a result of the way the proof system is specified in LF. The operations, axioms, and rules produce the following definite clauses.

```
bool perp.
bool (A imp B) :- bool A, bool B.
bool (box A) :- bool A.

valid A (c A P) :- bool A, taut A P.

valid B (r A B P1 P2) :-
  bool A, bool B, (pi PA\ ((taut A PA) => (valid B (P1 PA)))), valid A P2.

valid ((box A) imp B) (imp_i_v A B P) :-
  bool A, bool B, (pi PA\ ((valid (box A) PA) => (valid B (P PA)))).

taut A (perp_e A P) :- bool A, taut perp P.

taut A (2neg_e A P) :- bool A, taut ((A imp perp) imp perp) P.

taut (A imp B) (imp_i_t A B P) :-
  bool A, bool B, (pi PA\ ((taut A PA) => (taut B (P PA)))).

taut B (imp_e_t A B P1 P2) :- bool A, bool B, taut (A imp B) P1, taut A P2.

valid B (imp_e_v A B P1 P2) :- bool A, bool B, valid (A imp B) P1, valid A P2.

valid (box A) (box_i A P) :- bool A, valid A P.

valid A (box_e A P) :- bool A, valid (box A) P.
```

We can simplify this program by taking advantage of the type system of our logic programming language to handle the syntactic categories (declarations of the form $c : Type$) in the LF signature. In the above program, instead of the predicate bool used to prove that objects are formulas, we could include bool as one of the basic types, and replace the definite clauses

```
bool perp.
bool (A imp B) :- bool A, bool B.
bool (box A) :- bool A.
```

with the logic programming declarations

```
   perp : bool
    imp : bool -> bool -> bool
    box : bool -> bool.
```

As a result, all of the calls to the bool predicate in the old program are eliminated in the new program. In this new program, the type-checking for formulas is handled by the type-checking of the logic programming interpreter, and no longer needs to be included explicitly in the program.

A query to one of these programs is obtained using the same rules that were used to translate LF type declarations to definite clauses. In this case, an LF declaration $M : A$ is translated to a goal formula that will be presented to the logic programming interpreter. The example proof for modal S4 given in [Avron, Honsell & Mason 87] would generate the following query (to the program where bool is a basic type).

```
(pi A\ (pi B\ (valid ((box (A imp B)) imp ((box A) imp (box B)))
                     (imp_i_v (box (A imp B)) ((box A) imp (box B))
                        P1\ (imp_i_v A (box B)
                               P2\ (box_i B (imp_e_v A B
                                                (box_e (A imp B) P1)
                                                (box_e A P2))))))))
```

Since both the formula and the proof are specified, the above program would act as a proof-checker in satisfying this query. In actual execution, under the interpreter of Section 2 with depth-first search on the above ordering of clauses to resolve all non-determinism, this program will be able to answer such type-checking queries. There is much more non-determinism when attempting to use this program as a theorem prover, *i.e.* when a formula is specified and the proof is given as a logic variable. In this case, depth-first search is not sufficient. Alternatively, we can specify the above clauses as a set of tactics to use in a tactic prover as in Section 6. For example, the definite clause in the program above for the r object (the fifth clause) might be specified as the tactic below (where validgoal and tautgoal are atomic goal structures similar to proofgoal as in the earlier tactic prover examples).

```
prove r_tac (validgoal B (r A B P1 P2))
            (andgoal (allgoal PA\ (impgoal (taut A PA) (validgoal B (P1 PA))))
                     (validgoal A P2)).
```

Such a tactic module would give us a more controllable theorem prover for the modal S4 system.

# 9 Related Work

Other proof systems that are based on tactics and tacticals include LCF [Gordon, Milner & Wadsworth 79] and Nuprl [Constable *et al.* 86] as already mentioned, and Isabelle [Paulson 86]. The programming language ML is the meta-language used in all of these systems. ML is a functional language with several features that are useful for the design of theorem provers. It contains a secure typing scheme and is higher-order, allowing complex programs to be composed easily. In developing theorem provers, many extensions have been made to ML to increase its capabilities as demand requires. For example, Nuprl uses an extension of ML with term destructors so that terms can be decomposed and their components manipulated separately. Isabelle [Paulson 86] uses typed $\lambda$-terms to represent formulas, and higher-order unification is added to manipulate them. In [Paulson 87], the meta-theory of Isabelle is extended to include a fragment of higher-order logic with implication and universal quantification which is used to specify inference rules. The operation of "lifting" an object level rule over assumptions provides a mechanism for discharging assumptions. "Lifting" an object level rule over a universal variable provides a mechanism for reasoning about generic objects. In our setting, these capabilities were illustrated using the AUGMENT and GENERIC search operations respectively. Isabelle also allows goals containing variables which are instantiated by unification. This feature is provided in the logic programming setting by the INSTANCE search operation.

In contrast to ML which was originally designed as a meta-language for theorem proving, the theory of higher-order hereditary Harrop formulas on which our extended logic programming language is based was motivated by a desire to develop a clean semantics for a general purpose programming language. Other applications that have been explored include program transformations [Miller & Nadathur 87] and computational linguistics [Miller & Nadathur 86b]. Thus, in a sense, we are working in the opposite direction, examining theorem proving as a special application of this language. The kinds of features that are being added to ML and LCF are similar to those that have been identified as useful in the logic programming setting. In many ways, the two approaches seem to be converging.

One thing we have not been concerned with in this paper, that many of the other theorem proving efforts have addressed to some extent are issues of efficiency. Once the features of the logic programming language that are necessary for good implementations of proof systems have been fully identified, the issue of building efficient implementations for a subset of the language containing these features can be addressed.

Of the proof systems mentioned above, only Nuprl has been concerned with constructing and storing proof objects and using them in computation. The success of the term extraction algorithm for constructing executable programs from proofs will be a good model for continuing to examine the `realize` program in the logic programming setting. Also, as stated in Section 7.5, proofs in Nuprl are used for certain kinds of proof by analogy. In goal-directed proof, LCF constructs validations which map theorems to theorems. Their purpose is to insure that only provable formulas inhabit the type `thm`. They are not objects that can be manipulated. In Isabelle, as inference rules are applied, the internal structure of the proof is discarded as theorem proving proceeds. An incomplete proof is considered a derived rule whose premises are the subgoals that have not been completed.

We have discussed higher-order hereditary Harrop formulas in the context of being a specification language for a wide class of logics. As already mentioned, this aspect of our work is related to the Edinburgh Logical Framework [Harper, Honsell & Plotkin 87]. Again, the original motivation for these two approaches differs. LF was developed for the purpose of capturing the uniformities of a large class of logics, so that it can be used as the basis for implementing proof systems. In our case, we have been exploring the capabilities of our logic programming language, including an examination of its potential to naturally specify proof systems for certain logics. In addition, we also obtain implementations of theorem provers from these specifications. In [Paulson 87], the author is also concerned with expressing various logics within a uniform framework. In this case, a fragment of higher-order logic that is essentially a subclass of higher-order hereditary Harrop formulas is used to specify inference rules. Here, the development of a specification language is motivated by the desire for a more general theorem prover, and results from extensions to the meta-theory of Isabelle.

In [Schroeder-Heister 84], natural deduction is extended so that rules as well as assumptions may be discharged. Such an extension allows additional inference rules to be made available at different points in a proof. During proof construction in our tactic provers this capability is provided by the `use_module` tactical which allows modules that may contain additional tactics to be imported into the theorem proving environment. The proposed extension to natural deduction may provide a formal proof-theoretic semantics for this dynamic access to collections of tactics.

# 10 Conclusion and Proposal for Future Research

The main focus in this paper has been to demonstrate that our extended logic programming language based on the theory of higher-order hereditary Harrop formulas is well-suited to the task of specifying and implementing theorem provers. We have shown that the six search operations of an interpreter based on this language serve as a good mechanism for implementing the search required in proof discovery. In addition, we have illustrated that inference rules and other formula and proof manipulations can be specified quite naturally. We have demonstrated these characteristics on examples from Gentzen sequential and natural deduction systems and shown how to extend them to logics represented by LF signatures. The main goal in continuing this work will be to extend the techniques used in specifying and implementing theorem provers to larger systems. In particular, we will work with tactic style theorem provers because they have proven to be successful in providing a general framework for integrating user interaction with varying degrees of partial and even full automation in the search for proofs. Our goal will be to provide a diversified environment for interactive theorem proving, one that provides the user with many tools and techniques for proving and manipulating proofs. Such a system should allow reasoning in possibly many different logics, and include capabilities for theorem proving in more complicated mathematical domains.

## 10.1 Extending Tactic Theorem Provers

In this section, we discuss some of the extensions we will need to make and some of the issues we expect to encounter during the early stages of expanding the theorem proving environment. Many of the extensions will involve writing new tactics to expand the capabilities of the theorem prover and to increase the choices that are available to the user in constructing proofs. We will augment the natural deduction tactic theorem prover as needed to incorporate these capabilities. We will also want to choose some specific domain(s) and build a database of theorems and proofs in these domains. As work continues, we expect to address issues such as adding techniques and capabilities to further enhance the theorem proving environment, integrating the different techniques, and organizing the tactics, theorems, and proofs into a manageable system.

Specifically, one of the first additions will be the ability to introduce constants into the theorem proving environment and include axioms about these constants and the operations they represent. We will need to include tactics that allow us to use these axioms in searching for proofs and constructing proof terms. Induction and reasoning about equality are two more facets of theorem proving that will arise when expanding the theorem prover to handle more than simple logical manipulations. In Section 6.4.1 we gave example inference rule tactics that might be included for constructing proofs by induction. In the next subsection we will discuss some possible tactics for equality reasoning. In subsection 10.1.2, we will discuss some possible methods for building libraries to organize tactics and theorems so that they may be easily accessed during theorem proving.

### 10.1.1 Equality Reasoning

In any system with equality there are several general axioms such as symmetry and transitivity that must be included. In addition, when introducing a new domain, we will need to include axioms or equations that are specific to that domain, as well as tactics that reason about them. For example, in [Manna & Waldinger 85], when the non-negative integer domain is presented, the axioms $\neg(x+1=0)$, $(x=y) \supset (x+1=y+1)$, $x+0=0$, and $x+(y+1)=(x+y+1)$ are introduced. They are needed for even the simplest proofs in this domain. There are many ways to incorporate such reasoning. We plan to examine them and implement them to sufficiently handle large domains. Below are three examples of tactics that might be included in a theorem prover for induction on non-negative integers.

```
prove succ_axiom_tac (proofgoal ((X + 1) = (Y + 1)) (succ_axiom P))
                     (proofgoal (X = Y) P).

prove trans_query (proofgoal (X = Z) (trans P1 P2))
                  (andgoal (proofgoal (X = Y) P1) (proofgoal (Y = Z) P2)) :-
  write "Enter intermediate value", read Y.

prove rewrite_tac (proofgoal (F A) (rewrite P1 P2))
                  (andgoal (proofgoal (A = B) P1) (proofgoal (F B) P2)).
```

The first represents the second non-negative integer axiom above in inference rule form. It illustrates the fact that any axiom can be included as an inference rule. The second is an inference rule specification of transitivity, which allows the user to specify the intermediate value Y. The third is quite general. It takes an arbitrary formula, and substitutes an arbitrary subexpression with an equivalent expression. A unifies with some term in the formula and F is an abstraction over one or more occurrences of A (0 occurrences is also possible, but no rewriting is done in that case). If A is equal to some term B, then B replaces the occurrences of A that were removed by abstraction. This rule takes advantage of higher-order unification and as a result is quite powerful. There might be many unifiers for F and A and backtracking will be used to try each one. As it is written, the user has no control over the choice of A or B. An alternative version could query the user for this information and avoid some of the backtracking. The third tactic above incorporates transitivity, and thus when it is used in a theorem prover, the second tactic above would not be needed. On the other hand, many such tactics can exist simultaneously in a tactic prover, giving the user more choice in constructing proofs.

### 10.1.2 Building Libraries

**Referencing Existing Theorems** Another capability that we would like our theorem provers to have is the ability to use existing theorems that have already been proven as lemmas. This capability is very important, especially in mathematical domains. For example, in the non-negative integer domain, the proof for commutativity of addition (*i.e.* $x + y = y + x$) is based on another fact that must be proven first: $x + 1 = 1 + x$. The ability to reference such theorems requires that we have a mechanism for storing and retrieving proofs. One way to organize proofs is to give them names and define a predicate (which we will call `theorem`) to associate a name with a formula and its proof. Such a predicate will be declared as follows.

```
        theorem : name -> bool -> proof_object -> o.
```

The following is an example of a tactic that can retrieve existing proofs stored using this predicate during construction of a new proof.

```
prove lemma_tac (proofgoal A (lemma PB P))
                (impgoal (proof B PB) (proofgoal A P)) :-
  write "Enter theorem name", read Name,
  theorem Name B PB.
```

Here, the `proof` predicate is used to add the lemma to the goal structure. It will then be added to the program in the same manner as discharged assumptions.

**Definitions** We will also want to have the capability to incorporate definitions into the theorem proving process. We define a `definition` predicate with type `name -> A -> A -> o`. Here, `A` is a logical variable because we want to allow definitions of any type, though the defined object and its meaning must have the same type. For example, we may define logical equivalence as follows.

```
<=> : bool -> bool -> bool.

definition equiv (B <=> C) ((B imp C) and (C imp B)).
```

In this case type variable `A` gets assigned `bool`. The parameters to a definition are represented using (universally quantified) logic variables, so that as a result, instantiating them simply requires unification. One possibility for a tactic that does the instantiation using input from the user is as follows.

```
prove instantiate_def (proofgoal (F A) (def P)) (proofgoal (F B) P) :-
  write "Enter term", read A,
  write "Enter name of definition", read Name,
  definition Name A B.
```

This tactic uses higher-order unification in the same way as `rewrite_tac`. In this case it substitutes expressions with their expanded definitions. For example, if the input formula that must unify with `(F A)` is `(p imp (q <=> r))`, and the user inputs `(q <=> r)` and `equiv`, for `A` and `Name` respectively, then `X\ (p imp X)` is one unifier for `F`. `(F B)` will then be `(p imp ((q imp r) and (r imp q)))`.

**Accessing Libraries** The organization and modularization of libraries will become more important as the number of theorems, definitions, and tactics grows. In Section 6.4.3, we discussed accessing groups of tactics by importing modules using the `use_module` tactic. These modules could actually be libraries that contain definitions and theorems in addition to tactics. Then, the `use_module` tactical would allow a user to add a library to the current program which would be available during the execution of a tactical expression. This capability allows a user to limit the search environment. As an example of when this might be useful, a user might, during the proof of a theorem, first add a library containing the theorems about the properties of the objects in the theorem. If a proof is not found, then a library of definitions pertaining to the objects in the theorem could be loaded and a proof attempted in this environment. Limiting the amount of information in the search space could be informative in the sense that it tells the user what kind of information is and is not needed in the proof of a theorem. A smaller search space, of course, also contributes to greater efficiency.

Another possibility is that modules will contain groups of tactics that define the inference rules for a particular logic. This kind of module allows a user to define specialty logics. A user could then work exclusively in this specialized domain by using `use_module` to include this module, while excluding all others. The signatures of LF, for example, could be included into our theorem proving environment in this way.

## 10.2 Proof Objects

The programs in Section 7 demonstrated some potentially important applications for proof objects, and raised the issue of what information to include in these objects. As a result of implementing a larger domain, we will have a database of proof objects which we can examine closely to give us more insight into when and in what forms these objects are really useful.

Extracting programs from proofs has obvious value, and it will be important to examine the `realize` program more deeply. In the simple propositional example we presented in Section 7.3 the program simply constructed an object of a certain type based on the type of the input. We will need to extend this program to incorporate program constructs for quantifiers and possibly domain specific objects. We will need to discover what kinds of information is needed to extract executable programs from more complicated proofs.

Including more information in proof terms might be useful in general for practical and efficiency concerns. For example, in some of the programs such as cut-elimination and normalization, a lot of proof-checking was needed to obtain information. While this gives a clear declarative reading, it may mean a lot of extra work computationally. In cut-elimination, the situation is even worse since there is a lot of non-determinism in the proof-checking resulting from having to deduce the cut formula each time. Since knowing the formulas to which inference rules are applied is fundamental to this algorithm, it might be worthwhile to store these formulas explicitly in the proofs.

In analogy, the structure of the guiding proof is very important in the determination of the structure of the new proof. We might want to consider other forms that allow more flexibility in determining when two proofs are analogous. For example, we could define a new tactic like **`intro`** of Section 6.4.2, so that the proof term, instead of being a series of introduction rules (*e.g.* `(and_i (all_i (imp_i X\ (and_i ...)` would contain some other encoding that indicated that a series of introduction rules was applied. Since this could be any series of introduction rules, a proof would be considered analogous if it also had a series of introduction rules, though not necessarily the same ones. We might even replace certain subproofs with logic variables. Proof terms containing such variables will be "proof schemas" which can then be instantiated in different ways to obtain analogous proofs.

It appears that some algorithms will benefit from more information in proofs (*e.g.* cut-elimination), while others will require less (*e.g.* analogy). We may want to choose an intermediate representation during theorem proving and provide programs that do some sort of preprocessing of the proof terms to transform them to a form suitable for specific algorithms.

### 10.2.1 Analogy

With the exception of analogy, most of the algorithms in Section 7.5, although their exact form and degree of efficiency will depend on how they are implemented and on the form of proof objects, are fairly straightforward manipulations on proofs. Analogy on the other hand is not well-defined as an algorithm. The `copy_verification` program encompassed some ideas about when two proofs are structurally analogous, but it is difficult to assess how it will behave on larger proofs. For example, even the clause that involves some search (the second to last in Section 7.5) requires that the two proofs have exactly the same branching structure. This may be too rigid for larger proofs. Experimenting with alternative proof structures as mentioned above might prove advantageous. In any case, with a database of proofs to work with, we hope to gain some insight into this difficult problem.

## 10.3 Correctness of Programs

Tactic provers are inherently modular in the sense that each tactic can be considered a separate program that can be called during theorem proving. The theorem prover as a whole can be viewed as a collection of these smaller programs. In addition, as we have illustrated, these programs generally have a natural declarative reading. Both the relatively small size and the naturalness of specification should contribute to the facilitation of the usually very difficult task of proving programs correct. We are concerned with program correctness because we want to be able to make claims about the capabilities of tactics in our theorem prover. For example, we would like to prove the correctness of the LKC prover program, so that we can guarantee the integration of fully automatic theorem proving into the tactic theorem proving setting. In addition, in expanding the theorem prover we will want to prove correctness of new tactics as they are added.

## References

[Avron, Honsell & Mason 87] Arnon Avron, Furio A. Honsell, and Ian A. Mason. *Using Typed Lambda Calculus to Implement Formal Systems on a Machine.* Technical Report ECS-LFCS-87-31, University of Edinburgh, Edinburgh, Scotland, June 1987.

[Bates & Constable 85] Joseph L. Bates and Robert L. Constable. Proofs as Programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[Bledsoe 77] W. W. Bledsoe. Non-resolution Theorem Proving. *Artificial Intelligence*, 9:1–35, 1977.

[Bledsoe 86] W. W. Bledsoe. Some Thoughts on Proof Discovery. In *Third Annual IEEE Symposium on Logic Programming*, pages 2–10, Salt Lake City, Utah, September 1986. MCC Tech Report AI-208-86, June 1986.

[Brock, Cooper & Pierce 86] Bishop Brock, Shaun Cooper, and William Pierce. *Some Experiments with Analogy in Proof Discovery (Preliminary Report).* Technical Report AI-347-86, MCC, Austin, Texas, October 1986.

[Church 40] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Clocksin & Mellish 84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[Constable *et al.* 86] R. L. Constable *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[deBruijn 80] N.G. deBruijn. A Survey of the Project AUTOMATH. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, Academic Press, New York, 1980.

[de la Tour & Caferra 87] Thierry Boy de la Tour and Ricardo Caferra. Proof Analogy in Interactive Theorem Proving: A Method to Express and Use it via Second Order Pattern Matching. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 95–99, Seattle, WA, July 1987.

[Felty 86] Amy Felty. *Using Extended Tactics to do Proof Transformations.* Master's thesis, University of Pennsylvania, December 1986. Also available as MS-CIS-86-89.

[Gallier 86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper & Row, 1986.

[Gentzen 35] Gerhard Gentzen. Investigations into Logical Deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland Publishing Co., Amsterdam, 1969.

[Gordon, Milner & Wadsworth 79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[Hallnäs & Schroeder-Heister 87] Lars Hallnäs and Peter Schroeder-Heister. A Proof-Theoretic Approach to Logic Programming. Unpublished, 1987.

[Harper, Honsell & Plotkin 87] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. In *Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[Howard 80] W. A. Howard. The Formulae-as-Type Notion of Construction, 1969. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490, Academic Press, New York, 1980.

[Huet 75] G. P. Huet. A Unification Algorithm for Typed $\lambda$-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Manna & Waldinger 80] Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[Manna & Waldinger 85] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming.* Volume 1: Deductive Reasoning, Addison Wesley, 1985.

[Martin-Löf 82] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, North-Holland, Amsterdam, 1982.

[Martin-Löf 84] Per Martin-Löf. *Intuitionistic Type Theory. Studies in Proof Theory Lecture Notes*, BIBLIOPOLIS, Napoli, 1984.

[Miller 86] Dale Miller. A Theory of Modules for Logic Programming. In *Third Annual IEEE Symposium on Logic Programming*, Salt Lake City, Utah, September 1986.

[Miller & Felty 86] Dale Miller and Amy Felty. An Integration of Resolution and Natural Deduction Theorem Proving. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 198–202, Philadelphia, PA, August 1986.

[Miller & Nadathur 86a] Dale Miller and Gopalan Nadathur. Higher-Order Logic Programming. In *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[Miller & Nadathur 86b] Dale Miller and Gopalan Nadathur. Some Uses of Higher-Order Logic in Computational Linguistics. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 247–255, 1986.

[Miller & Nadathur 87] Dale Miller and Gopalan Nadathur. A Logic Programming Approach to Manipulating Formulas and Programs. In *IEEE Symposium on Logic Programming*, San Francisco, September 1987.

[Miller, Nadathur & Scedrov 87] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop Formulas and Uniform Proof Systems. In *Symposium on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.

[Miller 87] Dale A. Miller. A Compact Representation of Proofs. *Studia Logica*, 46(4), 1987.

[Milner 87] Robin Milner. Dialogue with a Proof System. In H. Ehrig *et al.*, editors, *TAPSOFT '87*, pages 271–275, Springer-Verlag, 1987.

[Nadathur 86] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, December 1986.

[Paulson 86] Lawrence C. Paulson. Natural Deduction as Higher-Order Resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Paulson 87] Lawrence C. Paulson. *The Representation of Logics in Higher-Order Logic.* Draft, University of Cambridge, July 1987.

[Prawitz 65] Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Uppsala, 1965.

[Schroeder-Heister 84] Peter Schroeder-Heister. A Natural Extension of Natural Deduction. *Journal of Symbolic Logic*, 49(4):1284–1300, 1984.

[Smullyan 68] Raymond M. Smullyan. *First-Order Logic.* Springer-Verlag New York Inc., 1968.

Engdahl, Elisabet: 1984, 'Parasitic gaps, resumptive pronouns, and subject extractions', ms., University of Wisconsin, Madison.

Gazdar, Gerald: 1981, 'Unbounded dependencies and coordinate structure', *Linguistic Inquiry*, **12**, 155-184.

Gazdar, Gerald: 1982, 'Phrase structure grammar', in Pauline Jacobson and Geoffrey K. Pullum, (eds), *On the Nature of Syntactic Representation*, Reidel, Dordrecht, pp. 131-186.

Gazdar, Gerald, Ewan Klein, Ivan A. Sag and Geoffrey K. Pullum: 1985, *Generalised Phrase Structure Grammar*, Blackwell, Oxford.

Geach, Paul T: 1972, 'A program for syntax', in Donald Davidson and Gilbert Harman, *Semantics of Natural Language*, Reidel, Dordrecht, pp. 483-497.

Joshi, Aravind: 1987, 'The convergence of mildly context-sensitive formalisms', Paper to CSLI Workshop on Processing of Linguistic Structure, Santa Cruz, Jan 1987, ms., U. Pennsylvania.

Katz, Jerrold and Paul Postal: 1964, *An Integrated Theory of Linguistic Descriptions*, MIT Press, Cambridge MA.

Keenan, Edward and Bernard Comrie: 1977, 'Noun phrase accessibility and Universal Grammar', *Linguistic Inquiry*, **8**, 63-100.

Klein, Ewan and Ivan A. Sag: 1984, 'Type-driven translation', *Linguistics and Philosophy*, **8**, 163-201.

Kuno, Susumo: 1973, 'Constraints on internal clauses and sentential subjects', *Linguistic Inquiry*, **4**, 363-386.

Lambek, Joachim: 1958, 'The mathematics of sentence structure', *American Mathematical Monthly*, **65**, 154-170.

Lambek, Joachim: 1961, 'On the calculus of syntactic types', *Structure of Language and its Mathematical Aspects, Proceedings of the Symposia in Applied Mathematics*, **XII**, American Mathematical Society, Providence, Rhode Island, pp. 166-178.

Lyons, John: 1968, *Introduction to Theoretical Linguistics*, Cambridge University Press.

McCloskey, M. James: 1978, *A Fragment of a Grammar of Modern Irish*, Ph.D Thesis, University of Texas at Austin. *Texas Linguistic Forum* **12**.

Moortgat, Michael: 1985, 'Mixed Composition and Discontinuous Dependencies', paper to the Conference on Categorial Grammar, Tucson, AR, June 1985, in Richard T. Oehrle, E.. Bach and D. Wheeler, (eds), *Categorial Grammars and Natural Language Structures*, Reidel, Dordrecht, (in press).

Pareschi, Remo: 1985, 'Combinatory Categorial Grammar, Logic Programming, and the Parsing of Natural Language', DAI Working Paper, University of Edinburgh.

Pareschi, Remo, and Mark Steedman: 1987, 'A lazy way to chart parse with categorial grammars', paper to ACL conference, Stanford July 1987, ms. CIS, University of Pennsylvania.

Perlmutter, David M: 1971, *Deep and Surface Structure Constraints in Syntax*, Holt Rhinehart and Winston, New York.

Pollard, Carl: 1985a, 'Lectures on HPSG', ms. Stanford University.