Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language

Amy Felty
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Abstract

We argue that a logic programming language with a higher-order intuitionistic logic as its foundation can be used both to naturally specify and implement tactic style theorem provers. The language extends traditional logic programming languages by replacing first-order terms with simply-typed λ-terms, replacing first-order unification with higher-order unification, and allowing implication and universal quantification in queries and the bodies of clauses. Inference rules for a variety of inference systems can be naturally specified in this language. The higher-order features of the language contribute to a concise specification of provisos concerning variable occurrences in formulas and the discharge of assumptions present in many inference systems. Tactics and tacticals, which provide a framework for high-level control over search for proofs, can be directly and naturally implemented in the extended language. This framework serves as a starting point for implementing theorem provers and proof systems that can integrate many diversified operations on formulas and proofs for a variety of logics. We present an extensive set of examples that have been implemented in the higher-order logic programming language λProlog.

Key Words. Tactics, tacticals, theorem proving, proof systems, natural deduction, logic programming, higher-order logic, logical frameworks.

1 Introduction

The operations of search and unification, which are essential for the implementation of most theorem provers, are directly available in logic programming languages. This fact suggests that such languages should provide good implementation languages for theorem provers. In addition, the declarative nature of logic programs should aid in providing high-level specifications of the tasks involved in theorem proving. For example, propositions in the logic programming language Prolog [40] are clauses with a top-level implication where a clause *body* implies its *head*. The specification of inference rules expressing provability in a particular logic should map directly to this setting: the conclusion of a rule maps to a head of a clause while the premises are specified by the body. A set of clauses specifying a set of inference rules then serves as a specification of a theorem prover for the logic in question. Operationally, search and unification can be used to determine which inference rules can be applied and to produce the proper instances of these rules. In addition, the declarative reading of such logic programs should help in both understanding and proving formal properties, such as soundness of an implementation of a particular logic or completeness of a representation of a logic as a logic program.

The functional programming language ML was originally developed as the metalanguage for the implementation of theorem provers and has been used for this purpose in such notable theorem proving systems as Edinburgh LCF [17], HOL [18], Nuprl [3], and Isabelle [35]. ML contains many features that are useful for the design of theorem provers. It has a secure typing scheme, and is higher-order, allowing complex operations to be composed easily. In addition, it contains pattern matching capabilities which allow flexible manipulation of data objects. While ML has been used with much success in implementing theorem provers, many of the characteristics of logic programming languages suggest that such languages are worth investigating as an alternative in which certain basic operations such as search and unification are available more directly.

The basic data structure of traditional logic programming languages such as Prolog is first-order terms. Such terms, however, cannot provide a direct representation of quantification in formulas in first-order logic, or in any other logic that contains quantifiers. In particular, they cannot be used to characterize the notions of variables and the scopes of variable bindings in such formulas. Of course, quantification can be specially encoded. For example, in Prolog, we can represent abstractions in formulas by representing bound variables as either Prolog free variables or constants. The formula $\forall x \exists y\, p(x,y)$, for instance, could be written as the first-order term

`forall(X,exists(Y,p(X,Y)))` or `forall(x,exists(y,p(x,y)))`

(where capital letters represent free variables and lower case letters represent constants). In either representation, occurrences of variables inside the scope of quantifiers must be distinguished from those outside it. In the first case, substitution and unification that is available on free variables in Prolog cannot be used directly to provide substitution for first-order formulas. In other words, Prolog's unification cannot provide unification at the object-level. In either case, the programmer would have to write special procedures that accomplish these tasks for the encoded representation. Programs that manipulate such encodings are not generally declarative in nature.

In this paper, we argue that a higher-order logic programming language based on *higher-order hereditary Harrop formulas* [30] is well-suited to the tasks of specifying and implementing theorem provers. This language replaces first-order terms with simply typed λ-terms. These terms can be used to elegantly express the *higher-order abstract syntax* of object-logics. For example, the abstractions built into λ-terms can be used to represent quantification. It is then possible to directly specify the operations of quantifier instantiation and substitution in terms of application of λ-terms. Abstraction in λ-terms also allows us to represent notions of abstraction found in many proof systems. For example, eigenvariables in natural deduction proofs provide a notion of variables bound inside proofs [39]. In addition, in natural deduction, a proof of an implication $A \supset B$ can be considered a function from proofs of A to proofs of B. Terms representing proofs can be constructed in which these notions are captured.

The extended language also permits queries and the bodies of clauses to contain both implication and universal quantification. We shall show how universal quantification can be used to specify the provisos on inference rules in many proof systems concerning the occurrences of variables in formulas. Such uses of universal quantification are in fact essential for the correct implementation of various kinds of theorem provers for these logics. In addition, implication is useful for specifying the discharge of assumptions in natural deduction systems.

In terms of implementation, depth-first search provided by most logic programming languages is rarely sufficient for the complex task of theorem proving. We shall show how *tactics* and *tacticals*, which provide more flexibility in controlling search, can be directly implemented in the higher-order logic programming language. For instance, quantification over higher-order objects such as predicates allows an elegant implementation of tacticals, which provide basic control mechanisms for proof search. Such procedures take as parameters the various primitive operations of a particular theorem prover and compose them in various ways to form more complex operations and proof search strategies, known as tactics. Tactics and tacticals provide a framework which can be extended modularly to integrate many potentially diverse operations on formulas and proofs for a variety of logics in one unified setting.

In Section 2, we present the extended logic programming language. In Section 3, we illustrate how this language can be used to specify inference rules. We specify both natural deduction and sequent proof systems for first-order logic, and then illustrate the specification of a higher-order logic with quantification over simply-typed λ-terms. While these examples illustrate the specification power of the language, they do not provide complete implementations of theorem provers because of the limitations of depth-first search. However, they do provide complete proof checkers. In addition, they provide simple direct encodings of logics which we use to illustrate a general pattern for providing formal proofs of correctness. In Section 4, we prove the correctness of the specification for natural deduction for first-order logic.

In the remaining sections, we focus on implementation of theorem provers. In Section 5, we illustrate how inference rule specifications can serve as the basic search operations of a tactic style theorem prover. Section 6 provides an implementation of the general theorem proving interpreter which includes the tacticals and basic operations for providing user interaction in searching for proofs. Building upon the tactics and tacticals implemented in Sections 5 and 6, we complete an implementation of a tactic style theorem prover for natural deduction for first-order logic in Section 7. Finally, we discuss related work in Section 8.

2 A Higher-Order Logic Programming Language

Higher-order hereditary Harrop (hohh) formulas extend positive Horn clauses in essentially two ways. The first extension permits richer logical expressions in both queries (goals) and the bodies of program clauses. In particular, this extension provides for implications and universal quantification, in addition to conjunctions, disjunctions, and existentially quantified formulas. The second extension to Horn clauses makes this language *higher-order* in the sense that it is possible to quantify over predicate and function symbols. For a complete realization of this kind of extension, several other features must be added. In order to instantiate predicate and function variables with terms, first-order terms are replaced by more expressive simply typed λ-terms. The application of λ-terms is handled by β-conversion, while the unification of λ-terms is handled by higher-order unification.

The types of the language include a set of primitive types containing at least the type symbol o which denotes the type of logic programming propositions, and is closed under the formation of functional types, *i.e.*, if τ_1 and τ_2 are types then so is $\tau_1 \rightarrow \tau_2$. The arrow type constructor associates to the right. If τ_0 is a primitive type then the type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ has $\tau_1, \ldots, \tau_n$ as *argument types* and τ_0 as *target type.* We say the *order* of a primitive type is 0 while the order of a non-primitive type is one greater than the maximum order of its argument types. We assume that there are denumerably many constants and variables of each type. Simply-typed λ-terms are built up in the usual way from these constants and variables using abstraction and application. Application associates to the left.

In this language, equality between λ-terms is taken to mean $\beta\eta$-convertibility. We shall assume that the reader is familiar with the usual notions and properties of α, β, and η conversion for the simply typed λ-calculus. See Hindley and Seldin [22] for a fuller discussion. We just review some basic notions and our notation here. The relation of convertibility up to α is written as $=$, up to α and β as $=_\beta$, and up to α, β, and η as $=_{\beta\eta}$. We say that a λ-term is in *β-normal form* if it contains no beta redexes, that is, subterms of the form $(\lambda x\ t)s$. A λ-term is in *$\beta\eta$-long form* if it is of the form

$$\lambda x_1 \ldots \lambda x_n (h t_1 \ldots t_m) \qquad (n, m \geq 0)$$

where h, called the *head* of the term, is either a constant or a variable, where the expression $ht_1 \ldots t_m$ is of primitive type, and where the terms $t_1, \ldots, t_m$ are also in $\beta\eta$-long form. All λ-terms $\beta\eta$-convert to a term in $\beta\eta$-long form, unique up to α-conversion. We write $[t/x]s$ to denote the term obtained by replacing all free occurrences of x in s with t, systematically renaming bound variables in order to avoid variable capture.

A λ-term which is of type o is called a *proposition.* Logical connectives and quantifiers are introduced into λ-terms by introducing suitable constants as in Church [2]. In particular, the constants $\wedge, \vee, \supset$ are all given type $o \rightarrow o \rightarrow o$, and the constants $\forall$ and $\exists$ are given type $(\alpha \rightarrow o) \rightarrow o$ for each type replacing the "type variable" α. (Negation is not used in this programming language.) The expressions $\forall(\lambda x\ A)$ and $\exists(\lambda x\ A)$ are abbreviated to be $\forall x A$ and $\exists x A$, respectively. $\wedge, \vee$, and $\supset$ will be written as infix constants. A function symbol whose target type is o will be considered a *predicate.* A λ-term of type

o such that the head of its $\beta\eta$-long form is not a logical constant will be called an *atomic formula*.

We now define two new classes of propositions, called *goal formulas* and *definite clauses* (or just *clauses*). Let A be a syntactic variable for atomic formulas, G a syntactic variable for goal formulas, and D a syntactic variable for definite clauses. These two classes of formulas are defined by the following mutual recursion.

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

$$D := A \mid G \supset A \mid \forall x D$$

Note that the top-level form of a definite clause is either $\forall x_1 \ldots \forall x_n A$ or $\forall x_1 \ldots \forall x_n (G \supset A)$ where $n \geq 0$. The atomic formula A is called the *head* of the clause, and in the latter case, G is called the *body*. There is one final restriction on definite clauses: the head of a definite clause must have a constant as its head. The heads of atomic goal formulas on the other hand may be either variable or constant. A *logic program* or just simply a *program* is a finite set of definite clauses.

Note that this logic allows for quantification over arbitrary predicates, a feature which we will not make use of until Section 6. Quantification over function symbols on the other hand will be used extensively throughout the paper, although it will generally be restricted to variables having types of order 1.

Given a set of primitive types $\mathcal{B}$, a *signature (over $\mathcal{B}$)* is a finite set Σ of constants and variables such that there is at least one constant or variable of every primitive type. If a is a constant or variable of type τ, we sometimes write $a\colon\tau$ to make the type explicit. Given a signature Σ, a term of type o is said to be a *Σ-clause* if it is a definite clause built using only the constants and variables in Σ, the logical constants, and application and abstraction. Similarly, a term of type o is a *Σ-goal* if it is a goal formula built from the constants and variables in Σ and the logical constants. A term of any type is said to be a *Σ-term* if it is built using the constants and variables in Σ and the logical constants except for $\supset$.

We present a search procedure for hohh whose soundness and completeness follows from properties shown by Miller et. al. [30]. In this procedure, it will be important that instances of existentially quantified goal formulas and universally quantified clauses are also goal formulas and clauses, respectively. One way to insure this property is to disallow implication in substitution terms. Thus, we will only consider Σ-terms as substitution terms. See [30] for more details. The theorem below provides a high-level description of the search procedure. In this theorem, given a signature Σ and a set of Σ-clauses $\mathcal{P}$, the set $|\mathcal{P}|_\Sigma$ is defined to be the smallest set of clauses such that $\mathcal{P} \subseteq |\mathcal{P}|_\Sigma$ and if $\forall x D \in |\mathcal{P}|_\Sigma$ and t is a Σ-term of the same type as x, then $[t/x]D \in |\mathcal{P}|_\Sigma$.

Theorem 2.1 A sound and complete (with respect to intuitionistic logic) *non-deterministic* search procedure for hohh can be organized using the following six search primitives. In these operations, Σ is the current signature and $\mathcal{P}$ the current program. The clauses in $\mathcal{P}$ are Σ-clauses and the current goal is a Σ-goal.

AND: $G_1 \wedge G_2$ is provable from Σ and $\mathcal{P}$ if and only if both G_1 and G_2 are provable from Σ and $\mathcal{P}$.

OR: $G_1 \vee G_2$ is provable from Σ and $\mathcal{P}$ if and only if G_1 or G_2 is provable from Σ and $\mathcal{P}$.

INSTANCE: $\exists x G$ is provable from Σ and $\mathcal{P}$ if and only if there is some Σ-term t of the same type as x such that $[t/x]G$ is provable from Σ and $\mathcal{P}$.

GENERIC: $\forall x G$ is provable from Σ and $\mathcal{P}$ if and only if $[c/x]G$ is provable from $\Sigma \cup \{c\}$ and $\mathcal{P}$ for any constant or variable c of the same type as x that does not occur in Σ.

AUGMENT: $D \supset G$ is provable from Σ and $\mathcal{P}$ if and only if G is provable from Σ and $\mathcal{P} \cup \{D\}$.

BACKCHAIN: The atomic formula A is provable from Σ and $\mathcal{P}$ if and only if either $A \in |\mathcal{P}|_\Sigma$ or $G \supset A \in |\mathcal{P}|_\Sigma$ and G is provable from Σ and $\mathcal{P}$.

Note that the AUGMENT search operation extends the current program, while the GENERIC search operation extends the current signature. Also, note that we do not include a separate operation for conversion since we consider terms to be equivalent up to $\beta\eta$-conversion. This search procedure defines a fairly rigid structure for meta-proofs showing that a given goal formula follows from a given program. Proofs of this structure are called *uniform proofs* in Miller et. al. [30].

An implementation of a deterministic interpreter must make choices which are left unspecified by the high-level description of the non-deterministic interpreter. We describe here the choices made in the λProlog language, many of which are similar to those routinely used in Prolog.

The order in which conjuncts and disjuncts are attempted and the order for backchaining over definite clauses is determined exactly as in conventional Prolog: conjuncts and disjuncts are attempted in the order they are presented. Definite clauses are backchained over in the order they are listed in $\mathcal{P}$ using a depth-first search paradigm to handle failures. In the extended language, clauses can be added dynamically by the AUGMENT operation. We specify that new clauses get added to the top of the list.

The non-determinism in the INSTANCE operation is extreme. Generally when an existential goal is attempted, there is very little information available as to what Σ-term should be inserted. Instead, the Prolog implementation technique of instantiating the existential quantifier with a logic (free) variable which is later "filled in" using unification is employed. Thus instead of picking a Σ-term t, the INSTANCE search operation will introduce a new logic variable as the substitution term. A similar use of logic variables is made in implementing BACKCHAIN: instead of choosing a clause from $|\mathcal{P}|_\Sigma$, a clause from $\mathcal{P}$ is chosen and an instance is made by replacing all outermost universally quantified variables with new logic variables. Such logic variables are not part of the meta-logic hohh and thus are distinct from the variables that occur in Σ. The universal instance of this clause is then unified with the current goal. This operation may partially or fully instantiate the new logic variables.

The addition of logic variables in this setting requires higher-order unification since these variables can occur inside λ-terms. Also the equality of terms is not a simple syntactic check but a more complex check of $\beta\eta$-conversion. Higher-order unification is not in general decidable and most general unifiers do not necessarily exist when unifiers do exist. λProlog addresses these issues by implementing a depth-first version of the unification search procedure described by Huet [24]. It was shown by Miller et. al. [30] that such unification is sufficient for determining substitutions, and by Nadathur and Miller [31, 33], that this unification procedure can be smoothly integrated into the usual backtracking mechanism of logic programming languages. The higher-order unification problems we shall encounter in this paper are all rather simple. In fact all such problems are decidable. The presence of logic variables requires that GENERIC be implemented slightly differently than is described above. In particular, if the goal $\forall x G$ or the current program $\mathcal{P}$ contains logic variables, the new signature item c must not appear in the terms eventually instantiated for those logic variables. Several ways of handling the constraints on unification imposed by the GENERIC operation are discussed by Miller [28]. Without these checks, logic variables would not be a sound implementation technique. Note that the new signature item in the GENERIC operation can be either a variable or constant. In describing operational behavior of programs we will think of these new signature items as constants to avoid confusion with logic variables, while in establishing formal results, it will be convenient to use variables as new signature items.

In presenting programs in this paper, we adopt the syntax of the eLP [8] implementation of λProlog. Variables are represented by tokens with an upper case initial letter and constants are represented by tokens with a lower case initial letter. λ-abstraction is represented using backslash as an infix symbol. Terms are most accurately thought of as being representatives of $\beta\eta$-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.

The symbols `,` and `;` represent $\wedge$ and $\vee$ respectively, and `,` binds tighter than `;`. The symbol `:-` denotes "implied-by" while `=>` denotes the converse "implies." The first symbol is often used to write the top-level connective of definite clauses as in Prolog. Implications in goals and the bodies of clauses are always written using `=>`. Free variables in a definite clause are assumed to be universally quantified, while free variables in a goal are assumed to be existentially quantified. Universal and existential quantification within goals and definite clauses are written using the constants `pi` and `sigma` in conjunction with a λ-abstraction.

Primitive types are introduced using *kind declarations* and signature items are introduced using *type declarations*. For example, the type a and signature member $f : a \to a \to a$ can be introduced as follows.

```
kind    a       type.
type    f       a -> a -> a.
```

When type and kind declarations are omitted, they will be inferred by the interpreter. λProlog permits a degree of polymorphism by allowing type declarations to contain type variables (written as capital letters). It is also possible to build new "primitive" types from other types, using type constructors. In this paper, we will need to have only one such type constructor, `list`. We also introduce the standard constructors for lists: `nil` represents an empty list of polymorphic type (list A), and :: is the polymorphic cons operator of type A -> (list A) -> (list A). The latter will be written as an infix symbol. The programs in this paper will use the following operations on lists.

```
memb X (X::L).
memb X (Y::L) :- memb X L.

memb_and_rest A (A::Rest) Rest.
memb_and_rest A (B::Tail) (B::Rest) :- memb_and_rest A Tail Rest.

nth_item 0 A List :- !, memb A List.
nth_item 1 A (B::Rest) :- !, A = B.
nth_item N A (B::Tail) :- M is (N - 1), nth_item M A Tail.

nth_and_rest 0 A List Rest :- !, memb_and_rest A List Rest.
nth_and_rest 1 A (B::Rest) Rest :- !, A = B.
nth_and_rest N A (B::Tail) (B::Rest) :-
  M is (N - 1), nth_and_rest M A Tail Rest.
```

The memb predicate implements the standard test for membership in a list. The predicate memb_and_rest is similar to memb but contains an additional argument for the rest of the list minus the chosen item. The nth_item and nth_and_rest programs are similar to memb and memb_and_rest, respectively, but use an integer argument to specify a particular member of the list. When the integer argument is 0, they behave like memb or memb_and_rest.

Several non-logical features of λProlog will be used in this paper. The cut (!), as used above is one such feature. It is used to eliminate backtracking points. It is a goal which always succeeds and commits the interpreter to all choices made since the parent goal was unified with the head of the clause in which the cut occurs (see Sterling and Shapiro [40]). We also make use of write and read predicates. As in Prolog, (write A) prints the current binding of A to the screen and will always succeed. The read predicate has polymorphic type (A -> o) -> o. A goal of the form (read G) prompts the user for input of some term M, and then solves the goal (G M). When this goal fails, (read G) also fails. The equality predicate (=) verifies that its two arguments are unifiable. In nth_item and nth_and_rest, the unification of A and B is placed after the cut rather than replacing B with A in the head of the clause. This implementation is chosen for control reasons. In particular, it verifies that only the second clause can succeed when the integer argument is 1.

3 Specifying Inference Rules

In this section, we illustrate the specification of inference rules in hohh using several examples. We begin by considering the specification of natural deduction for first-order intuitionistic logic. We then consider the specification of sequent calculi, classical logic, the simply-typed λ-calculus, and higher-order logic. Since we will be specifying logics within a logic, to avoid confusion we will refer to hohh as the *meta-logic* and the logic being specified as the *object-logic*.

To represent a first-order logic, we introduce two primitive types: `form` for object-level formulas and `i` for first-order individuals. The new type `form` serves to distinguish formulas of the object-logic from formulas of the meta-logic (of type `o`). The connectives of the meta-logic have a set meaning, for example as given by the non-deterministic interpreter of the previous section, while the object-level connectives will have only the meaning attributed to them by the programs that use them. Given these new primitive types, we introduce constants for the object-level connectives: `and`, `or`, and, `imp` of type `form -> form -> form`, `neg` of type `form -> form`, and `forall` and `exists` of type `(i -> form) -> form`. We use usual infix notation for the binary connectives. The constants `forall` and `exists` take a functional argument, and thus object-level binding of variables by quantifiers is defined in terms of meta-level λ-abstraction. This representation of formulas is commonly adopted to express the higher-order abstract syntax of object-logics (*e.g.*, [35, 29, 21, 4, 37]).

We may also introduce non-logical constants into the logic such as a binary function symbol `f` of type `i -> i -> i` and unary predicates `p` and `q` of type `i -> form`. Using these definitions, the first-order formula $\forall x \exists y (p(x) \supset q(f(x,y)))$, for example, is represented by the λ-term `(forall X\ (exists Y\ ((p X) imp (q (f X Y)))))`.

To specify the inference rules of natural deduction, we introduce the primitive type `nprf` for proofs, and the infix constant `#` of type `nprf -> form -> o` for the meta-level relation between an object-level formula and its proofs. The inference rules for natural deduction as in Prawitz [38] are given in Figure 1[1]. Each rule will be expressed as a simple declarative fact about the `#` relation. Operationally, `#` can be viewed as the theorem proving predicate. An operational reading will generally provide a goal-directed description of search for proofs in the object-logic.

Proof objects can be useful in theorem proving systems for various operations such as extracting programs, generating explanations, or building analogous proofs of related theorems. There are often many choices in representing and constructing proofs which depend on how proofs will be ultimately used. The examples given here serve to illustrate both how they can be specified and how they are constructed during execution. Of course, a theorem prover need not build explicit proofs at all. In this example, we may replace the binary predicate `#` by a unary predicate, say `provable`, of type `form -> o`.

First, consider the $\wedge$-I inference rule in Figure 1 which introduces a conjunction in the conclusion. The declarative reading of this inference rule is captured by the following definite clause.

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

This clause may be read as: `(and_i P1 P2)` is a proof of `(A and B)` if `P1` is a proof of `A` and `P2` is a proof of `B`. The rule can also be viewed as defining the constant `and_i`: it is a function from proofs of the premises to a proof of the conclusion. Operationally, this rule can be employed when the formula to be proved is a conjunction. Using the BACKCHAIN search command, the formula and proof of the query must unify with the formula and

[1]Note that substitution terms are given as parameters to the quantifier inference rules in this presentation. Making this information explicit will allow us to establish a more direct correspondence between deduction trees and our representation of them.

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

$$\frac{A}{A \vee B}\ \vee\text{-I} \qquad\qquad \frac{B}{A \vee B}\ \vee\text{-I} \qquad\qquad \frac{A \vee B \qquad \begin{array}{c}(A)\\ C\end{array} \qquad \begin{array}{c}(B)\\ C\end{array}}{C}\ \vee\text{-E}$$

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

$$\frac{\begin{array}{c}(A)\\ \bot\end{array}}{\neg A}\ \neg\text{-I} \qquad\qquad \frac{A \qquad \neg A}{\bot}\ \neg\text{-E}$$

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

$$\frac{[t/x]A}{\exists x A}\ \exists\text{-I}(t) \qquad\qquad \frac{\exists x A \qquad \begin{array}{c}([y/x]A)\\ B\end{array}}{B}\ \exists\text{-E}(y)$$

$$\frac{\bot}{A}\ \bot\text{-I}$$

The $\forall$-I rule has the proviso that the variable y cannot appear free in $\forall x A$, or in any assumption on which the deduction of $[y/x]A$ depends.

The $\exists$-E rule has the proviso that the variable y cannot appear free in $\exists x A$, in B, or in any assumption on which the deduction of the upper occurrence of B depends.

Figure 1: Natural Deduction Inference System for First-Order Intuitionistic Logic

proof in the head of this clause. If there is a match, the AND search operation is used to verify the two new subgoals in the body of this clause. The unification here is essentially first-order.

The two inference rules for proving disjunctions, the $\vee$-I rules in Figure 1, have a very natural rendering as the following definite clause.

```
(or_i P) # (A or B) :- P # A; P # B.
```

Declaratively, this clause specifies the meaning of a proof of a disjunction. For `(or_i P)` to be a proof of `(A or B)`, `P` must be a proof of either `A` or `B`. Operationally, this clause would use an OR search operation to determine which of the subgoals in the body should succeed. Alternatively, we could choose to specify the two rules for $\vee$-I with two clauses which serve to indicate which instance of the rule is used.

```
(or_i1 P) # (A or B) :- P # A.
(or_i2 P) # (A or B) :- P # B.
```

We next consider quantifier introduction rules whose operational reading will use the INSTANCE and GENERIC search operations and second-order unification. The $\exists$-I inference rule can be written as the following definite clause.

```
(exists_i P) # (exists A) :- sigma T\ (P # (A T)).
```

Here `A` is a functional variable of type `i -> form`. The meta-level application `(A T)` represents the object-level formula that is obtained by substituting `T` for the top-level bound variable in `A`. Declaratively, this clause reads: if there exists a term `T` such that `P` is a proof of `(A T)`, then `(exists_i P)` is a proof of `(exists A)`. Operationally, we rely on second-order unification to instantiate the logic variable `A`. The existential instance `(A T)` is obtained via the interpreter's operation of β-reduction. Note that the implementation of INSTANCE will choose a logic variable with which to instantiate `T`. By making use of a logic variable here, we do not commit to a specific term for the substitution at the time the clause is invoked in backchaining. It will later be assigned a value through unification if there is such a value which results in a proof.

It may be desirable here to keep a record of the substitution terms used by including them in proof terms. For example, we may may include `T` as an argument to `exists_i`, as in the following clause.

```
(exists_i T P) # (exists A) :- P # (A T).
```

Note that the existential quantification over the body of the previous example is replaced here with a universal quantification over the whole clause (not shown explicitly here since, by convention, we assume universal closure at the top level).

The $\forall$-I rule of natural deduction has the additional proviso that the variable y is not free in $\forall x A$, or in any assumption on which the deduction of the premise $[y/x]A$ depends. Although our programming language does not contain a check for "not free in" it is still possible to specify this inference rule. This proviso is handled by using a universal quantifier at the meta-level as in the following clause.

```
(forall_i Q) # (forall A) :- pi Y\ ((Q Y) # (A Y)).
```

This clause can be interpreted declaratively as follows: if `Q` is a function that maps arbitrary terms `Y` to proofs `(Q Y)` of the formula `(A Y)`, then `(forall_i Q)` is a proof of `(forall A)`. Operationally, the GENERIC search operation is used to introduce a new constant of type `i` to be used as the substitution term. Since this constant will not be permitted to appear in `A` the proviso will be satisfied.

The $\supset$-I rule illustrates the specification of the discharge of assumptions. This rule can be naturally specified as the definite clause below, which uses both universal quantification and implication at the meta-level.

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

Declaratively, this clause represents the fact that if `Q` is a "proof function" which maps an arbitrary proof of `A`, say `P`, to a proof of `B`, namely `(Q P)`, then `(imp_i Q)` is a proof of `(A imp B)`. This clause illustrates a second way in which abstractions are introduced in proof terms. Here, `Q` is an abstraction over proofs.

Operationally, the AUGMENT search operation plays a role in implementing the discharge of assumptions. To solve the universally quantified subgoal in the above clause, first the GENERIC operation is used to choose a new object, say p, to replace P and play the role of a proof of the formula A. The AUGMENT operation is used to add the assumption (p # A) to the current set of program clauses. This clause is then available to use in the search for a proof of B. The proof of B may contain occurrences of p. The function Q is the result of fully abstracting p out of the proof of B.

The elimination rules and $\bot_I$ can be specified similarly to the introduction rules. Figure 2 contains a complete specification of natural deduction for intuitionistic logic. In this specification, proof terms contain just enough information so that given the proof term and the formula at the root, the corresponding natural deduction can be completely reconstructed. Many of the proof terms for elimination rules contain formulas. For example, B is included as an argument to and_e1 to record the conjunct that is dropped when applying the rule.

```
(and_i P1 P2) # (A and B) :- P1 # A, P2 # B.
(or_i1 P) # (A or B) :- P # A.
(or_i2 P) # (A or B) :- P # B.
(imp_i Q) # (A imp B) :- pi P\ ((P # A) => ((Q P) # B)).
(neg_i Q) # (neg A) :- pi P\ ((P # A) => ((Q P) # false)).
(exists_i T P) # (exists A) :- P # (A T).
(forall_i Q) # (forall A) :- pi Y\ ((Q Y) # (A Y)).
(false_i P) # A :- P # false.
(and_e1 B P) # A :- P # (A and B).
(and_e2 A P) # B :- P # (A and B).
(or_e A B P Q1 Q2) # C :- P # (A or B),
                          pi P1\ ((P1 # A) => ((Q1 P1) # C)),
                          pi P2\ ((P2 # B) => ((Q2 P2) # C)).
(imp_e A P1 P2) # B :- P1 # A, P2 # (A imp B).
(neg_e A P1 P2) # false :- P1 # A, P2 # (neg A).
(exists_e A P1 Q) # B :- P1 # (exists A),
                         pi Y\ (pi P\ ((P # (A Y)) => ((Q Y P) # B))).
(forall_e T A P) # (A T) :- P # (forall A).
```

Figure 2: A Complete Specification of Natural Deduction for Intuitionistic Logic

To illustrate the operational behavior of these clauses, we consider the construction of a proof for the formula $pa \supset \exists x \forall y(py \supset px)$. Let $\mathcal{P}$ be the set of clauses in Figure 2, and Σ the signature containing all of the constants that appear in these clauses plus a unary predicate p and a constant a. To represent the state of the interpreter, we write $\mathcal{P};\Sigma$?- G where G is the goal to be attempted using program $\mathcal{P}$ and signature Σ. For this example, our initial state is

$\mathcal{P}$; Σ ?-   R # ((p a) imp (exists X\(forall Y\((p Y) imp (p X)))))

where R is the logic variable to be filled in with a proof. The following represents the series of states and operations of the interpreter used in solving the query. Given program $\mathcal{P}$

and clause D, we write $\mathcal{P}, D$ to denote $\mathcal{P} \cup \{D\}$. Similar notation is used for signatures. We write GEN for GENERIC, AUG for AUGMENT, and BC followed by the name of an inference rule to indicate that the clause specifying that rule is used by the BACKCHAIN operation.

BC $\supset$-I	$\mathcal{P}\,;\Sigma$ `?- pi P\((P # (p a)) =>` `((R1 P) # (exists X\(forall Y\((p Y) imp (p X))))))`
GEN	$\mathcal{P}\,;\Sigma,$`r ?- (r # (p a)) =>` `((R1 r) # (exists X\(forall Y\((p Y) imp (p X)))))`
AUG	$\mathcal{P},$`r # (p a)` $;\Sigma,$`r ?-` `(R1 r) # (exists X\(forall Y\((p Y) imp (p X))))`
BC $\exists$-I	$\mathcal{P},$`r # (p a)` $;\Sigma,$`r ?- R2 # (forall Y\((p Y) imp (p T)))`
BC $\forall$-I	$\mathcal{P},$`r # (p a)` $;\Sigma,$`r ?- pi Y\((R3 Y) # ((p Y) imp (p T)))`
GEN	$\mathcal{P},$`r # (p a)` $;\Sigma,$`r,c ?- (R3 c) # ((p c) imp (p T))`
BC $\supset$-I, GEN, AUG	$\mathcal{P},$`r # (p a),s # (p c)` $;\Sigma,$`r,c,s ?- (R4 s) # (p T)`

Here, `r`, `c`, and `s` are constants introduced by the GENERIC operation, and `R1`, `R2`, `R3`, `R4` and `T` are new logic variables introduced in backchaining. A backchain on an atomic clause completes the derivation. Note that there are two clauses in the program which unify with the final goal. The latter is ruled out by the restriction on the GENERIC operation. The constant `c` was introduced after the variable `T`, and thus cannot appear in `T`. Thus the other atomic clause must be used to complete the proof and `T` gets assigned `a`. The complete unification problem generated by unifying the goal with the head of a clause in each of the above uses of BACKCHAIN is the following:

$$\texttt{R} = (\texttt{imp\_i R1}), (\texttt{R1 r}) = (\texttt{exists\_i T R2}), \texttt{R2} = (\texttt{forall\_i R3}),$$
$$(\texttt{R3 c}) = (\texttt{imp\_i R4}), (\texttt{R4 s}) = \texttt{r}, \texttt{T} = \texttt{a}.$$

After assigning `a` to `T`, it is easy to solve for `R4`, `R3`, and `R2`: `R4` is assigned `P\r`, `R3` is assigned `Y\(imp_i P\r)`, and `R2` is assigned `(forall_i Y\(imp_i P\r))`. Since `R1` cannot contain `r`, it must be assigned the term `Q\(exists_i a (forall_i Y\(imp_i P\Q)))`. Thus, as a final solution for `R`, we obtain the term

```
(imp_i Q\(exists_i a (forall_i Y\(imp_i P\Q)))).
```

In specifying natural deduction, we have considered both declarative aspects as well as operational behavior of individual clauses. Now, consider the specification in Figure 2 with respect to deterministic depth-first control. If we view this program as a proof checker, that is, where initial queries contain closed proof terms, then there is little problem in controlling execution. The top-level constant of a proof term completely determines the unique definite clause which can be used in backchaining at each step. Not surprisingly, the execution of this program under depth-first control is not sufficient for theorem proving. When proof terms in queries are variables, there will in general be multiple definite clauses that could be applied to any one formula. The clause for `and_e1`, for example, can always be used in backchaining, since `A` in the head of the clause can be unified with any formula. The program may enter an infinite loop repeatedly applying this rule. In some cases, it may be possible to modify specifications so that they act as complete automatic theorem provers under depth-first control. Such a theorem prover for the classical sequent calculus can be found in Felty [10]. Beginning in Section 5, we consider the implementation of tactic style theorem provers which provide more flexible forms of control. There, we will discuss a modified specification of inference rules, such that clauses may be used as tactics implementing the basic operations of a theorem prover.

Before doing so, we continue using a simpler more direct specification in the remainder of this section and the next. Here, we present a few more specification examples and in the next section we discuss correctness of specifications and present the correctness proof for natural deduction. We next briefly consider specifying a sequent system for first-order intuitionistic logic. To define sequents, we introduce a new primitive type `seq` and a constant `-->` of type `(list form) -> form -> seq` written as an infix operator whose antecedent is a list of formulas and succedent is a single formula. The basic relation between a sequent and its proofs will be represented by the infix constant `>-` of type `sprf -> seq -> o` where `sprf` is the type of sequent proofs.

Rules that introduce a connective on the right of a sequent resemble introduction rules in natural deduction. For example, the following rule from Gentzen [16] introduces universal quantification on the right.

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

The following clause encodes this rule.

```
(forall_r Q) >- (Gamma --> (forall A)) :- pi Y\ ((Q Y) >- (Gamma --> (A Y))).
```

In this rule, the variable y cannot appear free in the lower sequent. As in natural deduction, universal quantification at the meta-level is used to handle this proviso. Introductions of logical constants into the antecedent of a sequent can be achieved similarly. The main difference here is that the antecedent is a list instead of a single formula. As an example, the rule introducing implication on the left and its specification are given below.

$$\frac{\Gamma \longrightarrow A \qquad B, \Gamma \longrightarrow C}{A \supset B, \Gamma \longrightarrow C}\ \supset\text{-L}$$

```
(imp_l P1 P2) >- ((A imp B)::Gamma --> C) :- P1 >- (Gamma --> A),
                                             P2 >- ((B::Gamma) --> C).
```

The structural rules of contraction, thinning, and interchange can be specified by simply manipulating lists of formulas. In addition, we need the clause below specifying initial sequents, that is, a sequent whose antecedent contains one formula which is also its succedent.

```
(initial A) >- ((A::nil) --> A).
```

Classical logic can be specified similarly to intuitionistic logic. For the sequent calculus, we must introduce lists on both sides of the sequent arrow, while for natural deduction, we simply replace the clause for the $\bot_I$ rule with a clause for the corresponding rule for classical logic [38].

We next consider the specification of a higher-order logic. For simplicity, we consider a logic containing only the $\supset$ and $\forall$ connectives. This logic will allow quantification over simply-typed λ-terms of arbitrary types. Since λ-terms are available at the meta-level, a first approach to specifying formulas might be to use these terms directly. We would then have to declare polymorphic quantifiers, *i.e.*, to introduce constants `forall` and `exists` of type `(S -> form) -> form`, where the type variable `S` indicates that the argument to `exists` or `forall` can be an abstraction over an object of any meta-level type. There are several reasons to avoid the use of type variables here. For instance, `S` can be instantiated with types we may not intend to quantify over at the object-level, such as the type `o` of formulas of the metalanguage. Also, such a use of polymorphism can cause undesirable operational behavior during unification. On simple unification problems, for example, there may be infinite branching in the search for instances of both types and terms. (See Felty [10] for a fuller discussion of these problems.) Instead, we introduce a new type `tm` to represent object-level terms, and a second type `ty` to represent object-level types. To construct function types, we use the infix arrow `-->` of type `ty -> ty -> ty`. If our object-language has a function symbol f of type $(i \to i) \to i$, we introduce meta-level constants `f` of type `tm`, `i` of type `ty`, and write `(i --> i) --> i` to represent the object-level type of `f`. To represent λ-terms, we introduce the constants `app` of type `tm -> tm -> tm` and `abs` of type `(tm -> tm) -> tm` to represent application and abstraction, respectively.

We must now express the relation between a term and its type at the meta-level. We introduce the infix predicate `#t` for this relation. The clauses below specify typing rules.

```
f #t ((i --> i) --> i).
(abs M) #t (R --> S) :- pi X\ ((X #t R) => ((M X) #t S)).
(app M N) #t S :- M #t (R --> S), N #t R.
imp #t (form --> form --> form).
(forall S) #t ((S --> form) --> form).
q #t ((i --> i) --> i -> form).
```

The first formula expresses the relation between `f` and its type. The next two formulas encode the usual rules for abstraction and application. In this example, formulas must also be terms of type `tm` at the meta-level, and must be type-checked in order to insure that they have object-level type `form`. The remaining three rules specify typing rules for formulas. In order to allow quantification at every type, the constant `forall` takes a type as an argument. Thus `(forall S)` represents the universal quantifier over objects of type `S`. Type assignment clauses must also be included for all predicates. Here, `q` is a predicate specifying a relation between a function and an element of type `i`.

In this example, we consider equality up to $\beta\eta$-conversion. This relation also must be specified explicitly. To do so, we introduce the binary predicate `conv` on terms, and provide the following clauses.

```
conv (app (abs M) N) (M N).
conv (abs X\(app M X)) M.
conv (app M N) (app P Q) :- conv M P, conv N Q.
conv (abs M) (abs N) :- pi X\ (conv (M X) (N X)).
conv M M.
```

```
conv M N :- conv N M.
conv M N :- conv M P, conv P N.
```

The first two clauses specify the β and η axioms, while the next two clauses specify convertibility inside an application and within the scope of an abstraction. Finally, the remaining clauses express reflexivity, symmetry, and transitivity for $\beta\eta$-convertibility.

To specify a natural deduction proof system for higher-order logic, we introduce the predicate `#p` of type `nprf -> tm -> o` to represent the basic relation between a formula and its proofs. The introduction and elimination rules of natural deduction are given below. In these clauses, we assume that the term on the right side of `#p` in the head of each clause has type `form`. As a result, very few type-checking subgoals will be needed in the bodies of these clauses.

```
(convert A B P) #p A :- B #t form, P #p B, conv A B.
(imp_i Q) #p (app (app imp A) B) :- pi P\ ((P #p A) => ((Q P) #p B)).
(imp_e A P1 P2) #p B :- P1 #p A, A #t form, P2 #p (app (app imp A) B).
(forall_i S Q) #p (app (forall S) A) :-
   pi Y\ ((Y #t S) => ((Q Y) #p (app A Y))).
(forall_e S T A P) #p (app A T) :- T #t S, P #p (app (forall S) A).
```

The first clause specifies the inference rule that states that if a formula is provable, any $\beta\eta$-equivalent formula is also provable. The next two clauses illustrate that propositional rules are specified similarly to those for first-order logic, except that formulas must now be written using the encoding of λ-terms. The clause for $\supset$-E also requires an additional type-checking subgoal to insure `A` has type `form`. The clause for $\forall$-I is also is similar to the first-order version except that here meta-level implication is necessary to add an assumption about the type of the new signature item introduced for `Y`. Note that in the case when `A` is an abstraction (has the form `(abs B)`), the object-level application `(app A Y)` is used to represent substitution of the new signature item introduced for `Y` for the bound variable in the abstraction, but no substitution occurs at the time that the rule is applied. However if an application of the inference rule for $\beta\eta$-convertibility is then applied, the β-reduction rule of the `conv` program can be used to perform the necessary substitution. Finally, in the $\forall$-E rule of higher-order logic, the quantified object can be of any type and a subgoal must be added to verify correct typing of the substitution term.

4 Correctness of Specifications

In specifying various logics, we represented terms, formulas, inference rules, and proofs at the object-level as terms and formulas of the metalanguage. In the specifications, there was always a clear correspondence between objects in the two languages. We now illustrate how to formalize this connection by proving the correctness of the specification of natural deduction in Figure 2. The correspondence between λ-abstraction and the discharge of assumptions and variables in natural deduction proofs is a well-known consequence of the Curry-Howard isomorphism [23]. The results presented here provide a formalization of this correspondence for our representation of proofs and the specification of inference rules as hohh formulas.

Although terms of the metalanguage are equivalent up to $\beta\eta$-convertibility, we will often need a representative from a $\beta\eta$-equivalence class of terms. In this section, we will always choose the $\beta\eta$-long form.

At the object-level, we assume a fixed set of constants, function symbols, propositions, and predicate symbols. We assume the existence of a bijective mapping Φ from these objects to constants of the metalanguage. Φ maps each object-level constant to a constant of type `i`, each proposition to a constant of type `form`, and each function symbol or predicate of arity n to a constant with target type `i` or `form`, respectively, and n argument types `i`. Using the functions and predicates given in the example execution in the previous section, for example, we can define Φ to be the mapping: $\Phi(a) = \mathtt{a : i}$ and $\Phi(p) = \mathtt{p : i \text{ -> } form}$. We write $\mathrm{dom}(\Phi)$ to denote the domain of Φ. We also assume a countably infinite set of first-order variables, and a fixed mapping ρ from these variables to the meta-variables of type `i`. Using these functions, we can define an encoding on terms and formulas in the obvious way which we give explicitly below. We write $\langle\!\langle A \rangle\!\rangle$ to denote the encoding of term or formula A.

$$
\begin{array}{rcl}
\langle\!\langle x \rangle\!\rangle & := & \rho(x) \text{ for variable } x \\
\langle\!\langle p(t_1, \ldots, t_n) \rangle\!\rangle & := & (\Phi(p)\ \langle\!\langle t_1 \rangle\!\rangle \ldots \langle\!\langle t_n \rangle\!\rangle) \\
 & & \text{for function or predicate symbol } p \in \mathrm{dom}(\Phi) \text{ of arity } n \geq 0 \\
\langle\!\langle A \wedge B \rangle\!\rangle & := & (\langle\!\langle A \rangle\!\rangle\ \mathtt{and}\ \langle\!\langle B \rangle\!\rangle) \\
\langle\!\langle A \vee B \rangle\!\rangle & := & (\langle\!\langle A \rangle\!\rangle\ \mathtt{or}\ \langle\!\langle B \rangle\!\rangle) \\
\langle\!\langle A \supset B \rangle\!\rangle & := & (\langle\!\langle A \rangle\!\rangle\ \mathtt{imp}\ \langle\!\langle B \rangle\!\rangle) \\
\langle\!\langle \neg A \rangle\!\rangle & := & (\mathtt{neg}\ \langle\!\langle A \rangle\!\rangle) \\
\langle\!\langle \forall x\, A \rangle\!\rangle & := & (\mathtt{forall\ X\backslash}\ \langle\!\langle A \rangle\!\rangle) \quad \text{where } \rho(x) = \mathtt{X} \\
\langle\!\langle \exists x\, A \rangle\!\rangle & := & (\mathtt{exists\ X\backslash}\ \langle\!\langle A \rangle\!\rangle) \quad \text{where } \rho(x) = \mathtt{X} \\
\langle\!\langle \perp \rangle\!\rangle & := & \mathtt{false}
\end{array}
$$

Note that any first-order term or formula is mapped to a term in the metalanguage in $\beta\eta$-long form. We will adopt the convention that ρ assigns an object-level variable written as a lower case letter to the corresponding meta-variable written as an upper case letter, *e.g.*, $\rho(x) = \mathtt{X}$. The above encoding has the following property: Given first-order terms or formulas M and N, and variable x, $\langle\!\langle [N/x]M \rangle\!\rangle = [\langle\!\langle N \rangle\!\rangle/\mathtt{X}]\langle\!\langle M \rangle\!\rangle$.

Our encoding for first-order terms and formulas is essentially the same as the encoding of first-order terms and formulas in LF given by Harper et. al. [21]. There, the encoding of terms and formulas is defined within the sublanguage of LF that corresponds to the simply-typed λ-calculus. The proofs given there for Adequacy of Syntax, I and II, can be applied here in a straightforward manner by replacing the notion of LF canonical forms there with $\beta\eta$-long forms here. These proofs use a function which is shown to be the inverse of the encoding, which we call a *decoding* here. Let $\mathcal{T}_0$ be the set containing the constants in $\{\mathtt{and}, \mathtt{or}, \mathtt{imp}, \mathtt{forall}, \mathtt{exists}, \mathtt{false}\}$, the constants in the codomain of Φ, and the variables in the codomain of ρ. Let $\mathcal{T}$ be the set of simply-typed λ-terms built up from abstraction, application, and the constants and variables in $\mathcal{T}_0$. The decoding is defined in the obvious way from terms in $\beta\eta$-long form of type `i` and `form` in $\mathcal{T}$ to first-order terms and formulas. We denote the decoding of term `M` as $\|\mathtt{M}\|$.

Before proving the correctness of the natural deduction specification, we make some notions about deductions precise. Several rules of natural deduction may discharge assumptions. For example, in the $\supset$-I rule, (A) indicates that occurrences of A at the leaves are discharged by the application of this rule. A formula occurrence B in a tree is said to *depend* on an assumption A if A occurs as a leaf and is not discharged by a rule application above B. A *deduction* of B from a set of formulas Γ is a tree with root B constructed using the inference rules in Figure 1 in which all assumptions on which B depends occur in Γ. Such a tree is a *proof* of B if Γ is empty. We often write a set of assumptions Γ as a list of formulas in which it is understood that a formula may occur more than once.

Given a deduction Π of B from assumptions $A_1, \ldots, A_n$, we say that a variable x *has a free occurrence in deduction* Π if x occurs free in $A_1, \ldots, A_n, B$, any node in Π, or in the substitution terms introduced in applications of $\exists$-I and $\forall$-E. In the results that follow, we will assume that all variables introduced by an application of $\forall$-I or $\exists$-E are distinct, do not have free occurrences in $A_1, \ldots, A_n, B$, and only have free occurrences in the subtrees of Π in which they are introduced, *i.e.*, in the subtree rooted at the premise of $\forall$-I or in the subtree rooted at the right premise of $\exists$-E. Such variables can always be renamed to meet this criteria [39]. We define $\Sigma(\Pi)$ to be the signature containing $\rho(x)$ for every variable x that has a free occurrence in Π except for those occurring as parameters to $\forall$-I or $\exists$-E.

Let $\mathcal{T}_0'$ be the set containing $\mathcal{T}_0$ plus all the constants used to build natural deduction proof terms. We denote the signature obtained by removing the infinite set of variables from $\mathcal{T}_0'$ as Σ_{ND}. Let $\mathcal{T}'$ be the set of simply-typed λ-terms built up from abstraction, application, and the constants and variables in $\mathcal{T}_0'$. Finally, let $\mathcal{P}_{ND}$ be the set of clauses in Figure 2.

It is easy to see that a variable x is free in a formula B if and only if $\rho(x)$ is free in $\langle\!\langle B \rangle\!\rangle$. For deduction Π of B from $A_1, \ldots, A_n$, it follows from this fact that $\langle\!\langle A \rangle\!\rangle, \langle\!\langle A_1 \rangle\!\rangle, \ldots, \langle\!\langle A_n \rangle\!\rangle, \langle\!\langle B \rangle\!\rangle$ are all $\Sigma_{ND} \cup \Sigma(\Pi)$-terms. Similarly, a meta-variable $\mathtt{X}$ is free in term $\mathtt{B} \in \mathcal{T}$ if and only if $\rho^{-1}(\mathtt{X})$ is free in $\|\mathtt{B}\|$.

Theorem 4.1 (Correctness)

1. Let Π be a natural deduction proof of B. Let Σ be $\Sigma_{ND} \cup \Sigma(\Pi)$. Then there exists a Σ-term $\mathtt{R}$ of type $\mathtt{nprf}$ such that $(\mathtt{R}\ \#\ \langle\!\langle B \rangle\!\rangle)$ is provable from $\Sigma; \mathcal{P}_{ND}$.

2. Let $\mathtt{B}$ be a term of type $\mathtt{form}$ in $\mathcal{T}$ and let $\mathtt{R}$ be a term of type $\mathtt{nprf}$ in $\mathcal{T}'$. Let Σ be the signature containing Σ_{ND} and possibly a finite number of variables of type $\mathtt{i}$ including at least all those that occur free in $\mathtt{B}$ and $\mathtt{R}$. If $(\mathtt{R}\ \#\ \mathtt{B})$ is provable from $\Sigma; \mathcal{P}_{ND}$, then $\|\mathtt{B}\|$ has a natural deduction proof.

Proof: (1) follows from the slightly more general statement: Let Π be a deduction of B from $A_1, \ldots, A_n$ where $n \geq 0$. Let $\mathtt{P}_1, \ldots, \mathtt{P}_n$ be n distinct variables of type $\mathtt{nprf}$. Let Σ and $\mathcal{P}$ be the following signature and set of clauses.

$$\begin{array}{rcl} \Sigma & := & \Sigma_{ND} \cup \Sigma(\Pi) \cup \{\mathtt{P}_1, \ldots, \mathtt{P}_n\} \\ \mathcal{P} & := & \mathcal{P}_{ND} \cup \{(\mathtt{P}_1\ \#\ \langle\!\langle A_1 \rangle\!\rangle), \ldots, (\mathtt{P}_n\ \#\ \langle\!\langle A_n \rangle\!\rangle)\} \end{array}$$

Then there exists a Σ-term `R` of type `nprf` such that `(R #` $\langle\!\langle B \rangle\!\rangle$`)` is provable from $\Sigma; \mathcal{P}$.

The proof is by induction on the height of Π. We show a few cases. First, if Π is a one node tree then it must be A_i for some i, $1 \leq i \leq n$. We know $(\mathtt{P}_i \ \texttt{\#} \ \langle\!\langle A_i \rangle\!\rangle)$ is provable by BACKCHAIN on an atomic clause, and thus we can take `R` to be $\mathtt{P}_i$.

If the last step in the deduction is an application of $\exists$-I(t), then B has the form $\exists x B'$, and the premise has the form $[t/x]B'$. All the variables free in $\langle\!\langle t \rangle\!\rangle$ are in $\Sigma(\Pi)$ and thus $\langle\!\langle t \rangle\!\rangle$ and $\langle\!\langle [t/x]B' \rangle\!\rangle$ are Σ-terms. By the induction hypothesis, there exists a Σ-term `S` of type `nprf`, such that `(S #` $\langle\!\langle [t/x]B' \rangle\!\rangle$`)` is provable from $\Sigma; \mathcal{P}$. Note that

$$\langle\!\langle [t/x]B' \rangle\!\rangle = [\langle\!\langle t \rangle\!\rangle / \mathtt{X}]\langle\!\langle B' \rangle\!\rangle =_\beta (\mathtt{X}\backslash\langle\!\langle B' \rangle\!\rangle \ \langle\!\langle t \rangle\!\rangle).$$

By BACKCHAIN on the clause for $\exists$-I, `(exists_i` $\langle\!\langle t \rangle\!\rangle$ `S) # (exists X\`$\langle\!\langle B' \rangle\!\rangle$`)` is provable from $\Sigma; \mathcal{P}$.

If the last step in the deduction is an application of $\forall$-I(y), then B has the form $\forall x B'$, and the premise has the form $[y/x]B'$. Since y is a parameter to this application of $\forall$-I, `Y` is not in Σ. Let Π' be the deduction rooted at the premise of this application, *i.e.*, of $[y/x]B'$ from $A_1, \ldots, A_n$. Note that $\Sigma(\Pi')$ is $\Sigma(\Pi) \cup \{\mathtt{Y} : \mathtt{i}\}$. Let $\Sigma' := \Sigma \cup \{\mathtt{Y} : \mathtt{i}\}$. By the induction hypothesis, there exists a Σ'-term `S` such that `(S #` $\langle\!\langle [y/x]B' \rangle\!\rangle$`)` is provable from $\Sigma'; \mathcal{P}$. We have

$$\langle\!\langle [y/x]B' \rangle\!\rangle = [\mathtt{Y}/\mathtt{X}]\langle\!\langle B' \rangle\!\rangle =_\beta (\mathtt{X}\backslash\langle\!\langle B' \rangle\!\rangle \ \mathtt{Y}).$$

Let `Q` be the Σ-term with bound variable `Y` and body `S`. The above goal can be rewritten as `((Q Y) # (X\`$\langle\!\langle B' \rangle\!\rangle$ `Y))`. Note that `Y` does not occur free in `Q` or in `X\`$\langle\!\langle B' \rangle\!\rangle$. By the GENERIC operation, `pi Y\((Q Y) # (X\`$\langle\!\langle B' \rangle\!\rangle$ `Y))` is provable from $\Sigma; \mathcal{P}$. Thus, by BACKCHAIN on the clause for $\forall$-I, `(forall_i Q) # (forall X\`$\langle\!\langle B' \rangle\!\rangle$`)` is provable from $\Sigma; \mathcal{P}$.

If the last step in the deduction is an application of $\supset$-I, then B has the form $B_1 \supset B_2$, and we know that B_2 is provable from $A_1, \ldots, A_n, B_1$. Let `P` be a variable of type `nprf` that does not occur in Σ. Let $\Sigma' := \Sigma \cup \{\mathtt{P} : \mathtt{nprf}\}$ and let $\mathcal{P}' := \mathcal{P} \cup \{\mathtt{P} \ \texttt{\#} \ \langle\!\langle B_1 \rangle\!\rangle\}$. By the induction hypothesis, there exists a Σ'-term `S` such that `(S #` $\langle\!\langle B_2 \rangle\!\rangle$`)` is provable from $\Sigma'; \mathcal{P}'$. By the AUGMENT operation, `(P #` $\langle\!\langle B_1 \rangle\!\rangle$`) => (S #` $\langle\!\langle B_2 \rangle\!\rangle$`)` is provable from $\Sigma'; \mathcal{P}$. Let `Q` be the Σ-term with bound variable `P` and body `S`. The above goal can be rewritten as `(P #` $\langle\!\langle B_1 \rangle\!\rangle$`) => ((Q P) #` $\langle\!\langle B_2 \rangle\!\rangle$`)`. Note that `P` does not occur free in `Q`, $\langle\!\langle B_1 \rangle\!\rangle$, or $\langle\!\langle B_2 \rangle\!\rangle$. By the GENERIC operation, `pi P\((P #` $\langle\!\langle B_1 \rangle\!\rangle$`) => ((Q P) #` $\langle\!\langle B_2 \rangle\!\rangle$`))` is provable from $\Sigma; \mathcal{P}$. Thus, by BACKCHAIN on the clause for $\supset$-I, `(imp_i Q) # (`$\langle\!\langle B_1 \rangle\!\rangle$ `imp` $\langle\!\langle B_2 \rangle\!\rangle$`)` is provable from $\Sigma; \mathcal{P}$.

The reverse direction (2) follows from the following more general statement: Let $\mathtt{B}, \mathtt{A}_1, \ldots, \mathtt{A}_n$ be terms of type `form` in $\mathcal{T}$ and let `R` be a term of type `nprf` in $\mathcal{T}'$. Let $\mathtt{P}_1, \ldots, \mathtt{P}_n$ be n distinct variables of type `nprf`. Let Σ be the signature containing Σ_{ND}, the variables $\mathtt{P}_1, \ldots, \mathtt{P}_n$, and possibly a finite number of variables of type `i` including at least all those that occur free in $\mathtt{B}, \mathtt{A}_1, \ldots, \mathtt{A}_n, \mathtt{R}$. Let $\mathcal{P}$ be the set of clauses $\mathcal{P}_{ND} \cup \{(\mathtt{P}_1 \ \texttt{\#} \ \mathtt{A}_1), \ldots, (\mathtt{P}_n \ \texttt{\#} \ \mathtt{A}_n)\}$. If `(R # B)` is provable from $\Sigma; \mathcal{P}$, then there is a deduction of $\|\mathtt{B}\|$ from assumptions $\{\|\mathtt{A}_1\|, \ldots, \|\mathtt{A}_n\|\}$.

The proof is by induction on the structure of R, and is similar to the proof of (1). It relies on the existence of proofs of the form described by Theorem 2.1. We start with a provable goal, and based on the structure of the goal, determine the last step that had to be taken by the interpreter described in that theorem. For an atomic goal, the constant at the head of the proof term determines which clause must be used in backchaining. ∎

Note that these proofs illustrate more than just correctness of the specification in Figure 2. They in fact illustrate a step-by-step correspondence between proofs in the object-language and proofs in the metalanguage. It is clear that each application of a rule at the object-level corresponds to a BACKCHAIN on a particular clause at the meta-level.

Proof terms can be considered as a means to record more precisely the correspondence between object- and meta-proofs. When proof terms contain enough information, it is possible to state and prove correctness results by defining an encoding and decoding between proof terms and object-level proof trees, and proceed by establishing the bijectivity of these functions. To do so for the natural deduction specification above would require a more precise formulation of deductions at the object-level and their correspondence to proof terms[2]. This is the approach taken by Harper et. al. [21] in the proof of Adequacy for Proofs for the LF specification of natural deduction. There, instead of natural deduction as defined by Prawitz [38], a notion of natural deduction that corresponds a bit more directly to the proof term representation is used. This approach is also taken by Gardner [15] where a bijection, called a *natural encoding*, is established between object-level proofs and their encoding as terms in the LF^+ type theory.

5 Inference Rules as Basic Tactics

In the remainder of this paper, we focus on a more general setting for proof search and construction: the implementation of tactic style theorem provers. Generally tactics and tacticals have been implemented in the functional programming language ML. We shall illustrate that their logic programming implementation is quite natural and extends the usual meaning of tacticals by permitting them to have access to logic variables and all six search operations. A comparison between the ML and λProlog implementations is contained in Section 8. In our setting, *primitive* tactics implement the basic inference rules of a particular proof system. A compact but powerful set of tacticals provides the basic control over search. They implement an interpreter on top of λProlog which must itself function well under depth-first search. They provide a mechanism for composing tactics in a principled manner, and can be viewed as a programming language for writing proof search strategies. Such proof strategies which are built up from primitive tactics and tacticals will be called *compound* tactics.

As an example, in this and the next two sections, we will implement a theorem prover for natural deduction for first-order logic. We begin in this section by modifying the specification of the natural deduction inference rules as discussed in Section 3, so that

[2]For example, a formulation of natural deduction that uses discharge functions is given in [38]. Such functions can be shown to correspond to abstractions from proofs to proofs in our definition of proof terms.

the clauses may now serve as the primitive search steps of the tactic theorem prover. We call the specification given here the *tactic* specification of inference rules as opposed to the *direct* specifications given there. In Section 6 we implement the theorem proving interpreter which includes an implementation of the basic tacticals, as well as capabilities for interactive theorem proving. This implementation of tacticals is generic in the sense that it is used without modification for any object-logic we may implement. Then, in Section 7 we complete the natural deduction theorem prover and give an example of its use.

First, we introduce a new primitive type `goal` for goal structures that will be manipulated by the new interpreter. These goals are distinct from logic programming goals of type o, which have a specific meaning given to them by the depth-first interpreter. Goals of type `goal` will only be given meaning by the new programs we write to manipulate them. We want to have each of the search operations of the metalanguage available to our interpreters, so we introduce one goal constructor corresponding to each and give them types as below.

```
type    tt      goal.
type    ff      goal.
type    &&      goal -> goal -> goal.
type    vv      goal -> goal -> goal.
type    all     (A -> goal) -> goal.
type    some    (A -> goal) -> goal.
type    ==>>    A -> goal -> goal.
```

Here, `tt` corresponds to the trivially satisfied goal, `ff` corresponds to failure, `&&` corresponds to the AND search operation, `vv` to OR, `all` to GENERIC, `some` to INSTANCE, and `==>>` to AUGMENT. Note that each of the goal constructors except `==>>` has a type similar to the corresponding logical connective of the metalanguage, where the type o is replaced by `goal` everywhere. The reason for the type of `==>>` will become apparent later. Note that the goal quantifiers `all` and `some` have polymorphic type. In general, for each theorem prover, quantification in goals will be limited to a small number of primitive types.

Primitive tactics specifying inference rules will be named facts where the name is a predicate of type `goal -> goal -> o`. The first argument is the input goal specifying the conclusion, and the second is the output goal specifying the premises. Basic goals will encode the relation between a formula and its proof, a sequent and its proof, a term and its type, etc., as they did in Section 3. We will call these goals *atomic goals* as opposed to *compound* goals built from the constructors above. (Note that atomic goals of type `goal` representing object-level relations are distinct from atomic goal formulas of the meta-logic of type o defined in Section 2.)

Any direct specification of inference rules can be converted to a set of tactics by a few minor syntactic changes. As an example, consider the specification of the $\wedge$-I rule of natural deduction in Figure 2. The following clause is the corresponding tactic.

```
and_i_tac  ((and_i P1 P2) # (A and B))  ((P1 # A) && (P2 # B)).
```

First, we provide a name for the tactic, in this case `and_i_tac`. Second, predicates used in the direct specification of inference rules become goal constructors for atomic goals. For the tactic specification of natural deduction, we again use the infix constant # to encode the relation between a formula and its proof, but in this case its target type is `goal`. Third, search connectives used in the direct specification must be replaced by the corresponding goal constructors defined above. In this tactic, the output goal is a conjunctive compound goal containing two atomic goals. The declarative reading of this clause is the same as in the direct specification. The operational reading, however, is similar but indirect since it depends on the fact that the goal structures `&&`, `vv`, etc., will be implemented in terms of their corresponding search connectives.

In Section 3 we showed that it was quite natural to specify the discharge of assumptions using universal quantification and implication at the meta-level. In interactive theorem proving, it may be desirable to have more direct control over the manipulation of assumptions. We can gain more explicit control of assumptions by storing them in a list and making the manipulation of these lists explicit in the definite clauses specifying the inference rules. To do so here, we will use lists of pairs of formulas associated with their proofs. We must make several modifications to incorporate such lists. We will again use the constant # for the relation between a formula and its proof, but now it will have target type `judg`, a new primitive type representing the basic judgment for natural deduction. We then use the sequent arrow `-->` to form "judgment sequents," of type `(list judg) -> judg -> goal`. A list of assumptions associated with their proofs appears on the left of the arrow, and the formula to be proved and its proof appear on the right. We will call such lists of pairs *contexts*.

To specify introduction rules that do not involve the discharge of assumptions, we simply add a list and sequent arrow to form a judgment sequent in the input and output goals of each tactic. For example, the tactic for $\wedge$-I becomes the following clause.

```
and_i_tac  (Gamma --> (and_i P1 P2) # (A and B))
           ((Gamma --> P1 # A) && (Gamma --> P2 # B)).
```

For readability, we assume that the infix operator # binds tighter than `-->`. The discharge of assumptions as in the $\supset$-I rule is specified as below where the new assumption gets added to the context rather than the program.

```
imp_i_tac  (Gamma --> (imp_i Q) # (A imp B))
           (all P\ (((P # A)::Gamma) --> (Q P) # B)).
```

The elimination rules can be specified similarly. For example, we may have the following two tactics for the $\wedge$-E rules.

```
and_e1_tac (Gamma --> (and_e1 B P) # A) (Gamma --> P # (A and B)).
and_e2_tac (Gamma --> (and_e2 A P) # B) (Gamma --> P # (A and B)).
```

Alternatively, we can specify elimination rules so that they are applied in a forward direction from the assumptions, another useful capability in interactive theorem proving. For example, we specify the $\wedge$-E rule as the following tactic.

```
and_e_tac N (Gamma --> PC # C)
            ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> PC # C) :-
   nth_item N (P # (A and B)) Gamma.
```

The integer argument provides the capability to choose a specific formula within the list to which the rule will be applied. The `nth_item` program was given in Section 2. This clause operates by finding a conjunction paired with its proof in position `N` in the context, applying both versions of the $\wedge$-E rule to it, and then expanding the context with the resulting new hypotheses, one for each conjunct. The attempt to find a proof `PC` of formula `C` then continues in the new context.

By similarly specifying the remaining natural deduction inference rules, we obtain the set of tactics in Figure 3. For the elimination rules, we include specifications that apply these rules in a forward direction since these are the tactics that will be used in the example execution in Section 7. We must also provide a tactic, `close_tac`, to complete proofs.

The specifications of proof systems in Section 3, on several occasions, made use of disjunctive, existential, and implicational goals in the bodies of clauses, which operationally correspond to the use of the OR, INSTANCE, and AUGMENT search operations, respectively. Note, on the other hand, that the disjunctive, existential, and implicational goal constructors are not used in Figure 3. In fact, although these three connectives are useful for specification, in general they are not essential. Specifications can always be modified to remove them.

The manner in which the rules are specified in Figure 3 is essentially the same as a specification given for natural deduction in Felty [12] which has the property that only deductions in "sharpened normal form" as defined by Prawitz [39] get built. We can state correctness theorems similar to those in Section 4 for this set of tactics. Such theorems have proofs similar to those in that section, except that they illustrate the correspondence between proof terms constructed by the program and deductions in normal form. Clearly, such a correctness theorem will play a large role in establishing the correctness of a tactic theorem prover. Full correctness will also depend on the correctness of the implementation of the interpreter described in the next section. Correctness of tactic theorem provers will be discussed further in Section 8.

For more flexibility, it may be desirable to include additional tactics for user-guided proof search. For example, it may be useful to remove an assumption when it is no longer needed. It is straightforward to do so when assumptions are stored as a list inside atomic goal structures. For example, we may want to include the following tactic for the $\wedge$-E rule in addition to the one in Figure 3.

```
and_e_rm N (Gamma --> PC # C)
           ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma1) --> PC # C) :-
   nth_and_rest N (P # (A and B)) Gamma Gamma1.
```

Such tactics can be useful in writing partially automated search strategies where we may only want to consider using each assumption once.

As another example, note that the tactic for $\exists$-I inserts a logic variable for the substitution term. In interactive proof, the user may want the flexibility to specify the substitution instance directly at the time the rule is invoked. This can be achieved with the following tactic.

```
exists_i_sbst (Gamma --> (exists_i T P) # (exists A)) (Gamma --> P # (A T)) :-
```

```
close_tac N (Gamma --> P # A) tt :- nth_item N (P # A) Gamma.
and_i_tac  (Gamma --> (and_i P1 P2) # (A and B))
           ((Gamma --> P1 # A) && (Gamma --> P2 # B)).
or_i1_tac  (Gamma --> (or_i1 P) # (A or B))  (Gamma --> P # A).
or_i2_tac  (Gamma --> (or_i2 P) # (A or B))  (Gamma --> P # B).
imp_i_tac  (Gamma --> (imp_i Q) # (A imp B))
           (all P\ (((P # A)::Gamma) --> (Q P) # B)).
neg_i_tac  (Gamma --> (neg_i Q) # (neg A))
           (all P\ (((P # A)::Gamma) --> (Q P) # false)).
forall_i_tac  (Gamma --> (forall_i Q) # (forall A))
              (all Y\ (Gamma --> (Q Y) # (A Y))).
exists_i_tac  (Gamma --> (exists_i T P) # (exists A)) (Gamma --> P # (A T)).
false_i_tac  (Gamma --> (false_i P) # A)  (Gamma --> P # false).
and_e_tac N (Gamma --> PC # C)
            ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> PC # C) :-
   nth_item N (P # (A and B)) Gamma.
imp_e_tac N (Gamma --> PC # C)
            ((Gamma --> P1 # A) &&
             ((((imp_e A P1 P2) # B)::Gamma) --> PC # C)) :-
   nth_item N (P2 # (A imp B)) Gamma.
neg_e_tac N (Gamma --> PC # C)
            ((Gamma --> P1 # A) &&
             ((((neg_e A P1 P2) # false)::Gamma) --> PC # C)) :-
   nth_item N (P2 # (neg A)) Gamma.
forall_e_tac N (Gamma --> PC # C)
               ((((forall_e T A P) # (A T))::Gamma) --> PC # C) :-
   nth_item N (P # (forall A)) Gamma.
or_e_tac N (Gamma --> (or_e A B P Q1 Q2) # C)
           ((all P1\ (((P1 # A)::Gamma) --> (Q1 P1) # C)) &&
            (all P2\ (((P2 # B)::Gamma) --> (Q2 P2) # C))) :-
   nth_item N (P # (A or B)) Gamma.
exists_e_tac N (Gamma --> (exists_e A P1 Q) # B)
               (all Y\ (all P\ (((P # (A Y))::Gamma) --> (Q Y P) # B))) :-
   nth_item N (P1 # (exists A)) Gamma.
```

Figure 3: Tactics for Natural Deduction

```
   write "Enter substitution term:", read X\ (T = X).
```

A user may enter a partially or fully instantiated term for T.

As a final example, in interactive theorem proving, the introduction and use of lemmas is quite useful, if not essential. Consider the following modus_ponens tactic for natural deduction.

```
modus_ponens  (Gamma --> P # A)
              ((Gamma --> Q # B) && (((Q # B)::Gamma) --> P # A)) :-
   write "Enter lemma: ", read X\ (B = X).
```

It allows the user to add a hypothesis as a lemma, prove it, and then use it in proving the original theorem.

6 Implementing a Tactic Interpreter

The core of the tactic interpreter is implemented by a small set of tacticals which define some basic control mechanisms. These tacticals generally take one or more tactics as arguments and compose them in various ways. We first present this basic set of tacticals, then implement some tactics and tacticals that are useful for interactive proof search, and finally, discuss the implementation of tactics which define general proof search strategies.

For clarity in displaying types in this and the next section, we will write `tactic` to abbreviate the type `(goal -> goal -> o)`[3]. First, the `maptac` tactical in Figure 4 applies tactics to compound goals. It takes a tactic as an argument and applies it to the input goal in a manner consistent with the meaning of the goal structure. In the clause implementing `==>>`, the polymorphic predicate `memo` allows the introduction of new clauses containing information of arbitrary type into the program. The last clause is used once the goal is reduced to an atomic form. It simply applies the tactic directly.

Six common tacticals are implemented by the clauses in Figure 5. The `then` tactical performs the composition of tactics. `Tac1` is applied to the input goal, and then `Tac2` is applied to the resulting goal. In this tactical and all others, we assume that the input goal is atomic. The `maptac` program is used in the second subgoal since the application of `Tac1` may result in an output goal (`MidGoal`) with compound structure. This tactical plays a fundamental role in combining the results of step-by-step proof construction. The substitutions resulting from applying these separate tactics get combined correctly since `MidGoal` provides the necessary sharing of logic variables between the two calls to tactics. 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 first of the two logic programming subgoals), then `Tac2` is tried. The third tactical, `idtac`, returns the input goal unchanged. This tactical is useful in constructing compound tactic expressions such as the one found in the `repeat` tactical. The `repeat` tactical is recursively defined using the three tacticals, `then`, `orelse`, and `idtac`. It repeatedly applies a tactic until it can no longer be applied. The `try` tactical prevents failure of the given argument tactic

[3] λProlog, however, does not allow such type abbreviations.

```
type    maptac  tactic -> tactic.
type    memo    A -> o.

maptac Tac tt tt.

maptac Tac (InGoal1 && InGoal2) (OutGoal1 && OutGoal2) :-
  maptac Tac InGoal1 OutGoal1, maptac Tac InGoal2 OutGoal2.

maptac Tac (all InGoal) (all OutGoal) :-
  pi T\ (maptac Tac (InGoal T) (OutGoal T)).

maptac Tac (InGoal1 vv InGoal2) OutGoal :-
  maptac Tac InGoal1 OutGoal; maptac Tac InGoal2 OutGoal.

maptac Tac (some InGoal) OutGoal :-
  sigma T\ (maptac Tac (InGoal T) OutGoal).

maptac Tac (D ==>> InGoal) (D ==>> OutGoal) :-
  (memo D) => (maptac Tac InGoal OutGoal).

maptac Tac InGoal OutGoal :- Tac InGoal OutGoal.
```

Figure 4: Interpreting Compound Goal Structures

```
type    then            tactic -> tactic -> tactic.
type    orelse          tactic -> tactic -> tactic.
type    idtac           tactic.
type    repeat          tactic -> tactic.
type    try             tactic -> tactic.
type    complete        tactic -> tactic.

then Tac1 Tac2 InGoal OutGoal :- Tac1 InGoal MidGoal,
                                 maptac Tac2 MidGoal OutGoal.

orelse Tac1 Tac2 InGoal OutGoal :- Tac1 InGoal OutGoal; Tac2 InGoal OutGoal.

idtac Goal Goal.

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

try Tac InGoal OutGoal :- orelse Tac idtac InGoal OutGoal.

complete Tac InGoal tt :- Tac InGoal OutGoal, goalreduce OutGoal tt.
```

Figure 5: Some Common Tacticals

by using `idtac` when `Tac` fails. 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 completely solve the given goal. It will fail if there is a non-trivial goal remaining after `Tac` is applied. It requires an auxiliary procedure `goalreduce` to simplify compound goal expressions by removing occurrences of `tt` from them. The `complete` tactical succeeds only if the output goal is simplified to `tt`.

The tactics and tacticals in Figure 6 provide an implementation of a simple interactive component. The first tactical provides an alternative implementation of the `orelse` tactical. In the previous version, if `Tac1` succeeds, a backtracking point will be set up so that if there is a subsequent failure, control may return to this clause to find other ways to apply `Tac1`, or if there are none, to attempt `Tac2`. The `orelse!` tactical eliminates this backtracking point by introducing a cut. If `Tac1` succeeds once, no other attempts will be made to apply `Tac1` or `Tac2`. This new version relies on a non-logical feature of the metalanguage to obtain the desired operational behavior. There are a few other occasions where the use of cut is crucial in defining operational behavior in clauses in this section since more fine-tuned control is important for good user interaction.

The `orelse!` tactical is used by the `query` tactic, which is the primitive operation of the interactive component. The task of this tactic can be divided into three steps. The first step is to output some information about the state of the interpreter. The second step is to get input from the user about what action to take, and the third step is to perform the action specified by the input. The first two steps are handled by the first subgoal of the `query` tactic. The predicate `IO` is a parameter to this tactic since the actual procedure for input and output will depend on the particular object-language for which we are building a theorem prover. The `basic_io` and `read_tac` clauses provide simple operations that can be used in building a specialized input/output procedure. For example, in a natural deduction theorem prover, if `ndoutput` is the name of a procedure to print out the state of a natural deduction theorem prover, the `IO` argument to `query` may be `(basic_io ndoutput readtac)`.

There are then three options in applying the tactic `Tac` input by the user, given by the three disjuncts in the `query` tactic. In the first disjunct, notice the use of cut (`!`) and `fail`. One requirement of a good interactive system is to provide the user with some capability to backup the search to previous points. In this implementation, the user will be allowed to incrementally backup the search one step at a time, by invoking the `backup` tactic. Here `backup` is implemented by causing the logic programming interpreter to fail to a previous point. The use of cut here insures that the other two disjuncts will not be attempted. Implementing `backup` using failure has both advantages and disadvantages. The main advantage is that all information about the previous state is handled automatically by the logic programming interpreter. We need not introduce extra data structures or implement additional control mechanisms to keep track of this information. The main disadvantage of this approach is that extreme care must be taken to strategically place cuts in all of the clauses that make up the interactive component so that invoking the `backup` tactic takes the interpreter to the desired backtracking point, the one corresponding to the last invocation of the `query` tactic.

```
type    orelse!         tactic -> tactic -> tactic.
type    query           (goal -> tactic -> o) -> tactic.
type    basic_io        (goal -> o) -> (tactic -> o) -> goal -> tactic -> o.
type    readtac         tactic -> o.
type    backup          tactic.
type    report_fail     tactic.
type    quit            tactic.
type    inter_repeat    tactic -> tactic.
type    inter           (goal -> tactic -> o) -> tactic.
type    with_tacs       modul -> tactic -> tactic.

orelse! Tac1 Tac2 InGoal OutGoal :- Tac1 InGoal OutGoal,!; Tac2 InGoal OutGoal.

query IO InGoal OutGoal :-
   IO InGoal Tac,
   ((Tac = backup), !, fail;
    orelse! Tac report_fail InGoal OutGoal;
    query IO InGoal OutGoal).

basic_io PrintPred ReadPred Goal Tac :- PrintPred Goal, ReadPred Tac.

readtac Tac :- writesans "Enter tactic: ", read X\ (Tac = X).

report_fail Goal Goal :- writesans "Tactic failed.", nl.

quit InGoal ff.

inter_repeat Tac InGoal OutGoal :-
   Tac InGoal MidGoal,
   ((MidGoal = ff), !, (OutGoal = InGoal);
   maptac (inter_repeat Tac) MidGoal OutGoal).

inter IO InGoal OutGoal :- inter_repeat (query IO) InGoal OutGoal.

with_tacs M Tac InGoal OutGoal :- M ==> (Tac InGoal OutGoal).
```

Figure 6: Interactive Component for Tactic Interpreter

If a backup is not requested by the user, the second disjunct attempts to apply the requested tactic. It either applies the tactic successfully, or reports failure when the tactic fails. The orelse! tactical is used inside query since if the tactic succeeds, we do not want the interpreter to be able to later report failure. The third disjunct is best understood in the context of an interactive loop which repeatedly calls the query tactic. A simple loop might be defined simply as the following tactic.

```
inter IO InGoal OutGoal :- repeat (query IO) InGoal OutGoal.
```

At a particular invocation of query, if a backup is requested by the user, control will return to the previous invocation of query. At this point, we do not want to fail further, but instead return the search to the state it was in upon entering this invocation of query. In order to achieve this behavior, the third disjunct of query makes a recursive call to itself.

Since the repeat tactical loops until the tactic fails, and since the query tactic only fails when the backup tactic is invoked, the above implementation of inter will terminate in one of two ways. It will fail if backed up all the way to the beginning, or will succeed when the input goal is completely solved by the user. In the latter case, the first clause of maptac terminates each branch of the search as the input goals are reduced to tt. A good interactive interpreter must also provide the user with the capability to stop the search without losing the work done so far, or to stop particular branches of search in favor of pursuing others. For this task we define the quit tactic, and the inter_repeat looping tactical given in Figure 6. This tactical terminates search when the quit tactic is invoked and returns the current input goal as the output goal. The inter tactic in Figure 6 uses inter_repeat and thus provides a top-level interactive loop that allows both backing up and stopping search branches.

For a particular theorem prover, as the set of existing theorems and specialized tactics grows, it may be desirable to organize them into modules containing sets of related tactics and theorems. We can provide the user with the flexibility to access modules as they are needed at different points during proof construction. The with_tacs tactical in Figure 6 allows the user to dynamically extend the current theorem proving environment. The type of the first argument M is modul, the meta-level primitive type for λProlog module names. The ==> symbol is the meta-level connective that instructs the interpreter to load the module M into memory and add all of the clauses in M to the current program. The tactic Tac is applied in the new environment. Tac may be, for example, a new call to a top-level interactive loop. If execution continues after successful completion or failure of this tactic, the clauses of M will no longer be available unless explicitly added again. Such a tactical may also be used to extend the definition of existing tactics. In particular, M may contain clauses with the same name as a tactic in the current environment. In the new environment, the new clauses get attempted before the existing ones when this tactic is invoked.

Using the data structures for goals that we have defined and the maptac program for traversing goal structures, several general search strategies can be implemented as tactics. For example, the program below implements depth-first search.

```
type    app_lis_tac     list tactic -> tactic.
```

```
type    dfs             list tactic -> tactic.

app_lis_tac (Tac::Rest) InGoal OutGoal :- Tac InGoal OutGoal.
app_lis_tac (Tac::Rest) InGoal OutGoal :- app_lis_tac Rest InGoal OutGoal.

dfs Tacs InGoal OutGoal :- app_lis_tac Tacs InGoal MidGoal,
                           maptac (dfs Tacs) MidGoal OutGoal.
```

Such a strategy may be useful in domains where depth-first search may be sufficient for proving various simple subproofs.

Using this strategy, if all subgoals are reduced to `tt`, the top-level call to `dfs` terminates with success. `OutGoal` will then be a compound goal structure containing only `tt` as its atomic subgoals. Otherwise depth-first search fails. Since the `dfs` tactic will either completely solve the input goal or loop indefinitely, the output goal will contain no useful information. Alternatively, we could modify this tactic so that it applies as many tactics from `Tacs` as possible until no more can be done, and then returns the unfinished subgoals to be solved by the user in some other manner. This can be achieved by simply adding the following clause to the end of the definition of `app_lis_tac`.

```
app_lis_tac Tacs Goal Goal.
```

It is straightforward to implement several other general search strategies within this framework. For example breadth-first search, a complete search strategy, can be implemented. For an implementation of depth-first iterative deepening as defined by Korf [25], a variant of depth-first search that is also a complete search strategy, see Felty [10].

7 A Tactic Theorem Prover for Natural Deduction

We have presented a general tactic interpreter with an interactive component as well as a set of primitive tactics for proof search in natural deduction. We can continue to build on this structure adding new tactics and strategies for natural deduction. We illustrate here with a few simple examples. Depending on the logic or theory being implemented, a user will want to provide more sophisticated automation tactics specialized to that logic. For example, term rewriting tactics can be useful in logics that have a notion of equality between terms. For more on how both general and specific rewriting tactics can be implemented in this setting, see Felty [13].

The following tactics implement an interactive loop and a simple tactic that repeatedly applies some of the introduction rules, respectively.

```
type    inter_nd    form -> nprf -> goal -> o.
type    intros_tac  tactic.

inter_nd A P OutGoal :-
  inter (basic_io ndoutput readtac) (nil --> P # A) OutGoal.

intros_tac (Gamma --> J) OutGoal :-
  repeat (orelse! and_i_tac (orelse! imp_i_tac
          (orelse! neg_i_tac forall_i_tac))) (Gamma --> J) OutGoal.
```

We assume here that ndoutput is a procedure to print out the current list of assumptions and formula to be proved in a natural deduction goal. To execute this interactive loop, the user provides a formula A, and after an interactive session, P will be instantiated to the proof or partial proof and OutGoal will contain the subgoals that remain to be proved.

In the same setting, we can build a complete proof checking tactic in one of many ways. For instance, we could use dfs with a list containing tactics for all of the inference rules of natural deduction. If we use the specification of tactics that is a direct modification of the clauses in Figure 2 so that elimination as well as introduction rules are applied in a backward direction, we obtain a tactic with the same operational behavior as the clauses in the figure. Such a tactic would not be useful for theorem proving in general, but illustrates how the tactic setting provides a uniform framework for both theorem proving and proof checking.

The following is a simple example session with the tactic theorem prover for natural deduction where the formula $q(a) \vee q(b) \supset \exists x q(x)$ is proved. Notice that a bad attempt to prove this formula is backed out of before the right solution is found.

```
?- inter_nd (((q a) or (q b)) imp (exists X\(q X))) Proof OutGoal.

Assumptions:

Conclusion:
((q a) or (q b)) imp (exists X\(q X))
Enter tactic:  ?- imp_i_tac.

Assumptions:
1 (q a) or (q b)

Conclusion:
exists X\(q X)
Enter tactic:  ?- exists_i_tac.

Assumptions:
1 (q a) or (q b)

Conclusion:
q T
Enter tactic:  ?- or_e_tac 1.

Assumptions:
1 q a
2 (q a) or (q b)

Conclusion:
q T
Enter tactic:  ?- close_tac 0.

Assumptions:
```

```
1 q b
2 (q a) or (q b)

Conclusion:
q a
Enter tactic:  ?- backup.

Assumptions:
1 q a
2 (q a) or (q b)

Conclusion:
q T
Enter tactic:  ?- backup.

Assumptions:
1 (q a) or (q b)

Conclusion:
q T
Enter tactic:  ?- backup.

Assumptions:
1 (q a) or (q b)

Conclusion:
exists X\(q X)
Enter tactic:  ?- then (or_e_tac 1) (then exists_i_tac (close_tac 0)).

Proof = (imp_i P\(or_e (q a) (q b) P (P1\(exists_i a P1)) (P2\(exists_i b P2)))
OutGoal = (all P\((all P1\tt) && (all P2\tt))).
```

The application of the `exists_i_tac` tactic introduces a logic variable `T` for the substitution term. Then, when `(close_tac 0)` is applied, `T` is instantiated to `a`, the proof branch is completed, and this unifier is carried over to the second branch of the proof. Since this branch cannot be completed, the user backs the proof up to the point where it can be corrected. The final compound expression that completes the proof first applies `or_e_tac` causing the search to branch, and then applies `exists_i_tac` followed by `(close_tac 0)` to each of the branches. Thus a new logic variable is introduced in each branch separately. The first is instantiated to `a` and the second to `b`, allowing the proof to be completed. Thus `Proof` is instantiated to a complete proof, and `OutGoal` is a compound goal expression whose atomic subgoals are all instances of `tt`.

8 Related Work

The programming language ML is the metalanguage used in all of the other tactic theorem provers mentioned earlier. There are several differences in the implementations of both tactics and tacticals in these two languages. First, in ML, tactics are functions that take a goal as input and return a list of subgoals, and in some cases (such as LCF) also return a validation. In contrast, tactics in λProlog are relational, which is natural when the relation being modeled is "is a proof of." Although, as we noted earlier, the direct specification of natural deduction in Figure 2 can only be used for proof checking, tactics can be used for both theorem proving and proof checking. The functional aspects of ML do not permit input and output distinctions to be blurred in this manner.

As stated in Gordon et. al. [17], tacticals encourage the programming of valid tactics. In fact, as long as the tacticals are implemented correctly, all compound tactics built from valid tactics will also be valid. This fact holds for both the ML and logic programming settings.

In LCF, the basic inference rules for a particular logic are implemented as functions taking instances of the premises of the rule to an instance of the conclusion. The input arguments are required to be theorems (type `thm`), and thus the result is also a theorem. When writing tactics for backward proof search, these functions are the building blocks used by the programmer to construct validations. The use of validations in tactics in this way provides an extra level of security. The programmer has complete freedom to write tactics without being concerned with their validity. After successful completion of backward search for the proof of a particular formula, the resulting validation must be executed. If invalid tactics were used, the validation will fail, and the formula will not be added as a theorem. Of course the functions implementing the primitive rules in the forward direction must be sound. As long as this is the case, only formulas that are truly theorems will be recorded as such.

The ML notion of validations is replaced in our system by (potentially much larger and more complex) proof objects. If we also give the programmer complete freedom to write tactics that construct such objects, we must insure that the terms constructed during proof search correspond to actual proofs. To do so, we can simply use the core set of tactics implementing the basic inference rules for the purpose of proof checking. We need not implement additional code for this task. We can even require that such a check is performed before accepting a formula as a theorem. In contrast to ML, such security must be provided at the program level rather than by the type system. While in ML, it is the functions implementing the primitive rules in the forward direction that must be sound, here it is the basic set of tactics that must be implemented correctly. The declarative nature of tactics in the logic programming setting makes it straightforward to prove their validity formally as was illustrated by Theorem 4.1.

The implementation of the `then` tactical in λProlog is quite different from its ML counterpart. The λProlog implementation of `then` reveals its very simple nature: `then` is very similar to the natural join of two relations. In ML, the `then` tactical applies the first tactic to the input goal and then maps the application of the second tactic over the list of intermediate subgoals. The full list of subgoals must be built as well as the compound validation function from the results. These tasks can be quite complicated, requiring some auxiliary list processing functions. In λProlog, the composition of tactics is handled correctly by the sharing of logic variables between the two calls to tactics. The analogue of a list of subgoals is a nested `&&` structure. These are processed by the clause of `maptac` which handles `&&`. The `maptac` procedure is richer than the usual notion of a mapping function in that, in addition to nested `&&` structures, it handles all of the other goal structures. The `all` goal structure, for example, provides a principled way in which to descend through abstractions in formulas and proofs as illustrated by its uses in the primitive tactics for natural deduction in Section 5.

In the λProlog implementation of `then` that we presented in Section 6, if the first tactic succeeds and the second fails, the logic programming interpreter will backtrack and try to find a new way to successfully apply the first tactic, exhausting all possibilities before completely failing. For example, if there are several possible instantiations of the substitution term used to instantiate the existential quantifier in `exists_i_tac` in Figure 3, they will all be attempted before failure occurs. As another example, if the first tactic is `orelse` applied to two arguments, all possible ways in which either of these arguments can succeed will be examined before failure occurs. It is also possible to implement `then` so that if the second tactic fails after a successful call to the first tactic, the full tactic still fails. To do so requires the use of cut (`!`) after the first subgoal to restrict its backtracking behavior.

Another difference in the ML and logic programming approaches is in the manipulation of quantified formulas. In ML, first-order syntax is used and thus manipulating quantified formulas requires that the binding be separated from its body. In logic programming, we use higher-order syntax. We identify a term as a universal quantification if it can be unified with the term `(forall A)`. However, since terms in λProlog represent $\beta\eta$-equivalence classes of λ-terms, the programmer does not have access to bound variable names. Although such a restriction may appear to limit access to the structure of λ-terms, we have seen that sophisticated analysis of λ-terms is still possible to perform using higher-order unification and the universal quantifier `pi`. In addition, there are certain advantages to such a restriction. For example, in the case of applying substitutions, all the renaming of bound variables is handled by the metalanguage, freeing the programmer from such concerns. The programmer may find it desirable to have more control over the names of variables in the printed form of a λ-term. Current implementations do not allow this, although, whenever possible, names are generated based on the names given by the user either in the original input or in program clauses, using a numbering scheme to avoid name clashes.

In the λProlog setting, we make use of logic variables for lazy determination of substitution instances. The example in Section 7 illustrated how such variables can be used so that substitution instances do not have to be given at the point where the substitution takes place.

The Isabelle theorem prover [35] contains a specification language based on a fragment of higher-order logic with implication and universal quantification that is essentially a subset of higher-order hereditary Harrop formulas. This language is used to specify inference rules for various object-logics. In fact, if we drop proofs from the specification of natural deduction in Figure 2, the resulting specification is similar to the one given in [35]. In addition, all the object-logics we have specified here could be very similarly specified in Isabelle.

The proof theory of the meta-logic of Isabelle is given in terms of natural deduction. In [35], a proof of correctness of the specification of natural deduction as an object-logic is given, illustrating the correspondence between object-level proofs and meta-level natural deduction proofs. The existence of proofs in the meta-logic in the form described by Theorem 2.1 (uniform proofs) corresponds in [35] to the existence of expanded normal form proofs in natural deduction [39].

In this paper, we have used the same metalanguage for specification of inference rules and for implementation of search. In Isabelle, ML is used to implement tacticals and specify tactics. As a result, there is a more significant difference operationally in the two approaches. It seems very likely that Isabelle could be rather directly implemented inside λProlog. Although such an implementation might achieve the same functionality as is currently available in Isabelle, it is not likely to be nearly as efficient. This is due partly to the fact that a λProlog implementation implements a general purpose programming language. The development of more efficient implementations of λProlog is currently underway [1, 32, 9]. Another approach is to modify λProlog's depth-first interpreter to use a different control strategy. For example, tacticals and interactive tactics could be implemented at the level of the metalanguage. This approach to control is more like that found in Isabelle.

Universal quantification and implication are used in the same manner here and in Isabelle to specify eigenvariable conditions and the discharge of assumptions. Operationally, although both systems are similar in their use of goal directed proof search, it is worth noting that the mechanisms to handle these two constructs differ. In λProlog, eigenvariables are handled by the introduction of new constants by the GENERIC operation. When backchaining on the clause for the $\forall$-I rule of natural deduction, for example, the universal quantifier is "stripped off" the goal and a new constant is introduced to replace the bound variable and allow us to descend through the λ-abstraction in the formula. This constant may appear in instances of clauses used in subsequent backchaining steps. Neither backchaining nor the GENERIC operation are present in Isabelle. There, a technique called *lifting* [35] is used. When applying the $\forall$-I rule, the object-level universal quantification is replaced by a meta-level quantifier exactly as in λProlog, but this quantifier is not stripped off. Instead, the form of clauses used in further proof steps is modified to take into account the universal quantifier in the goal. For example, to apply the following $\vee$-I rule:

```
provable (A or B) :- provable A.
```

Isabelle would first modify this formula to obtain the following formula.

```
provable ((A X) or (B X)) :- pi X\ (provable (A X)).
```

(Recall that there is implicit universal quantification at the top level over the variables `A`, `B`, and the free occurrences of `X`.) Using this formula, a goal of the form `pi X\(provable`

`((A X) or (B X)))` would be replaced by the subgoal `pi X\(provable (A X))`. A similar lifting operation is used to handle implication in goals. Although the mechanism is different, the behavior is quite similar to the use of AUGMENT in λProlog.

Both the theorem provers developed here and the Isabelle system adopt an intuitionistic logic with quantification over the simply typed λ-terms as a metalanguage for specifying inference rules. Various forms of typed lambda calculi with dependent types have also been proposed as specification languages for representing a wide variety of logics. Examples include the AUTOMATH languages [5], type theories developed by Martin-Löf [26], the Logical Framework (LF) [21], LF^{+} [15], and the Calculus of Constructions [4]. In [11], we show that LF signatures can be encoded directly and naturally as formulas in the subset of hohh that does not allow predicate quantification. This encoding demonstrates a close correspondence between the two approaches. In addition, an encoded signature can serve as a set of tactics providing a direct implementation of a simple tactic theorem prover for the object-logic.

Pfenning [36] adopts LF as the logical foundation of the higher-order logic programming language Elf. A non-deterministic interpreter can be described for Elf in much the same way as for hohh by providing a small set of search operations which, in this case, give an operational interpretation to types. To implement this language, a more complex unification procedure is required to handle dependent types [7]. Proof checkers and theorem provers similar to those presented here can also be implemented in this language. In such implementations, the Elf interpreter will construct LF terms corresponding to object-level proofs, and thus explicit proof terms need not be included in the programs. If proof terms other than those of the form constructed by the interpreter are desired, as is often the case, programs to transform proofs must be written. In many cases, partial correctness of such proof transformers (as well as many other programs) can be guaranteed by Elf. The operational behavior of proof checking and theorem proving programs under the two kinds of interpreters is also different. In Elf, the type checker for dependent types can handle some of the work that must be performed by logic programming search in the corresponding λProlog programs.

Although the programs in this paper make extensive use of many of the higher-order features of the metalanguage, such features are used in a fairly limited way. For example, quantification over both functions and predicates has been restricted to types of order at most 2. Operationally, the unification problems that arise in executing these programs are all fairly simple. In fact, with minor modification, most of the programs presented here fall within the L_λ sublanguage of hohh described by Miller [27]. The most significant modification required is to eliminate uses of application of terms at the meta-level to perform object-level substitution such as those found in the specification of the $\exists$-I and $\forall$-E rules of natural deduction. Instead, an explicit implementation of substitution as described in [27] must be used. In this language, quantification over predicates is not allowed and quantification over function variables is greatly restricted. As a result, unification for this language is very simple; it is decidable and most general unifiers always exist. An efficient implementation of L_λ should contribute significantly to the efficiency of our programs.

In this paper we have shown how various features and techniques of higher-order logic programming are useful for the specific task of manipulating formulas and proofs. λProlog and related metalanguages have also been successfully applied to several other meta-programming tasks. Other applications that have been explored include program manipulation [20], natural language processing [34], and generalization [6, 19].

Acknowledgements

The author would like to thank Robert Constable, Elsa Gunter, John Hannan, Dale Miller, and Frank Pfenning for valuable discussions and comments, as well as the anonymous reviewers for many very helpful suggestions. This research was supported in part by grants ARO-DAA29-84-9-0027, ONR N00014-88-K-0633, NSF CCR-87-05596, DARPA N00014-85-K-0018, and ESPRIT Basic Research Action 3245.

References

[1] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of λProlog: Prolog/mali. In Dale Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, August 1992. University of Pennsylvania, Technical Report MS-CIS-92-86.

[2] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

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

[4] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[5] N.G. deBruijn. A survey of the project AUTOMATH. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 579–606. Academic Press, 1980.

[6] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. In Alberto Maria Segre, editor, *Sixth International Workshop on Machine Learning*, pages 447–449. Morgan Kaufmann, 1989.

[7] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, pages 121–136. Springer-Verlag Lecture Notes in Computer Science, April 1989.

[8] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Feb 1990.

[9] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[10] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, Technical Report MS-CIS-89-53, August 1989.

[11] Amy Felty. Encoding dependent types in an intuitionistic logic. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 215–251. Cambridge University Press, 1991.

[12] Amy Felty. A logic program for transforming sequent proofs to natural deduction proofs. In Peter Schroeder-Heister, editor, *Proceedings of the First International Workshop on Extensions of Logic Programming*, pages 157–178. Springer-Verlag Lecture Notes in Artificial Intelligence, 1991.

[13] Amy Felty. A logic programming approach to implementing higher-order term rewriting. In Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 135–161. Springer-Verlag Lecture Notes in Artificial Intelligence, 1992.

[14] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pages 61–80. Springer-Verlag Lecture Notes in Computer Science, May 1988.

[15] Philippa Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, Technical Report CST-93-92, July 1992.

[16] 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.

[17] 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.

[18] Mike Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, July 1985.

[19] Masami Hagiya. Programming by example and proving by example using higher-order unification. In *Tenth International Conference on Automated Deduction*, pages 588–602. Springer-Verlag Lecture Notes in Artificial Intelligence, July 1990.

[20] John Hannan and Dale Miller. A meta language for functional programs. In H. Abramson and M. Rogers, editors, *Meta Programming in Logic Programming*, chapter 24, pages 453–476. MIT Press, 1989.

[21] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[22] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.

[23] William 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, 1980.

[24] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[25] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

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

[27] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[28] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[29] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming*, pages 379–388, September 1987.

[30] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[31] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming*. PhD thesis, University of Pennsylvania, Technical Report MS-CIS-87-48, June 1987.

[32] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM model for λProlog. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1180–1198, October 1989.

[33] Gopalan Nadathur and Dale Miller. Higher-order horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[34] Remo Pareschi and Dale Miller. Extending definite clause grammars with scoping constructs. In D. H. D. Warren and P. Szeredi, editors, *International Conference in Logic Programming*, pages 373–389. MIT Press, June 1990.

[35] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[36] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[37] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

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

[39] Dag Prawitz. Ideas and results in proof theory. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 235–307. North-Holland, 1971.

[40] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge MA, 1986.