Specifying Theorem Provewrs In A Higher-Order Logic Programming Language

MS-CIS-88-12
LINC LAB 99

Amy Felty
DaleMiller

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

February 1988

Acknowledgements:
This research was supported in party by DARPA grant N00014-85-K-0018, NSF grants CCR-87-05596, MCS-8219196-CER and U.S. Army Research Office grants DAA29-84-K-0061, DAA29-84-9-0027

Specifying Theorem Provers in a Higher-Order Logic Programming Language

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

Abstract

Since logic programming systems directly implement search and unification and since these operations are essential for the implementation of most theorem provers, logic programming languages should make ideal implementation languages for theorem provers. We shall argue that this is indeed the case if the logic programming language is extended in several ways. We present an extended logic programming language where first-order terms are replaced with simply-typed λ-terms, higher-order unification replaces first-order unification, and implication and universal quantification are allowed in queries and the bodies of clauses. This language naturally specifies inference rules for various proof systems. The primitive search operations required to search for proofs generally have very simple implementations using the logical connectives of this extended logic programming language. Higher-order unification, which provides sophisticated pattern matching on formulas and proofs, can be used to determine when and at what instance an inference rule can be employed in the search for a proof. Tactics and tacticals, which provide a framework for high-level control over search, can also be directly implemented in this extended language. The theorem provers presented in this paper have been implemented in the higher-order logic programming language λProlog.

1 Introduction

Logic programming languages have many characteristics that indicate that they should serve as good implementation languages for theorem provers. First, at the foundation of computation in logic programming is search, and search is also fundamental to theorem proving. The process of discovering a proof involves traversing an often very large and complex search space in some controlled manner. Second, unification, which is required in most theorem provers, is immediately and elegantly accessible in most logic programming systems. Third, if in fact theorem provers can be written directly in logic programming, the simple and declarative reading of such logic programs should help in understanding formal properties, such as completeness and soundness, of the resulting implementation. This potential advantage is very important for theorem prover implementations not only because such formal properties are important considerations but also because such implementations are often very complex and hard to understand.

Traditional logic programming languages such as Prolog [SS86] are not sufficient for providing natural implementations of several aspects of theorem provers. One deficiency, as

argued in [MN87b], is that first-order terms are quite inadequate for a clean representation of formulas. For instance, first-order terms provide no mechanism for representing variable abstraction required for quantification in first-order formulas. Of course, quantification could be specially encoded. For example, in Prolog, we can represent abstractions in formulas by representing bound variables as either constants or logical variables. The formula $\forall x \exists y\ P(x,y)$ could be written as the first-order term `forall(x,exists(y,p(x,y)))`. This kind of encoding is very unnatural and spoils the elegance which logic programming often offers. We shall mention other similar shortcomings of conventional Prolog systems.

In this paper, we use a higher-order logic programming language based on *higher-order hereditary Harrop formulas* [MNS87]. This language replaces first-order terms with simply typed λ-terms. The abstractions built into λ-terms can be used to naturally represent quantification. Our extended language also permits queries and the bodies of clauses to be both implications and universally quantified. We shall show how such queries are, in fact, necessary for implementing various kinds of theorem provers. The programs that we present in this paper have been tested using λProlog which is a partial implementation of higher-order hereditary Harrop formulas [MN87a]. Various aspects of this language have been discussed in [MN86a, MN86b, MN87b, Nad86].

Our main claim in this paper is that such a language is a very suitable environment for implementing theorem provers. We will show that search and unification accommodate the tasks involved in theorem proving very naturally. Most of our theorem provers will have a clean declarative reading which provides them with implementation independent semantics and makes establishing their formal properties more tractable.

In the next section, we will briefly present higher-order hereditary Harrop formulas. In Section 3, we discuss how to directly specify various inference rules as such formulas. In Sections 4 and 5 we discuss the implementation of tactic style theorem provers which allow greater control in searching for proofs and provide means for user participation in the theorem proving process. In Section 4 we show how to implement high-level tacticals, and in Section 5 we illustrate the tactic specification of inference rules for a particular proof system. Finally, in Section 6, we discuss related work.

2 Extended Logic Programs

Higher-order hereditary Harrop formulas extend positive Horn clause 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, disjunctions, and universally and existentially quantified formulas, as well as conjunction. The addition of disjunctions and existential quantifiers into the bodies of clauses does not depart much from the usual presentation of Horn clauses since such extended clauses are classically equivalent to Horn clauses. The addition of implications and universal quantifiers, however, makes a significant departure. 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 λ-conversion, while the unification of λ-terms is handled by higher-order unification.

There are four major components to our extended logic programming language: types, λ-terms, definite clauses, and goal formulas. Types and terms are essentially those of the simple theory of types [Chu40]. We assume that a certain set of non-functional (primitive) types is provided. This set must contain the type symbol o which will denote the type of logic programming propositions: other primitive types are supplied by the programmer. Function types of all orders are also permitted: if α and β are types then so is $\alpha \to \beta$. The arrow type constructor associates to the right: read $\alpha \to \beta \to \gamma$ as $\alpha \to (\beta \to \gamma)$. A function symbol whose target type is o will also be considered a predicate symbol. No abstractions or quantifications are permitted in types (hence the adjective "simple").

Simply typed λ-terms are built in the usual fashion. Through the writing of programs, the programmer specifies (explicitly or implicitly) typed constants and variables. λ-terms can then be built up using constants, variables, applications, and abstractions in the usual way. Logical connectives and quantifiers are introduced into these λ-terms by introducing suitable constants; in particular, the constants $\wedge, \vee, \supset$ are all assumed to have type $o \to o \to o$, and the constants Π and Σ are given type $(\alpha \to o) \to o$ for each type replacing the "type variable" α. (Negation is not used in this programming language.) The expressions $\Pi\lambda x\ A$ and $\Sigma\lambda x\ A$ are abbreviated to be $\forall x\ A$ and $\exists x\ A$, respectively. A λ-term which is of type o is called a *proposition.*

Equality between λ-terms is taken to mean $\beta\eta$-convertible. We shall assume that the reader is familiar with the basic facts about λ-conversion. It suffices to say here that this equality check is decidable, although if the terms being compared are large, this check can be very expensive: β-reduction can greatly increase the size of λ-terms.

A proposition in λ-normal form whose head is not a logical constant will be called an *atomic* formula. In this section, A denotes a syntactic variable for atomic formulas.

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

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

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

Here, the universal quantifier $\forall c\ G$ in goal formulas must be over a constant c in G and the existential quantification $\exists v\ G$ must be over a variable v in G. Similarly, to form the formula $\forall v\ D$, v is a variable in D. There is one final restriction: if an atomic formula is a definite clause, it must have a constant as its head. The heads of atomic goal formulas may be either variable or constant. A *logic program* or just simply a *program* is a finite set, generally written as $\mathcal{P}$, of closed definite formulas.

Several abstract properties of this logic programming language are presented in [MNS87]. It is stated there that although substitutions for predicates in the underlying impredicative and unramified logic can be very complex, in this setting, substitutions for predicates can be determined through unifications. Furthermore, substitution terms can be restricted to be those λ-terms whose embedded logical connectives (if any) satisfy the constraints placed on goal formulas above.

Another property known for this logic programming language is that a sound and complete (with respect to intuitionistic logic) *non-deterministic* interpreter can be implemented

by employing the following six *search operations.* Here, the interpreter is attempting to determine if the goal formula G follows from the program $\mathcal{P}$. The substitution instances used by GENERIC and BACKCHAIN are those described above.

AND	If G is $G_1 \wedge G_2$ then try to show that both G_1 and G_2 follow from $\mathcal{P}$.
OR	If G is $G_1 \vee G_2$ then try to show that either G_1 or G_2 follows from $\mathcal{P}$.
AUGMENT	If G is $D \supset G'$ then add D to the current program and try to show G'.
GENERIC	If G is $\forall x\ G'$ then pick a new parameter c and try to show $[x/c]G'$.
INSTANCE	If G is $\exists x\ G'$ then pick some closed λ-term t and try to show $[x/t]G'$.
BACKCHAIN	If G is atomic, we must now consider the current program. If there is a universal instance of a definite clause which is equal to G then we are done. If there is a definite clause with a universal instance of the form $G' \supset G$ then try to show G' from $\mathcal{P}$. If neither case holds then G does not follow from $\mathcal{P}$.

In order to implement such an interpreter, it is important to make choices which are left unspecified in the high-level description above. There are, of course, many ways to make such choices. We will assume for the purposes of this paper that choices similar to those routinely used in Prolog are employed. In particular, we are following the conventions established in the λProlog system which implements most of the language we are describing (see [MN86a] and [MN87a]).

The order in which conjuncts and disjuncts are attempted and the order for backchaining over definite clauses is determined exactly as in conventional Prolog systems: 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.

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 closed λ-term should be inserted. Instead, the Prolog implementation technique of instantiating the existential quantifier with a logical (free) variable which is later "filled in" using unification is employed. A similar use of logical variables is made in implementing BACKCHAIN: universal instances are made using new logical variables.

The addition of logical variables in our setting, however, forces the following extensions to conventional Prolog implementations. First, higher-order unification becomes necessary since these logical 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. Since higher-order unification is not in general decidable and since most general unifiers do not necessarily exist when unifiers do exist, unification can contribute to the search aspects of the full interpreter. λProlog addresses this by implementing a depth-first version of the unification search procedure described in [Hue75]. For more information on how logic programming behaves with such a unification procedure, see [MN86a, Nad86]. The higher-order unification problems we shall encounter in this paper are all rather simple: it is easy to see, for example, that all such problems are decidable.

The presence of logical variables in an implementation also requires that GENERIC be implemented slightly differently than is described above. In particular, if the goal $\forall x\ G'$

contains logical variables, the new parameter c must not appear in the terms eventually instantiated for the logical variables which appear in G' or in the current program. Without this check, logical variables would not be a sound implementation technique.

Since much of this paper is concerned with how to implement theorem provers using the class of hereditary Harrop formulas presented above, we shall need to present many such formulas. We will make such presentations by using the syntax adopted by the λProlog system, which itself borrows from conventional Prolog systems.

Variables are represented by tokens with an upper case initial letter and constants are represented by tokens with a lower case initial letter. Function application is represented by juxtaposing two terms of suitable types. Application associates to the left, except when a constant is declared to be infix and then normal infix conventions are adopted. λ-abstraction is represented using backslash as an infix symbol: a term of the form $\lambda x\ T$ is written as `X\T`. Terms are most accurately thought of as being representatives of $\beta\eta$-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 used to write the top-level connective of definite clauses: the clause $G \supset A$ is written `A :- G`. Implications in goals and the bodies of clauses are 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.

Below is an example of a (first-order) program using this syntax.

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

The goal `(sterile j)`, for example, follows from these clauses.

In order to base a logic programming language on the simply typed λ-calculus, all constants and variables must be assigned a type. In λProlog, types are assigned either explicitly by user declarations or by automatically inferring them from their use in programs. Explicit typings are made by adding to program clauses declarations such as:

```
type  sterile  jar -> o.
type  in       insect -> jar -> o.
```

where `jar` and `insect` are primitive types. Notice that from this declaration, the types of the variables and other constants in the example clauses above can easily be inferred.

λProlog permits a degree of polymorphism by allowing type declarations to contain type variables (written as capital letters). For example, `pi` is given the polymorphic typing `(A -> o) -> o`. It is also convenient to be able to build new "primitive" types from other types. This is done using type constructors. In this paper, we will need to have only one such type constructor, `list`. For example, `(list jar)` would be the type of lists all of

whose entries are of type `jar`. Lists are represented by the following standard construction: `[]` represents an empty list of polymorphic type `(list A)`, and if `X` is of type `A` and `L` is of type `(list A)` then `[X|L]` represents a list of type `(list A)` whose first element is `X` and whose tail is `L`. Complex expressions such as `[X|[Y|[]]]` are abbreviated as simply `[X,Y]`.

3 Specifying Inference Rules

In this section we briefly outline how to use definite clauses to specify inference rules in two kinds of proof systems. Although we only consider theorem provers for first-order logic, the techniques described in this section are much more general (see Section 6).

Since we wish to implement a logic within a logic, we will find it convenient to refer to the logic programming language as the *metalogic* and the logic being specified as the *object logic*. An object logic's syntax can be represented in much the same way as the metalogic's syntax is represented. To represent a first-order logic, we introduce two primitive types: `bool` for the object-level boolean and `i` for first-order individuals. Given these types, we introduce the following typed constants.

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

It is easy to identify closed λ-terms of type `bool` as first-order formulas and of type `i` as first-order terms. For example, the λ-term

```
(forall X\ (exists Y\ ((p X Y) imp (q (f X Y)))))
```

represents the first-order formula $\forall x \exists y (P(x,y) \supset Q(f(x,y)))$.

We shall outline how to specify both sequential and natural deduction inference rules for a first-order logic. To define the sequential proof system we introduce a new infix constant `-->` of type `(list bool) -> bool -> sequent`; that is, a sequent contains a list of formulas as its antecedent and a single formula as its succedent (much as in the LJ sequent system in [Gen35]). We also want to retain proofs as they are built, so we shall introduce another primitive type `proof_object` which will be the type of proofs.

The basic relation between a sequent and its proofs will be represented as a binary relation on the metalevel by the constant `proof` of type `sequent -> proof_object -> o`. The inference rules of sequential calculus can be considered as simple declarative facts about the `proof` relation. As we shall see, all these declarative facts are expressible as definite clauses.

Consider the $\wedge$-R inference rule which introduces a conjunction on the right side of the sequent.

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

The declarative reading of this inference rule is captured by the following definite clause.

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

This clause may be read as: if P1 is a proof of (Gamma --> A) and P2 is a proof of (Gamma --> B), then (and_r P1 P2) is a proof of (Gamma --> (A and B)). The rule can also be viewed as defining the constant and_r: it is a function from two proofs (the premises of the $\wedge$-R rule) to a new proof (its conclusion). Its logic program type is `proof_object -> proof_object -> proof_object`.

Operationally, this rule could be employed to establish a proof-goal: using the BACK-CHAIN search command, first unify the sequent and proof in the head of this clause with the sequent and proof in the query. If there is a match, use the AND search operation to verify the two new proof-goals in the body of this clause. The unification here is essentially first-order.

Next we consider the two inference rules for proving disjunctions.

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

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

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

Declaratively, this clause specifies the meaning of a proof of a disjunction. For (or_r P) to be a proof of (Gamma --> (A or B)), P must be a proof of either (Gamma --> A) or (Gamma --> B). Operationally, this clause would cause an OR search operation to be used to determine which of the proof-goals in the body should succeed.

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. Consider the following implication introduction rule.

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

This could be specified as the following definite clause.

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

All propositional rules for Gentzen sequential systems can be very naturally understood as combining a first-order unification step with possibly an AND or an OR search operation. The structural rules of contraction, thinning, and interchange could be specified by simply manipulating lists of formulas. For example, the following clauses specify these three structural rules.

```
proof ([C|Gamma] -> A) (contract P) :- proof ([C,C|Gamma] -> A) P.
proof ([C|Gamma] -> A) (thin P) :- proof (Gamma -> A) P.
proof (Gamma1 -> A) (interchange P) :- append S1 [B,C|S2] Gamma1,
                                       append S1 [C,B|S2] Gamma2,
                                       proof (Gamma2 -> A) P.
```

Here, append is the standard Prolog procedure for appending lists.

We now look at specifying quantifier introduction rules. Here, the operational reading of definite clauses will use the INSTANCE and GENERIC search operations and higher-order unification. Consider the following $\exists$-R inference rule:

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

which can be written as the following definite clause.

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

The existential formula of the conclusion of this rule is written `(exists A)` where the logical variable `A` has functional type `i -> bool`. Since `A` is an abstraction over individuals, `(A T)` represents the formula that is obtained by substituting `T` for the bound variable in `A`. Declaratively, this clause reads: if there exists a term `T` (of type `i`) such that `P` is a proof of `(Gamma --> (A T))`, then `(exists_r P)` is a proof of `(Gamma --> (exists A))`. Operationally, we rely on higher-order unification to instantiate the logical variable `A`. The existential instance `(A T)` is obtained via the interpreter's operations of λ-application and normalization. Of course, the implementation of INSTANCE will choose a logical variable with which to instantiate `T`. By making `T` a logical variable, we do not need to commit to a specific term for the substitution. It will later be assigned a value through unification if there is a value which results in a proof.

The following consideration of the $\forall$-R inference figure raises a slight challenge to our specification of inference rules.

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

There is the additional proviso that y is not free in Γ or $\forall x\ B$. 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 metalevel.

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

Again `A` has functional type. In this case, so does `P`, and the type of `forall_r` is `(i -> proof_object) -> proof_object`. Declaratively, this clause reads: if we have a function `P` that maps arbitrary terms `Y` to proofs `(P Y)` of the sequent `(Gamma --> (A Y))`, then `(forall_r P)` is a proof of `(Gamma --> (forall A))`. In order to capture the proviso on y it is necessary to introduce a λ-abstraction over type `i` into proof objects.

Operationally, the GENERIC search operation is used to insert a new parameter of type `i` into the sequent. Since that parameter will not be permitted to appear in `Gamma`, `A`, or `P`, the proviso will be satisfied.

The following simple definite clause specifies initial sequents, that is, a sequent whose antecedent contains one formula which is also its succedent.

```
proof ([A] --> A) (initial A).
```

Here the constant `initial` is of type `bool -> proof_object`. It represents one way in which formulas get placed inside a proof.

We next briefly consider specifying inference rules in a natural deduction setting (see [Gen35] or [Pra65]). Here, the basic proof relation is between proofs and formulas (instead of sequents). Hence, for these examples, we assume that `proof` is of the type `bool -> proof_object -> o`. Several of the introduction rules for this system resemble rules that apply to succedents in the sequential system just considered. Those that correspond to the example inference rules given in the previous section are as follows:

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

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

The $\forall$-I rule also has the proviso that y cannot appear in $\forall x\ B$ or in any assumptions on which that formula depends. These inference rules can be specified naturally as the following definite clauses.

```
proof (A and B) (and_i P1 P2) :- proof A P1, proof B P2.
proof (A or B) (or_i P) :- proof A P; proof B P.
proof (exists A) (exists_i P) :- sigma T\ (proof (A T) P).
proof (forall A) (forall_i P) :- pi Y\ (proof (A Y) (P Y)).
```

In natural deduction, unlike sequential systems, we have the additional task of specifying the operation of discharging assumptions. Consider the following implication introduction rule.

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

This rule can very naturally be specified using the definite clause:

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

This clause represents the fact that if P is a "proof function" which maps an arbitrary proof of A, say PA, to a proof of B, namely (P PA), then (imp_i P) is a proof of (A imp B). Here, the proof of an implication is represented by a function from proofs to proofs. The constant imp_i has the type (proof_object -> proof_object) -> proof_object. Notice that while sequential proofs only contain abstractions of type i, natural deduction proofs contain abstractions of both types i and proof_object.

Operationally, the AUGMENT search operation plays a role in representing the discharge of assumptions. In this case, to solve the subgoal (pi PA\ ((proof A PA) => (proof B (P PA)))), the GENERIC operation is used to choose a new object, say pa, to play the role of a proof of the formula A. The AUGMENT goal is used to add this assumption about A and pa, that is (proof A pa), 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 will most likely contain instances of the proof of A (the term pa). The function P is then the result of abstracting out of this proof of B this generic proof object.

There are several aspects of both sequential and natural deduction proof systems which the above discussion does not cover. We elaborate a bit further on two such aspects: the representation of proof objects and controlling search in the resulting specification.

The proof objects built in the previous examples serve only to show how proofs might be built and to illustrate the differences between the two styles of proof systems. Proof terms could contain more information. For example, it might be desirable to have two or-introduction rules in both proof systems. The two proof building constants, say or_r1 and or_r2 for the sequential system, would indicate whether the left or right disjunct had been proved. Similarly, in the introduction of existential quantifiers, it might be sensible to store inside the proof the actual substitution term used. In that case, the exists_r would be given the type i -> proof_object -> proof_object.

Depending on the later use made of proofs, it might be desirable for proof objects to contain less information than we have specified. For example, it might be desirable for a single sequential proof to be a proof of many different sequents, that is, the proof terms should be polymorphic. In that case, it might be desirable for the initial proof term to not store a formula within the proof. Instead, initial could have the simpler type proof_object. Of course, proof objects do not need to be built at all. The predicate proof could be replaced with similar predicates, say provable or true, of type sequent -> o or bool -> o.

Another aspect of these theorem provers not yet considered is control. Assume that we have a complete set of definite clauses which specify the inference rules needed to implement some sequent proof system. These definite clauses could be used, for example, to do both proof checking and theorem proving. To do proof checking, assume that we are given a sequent (Gamma --> A) and a proof term P. If the goal (proof (Gamma --> A) P) succeeds, then P is a valid proof of (Gamma --> A). In this example, the proof in the initial proof-goal is closed, and this causes all subsequent proofs in proof-goals to also be closed. Since the top-level constant of a proof term completely determines the unique definite clause which can be used in backchaining, there is little problem of controlling proof checking. Even the simple minded depth-first discipline of λProlog will work.

Such definite clauses could also be used to do theorem proving. Here, we start with a

proof object which is just a logical variable which we wish to have instantiated. Since the proof object is a variable, multiple definite clauses could be applied to any one sequent. In particular, the structural rules could always be applied: as written above, they are much too non-deterministic to be useful in this setting. The same control problem is true for elimination rules in the natural deduction setting (see Section 5). It is possible to not implement thinning directly if the definition of initial sequents is extended to be sequents whose succedent is contained in its antecedent. Interchange does not need to be implemented directly if all rules introducing logical constants into the antecedent can operate on any formula in the antecedent instead of just the first formula. Contraction, of course, cannot be so simply removed. Implementing contraction is the great challenge to automating theorem provers. It is possible to build systems of definite clauses which can provide complete theorem provers under depth-first processing of backtracking. Generally, the proof system must be modified somewhat and careful controlling of contraction must be observed. For example, we have implemented a variant of Gentzen's LK sequent system [Gen35] so that it is a complete theorem prover for first-order classical logic [Fel87].

In the next two sections, we describe a different approach to specifying a set of inference rules so that it is easier to be explicit about controlling search.

4 A Logic Programming Implementation of Tacticals

Tactic style theorem provers were first built in the early LCF systems and have been adopted as a central mechanism in such notable theorem proving systems as Edinburgh LCF [GMW79], Nuprl [Con86], and Isabelle [Pau87]. Primitive tactics generally implement inference rules while compound tactics are built from these using a compact but powerful set of *tacticals*. Tacticals provide the basic control over search. Tactics and tacticals have proved valuable for several reasons. They promote modular design and provide flexibility in controlling the search for proofs. They also allow for blending automatic and interactive theorem proving techniques in one environment. This environment can also be grown incrementally.

We shall argue in this section and the next that logic programming provides a very suitable environment for implementing both tactics and tacticals. Generally tactics and tacticals have been implemented in the functional programming language ML. Here we shall show how they can be implemented in λProlog. This implementation is very natural and extends the usual meaning of tacticals by permitting them to have access to logical variables and all six search operations. A comparison between the ML and λProlog implementations is contained in Section 6.

In this section, we assume that tactics are primitive, and show how to implement the higher-level tacticals. In the next section, we show how to implement individual tactics for the proof systems considered in the previous section. Our presentation in these two sections is cursory: for more details, the reader is referred to [Fel87].

We introduce the primitive type `goalexp` to denote *goal expressions*. In the next section, we will define primitive goals which encode such propositions as "this sequent has this proof" or "this formula is provable." For this section, we wish to think more abstractly of goals: they simply denote judgments which could succeed or fail. We define the following goal

constructors used to build compound forms of such judgments.

```
type  truegoal    goalexp.
type  andgoal     goalexp -> goalexp -> goalexp.
type  orgoal      goalexp -> goalexp -> goalexp.
type  allgoal     (A -> goalexp) -> goalexp.
type  existsgoal  (A -> goalexp) -> goalexp.
type  impgoal     A -> goalexp -> goalexp.
```

Here, `truegoal` represents the trivially satisfied goal, `andgoal` corresponds to the AND search operation, `orgoal` to OR, `allgoal` to GENERIC, `existsgoal` to INSTANCE, and `impgoal` to AUGMENT. Notice that `allgoal`, `existsgoal`, and `impgoal` are polymorphic. We will use `allgoal` and `existsgoal` in the next section with `A` instantiated to types `i` and `proof_object`.

The meaning of a tactic will be a *relation* between two goals: that is, its type is `goalexp -> goalexp -> o`. Abstractly, if a tactic denotes the relation R, then $R(g_1, g_2)$ means that to satisfy goal g_1, it is sufficient to satisfy goal g_2. Primitive tactics are implemented directly in the underlying λProlog language in a fashion similar to that used in the preceding section; they are also assumed to work only for primitive goals. Compound tactics and the application of tactics to compound goals are implemented completely by the program clauses below.

The first program we describe, called `maptac`, 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. For example, on an `andgoal` structure, `maptac` will apply the tactic to each subgoal separately, forming a new `andgoal` structure to combine the results. The type of `maptac` is `(goalexp -> goalexp -> o) -> goalexp -> goalexp -> o`, that is, the metalevel predicate `maptac` takes as its first argument a metalevel predicate which represents a tactic.

```
maptac Tac truegoal truegoal.

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

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

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

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

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

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

The last clause above is used once the goal is reduced to a primitive form. Note that an auxiliary predicate `memo` (of polymorphic type `A -> o`) was introduced in the clause

implementing **impgoal**. This allows the introduction of new clauses into the program. The type and content of these clauses will be specific to a particular theorem prover. In the next section we will illustrate how it is used to add assumptions that are discharged during the construction of natural deduction proofs.

The following definite clauses implement several of the familiar tacticals found in many tactic style theorem provers.

```
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 truegoal :- Tac InGoal OutGoal, goalreduce OutGoal truegoal.
```

The **then** tactical performs the composition of tactics. **Tac1** is applied to the input goal, and then **Tac2** is applied to the resulting goal. **maptac** is used in the second case 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 logical variables between these 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 corresponding metalevel goal), 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. **repeat** is recursively defined using the three tacticals, **then**, **orelse**, and **idtac**. It repeatedly applies a tactic until it is no longer applicable. The **try** tactical prevents failure of the given tactic 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** of type **goalexp -> goalexp -> o** which simplifies compound goal expressions by removing occurrences of **truegoal** from them. The code for **goalreduce** is as follows:

```
goalreduce (andgoal truegoal Goal) OutGoal :- goalreduce Goal OutGoal.
goalreduce (andgoal Goal truegoal) OutGoal :- goalreduce Goal OutGoal.
goalreduce (allgoal T\ truegoal) truegoal.
goalreduce (impgoal A truegoal) truegoal.
goalreduce Goal Goal.
```

Although the **complete** tactical is the only one that requires the use of the **goalreduce** procedure, it is also possible and probably desirable to modify the other tacticals so that they use it to similarly reduce their output goal structures whenever possible.

Notice that the notion of success and failure of the interpreter in Section 2 carries over to the success and failure of a tactic to solve a goal. The failure of the interpreter to succeed on a goal of the form `(Tac InGoal OutGoal)` indicates the failure of `Tac` to make progress toward solving `InGoal`.

The definite clauses listed above provide a complete implementation of tacticals. We now illustrate how tactics for a theorem prover can be implemented.

5 Inference Rules as Tactics

In this section we illustrate how to specify the inference rules of Gentzen's NK natural deduction system [Gen35]. Each inference rule will be specified as a primitive tactic. The basic goal needed to be established in this system is that a certain formula has a certain proof. This is encoded by the constant `proofgoal` which is of type `bool -> proof_object -> goalexp`. Goals of the form `(proofgoal A P)` will be called *atomic* goals of the natural deduction theorem prover, in contrast to *compound* goals built from using the goal constructors from the last section.

The implementation of inference rules is done in a fashion similar to the techniques in Section 3, except the clauses are made into named facts. For example, $\wedge$-I is specified as:

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

This tactic can be applied whenever the formula in the input goal is a conjunction. This clause has essentially the same meaning as the definite clause for the $\wedge$-I rule of Section 3, except that this clause is not automatically BACKCHAINed on by the interpreter. Put another way, the inference rule is represented here declaratively instead of procedurally. The procedural representation is at the mercy of the depth-first logic programming interpreter. In this other form, however, tacticals can specify their own forms of control.

In order to handle the hypotheses in this proof system, we introduce the new primitive type `assump` and the additional metalevel predicate `hyp` of type `bool -> proof_object -> assump`. This new symbol will be used in conjunction with `impgoal` and AUGMENT to represent assumptions. For example, $\supset$-I can be implemented using the clause:

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

The AUGMENT goal will then add a clause of the form `(memo (hyp A pa))` to the program (where `pa` is a new constant generated by the GENERIC goal to replace `PA`).

Elimination rules are used to do forward reasoning from assumptions stored as `memo` facts. For example, the following clause specifies the $\wedge$-E inference rule.

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

This clause works by moving the goal `(proofgoal C PC)` into the expanded context containing the hypotheses `(hyp A (and_e1 P))` and `(hyp B (and_e2 P))` if the hypothesis `(hyp (A and B) P)` already exists.

The remaining inference rules for NK are given below. We use the constant `perp` of type `bool` to represent the formula $\perp$.

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

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

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

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

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

imp_e_tac (proofgoal B (imp_e P PA)) (proofgoal A PA) :-
  memo (hyp (A imp B) P).

forall_e_tac (proofgoal B PB)
             (existsgoal T\ (impgoal (hyp (A T) (forall_e P)) (proofgoal B PB))) :-
  memo (hyp (forall A) P).

exists_e_tac (proofgoal B (exists_e P PB))
             (allgoal T\ (allgoal PA\ (impgoal (hyp (A T) PA)
                                               (proofgoal B (PB T PA))))) :-
  memo (hyp (exists A) P).

neg_e_tac (proofgoal perp (neg_e P PA)) (proofgoal A PA) :-
  memo (hyp (neg A) P).

perp_tac (proofgoal A (contra PA))
         (allgoal P\ (impgoal (hyp (neg A) P) (proofgoal perp (PA P)))).

close_tac (proofgoal A P) truegoal :- memo (hyp A P).
```

Given these primitive tactics, we can then write compound tactics using the tacticals of the last section. For example, consider the following query (metalevel goal):

```
repeat (orelse or_i_tac and_i_tac) (proofgoal ((r or (p imp q)) and s) P) OutGoal.
```

This goal would succeed by instantiating `P` to the open proof term `(and_i (or_i P1) P2)` and instantiating `OutGoal` to the open goal expression:

```
(andgoal (orgoal (proofgoal r P1) (proofgoal (p imp q) P1)) (proofgoal s P2)).
```

This latter goal could then get processed by other tactics. Such processing would then instantiate the logical variables P1 and P2 which would then further fill in the original proof variable P in the query above.

Providing a means for accommodating user interaction is one of the strengths of tactic theorem provers. One way to provide an interface to the user in this paradigm is by writing tactics that request input. The following is a very simple tactic which asks the user for direction.

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

Here we have a tactic that, for any atomic input goal, will present the formula to be proved to the user, query the user for a tactic to apply to the input goal, then apply the input tactic. As in Prolog, `(write A)` prints `A` to the screen and will always succeed while `(read A)` prompts the user for input and will succeed if `A` unifies with the input. In this case `(read Tac)` will accept any term of type `goalexp -> goalexp -> o`.

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

```
interactive InGoal OutGoal :- repeat query InGoal OutGoal.
```

These additions to the tactic prover will still not be sufficient, in general, for interactive theorem proving in a natural deduction setting. For example, if there is more than one conjunction among the discharged assumptions, the $\wedge$-I rule will be applicable in more than one way. The user needs the capability to specify which formula to apply the tactic to. One way to solve this problem is to extend the program with tactics that request input from the user. Such tactics could be written as:

```
and_e_query (proofgoal C PC)
            (impgoal (hyp A (and_e1 P))
              (impgoal (hyp B (and_e2 P)) (proofgoal C PC))) :-
  memo (hyp (A and B) P), write "Eliminate this conjunction?",
  write (A and B), read "yes".
```

The tactic would enumerate conjunctive hypotheses until the user types in the word "yes."

6 Related Work

The development of the theory of higher-order hereditary Harrop formulas was motivated by a desire to develop a clean semantics for a programming language which embraced many more aspects of logic than first-order Horn clauses embrace. Other applications of this language that have been explored using the λProlog implementation lie in the areas of program transformations [MN87b] and computational linguistics [MN86b]. In this paper we have emphasized particular aspects that are useful for the specific task of implementing theorem provers.

The UT prover is well-known for its implementation of a natural deduction style proof system [Ble77, Ble83]. Since some aspects of its implementation have been designed to handle quantifiers and substitutions in a principled fashion, it is interesting to compare it to the systems described in this paper. In the UT prover, the IMPLY procedure is based on a set of rules for a "Gentzen type" system for first-order logic. In this procedure, formulas keep their basic propositional structure although their quantifiers are removed. In the AND-SPLIT rule of this prover (which corresponds to the $\wedge$-R rule in a sequent system), the first conjunctive subgoal returns a substitution which must then be applied to the second subgoal before it is attempted. In logic programming, such composition of substitutions obtained from separate subgoals is handled automatically via shared logical variables. An issue that arises as a result of the AND-SPLIT rule is the occurrence of "conflicting bindings" due to the need to instantiate a quantified formula more than once. The UT prover uses *generalized substitutions* [TB79] to handle such multiple instances. A substitution is the final result returned when a complete proof is found. In contrast, in our logic programming language, quantification in formulas is represented by λ-abstraction. As a result, we do not need to remove quantifiers before attempting a proof. Instead we implement the inference rules for quantified formulas as illustrated in Section 3. For example, in a sequent system for classical logic, the inference rules for a universally quantified hypothesis or an existentially quantified conclusion are two rules which must allow multiple instantiations. This is easily accomplished by introducing a new logical variable for each instantiation. As the result of a successful proof we obtain a proof term rather than a substitution. As mentioned in Section 3, the construction of such proof terms may be defined to include the substitution information if desired. In fact such proof terms could be simplified to only return substitution information. Such simplified proof terms would be very similar to the generalized substitutions of the UT prover.

Other theorem provers that are based on tactics and tacticals include LCF [GMW79], Nuprl [Con86], and Isabelle [Pau86]. The programming language ML is the metalanguage used in all of these systems. ML is a functional language with several features that are useful for the design of theorem provers. It contains a secure typing scheme and is higher-order, allowing complex programs to be composed easily. There are several differences in the implementations of tactics in ML and in λProlog. First, tactics in ML are functions that take a goal as input and return a pair consisting of a list of subgoals and a validation. In contrast, tactics in λProlog are relational, which is very natural when the relation being modeled is "is a proof of." The fact that input and output distinctions can be blurred makes it possible, as described in Section 3, for tactics to be used in both a theorem proving and proof checking context. The functional aspects of ML do not permit this dual use of tactics. The ML notion of validations is replaced in our system by (potentially much larger and more complex) proof objects.

Second, it is worth noting the differences between the ML and λProlog implementations of the `then` tactical. 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 analogue of a list of subgoals is a nested `andgoal` structure. These are processed by the `andgoal` clause of `maptac`. The behavior of `then` (in conjunction with `maptac`) in λProlog is also a bit richer in two ways. First of all,

`maptac` is richer than the usual notion of a mapping function in that, in addition to nested `andgoal` structures, it handles all of the other goal structures corresponding to the λProlog search operations. Secondly, in the ML version of `then`, if the second tactic fails after a successful call to the first tactic, the full tactic still fails. In contrast, in λProlog, 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. Alternatively, we could use the cut (`!`) as in Prolog which prevents backtracking beyond a specified point and thus restrict the meaning of `then` to match its ML counterpart.

A third difference with ML is that for every constructor used to build a logic, explicit discriminators and destructors must also be introduced. In logic programming, however, the purpose served by these explicit functions is achieved within unification. The difference here is particularly striking if we look at the different representations of quantified formulas. A universal formula is constructed in ML by calling a `mk_forall` function which takes a variable and a formula and returns a universally quantified formula. `is_forall` and `dest_forall` are the corresponding discriminator and destructor to test for universal formulas and to obtain its components, respectively. Manipulating quantified formulas requires that the binding be separated from its body. In logic programming, 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, sophisticated analysis of λ-terms is still possible to perform using higher-order unification. 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 Isabelle theorem prover [Pau87] uses a fragment of higher-order logic with implication and universal quantification which is used to specify inference rules. That fragment is essentially a subset of the higher-order hereditary Harrop formulas. Hence, 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.

Although our example theorem provers have been for first-order logic, we have also considered the logic of higher-order hereditary Harrop formulas as a specification language for a wide variety of logics. In this respect, we share a common goal with the Edinburgh Logical Framework (LF) project [HHP87]. LF was developed for the purpose of capturing the uniformities of a large class of logics so that it can be used as the basis for implementing proof systems. The two approaches are actually similar in ways that go beyond simply sharing common goals. First, the LF notions of *hypothetical* and *schematic* judgments can be implemented with the GENERIC and AUGMENT search operations in the logic programming setting. The LF hypothetical judgment takes the form $J_1 \vdash J_2$ and represents the assertion that J_2 follows from J_1. Objects of this type are functions mapping proofs of J_1 to proofs of J_2. Such a judgment can be implemented by having the GENERIC search operation introduce a new proof object, and then using AUGMENT to assume the fact that this new object is a proof of J_1. A proof of J_2 would then be the intended function applied to this new object. A schematic judgment in LF is of the form $\bigwedge_{x:A} J(x)$. It is a statement

about arbitrary objects x of type A and is proved (inhabited) by a function mapping such objects to proofs of $J(x)$. This is implemented by using the GENERIC and AUGMENT search operations to first introduce an arbitrary constant and then assume it to be of LF type A.

Second, and more specifically, we have developed an algorithm that systematically translates all of the example LF signatures in [HHP87] and [AHM87] to logic programs [Fel87]. In the logic programming setting, in addition to being natural specifications, the resulting definite clauses also represent non-deterministic theorem provers. LF signatures could also be translated to sets of tactics to be used in a tactic theorem prover. The formal properties of this translation have yet to be established.

Acknowledgements The authors would like to thank Robert Constable, Elsa Gunter, Robert Harper, and Frank Pfenning for valuable comments and discussions. We are also grateful to the reviewers of an earlier draft of this paper for their comments and corrections. The first author is supported by US Army Research Office grant ARO-DAA29-84-9-0027, and the second author by NSF grant CCR-87-05596 and DARPA N000-14-85-K-0018.

References

[AHM87] Arnon Avron, Furio A. Honsell, and Ian A. Mason. *Using Typed Lambda Calculus to Implement Formal Systems on a Machine.* Technical Report ECS-LFCS-87-31, Laboratory for the Foundations of Computer Science, University of Edinburgh, June 1987.

[Ble77] W. W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9:1–35, 1977.

[Ble83] W. W. Bledsoe. *The UT Prover.* Technical Report ATP-17B, University of Texas at Austin, April 1983.

[Con86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Fel87] Amy Felty. Implementing theorem provers in logic programming. November 1987. Dissertation Proposal, University of Pennsylvania.

[Gen35] 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.

[GMW79] 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.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[Hue75] G. P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[MN86a] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[MN86b] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 247–255, 1986.

[MN87a] Dale Miller and Gopalan Nadathur. λProlog Version 2.6. August 1987. Distribution in C-Prolog code.

[MN87b] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming*, San Francisco, September 1987.

[MNS87] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary harrop formulas and uniform proof systems. In *Symposium on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.

[Nad86] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, December 1986.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau87] Lawrence C. Paulson. *The Representation of Logics in Higher-Order Logic.* Draft, University of Cambridge, July 1987.

[Pra65] Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Uppsala, 1965.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge MA, 1986.

[TB79] Mabry Tyson and W. W. Bledsoe. Conflicting bindings and generalized substitutions. In *4th International Conference on Automated Deduction*, pages 14–18, Springer-Verlag, February 1979.