CS-1992-17

A Proof Procedure for the Logic of Hereditary Harrop Formulas

Gopalan Nadathur

Department of Computer Science

Duke University

Durham, North Carolina 27708-0129

November 1992

A Proof Procedure for the Logic of Hereditary Harrop Formulas*

Gopalan Nadathur
Department of Computer Science
Duke University, Durham, NC 27706
gopalan@cs.duke.edu

Abstract

A proof procedure is presented for a class of formulas in intuitionistic logic. These formulas are the so-called *goal formulas* in the theory of hereditary Harrop formulas. Proof search in intuitionistic logic is complicated by the non-existence of a Herbrand-like theorem for this logic: formulas cannot in general be preprocessed into a form such as the clausal form and the construction of a proof is often sensitive to the order in which the connectives and quantifiers are analyzed. An interesting aspect of the formulas we consider here is that this analysis can be carried out in a relatively controlled manner in their context. In particular, the task of finding a proof can be reduced to one of demonstrating that a formula follows from a set of assumptions with the next step in this process being determined by the structure of the conclusion formula. An acceptable implementation of this observation must utilize unification. However, since our formulas may contain universal and existential quantifiers in mixed order, care must be exercised to ensure the correctness of unification. One way of realizing this requirement involves labelling constants and variables and then using these labels to constrain unification. This form of unification is presented and used in a proof procedure for goal formulas in a first-order version of hereditary Harrop formulas. Modifications to this procedure for the relevant formulas in a higher-order logic are also described. The proof procedure that we present has a practical value in that it provides the basis for an implementation of the logic programming language λProlog.

Key Words: hereditary Harrop formulas, proof procedure, logic programming, intuitionistic logic.

1 Introduction

The basis for logic programming has traditionally been provided by the logic of Horn clauses [26]. Using this logic has lead to the realization of novel and genuinely useful features in programming. This logic has, for instance, provided for a paradigm that supports search as a primitive operation and has revealed novel uses for the operation of unification in programming. The simplicity of this logic, nevertheless, prevents the natural realization of several features considered important in modern day programming languages. One example of a facet that is not directly supported by this logic is that of abstraction: using Horn clauses alone, there is no transparent method for capturing the idea that some parts of the program are to be used only in solving specific tasks or for deeming that certain names (of constants, functions or predicates) are to be visible only in specific contexts.

* This paper is to appear in the *Journal of Automated Reasoning*. Comments on its contents are welcome and may be sent to the author at the indicated address.

Shortcomings such as these have lead to an interest in describing richer logics that, on the one hand, preserve the features of Horn clause logic that are important to their programming use and, on the other hand, provide a means for realizing additional desirable features.

A logic that has been proposed in this regard is that of hereditary Harrop formulas [11, 16]. This logic has a first-order and a higher-order variant. The essential sense in which (the first-order version of) this logic extends the logic of Horn clauses is by permitting implications and universal quantifiers in goals. (The precise syntax of first-order hereditary Harrop formulas is presented in Section 3). Hereditary Harrop formulas, when interpreted via the notion of intuitionistic provability, constitute an abstract logic programming language in the sense defined in [16]. From an intuitive perspective, this guarantees that the logical connectives that appear within these formulas can be interpreted as symbols having a fixed search semantics. With regard to the new logical symbols, this amounts to the following: A goal of the form $D \supset G$ can be interpreted as an instruction to augment the program with D in the course of solving G. A goal of the form $\forall x G$ can be interpreted as an instruction to generate a new name and to use it for x in the course of solving G. These logical symbols thus provide a means for realizing scoping with respect to program code and names and detailed illustrations of this aspect are provided in [12]. The higher-order version of this logic also provides for higher-order programming and for the use of higher-order terms as data structures. A logic programming language called λProlog that is based on this logic is described in [20] and has been used in numerous applications (*e.g.*, see [3, 6, 15, 24]).

Our interest in this paper is in describing a proof procedure for the logic of hereditary Harrop formulas. The practical motivation for this endeavor is obvious: such a procedure could provide the basis for an interpreter for λProlog. From a theoretical perspective, the exercise undertaken is interesting because it is provability in intuitionistic logic that is considered. In the context of classical logic, the existence of certain logical equivalences permits the search for proofs for formulas to be conducted in a carefully controlled fashion. In particular, any given formula can be converted into a form in which a sequence of existential quantifiers govern a quantifier free matrix, and determining provability then amounts to finding instantiations for the quantifiers that produces a tautology. A similar observation can unfortunately not be made with regard to intuitionistic logic. The construction of a proof in this context is much more sensitive to the order in which the connectives and quantifiers are analyzed and an important component of proof search is in fact determining a satisfactory order.

The inherent complexity in finding proofs in intuitionistic logic, and the fact that this is a relatively unexplored domain, makes this an interesting topic for investigation. There are at least two different directions that can be followed in such a study. First, the issue of finding proofs in the general context can be examined with a view towards controlling the search effort for any arbitrary formula. Some efforts have been invested in this direction, *e.g.*, those in [25, 27]. An alternative direction for exploration is that of finding restricted but interesting classes of formulas for which simple search procedures can be used. Hereditary Harrop formulas are an example of such a class of formulas. The interest in this class is apparent from the use for these formulas that we have described above. From the perspective of finding proofs, it turns out that the analysis of the logical symbols in these formulas can be carried out in a relatively deterministic fashion: this is in fact a consequence of the logic of these formulas possessing the property of *uniform proofs* in the sense of [16]. Unlike the case in classical logic, however, the quantifier structure of these formulas cannot be simplified prior to a search for a proof. Methods must therefore be provided for dealing with existential and universal quantifiers in the course of constructing proofs. Fortunately, ideas similar to the dynamic Skolemization described in [4] can be used and, in conjunction with the other properties of these formulas, this leads to a rather simple proof procedure.

We describe such a procedure in this paper and prove its soundness and completeness. It is to be noted that the ideas used in this procedure are not completely novel. As mentioned above, the adequacy of searching for uniform proofs in the logic of hereditary Harrop formulas is intrinsic to this procedure, and this fact is demonstrated in [16]. Similarly, the problem of unifying terms embedded under arbitrary sequences of quantifiers is central to our proof procedure, and approaches to this problem have been described in [23] and examined in detail in [14]. The specific solution to this problem that is used here involves labelling constants and variables and using these labels to constrain unification. This possibility has also been appreciated previously: the author first heard of it from Frank Pfenning in 1988. Finally, actual implementations of λProlog exist [1, 2, 20] that have closely related "proof" procedures as their bases. Despite these observations, we believe the discussions in this paper are of interest for at least two reasons. First, there has been, to our knowledge, no prior presentation of the particular proof procedure we describe here together with a demonstration of the fact that it is indeed a proof procedure. As should be apparent from this paper, this is a matter of some complexity and therefore worthy of careful treatment. Second, the procedure that we describe here, especially the manner of constraining unification that is employed in it, is, we feel, congenial to an efficient implementation of λProlog[1]. This procedure (or one closely related to it) has apparently been used in an implementation undertaken by Conal Elliott and Frank Pfenning [1] (see the comments in [2]) and is also being used in an abstract machine being developed by us for the language [8, 19]. The correctness of this procedure is, however, not readily apparent from other discussions. The proofs in this paper are, in this sense, essential for ensuring the correctness of possible implementations.

The rest of this paper is structured as follows. In the next section we summarize the various logical notions that we need, including the notion of intuitionistic provability. In Section 3, we describe the logic of first-order hereditary Harrop formulas and we present some properties that are relevant to the construction of a simple proof procedure for this logic. We also outline here the need for some mechanism for constraining unification and describe informally a scheme for realizing these constraints through a labelling of constants and variables. In Section 4 we describe this labelled unification — which is essentially first-order unification restricted to respect constraints represented by the labels on the variables and constants — and show the existence of most general unifiers with respect to it. In Section 5 we finally present our proof procedure for first-order hereditary Harrop formulas and prove it correct. The procedure, as we present it, is non-deterministic. However we show that the nondeterminism is inconsequential in several respects. In Section 6 we outline the manner in which the the procedure described in the previous section can be adapted to the context of higher-order hereditary Harrop formulas. A detailed presentation and a proof of correctness are somewhat tedious and we therefore do not undertake these in this paper. We conclude the paper with a brief discussion of the manner in which the procedure described here is actually being

[1]Proof procedures have been presented together with proofs of correctness for closely related formula classes in [10] and [13]. The structure of the procedure in [10] differs from the one considered here. Further, the discussions in [10] seem largely to note the constraints that must be placed on substitution terms without detailing simple methods for ensuring the satisfaction of these constraints. The procedure in [13] has a similar structure to our procedure and this paper also studies the unification problem for an interesting class of higher-order terms. However, the method for realizing constraints on substitutions that is used in the procedure in [13] differs from the one we describe here. We feel the that the method presented in this paper is better suited to an embedding in an abstract machine for a λProlog-like language. The discussions here complement, in this sense, those in [10] and [13].

implemented.

2 Logical Preliminaries

We shall use first-order intuitionistic logic in the discussions in this paper. The formulas in the logic are defined in the customary fashion: The language has variables, constants and function symbols, and terms are freely generated from these. There are predicate symbols and these are used in conjunction with terms to obtain atomic formulas. Finally the connectives and quantifiers are used to construct arbitrary formulas. We assume that $\sim$, $\vee$, $\wedge$, and $\supset$ are the primitive connectives available and $\exists$ and $\forall$ are the quantifiers.

To facilitate the description of our proof procedure we will need a (denumerably) infinite supply of constant symbols and we assume that this is in fact available. Further, we assume that these symbols are partitioned into a denumerable collection of denumerable sets and that there is an injective function from this collection to the natural numbers. Finally we shall need to talk about labelling functions on constants and variables. We assume that the behavior of such functions is *fixed* on the constants: any *labelling* function $\mathcal{L}$ must map a given constant to the natural number associated with the set to which the constant belongs.

The notion of free variables for formulas and terms is defined in the customary fashion. We shall use the notation $\mathcal{F}(t)$ to denote the set of variables free in (the term or formula) t. The notation is extended to arbitrary structures containing formulas and terms, such as sets, tuples, *etc*, in the following fashion: if S is such a structure, then $\mathcal{F}(S)$ is the collection of all the variables free in the formulas and terms appearing in S.

We shall consider performing substitutions for the free variables in terms and formulas. Care must be exercised in this process to avoid the usual capture problems. We describe one way in which this may be done.

Definition 1. A substitution θ is a finite set $\{\langle x_i, t_i\rangle | 1 \leq i \leq n\}$ of variable-term pairs such that, for $1 \leq i, j \leq n$, x_i and x_j are distinct variables if $i \neq j$. A substitution is, as usual, to be interpreted as a mapping on variables that is the identity except at the points specified. By an abuse of terminology, we refer to the set $\{x | \langle x, t\rangle \in \theta\}$ as the domain of θ, and to the substitution $\{\langle x, t\rangle | \langle x, t\rangle \in \theta \text{ and } x \in \mathcal{V}\}$ as the restriction of θ to (the set of variables) $\mathcal{V}$. The mapping on variables denoted by a substitution is extended in the usual fashion to terms. The application of θ to a formula is defined by recursion on the structure of the formula:

(i) G is atomic. In this case G is of the form $(P\ t_1\ \ldots\ t_n)$. Simply replace G by the formula that results from applying θ to each t_i.

(ii) G is $\sim G_1$. Let G_1' be the result of applying θ to G_1. Replace G by $\sim G_1'$.

(iii) G is $G_1 \vee G_2$, $G_1 \wedge G_2$ or $G_1 \supset G_2$. Let G_1' and G_2' result from applying θ to G_1 and G_2. Replace G by $G_1' \vee G_2'$, $G_1' \wedge G_2'$ or $G_1' \supset G_2'$ as the case might be.

(iv) G is $\forall y G_1$ or $\exists y G_1$. Using the notation introduced already, $\mathcal{F}(\theta)$ denotes the set of variables free in the substitution θ, *i.e.*, the set $\bigcup\{\mathcal{F}(t) \cup \{x\} | \langle x, t\rangle \in \theta\}$. If $y \notin \mathcal{F}(\theta)$ and applying the substitution to G_1 yields G_1', the desired result is $\forall y G_1'$ or $\exists y G_1'$. If $y \in \mathcal{F}(\theta)$, pick a z such that $z \notin \mathcal{F}(\theta) \cup \mathcal{F}(G_1)$ and let G_1' be obtained by first substituting z for y in G_1 and then applying the given substitution to the result. The desired result is now $\forall z G_1'$ or $\exists z G_1'$, as the case might be.

Once again, we shall need to consider the application of a substitution to all the formulas and terms contained in an arbitrary structure. We shall refer to this operation as the application of the substitution to the structure and, if the structure is S and the substitution is θ, we shall denote the result of performing it by $\theta(S)$.

We shall often consider singleton substitutions and we find it convenient to use an alternative notation for the application of these to formulas and terms. If θ is the substitution $\{\langle x, t\rangle\}$, then the application of θ to G may be written as $[t/x]G$. One formula is considered to be an *alphabetic variant* of another if it is obtained by replacing some (possibly no) subparts of the form $\forall y G$ or $\exists y G$ by $\forall z([z/y]G)$ or $\exists z([z/y]G)$, where z is a variable not free in G. It may be observed that the definition of substitution provided above does not identify a unique formula as the result, but rather a class of formulas that are alphabetic variants of each other. However, any member of this class is satisfactory for our purposes as will be apparent from the discussions below.

We need a concrete description of the notion of provability in the sequel. We adopt a formalization based on the sequent calculus. In the setting of intuitionistic logic, a sequent is a pair $\langle \Delta, \Theta\rangle$ of sets of formulas such that Δ is finite (possibly empty) and Θ is either empty or a singleton. The pair is usually written as $\Delta \longrightarrow \Theta$, with Δ and Θ themselves being written as sequences. The set Δ is referred to as the antecedent of the sequent and Θ as the succedent. Such a sequent corresponds intuitively to the assertion that Δ is inconsistent in the case that Θ is empty and to the assertion that the formula in Θ follows from those in Δ in the case that Θ is a singleton. Proofs for sequents are constructed by putting sequents together using the inference rule schemata in Figure 1. In generating instances of these schemata, we assume that Δ and Θ are instantiated so as to produce sequents, that t is instantiated by a term and c by a constant that does not appear in the instantiation of the lower "sequent" of the relevant rule schema. In those sequents where the antecedent has the form F, Δ for some formula F, we assume that F may appear in Δ, *i.e.*, formulas have arbitrary multiplicity. A proof for $\Delta \longrightarrow \Theta$ is a finite tree constructed using inference rules that are so defined and that has its root labelled with $\Delta \longrightarrow \Theta$ and its leaves labelled with sequents that have an atomic formula common to their antecedent and succedent.

We shall write $\Gamma \vdash_I B$ if the sequent $\Gamma \longrightarrow B$ has a proof in the calculus described above. The relation thus described corresponds to the provability in intuitionistic logic of the formula B from a set of premises Γ. The set of premises may be empty, and in this case we simply write $\vdash_I B$. The sequent calculus presented here to formalize this relation bears several similarities to the one used, for example, in [5]; the main difference is that the cut-elimination theorem has been incorporated here into the presentation of the sequent calculus.

The height of a proof is its height when viewed as a tree. The length of a proof is defined by recursion on its height: If the height is 1, then the length is 1. Otherwise we consider the cases for the last inference rule. If this is a rule with one upper sequent whose proof has length l, then the length of the entire proof is $l + 1$. If the rule has two upper sequents with proofs of length l_1 and l_2 respectively, then the length of the entire proof is $l_1 + l_2 + 1$.

We observe some meta-theorems about proofs in our sequent calculus.

Theorem 1 *If $\Gamma \longrightarrow \Theta$ has a proof of length l (height h) and Γ', Θ' are obtained from Γ and Θ by replacing some formulas by one of their alphabetic variants, then $\Gamma' \longrightarrow \Theta'$ also has a proof of length l (height h).*

$$\frac{B, D, \Delta \longrightarrow \Theta}{B \wedge D, \Delta \longrightarrow \Theta}\ \wedge\text{-L} \qquad\qquad \frac{\Delta \longrightarrow B \qquad \Delta \longrightarrow D}{\Delta \longrightarrow B \wedge D}\ \wedge\text{-R}$$

$$\frac{B, \Delta \longrightarrow \Theta \qquad D, \Delta \longrightarrow \Theta}{B \vee D, \Delta \longrightarrow \Theta}\ \vee\text{-L}$$

$$\frac{\Delta \longrightarrow B}{\Delta \longrightarrow B \vee D}\ \vee\text{-R} \qquad\qquad \frac{\Delta \longrightarrow D}{\Delta \longrightarrow B \vee D}\ \vee\text{-R}$$

$$\frac{\Delta \longrightarrow B}{\sim B, \Delta \longrightarrow \Theta}\ \sim\text{-L} \qquad\qquad \frac{B, \Delta \longrightarrow}{\Delta \longrightarrow \sim B}\ \sim\text{-R}$$

$$\frac{\Delta \longrightarrow B \qquad D, \Delta \longrightarrow \Theta}{B \supset D, \Delta \longrightarrow \Theta}\ \supset\text{-L} \qquad \frac{B, \Delta \longrightarrow D}{\Delta \longrightarrow B \supset D}\ \supset\text{-R}$$

$$\frac{[t/x]P, \Delta \longrightarrow \Theta}{\forall x P, \Delta \longrightarrow \Theta}\ \forall\text{-L} \qquad\qquad \frac{\Delta \longrightarrow [t/x]P}{\Delta \longrightarrow \exists x P}\ \exists\text{-R}$$

$$\frac{[c/x]P, \Delta \longrightarrow \Theta}{\exists x P, \Delta \longrightarrow \Theta}\ \exists\text{-L} \qquad\qquad \frac{\Delta \longrightarrow [c/x]P}{\Delta \longrightarrow \forall x P}\ \forall\text{-R}$$

Figure 1: Inference Figure Schemata

Proof. By induction on the length (height) of the given proof. We need to observe that (a) two atomic formulas are alphabetic variants only if they are identical, (b) the constants appearing in alphabetic variants are identical, and (c) if $\forall x P$ and $\forall y P'$ are alphabetic variants then $[t/x]P$ and $[t/y]P'$ must also be alphabetic variants and similarly for existential quantifiers.

$\square$

Theorem 2 *If* $\Gamma \longrightarrow \Theta$ *has a proof of length* l *(height* h*), then, for any substitution* σ*, the sequent* $\sigma(\Gamma) \longrightarrow \sigma(\Theta)$ *has a proof of length* l *(height* h*).*

Proof. The intuitive idea is to apply the substitution to every sequent in the given proof. However, the quantifier introduction rules require some care. The $\forall$-L and the $\exists$-R still work fine. For example, consider that the original proof had the following rule in it:

$$\frac{[t/x]P, \Delta \longrightarrow \Theta}{\forall x P, \Delta \longrightarrow \Theta}$$

Now, for any $y \notin \mathcal{F}(\sigma)$, $\forall y\, \sigma([y/x]P)$ is an alphabetic variant of $\sigma(\forall x P)$. Further, for such a choice of y, $\sigma([t/x]P)$ is an alphabetic variant of $[\sigma(t)/y]\sigma([y/x]P)$. Using Theorem 1 and the proof for $\sigma([t/x]P), \sigma(\Delta) \longrightarrow \sigma(\Theta)$ (whose existence follows from the induction hypothesis), we see that $[\sigma(t)/y]\sigma([y/x]P), \sigma(\Delta) \longrightarrow \sigma(\Theta)$ has a proof. Using a $\forall$-L rule below this, we obtain a proof for $\forall y\, \sigma([y/x]P), \sigma(\Delta) \longrightarrow \sigma(\Theta)$. Using Theorem 1 again, we get a proof for $\sigma(\forall x P), \sigma(\Delta) \longrightarrow \sigma(\Theta)$. Finally, the length (height) of this proof must be the same as that of the proof we started with, *i.e.*, the proof for $\forall x P, \Delta \longrightarrow \Theta$.

For $\exists$-L and $\forall$-R we have the additional problem that the constant being generalized upon may appear in the substitution. For this purpose, we must rename these constants to be distinct from all those in the substitution prior to performing the transformation outlined in this proof. This can be done, for instance, by using the method outlined in [5].

$\square$

Theorem 3 *If* $\Gamma \longrightarrow \Theta$ *has a proof of length* l *(height* h*) and* Γ' *and* Θ' *are obtained from* Γ *and* Θ *by replacing certain constants in a consistent manner by other constants or variables not bound in the formulas in* $\Gamma \cup \Theta$*, then* $\Gamma' \longrightarrow \Theta'$ *also has a proof of length* l *(height* h*).*

Proof. By an obvious induction on the length (height) of the proof. We omit the details but only note that the argument is similar to that employed for renaming the constants generalized on by $\forall$-R and $\exists$-L.

$\square$

3 First-Order Hereditary Harrop Formulas

We are interested in the G- and D-formulas defined by the following syntax rules in which we assume A represents atomic formulas:

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

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

The D-formulas defined here are called (first-order) hereditary Harrop formulas [17]. These formulas define a logic programming language in the following sense: a G-formula can be thought of as a query or goal, a finite set of closed D-formulas constitutes a program, and the process of answering a query consists of constructing an intuitionistic proof of the existential closure of the query from the given program. In keeping with this interpretation, we shall refer to a G-formula as a goal formula and to a D-formula as a program clause. The proof-theoretic properties of G- and D-formulas that justify this identification and the usefulness of the logic programming language thus described have, as we have mentioned already, been explored at length elsewhere and we do not dwell on these aspects here.

Our objective in this paper is that of providing the basis for an interpreter for the logic programming language described above. We do this in a later section by describing a non-deterministic procedure for determining whether a proof exists for a goal formula from a finite set of program clauses. In proving the completeness of this procedure, we need certain relationships between lengths of proofs of goal formulas. We observe these relationships in Theorem 5 below. First we define the notions of an instance and an elaboration of a D-formula.

Definition 2. The elaboration of a program clause D, denoted by $elab(D)$, is the set of formulas defined as follows:

(i) If D is an atomic formula or of the form $G \supset A$, then it is $\{D\}$.

(ii) If D is $D_1 \wedge D_2$, then it is $elab(D_1) \cup elab(D_2)$.

(iii) If D is $\forall x D'$ then it is $\{\forall x D'' | D'' \in elab(D')\}$.

Evidently all the formulas in $elab(D)$ are of the form $\forall x_1 \ldots \forall x_n A$ or $\forall x_1 \ldots \forall x_n (G \supset A)$, where A is atomic and G is a goal formula. An *instance* of such a formula is any formula that can be written as $\theta(A)$ or $\theta(G \supset A)$ where θ is a substitution whose domain is $\{x_1, \ldots, x_n\}$. The instances of a D-formula are all the instances of the formulas in $elab(D)$. The elaboration of $\mathcal{P}$, a finite set of program clauses, is the union of the elaborations of the formulas in $\mathcal{P}$. This collection is denoted by $elab(\mathcal{P})$.

There is an alternative characterization of the set of instances of a program clause that is useful in the proof of Theorem 5. This is provided in the following lemma.

Lemma 4 *Let D be a program clause. Then a formula D' is an instance of D if and only if one of the following is true:*

(i) D is of the form A or $G \supset A$ and D' is identical to D.

(ii) D is $D_1 \wedge D_2$ and D' is an instance of either D_1 or D_2.

(iii) D is $\forall x D_1$ and D' is an instance of $[t/x]D_1$ for some choice of term t.

Proof. By an obvious induction on the structure of D.

□

Now we observe the following property concerning the lengths of proofs.

Theorem 5 *Let $\mathcal{P}$ be a finite set of program clauses and let G be a goal formula such that $\mathcal{P} \longrightarrow G$ has a proof of length l.*

(1) If G is atomic, it is either identical to an instance of a formula in $\mathcal{P}$ or there is an instance $G' \supset G$ of some formula in $\mathcal{P}$ such that $\mathcal{P} \longrightarrow G'$ has a proof of length less than l.

(2) If G is $G_1 \wedge G_2$, then $\mathcal{P} \longrightarrow G_1$ and $\mathcal{P} \longrightarrow G_2$ have proofs of length less than l.

(3) If G is $G_1 \vee G_2$, then, for $i = 1$ or $i = 2$, the sequent $\mathcal{P} \longrightarrow G_i$ has a proof of length less than l.

(4) If G is $\exists x G_1$, then there is a term t such that $\mathcal{P} \longrightarrow [t/x]G_1$ has a proof of length less than l.

(5) If G is $D \supset G_1$, then $D, \mathcal{P} \longrightarrow G_1$ has a proof of length less than l.

(6) If G is $\forall x G_1$, then there is a constant c that does not appear in $\mathcal{P}$ or in G_1 such that $\mathcal{P} \longrightarrow [c/x]G_1$ has a proof of length less than l.

Proof. By induction on the length of the proof.

If the length is 1, then G is atomic and identical to some formula in $\mathcal{P}$ and thus the theorem must be true.

If the length is greater than 1, we consider by cases the last rule used in the proof. The claim is obviously true if this rule pertains to the formula in the succedent. Thus we only need to consider $\supset$-L, $\wedge$-L and $\forall$-L.

If the last rule is an $\supset$-L, then it has the form

$$\frac{\Delta \longrightarrow G' \qquad A, \Delta \longrightarrow G}{G' \supset A, \Delta \longrightarrow G}$$

The upper sequents have the form required by the theorem and the hypothesis applies to them. We now consider the cases for the structure of G and show that the theorem holds in each case. Suppose G is of the form $D \supset G_1$. We observe first that $D, \Delta \longrightarrow G'$ must have a proof of the same length as $\Delta \longrightarrow G'$; we obtain one for the former by simply affixing D to the antecedent of each sequent in the proof for the latter and possibly "renaming" the constant generalized upon in some of the $\forall$-R and $\exists$-L rules. By the hypothesis, $D, A, \Delta \longrightarrow G_1$ has a shorter proof than that for $A, \Delta \longrightarrow G$. Putting the proofs for $D, \Delta \longrightarrow G'$ and $D, A, \Delta \longrightarrow G_1$ together using a $\supset$-L rule, we obtain a proof satisfying the theorem. Similar arguments can be supplied for the other cases when G is non-atomic; the case where G is of the form $\forall x G_1$ may require a renaming of a constant in a proof but this is, by now, straightforward. If G is atomic, then the hypothesis applied to $A, \Delta \longrightarrow G$ yields the theorem in all cases except when G is identical to A. However, in the last case, the theorem follows by observing that $G' \supset A \in \mathcal{P}$ and $\mathcal{P} \longrightarrow G'$ must have a proof of length identical to that for $\Delta \longrightarrow G'$.

The arguments for $\wedge$-L and $\forall$-L follow a similar pattern. The only additional observation to be made is that the characterization of instances provided in Lemma 4 is designed for these cases.

$\square$

We shall need a slightly stronger observation than is contained in the above theorem for universally quantified G-formulas. This is stated below.

Corollary 1 *Let $\mathcal{P}$ be a finite set of D-formulas and let $\forall x G$ be a goal formula such that the sequent $\mathcal{P} \longrightarrow \forall x G$ has a proof of length l. Then, for any constant c, $\mathcal{P} \longrightarrow [c/x]G$ has a proof of length less than l.*

Proof. From Theorem 5 we see that $\mathcal{P} \longrightarrow [c'/x]G$ has a proof of length less than l for some constant c' that does not appear in $\mathcal{P}$ or G. We now invoke Theorem 3 to reach the desired conclusion.

□

Our interest, as we have mentioned before, is in a procedure for determining if a proof exists for a given goal formula from a set of program clauses. Theorem 5 contains information that might be used in the design of a such procedure. In particular, the theorem indicates the manner in which the search for a proof for a given goal formula can be reduced to a search for simpler proofs for certain other formulas. When a non-atomic goal is encountered, the suggested step is one of goal simplification. The particular steps to be carried out are apparent when the top-level logical symbol is a propositional connective; the procedure may conduct a conjunctive search, a disjunctive search or a search based on augmenting the program clauses depending on whether this symbol is $\wedge$, $\vee$ or $\supset$. However, there is some question as to what should be done when the top-level symbol in the goal is a quantifier. In particular, an instantiation term needs to be picked when this symbol is an existential quantifier and there is little information as to what this term should be. A suggestion that is in keeping with the technique used with Horn clauses is to delay the instantiation. The quantified variable may be replaced by a "place-holder", usually referred to as a logic variable. In implementing the condition pertaining to atomic goals, use may be made of the operation of unification, in this process also determining instantiations for logic variables.

While the above suggestions appear to provide the structure for a satisfactory procedure, care needs to be exercised in their actual implementation to ensure correctness. The need for caution arises from the presence of universal quantifiers. The simplification step that is indicated for a quantifier of this kind is that of instantiating it with a new constant and then searching for a proof for the resulting goal. Notice that the newness of the constant is crucial for the correctness of the search strategy: this is a requirement on the universal generalization step that produces a proof for the quantified formula from a proof of the formula that results from the simplification. This newness condition places a constraint on the instantiations that are permitted for the logic variables appearing in the formula. To illustrate this situation, let us assume that we are searching for a proof of the goal formula $\exists x \forall y (p\ x\ y)$ from a set of program clauses containing only the formula $\forall x (p\ x\ x)$; we assume that p is a predicate symbol in these formulas and adopt the convention of using lower case letters to denote constants and bound variables here and below. A cursory inspection of the formulas in question reveals that the attempt to construct a proof should not succeed. Following the "recipe" described above, we proceed to simplify the given goal formula first to $\forall y (p\ X\ y)$ and then to $(p\ X\ c)$, where X denotes a logic variable and c is a new constant. Now this formula unifies with an instance of the given program clause and we might therefore be tempted to conclude that the attempted proof search is successful. This conclusion is obviously erroneous, and a closer look at the instantiation for X reveals the source of the problem: our "solution" requires instantiating X with the constant c, thereby constructing a "proof" for $\forall y (p\ c\ y)$ from one for $(p\ c\ c)$.

The problem discussed above indicates the need for some method for constraining the permitted substitutions for logic variables. In the context of classical logic the idea of Skolemization is generally used for this purpose. Within this context, formulas are converted into a prenex normal form and universal quantifiers are then instantiated by a (new) function of the existentially quantified variables whose quantifier scope governs them. Thus, the goal formula considered above would be converted into the form $\exists x(p\ x\ (f\ x))$, where f is a new function symbol. Such a conversion process, when used with the recipe discussed above, makes logic variables appear within terms that should not appear in their instantiations; as a particular example, our Skolemized goal will be simplified to $(p\ X\ (f\ X))$. The notion of "occurs-checking" that is part of the unification operation now ensures that the necessary constraints on instantiation terms are satisfied.

The preprocessing phase that solves the problem within classical logic can unfortunately not be employed within intuitionistic logic. An appreciation of this fact might be obtained by considering the formula $((\forall x(p\ x) \supset q) \supset \exists x((p\ x) \supset q))$. This formula is a goal formula as defined in this section. One interesting observation to make about this formula is that it cannot be converted into an equivalent prenex normal form. The reason for this is that certain logical equivalences needed in the conversion process do not hold in intuitionistic logic. In particular, if $F_1(x)$ and F_2 respectively represent formulas in which x does and does not appear free, then neither $(\forall x F_1(x) \supset F_2)$ and $\exists x(F_1(x) \supset F_2)$ nor $(F_2 \supset \exists x F_1(x))$ and $\forall x(F_2 \supset F_1(x))$ are intuitionistically equivalent. One might, nevertheless, attempt to convert a formula into a Skolemized form by instantiating essential universal quantifiers by Skolem functions of the essential existentially quantified variables whose quantifier scope governs them. Applied to the formula at hand, this process would yield the formula $(((p\ c) \supset q) \supset \exists x((p\ x) \supset q))$, where c is assumed to be a new constant. However, such a "Skolemization" process is not sound for intuitionistic logic. For example, for the formula considered here, it is easily verified that the Skolemized version is provable using the sequent calculus presented in the last section whereas the original formula is not[2].

Although static Skolemization is not possible, a dynamic version of Skolemization as described in [4] can be used. The manner in which this would work is the following. Together with the goal formula to be proved, we maintain a list of all the logic variables that have been introduced in the search conducted up to that point. Now imagine that the top-level symbol in the goal formula is a universal quantifier. The search is then continued by instantiating the quantifier by a Skolem function of the logic variables present in the list. The use of such a function ensures, as before, that the logic variables present at that point cannot be instantiated by a term that contains the instantiation for the universal quantifier. Thus, satisfaction of the "newness" constraint that goes with the universal quantifier is ensured. As an illustation of this idea, we may consider a search for a proof for the formula presented above, *i.e.*, for $((\forall x(p\ x) \supset q) \supset \exists x((p\ x) \supset q))$. Using the recipe suggested, this would reduce to finding a proof for $\exists x((p\ x) \supset q)$ from the program clause $(\forall x(p\ x) \supset q)$; the logic variable list is empty at this point. At this stage, the existential quantifier is encountered leading to a search for a proof for $((p\ X) \supset q)$ from the program clause $(\forall x(p\ x) \supset q)$ with the logic variable list containing the sole variable X. Iterating through a few more steps, the goal becomes one of finding a proof for $\forall x(p\ x)$ from the set of program clauses $\{(p\ X), (\forall x(p\ x) \supset q)\}$ in the context of the same logic variable list. The goal formula is at this stage reduced to $(p\ (f\ X))$ where f is a new function symbol. It is easily seen that the search now reaches a dead-end and this leads to the conclusion that no proof exists for the original formula.

Although the scheme outlined above functions correctly, it involves keeping track of a potentially

[2]It follows from this that a Herbrand-like theorem does not hold for hereditary Harrop formulas, contrary to the claim in [17]. A deeper analysis reveals that the source of the problem is that, in contrast to the classical case, certain propositional inference rules — in this case the $\supset$ -L and $\supset$ -R rules — cannot be permuted in our intuitionistic sequent calculus. This observation, coincidentally using the same example, is also made in [25].

long list of logic variables and forming Skolem functions of these variables when universal quantifiers are encountered. A direct implementation of this scheme would therefore be rather cumbersome[3]. However, an alternative scheme that has an efficient implementation can be used. Under this scheme, instead of using Skolem functions, we think of tagging logic variables with the set of constants that may appear in terms instantiating them; this set can then be used in a modified occurs-check to be performed in the course of unification. Fortunately, the different sets of constant symbols constitute a hierarchy of universes and a practical realization of this idea can be obtained by using a numerical label with each constant and logic variable. The level 0 universe consists of all the constant symbols that appear in the program clauses and the original goal. These symbols may be labelled by 0 to indicate their position in the hierarchy. Each time a universal quantifier is encountered, the "universe index" is increased by 1 and a new constant labelled with this index is introduced; thus, the universe at this level consists of the new constant and those in the universes below it. When an existential quantifier is encountered, it is instantiated by a logic variable labelled with the current value of the universe index. The labels are then used in the following fashion: the process of unification culminates with trying to instantiate a logic variable with a term. In the present context this would amount to setting a variable X with label i to a term t. This instantiation is only permitted if t does not contain any constants with a label greater than i.

The actual realization of the latest scheme thus depends on a unification process that respects the constraints represented by labels on constants and variables. We describe such a notion of unification in the next section. We then use this in a precise description of the proof procedure outlined above and in a proof of its correctness.

4 Labelled Unification

The interpreter that we describe in the next section will have to consider unifying terms under certain restrictions pertaining to substitutions. These restrictions are obtained from the labels on constants and variables that are given by the labelling functions alluded to in Section 2. As mentioned, the behavior of these functions is fixed on the constants but may vary on the variables. The particular behavior of any given function will not concern us in this section but will be relevant to the discussions in Section 5.

Definition 3. Let $\theta = \{\langle x_i, t_i\rangle | 1 \leq i \leq n\}$ be a substitution and let $\mathcal{L}$ be a labelling function. θ is *proper* with respect to $\mathcal{L}$ if, for $1 \leq i \leq n$, it is the case that $\mathcal{L}(c) \leq \mathcal{L}(x_i)$ for every constant c appearing in t_i. The labelling *induced* by θ from $\mathcal{L}$ is then the labelling function $\mathcal{L}'$ whose behavior on variables is given as follows:

$$\mathcal{L}'(x) = min(\{\mathcal{L}(x)\} \cup \{\mathcal{L}(x_i) \mid \langle x_i, t_i\rangle \in \theta \text{ and } x \text{ appears in } t_i\}).$$

As mentioned already, the behavior of labelling functions on constants is fixed, and hence $\mathcal{L}'$ is identical to $\mathcal{L}$ with respect to these symbols.

[3]There is a dual to Skolemization, called raising in [14] and lifting in [23], that can be used in a higher-order context. Raising requires maintaining a list of universal, as opposed to existential, quantifiers encountered in a proof search. In practical contexts this entails less bookkeeping and Skolemization also has other ills in the higher-order context [14]. However, even raising appears not to be an operation at low enough a level to be incorporated into an abstract machine for λProlog. The scheme that is eventually used here seems to be such an operation and also captures directly the constraints Skolemization and raising are designed to capture.

Definition 4. The composition of two substitutions is the composition of these when viewed as functions. The composition of θ_1 and θ_2 will be written as $\theta_1 \circ \theta_2$, *i.e.*, for any term t, $\theta_1 \circ \theta_2(t) = \theta_1(\theta_2(t))$. This operation is easily seen to be associative, so there is no essential ambiguity in an expression of the form $\theta_1 \circ \theta_2 \circ \theta_3$. A substitution θ_1 is more general than θ_2 relative to a labelling function $\mathcal{L}$ if θ_1 and θ_2 are proper with respect to $\mathcal{L}$ and there is a substitution σ that is proper with respect to the labelling induced by θ_1 from $\mathcal{L}$ such that $\theta_2 = \sigma \circ \theta_1$.

Definition 5. We refer to a finite set of pairs of terms or atomic formulas as a disagreement set. Let $T = \{\langle t_i, s_i\rangle | 1 \le i \le n\}$ be a disagreement set and let $\mathcal{L}$ be a labelling function. A *unifier* for T under $\mathcal{L}$ is a substitution θ that is proper with respect to $\mathcal{L}$ and such that $\theta(t_i) = \theta(s_i)$ for $1 \le i \le n$. In the case that T is a singleton containing the pair $\langle t, s\rangle$, we shall refer to a unifier for T as a unifier of t and s. A most general unifier for T under $\mathcal{L}$ is a unifier θ that is more general as a substitution that any other unifier relative to $\mathcal{L}$.

Although we have defined the notion of a most general unifier for a unification problem posed by disagreement set T and a labelling function $\mathcal{L}$, it is not clear that this notion makes sense. We show that it does by outlining a procedure that finds a most general unifier for a disagreement set whenever the set has a unifier. This procedure is modelled on the first nondeterministic algorithm presented in [9]. No attention is given to efficiency at this stage; the only purpose is to show the existence and computability of most general unifiers.

We begin by defining the following transformations on disagreement sets and labelling functions. The former is given generically by T below and the latter by $\mathcal{L}$.

(1) *Term Reduction.* Let $\langle (f\ t_1\ \ldots\ t_n), (f\ s_1\ \ldots\ s_n)\rangle \in T$. Then transform T by replacing this pair by the pairs $\langle t_1, s_1\rangle, \ldots, \langle t_n, s_n\rangle$. The labelling function is preserved.

(2) *Variable Elimination.* Let $\langle x, t\rangle$ be a pair in T with x being a variable. Transform T by applying $\{\langle x, t\rangle\}$ as a substitution to all the other pairs in T. The labelling function is preserved.

(3) *Label Adjustment.* Let $\langle x, t\rangle$ be a pair in T with x being a variable. Transform $\mathcal{L}$ into the labelling induced from $\mathcal{L}$ by $\{\langle x, t\rangle\}$.

We now observe the following properties that go towards showing that the set of unifiers is preserved under the transformations described above.

Lemma 6 *Let T be a disagreement set and let $\langle (f\ t_1\ \ldots\ t_n), (g\ s_1\ \ldots\ s_m)\rangle \in T$. If $f \neq g$ then T has no unifiers. Otherwise the set obtained from T by applying term reduction has the same set of unifiers as does T. Furthermore, these observations are true relative to any labelling function.*

Proof. If $f \neq g$, no substitution can make the two terms in the given pair identical. Otherwise a substitution makes the two terms identical if and only if it makes the pairs produced by term reduction identical. Since the labelling function is preserved, the set of proper substitutions remains unchanged.

□

Lemma 7 *Let T be a disagreement set containing the pair $\langle x, t\rangle$ where x is a variable and let $\mathcal{L}$ be a labelling function. If x occurs in t and $t \neq x$ or if there is a constant c in t such that $\mathcal{L}(x) < \mathcal{L}(c)$*

then T has no unifiers relative to $\mathcal{L}$. Otherwise let T' be obtained from T by applying variable elimination with respect to $\langle x, t\rangle$. Then the set of unifiers for T' relative to $\mathcal{L}$ is identical to that for T relative to $\mathcal{L}$.

Proof. Let θ be a unifier for T relative to $\mathcal{L}$. Then $\langle x, \theta(t)\rangle \in \theta$. If x occurs in t, this would be impossible since $\theta(t)$ must be finite. Further, any constant occurring in t must occur in $\theta(t)$ as well. Thus if there is a constant c in t such that $\mathcal{L}(x) < \mathcal{L}(c)$, then θ cannot be proper with respect to $\mathcal{L}$. Finally, observe that if s' is obtained from s by replacing x by t and $\theta(x) = \theta(t)$, then $\theta(s) = \theta(s')$. Thus the substitutions making the elements in each pair in T identical must be the same as those making the elements of each pair in T' identical.

□

Lemma 8 *Let T be a disagreement set, let $\mathcal{L}$ be a labelling function and let $\mathcal{L}'$ be the new labelling function obtained by applying label adjustment relative to some pair $\langle x, t\rangle$ in T. Then T has the same sets of unifiers relative to $\mathcal{L}'$ as it does relative to $\mathcal{L}$.*

Proof. Any substitution that is proper with respect to $\mathcal{L}'$ must clearly be proper with respect to $\mathcal{L}$ as well. Hence any unifier for T relative to $\mathcal{L}'$ must also be one relative to $\mathcal{L}$. In the converse direction, let θ be a substitution that is proper with respect to $\mathcal{L}$ but not $\mathcal{L}'$. Then for some variable y occurring in t there must be a pair $\langle y, s\rangle$ in θ with a constant c occurring in s such that $\mathcal{L}(x) < \mathcal{L}(c)$. But then θ cannot be a unifier for T relative to $\mathcal{L}$: if it were, $\langle x, \theta(t)\rangle \in \theta$ and thus θ is not proper with respect to $\mathcal{L}$.

□

We think of a disagreement set T as being in *solved form* in the context of a labelling function $\mathcal{L}$ if the following conditions are satisfied:

(i) the first component of each pair in T is a variable,

(ii) a variable occurring as the first component of any pair in T occurs only there, and

(iii) for each pair $\langle x, t\rangle \in T$ it is the case that $\mathcal{L}(a) \leq \mathcal{L}(x)$ for all constants and variables a occurring in t.

Lemma 9 *A disagreement set T that is in solved form in the context of a labelling function $\mathcal{L}$ is its own most general unifier relative to $\mathcal{L}$.*

Proof. It is obvious that T is its own unifier relative to $\mathcal{L}$. The labelling induced by T from $\mathcal{L}$ is $\mathcal{L}$. If θ is any other unifier, then it must be the case that $\theta = \theta \circ T$. Finally, θ is by definition proper with respect to $\mathcal{L}$.

□

We now present a nondeterministic algorithm for finding most general unifiers in our context.

Algorithm 1
Given a disagreement set T and a labelling function $\mathcal{L}$, perform the following transformations. If none applies, return the resulting set of term pairs as the most general unifier.

(1) Replace any pair in T of the form $\langle t, x\rangle$ where x is a variable and t is not by the pair $\langle x, t\rangle$.

(2) Remove from T any pair of the form $\langle x, x\rangle$ where x is a variable.

(3) Pick any pair in T of the form $\langle t, s\rangle$ where t and s are not variables. If the root function symbols of t and s are distinct, declare non-unifiability and stop. Otherwise apply term reduction.

(4) Pick any pair in T of the form $\langle x, t\rangle$ where x is a variable that occurs somewhere else in T and t is distinct from x. If x occurs in t or if $\mathcal{L}(x) < \mathcal{L}(c)$ for some constant c occurring in t, declare non-unifiability and stop. Otherwise apply variable elimination.

(5) Pick a pair in T of the form $\langle x, t\rangle$ where x is a variable and for some variable y occurring in t it is the case that $\mathcal{L}(x) < \mathcal{L}(y)$. Apply label adjustment.

The above algorithm purportedly produces a most general unifier or determines that the given set has no unifiers under the corresponding labelling function. We show now that this is in fact the case, *i.e.*, that the algorithm is correct in its judgement.

Theorem 10 *Algorithm 1 will terminate for any given disagreement set T and labelling function $\mathcal{L}$. If it terminates after declaring non-unifiability, T has no unifiers relative to $\mathcal{L}$. Otherwise the set returned is the most general unifier.*

Proof. The argument for termination is similar to that in [9]. Specifically, we associate with each disagreement set T and labelling function $\mathcal{L}$ a quadruple of natural numbers $\langle n_1, n_2, n_3, n_4\rangle$ as follows. The number n_1 is a count of the number of variables that do not occur only once as the first element of a pair in T. The second number n_2 is the count of the number of occurrences of function symbols in T. The third number n_3 is a count of the number of pairs in T of the form $\langle x, x\rangle$ and $\langle t, x\rangle$ where x is a variable and t is not. The fourth number n_4 is the summation of $\mathcal{L}(x)$ over all the variables x occurring in T. We now assume an ordering on disagreement sets and labelling functions given by the lexicographic ordering on the associated quadruples. It is then easily shown that each application of the steps in Algorithm 1 produces a disagreement set and a labelling function that is smaller with respect to this ordering. Termination follows, the ordering being well-founded.

Using the Lemmas 6, 7 and 8 and an induction on the number of steps applied, we see that for the disagreement set T' and corresponding labelling function $\mathcal{L}'$ produced at any intermediate stage by the algorithm, the set of unifiers for T' relative to $\mathcal{L}'$ is identical to the set of unifiers for T relative to $\mathcal{L}$. The correctness of the algorithm when it declares non-unifiability follows from this by once again using Lemmas 6, 7 and 8. If the algorithm succeeds after producing the set T' with the associated labelling function $\mathcal{L}'$, it follows from Lemma 9 that (a) T' is a unifier and (b) for any other unifier θ there is a substitution σ that is proper with respect to $\mathcal{L}'$ such that $\theta = \sigma \circ T'$. But it is easily seen that σ must be proper with respect to the labelling induced by T' from $\mathcal{L}$ and thus T' is a most general unifier for T relative to $\mathcal{L}$.

□

5 A Non-Deterministic Proof Procedure

We now wish to describe a procedure for determining whether a proof exists for an instance of a goal formula from a finite set of D-formulas. The procedure we describe will operate in a context provided by a set of constants, a set of variables and a labelling function. Its purpose will be to transform a set of tuples of the form $\langle G, \mathcal{P}, I\rangle$ where G is a goal formula, $\mathcal{P}$ is a finite set of program clauses and I is a natural number that, intuitively, bounds the labels of the constants and variables appearing in G and $\mathcal{P}$. We assume hereafter that $\mathcal{G}$, used perhaps with subscripts, denotes a collection of such tuples, that $\mathcal{C}$ and $\mathcal{V}$ similarly denote sets of constants and variables, that $\mathcal{L}$ denotes a labelling function and that θ is a syntactic variable for a substitution. A state in our procedure is defined by a tuple of the form $\langle \mathcal{G}, \mathcal{C}, \mathcal{V}, \mathcal{L}, \theta\rangle$ and the transformation that this procedure affects on states is given by the following relation between them.

Definition 6. A tuple $\langle \mathcal{G}_2, \mathcal{C}_2, \mathcal{V}_2, \mathcal{L}_2, \theta_2\rangle$ is derived from another tuple $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1\rangle$ if one of the following holds:

(i) $\langle G_1 \wedge G_2, \mathcal{P}, I\rangle \in \mathcal{G}_1$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{\langle G_1 \wedge G_2, \mathcal{P}, I\rangle\}) \cup \{\langle G_1, \mathcal{P}, I\rangle, \langle G_2, \mathcal{P}, I\rangle\}$, $\mathcal{C}_2 = \mathcal{C}_1$, $\mathcal{V}_2 = \mathcal{V}_1$, $\mathcal{L}_2 = \mathcal{L}_1$ and $\theta_2 = \emptyset$.

(ii) $\langle G_1 \vee G_2, \mathcal{P}, I\rangle \in \mathcal{G}_1$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{\langle G_1 \vee G_2, \mathcal{P}, I\rangle\}) \cup \{\langle G_i, \mathcal{P}, I\rangle\}$ for $i = 1$ or $i = 2$, $\mathcal{C}_2 = \mathcal{C}_1$, $\mathcal{V}_2 = \mathcal{V}_1$, $\mathcal{L}_2 = \mathcal{L}_1$ and $\theta_2 = \emptyset$.

(iii) $\langle \exists x G, \mathcal{P}, I\rangle \in \mathcal{G}_1$ and, for some variable w not in $\mathcal{V}_1$,

$$\mathcal{G}_2 = (\mathcal{G}_1 - \{\langle \exists x G, \mathcal{P}, I\rangle\}) \cup \{\langle [w/x]G, \mathcal{P}, I\rangle\},$$

$\mathcal{C}_2 = \mathcal{C}_1$, $\mathcal{V}_2 = \mathcal{V}_1 \cup \{w\}$, $\mathcal{L}_2$ is like $\mathcal{L}_1$ except that $\mathcal{L}_2(w) = I$, and $\theta_2 = \emptyset$.

(iv) $\langle D \supset G, \mathcal{P}, I\rangle \in \mathcal{G}_1$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{\langle D \supset G, \mathcal{P}, I\rangle\}) \cup \{\langle G, \mathcal{P} \cup \{D\}, I\rangle\}$, $\mathcal{C}_2 = \mathcal{C}_1$, $\mathcal{V}_2 = \mathcal{V}_1$, $\mathcal{L}_2 = \mathcal{L}_1$ and $\theta_2 = \emptyset$.

(v) $\langle \forall x G, \mathcal{P}, I\rangle \in \mathcal{G}_1$ and for some constant c not in $\mathcal{C}_1$ and such that $\mathcal{L}_1(c) = I + 1$

$$\mathcal{G}_2 = (\mathcal{G}_1 - \{\langle \forall x G, \mathcal{P}, I\rangle\}) \cup \{\langle [c/x]G, \mathcal{P}, I+1\rangle\}$$

and $\mathcal{C}_2 = \mathcal{C}_1 \cup \{c\}$, $\mathcal{V}_2 = \mathcal{V}_1$, $\mathcal{L}_2 = \mathcal{L}_1$, and $\theta_2 = \emptyset$.

(vi) Let $\langle A, \mathcal{P}, I\rangle \in \mathcal{G}_1$, let $\forall x_1 \ldots \forall x_n A' \in elab(\mathcal{P})$ and let $\theta = \{\langle x_1, w_1\rangle, \ldots, \langle x_n, w_n\rangle\}$ be a renaming substitution such that, for $1 \leq i \leq n$, w_i is a distinct variable not in $\mathcal{V}_1$. Then A and $\theta(A')$ are unifiable with a most general unifier σ relative to the labelling function $\mathcal{L}'$ which is like $\mathcal{L}_1$ except that it maps each w_i to I and $\mathcal{G}_2 = \sigma(\mathcal{G}_1 - \{\langle A, \mathcal{P}, I\rangle\})$, $\mathcal{C}_2 = \mathcal{C}_1$, $\mathcal{V}_2 = \mathcal{V}_1 \cup \{w_1, \ldots, w_n\}$, $\theta_2 = \sigma$ and $\mathcal{L}_2$ is the labelling induced by σ from $\mathcal{L}'$.

(vii) Let $\langle A, \mathcal{P}, I\rangle \in \mathcal{G}_1$, let $\forall x_1 \ldots \forall x_n (G \supset A') \in elab(\mathcal{P})$ and let $\theta = \{\langle x_1, w_1\rangle, \ldots, \langle x_n, w_n\rangle\}$ be a renaming substitution such that, for $1 \leq i \leq n$, w_i is a distinct variable not in $\mathcal{V}_1$. Also let A and $\theta(A')$ be unifiable with a most general unifier σ relative to the labelling function $\mathcal{L}'$ which is like $\mathcal{L}_1$ except that it maps each w_i to I. Then $\mathcal{G}_2 = \sigma((\mathcal{G}_1 - \{\langle A, \mathcal{P}, I\rangle\}) \cup \{\langle \theta(G), \mathcal{P}, I\rangle\})$, $\mathcal{C}_2 = \mathcal{C}_1$, $\mathcal{V}_2 = \mathcal{V}_1 \cup \{w_1, \ldots, w_n\}$, $\theta_2 = \sigma$ and $\mathcal{L}_2$ is the labelling induced by σ from $\mathcal{L}'$.

Definition 7. A sequence $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1 \rangle, \ldots, \langle \mathcal{G}_n, \mathcal{C}_n, \mathcal{V}_n, \mathcal{L}_n, \theta_n \rangle$ is a *derivation* sequence if the $(i+1)$th tuple in it is derived from the ith tuple. Such a derivation sequence terminates if no tuple can be derived from $\langle \mathcal{G}_n, \mathcal{C}_n, \mathcal{V}_n, \mathcal{L}_n, \theta_n \rangle$. The sequence terminates successfully if $\mathcal{G}_n = \emptyset$.

Definition 8. Let G be a goal formula and $\mathcal{P}$ be a finite set of closed program clauses such that the label associated with each constant in these formulas is 0. Also let $\mathcal{G}_1 = \{\langle G, \mathcal{P}, 0 \rangle\}$, let $\mathcal{C}_1$ and $\mathcal{V}_1$ be, respectively, the set of constants and the set of free variables appearing in the formulas in $\{G\} \cup \mathcal{P}$, let $\mathcal{L}_1$ be the constant 0 valued function over $\mathcal{V}_1$ and let $\theta_1 = \emptyset$. Then a derivation sequence of the form $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1 \rangle, \ldots, \langle \mathcal{G}_n, \mathcal{C}_n, \mathcal{V}_n, \mathcal{L}_n, \theta_n \rangle$ is said to be a *derivation for* G relative to $\mathcal{P}$. It is a *derivation of* G from $\mathcal{P}$ if it is successfully terminated and, in this case, its associated *answer substitution* is the restriction of $\theta_n \circ \cdots \circ \theta_1$ to the free variables of G, *i.e.*, to $\mathcal{V}_1$.

A (non-deterministic) procedure for determining if a proof exists for an instance of a goal formula G from a set of closed program clauses $\mathcal{P}$ can now be described as one that searches for a derivation of G from $\mathcal{P}$. The correctness and adequacy of such an identification is demonstrated through Theorems 13 and 15 below. First we introduce a definition and observe a property that will be useful in proving these theorems.

Definition 9. A tuple $\langle \mathcal{G}_i, \mathcal{C}_i, \mathcal{V}_i, \mathcal{L}_i, \theta_i \rangle$ is said to be *proper* if the following conditions hold:

(i) $\mathcal{C}_i$ and $\mathcal{V}_i$ include all the constants and free variables in the formulas appearing in $\mathcal{G}_i$, and

(ii) for each $\langle G, \mathcal{P}, I \rangle \in \mathcal{G}_i$ it is the case that $\mathcal{L}_i(a) \leq I$ for each constant or free variable a appearing in the formulas in $\{G\} \cup \mathcal{P}$.

Lemma 11 *Every tuple in a derivation for a goal formula G from a set of closed program clauses $\mathcal{P}$ is proper.*

Proof. Obvious from an inspection of Definitions 8 and 6. We only mention that most general unifiers do not introduce any constants or variables not already in the formulas and that the labelling induced by a substitution from a given labelling may only reduce the label values for some variables.

□

The property of derivations that is observed in the following lemma is central to ensuring the soundness of the suggested proof procedure.

Lemma 12 *Let $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1 \rangle, \ldots, \langle \mathcal{G}_n, \mathcal{C}_n, \mathcal{V}_n, \mathcal{L}_n, \theta_n \rangle$ be a derivation of G from $\mathcal{P}$. Let σ_n denote the empty substitution and, for $1 \leq i < n$, let σ_i denote the substitution $\theta_n \circ \cdots \circ \theta_{i+1}$; alternatively $\sigma_i = \sigma_{i+1} \circ \theta_{i+1}$. Then, for $1 \leq i \leq n$,*

(1) σ_i restricted to $\mathcal{V}_i$ is proper with respect to $\mathcal{L}_i$, and

(2) for each $\langle G', \mathcal{P}', I \rangle \in \mathcal{G}_i$ it is the case that $\sigma_i(\mathcal{P}') \vdash_I \sigma_i(G')$.

Proof. The lemma is proved by a backward induction on the given sequence. It is vacuously true for the case when $i = n$. For the case when $i < n$ we consider the possibilities by which the $(i+1)$th tuple may have been derived from the ith one. Referring to the cases in Definition 6, a simple use of the induction hypothesis suffices for (i), (ii) and (iv). For (iii) and (v), we observe first that $\sigma_i = \sigma_{i+1}$. The requirement in the lemma concerning the "properness" of σ_i now follows in these cases from the induction hypothesis by observing that $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ agree on all the variables in $\mathcal{V}_i$. As for the second requirement, the induction hypothesis immediately verifies its truth for all the tuples in $\mathcal{G}_i$ that also belong to $\mathcal{G}_{i+1}$. This leaves only one other tuple to be argued for.

In the case of (iii), this tuple is of the form $\langle \exists x G', \mathcal{P}', I\rangle$. We observe here that $\langle [w/x]G', \mathcal{P}', I\rangle$ is a member of $\mathcal{G}_{i+1}$ and hence, by hypothesis, $\sigma_i(\mathcal{P}') \vdash_I \sigma_i([w/x]G')$. Now, for some $y \notin \mathcal{F}(G') \cup \mathcal{F}(\sigma_i)$, $[\sigma_i(w)/y]\sigma_i([y/x]G')$ is an alphabetic variant of $\sigma_i([w/x]G')$. Thus, it follows from Theorem 1 that

$$\sigma_i(\mathcal{P}') \vdash_I [\sigma_i(w)/y]\sigma_i([y/x]G'),$$

and, observing the structure of the $\exists$-R rule, therefore $\sigma_i(\mathcal{P}') \vdash_I \exists y \sigma_i([y/x]G')$. But $\exists y \sigma_i([y/x]G')$ is an alphabetic variant of $\sigma_i(\exists x G')$. Hence, by virtue of Theorem 1, the second requirement must be true for the tuple under consideration as well.

In the case of (v), the remaining tuple is of the form $\langle \forall x G', \mathcal{P}', I\rangle$. We know that, for some constant c with label $I+1$, $\langle [c/x]G', \mathcal{P}', I+1\rangle \in \mathcal{G}_{i+1}$. Thus $\sigma_i(\mathcal{P}') \vdash_I \sigma_i([c/x]G')$. Once again, for some y that is distinct from the free variables of G' and of σ_i, $[c/y]\sigma_i([y/x]G')$ is an alphabetic variant of $\sigma_i([c/x]G')$ and hence $\sigma_i(\mathcal{P}') \vdash_I [c/y]\sigma_i([y/x]G')$. Now, we have just noted that σ_i restricted to $\mathcal{V}_i$ is proper with respect to $\mathcal{L}_i$. Further, by lemma 11, the tuple $\langle \mathcal{G}_i, \mathcal{C}_i, \mathcal{V}_i, \mathcal{L}_i, \theta_i\rangle$ is proper. From these observations it is clear that the constants appearing in $\sigma_i(\mathcal{P}')$ and in $\sigma_i([y/x]G')$ must have labels that are bounded from above by I. Thus, c cannot appear in either $\sigma_i(\mathcal{P}')$ or $\sigma_i([y/x]G')$. But then we can use the $\forall$-R rule in conjunction with the proof of $\sigma_i(\mathcal{P}') \longrightarrow [c/y]\sigma_i([y/x]G')$ to obtain a proof for $\sigma_i(\mathcal{P}') \longrightarrow \forall y \sigma_i([y/x]G')$. The second requirement in the lemma is now verified for the tuple under consideration by observing that $\forall y \sigma_i([y/x]G')$ is an alphabetic variant of $\sigma_i(\forall x G')$.

The only remaining cases, then, are (vi) and (vii). In these cases, θ_{i+1} must be proper with respect to a labelling function that is like $\mathcal{L}_i$ on all the variables in $\mathcal{V}_i$. By the induction hypothesis, σ_{i+1} is proper with respect to a labelling induced from this labelling by θ_{i+1}. It is easily seen from these facts that $\sigma_{i+1} \circ \theta_{i+1}$ restricted to the variables in $\mathcal{V}_i$ must be proper with respect to $\mathcal{L}_i$. But this substitution is identical to the restriction of σ_i to $\mathcal{V}_i$.

The second requirement in the lemma follows directly from the hypothesis for all the tuples $\langle G_1, \mathcal{P}_1, I_1\rangle \in \mathcal{G}_i$ for which $\langle \theta_{i+1}(G_1), \theta_{i+1}(\mathcal{P}_1), I_1\rangle \in \mathcal{G}_{i+1}$. There is only one other tuple to be considered. Let this be $\langle A', \mathcal{P}', I\rangle$. We provide an argument for only (vii), the argument for (vi) being similar but simpler. In case (vii), there is a formula $\forall x_1 \ldots \forall x_m (G'' \supset A'')$ that is an alphabetic variant of some formula in $elab(\mathcal{P}')$ such that

(a) $\theta_{i+1}(A'') = \theta_{i+1}(A')$ and hence $\sigma_i(A'') = \sigma_i(A')$, and

(b) $\langle \theta_{i+1}(G''), \theta_{i+1}(\mathcal{P}'), I\rangle \in \mathcal{G}_{i+1}$.

From (a) it follows that $\sigma_i(A''), \sigma_i(\mathcal{P}') \longrightarrow \sigma_i(A')$ is an initial sequent. From (b) and the induction hypothesis it follows that $\sigma_i(\mathcal{P}') \longrightarrow \sigma_i(G'')$, *i.e.*, $\sigma_{i+1}(\theta_{i+1}(\mathcal{P}')) \longrightarrow \sigma_{i+1}(\theta_{i+1}(G''))$, has a proof. Now, we can rewrite $\sigma_i(A'')$ as

$$[\sigma_i(x_m)/y_m] \ldots [\sigma_i(x_1)/y_1]\sigma_i([y_m/x_m] \ldots [y_1/x_1]A'')$$

and, similarly, $\sigma_i(G'')$ as

$$[\sigma_i(x_m)/y_m]\ldots[\sigma_i(x_1)/y_1]\sigma_i([y_m/x_m]\ldots[y_1/x_1]G'')$$

for some distinct variables $y_1,\ldots,y_m \notin \mathcal{F}(A'')\cup\mathcal{F}(G'')\cup\mathcal{F}(\sigma_i)$. Using this observation and noting that the proofs for $\sigma_i(A''), \sigma_i(\mathcal{P}') \longrightarrow \sigma_i(A')$ and $\sigma_i(\mathcal{P}') \longrightarrow \sigma_i(G'')$ can be combined by an $\supset$-L rule, we see that a proof exists for the sequent

$$[\sigma_i(x_m)/y_m]\ldots[\sigma_i(x_1)/y_1]\sigma_i([y_m/x_m]\ldots[y_1/x_1](G''\supset A'')), \sigma_i(\mathcal{P}') \longrightarrow \sigma_i(A').$$

By repeated uses of $\forall$-L below this proof, we obtain one for

$$\forall y_1\ldots\forall y_m\,\sigma_i([y_m/x_m]\ldots[y_1/x_1](G''\supset A'')), \sigma_i(\mathcal{P}') \longrightarrow \sigma_i(A')$$

and thus for $\sigma_i(\forall y_1\ldots\forall y_m(G''\supset A'')), \sigma_i(\mathcal{P}') \longrightarrow \sigma_i(A')$. Noting that $\sigma_i(\forall y_1\ldots\forall y_m(G''\supset A''))$ is an alphabetic variant of a formula in $elab(\sigma_i(\mathcal{P}'))$, it is easily seen that $\sigma_i(\mathcal{P}') \longrightarrow \sigma_i(A')$ has a proof. The desired conclusion is thus obtained.

□

The soundness of a proof procedure that essentially searches for a derivation of a goal formula from a given set of program clauses is now stated and proved.

Theorem 13 *Let there be a derivation of a goal formula G from the set of closed program clauses $\mathcal{P}$ and let θ be the associated answer substitution. Then, for any formula G' that can be obtained by applying a substitution to $\theta(G)$, it is the case that $\mathcal{P} \vdash_I G'$.*

Proof. Let $\langle\mathcal{G}_1,\mathcal{C}_1,\mathcal{V}_1,\mathcal{L}_1,\theta_1\rangle,\ldots,\langle\mathcal{G}_n,\mathcal{C}_n,\mathcal{V}_n,\mathcal{L}_n,\theta_n\rangle$ be a derivation of G relative to $\mathcal{P}$. We observe that $\langle G,\mathcal{P},I\rangle \in \mathcal{G}_1$. Using Lemma 12 and noting that θ_1 is the empty substitution it follows then that

$$\theta_n\circ\cdots\circ\theta_1(\mathcal{P}) \vdash_I \theta_n\circ\cdots\circ\theta_1(G).$$

Since the formulas in $\mathcal{P}$ are closed, $\theta_n\circ\cdots\circ\theta_1(\mathcal{P})$ is an alphabetic variant of $\mathcal{P}$. Since θ is the restriction of $\theta_n\circ\cdots\circ\theta_1$ to the free variables of G, $\theta(G)$ is an alphabetic variant of $\theta_n\circ\cdots\circ\theta_1(G)$. Thus, with a possible recourse to Theorem 1, we see that $\mathcal{P} \vdash_I \theta(G)$. The desired conclusion now follows from Theorem 2.

□

We are now interested in a converse to Theorem 13, *i.e.*, we would like to show that our nondeterministic procedure is adequate as a device for determining whether a proof exists for a goal formula from a given set of program clauses. The strategy that we adopt in this direction can be characterized as follows. Let us say that a tuple of the form $\langle\mathcal{G},\mathcal{C},\mathcal{V},\mathcal{L},\theta\rangle$ is "solvable" if there is some (proper) substitution σ such that for every $\langle G,\mathcal{P},I\rangle \in \mathcal{G}$ it is the case that $\sigma(\mathcal{P}) \vdash_I \sigma(G)$. We show then that, given any solvable tuple that is not a terminated derivation, a new solvable tuple can be derived from it that is in a certain sense closer to being a terminated derivation. In fact we show that this is true independently of several choices that can be made in generating the new tuple. This fact is then used to show that if our procedure is started out with a solvable tuple, then it will construct a successfully terminated derivation, provided, of course, that it makes the correct choices at the critical points.

In order to execute the strategy outlined above, we need a measure that indicates the complexity of proofs for a goal formula from a finite set of program clauses. Such a measure is now defined.

Definition 10. Let $\mathcal{P}$ be a finite set of program clauses and let G be a goal formula such that $\mathcal{P} \vdash_I G$. Further, let l be the length of the shortest proof for $\mathcal{P} \longrightarrow G$. Then $\pi(\mathcal{P}, G) = 3^l$.

The first step in our strategy, then, is the content of the following lemma.

Lemma 14 *Let $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1 \rangle$ be a proper tuple and let σ be a substitution that is proper with respect to $\mathcal{L}_1$ and such that $\sigma(\mathcal{P}) \vdash_I \sigma(G)$ for every $\langle G, \mathcal{P}, I \rangle \in \mathcal{G}_1$. If $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1 \rangle$ is not a successfully terminated derivation sequence, then there is a tuple $\langle \mathcal{G}_2, \mathcal{C}_2, \mathcal{V}_2, \mathcal{L}_2, \theta_2 \rangle$ that can be derived from it and a substitution φ that together satisfy the following properties:*

(i) φ *is proper with respect to* $\mathcal{L}_2$*,*

(ii) σ *and* $\varphi \circ \theta_2$ *agree on* $\mathcal{V}_1$*,*

(iii) $\varphi(\mathcal{P}) \vdash_I \varphi(G)$ *for every* $\langle G, \mathcal{P}, I \rangle \in \mathcal{G}_2$*, and*

(iv) $\sum_{\langle G,\mathcal{P},I \rangle \in \mathcal{G}_2} \pi(\varphi(\mathcal{P}), \varphi(G)) < \sum_{\langle G,\mathcal{P},I \rangle \in \mathcal{G}_1} \pi(\sigma(\mathcal{P}), \sigma(G))$

Furthermore, such a tuple and a corresponding substitution can be obtained by picking the element from $\mathcal{G}_1$ that is to be acted upon in an arbitrary fashion.

Proof. Let $\langle G, \mathcal{P}, I \rangle \in \mathcal{G}_1$. We consider by cases the structure of G and exhibit the desired tuple and substitution in each case. Implicit in this argument is the fact that it is irrelevant which tuple is picked from $\mathcal{G}_1$.

Let G be of the form $G_1 \wedge G_2$, $G_1 \vee G_2$ or $D \supset G_1$. Then one of the cases (i), (ii) or (iv) in Definition 6 is applicable. We let $\varphi = \sigma$ and also let $\langle \mathcal{G}_2, \mathcal{C}_2, \mathcal{V}_2, \mathcal{L}_2, \theta_2 \rangle$ be the tuple indicated in the relevant case. It is easily seen that all the requirements are met by this choice. We only note that Theorem 5 is needed to ensure that the measure decreases.

Let G be of the form $\exists x G'$. Then case (iii) in Definition 6 is applicable. Let $\langle \mathcal{G}_2, \mathcal{C}_2, \mathcal{V}_2, \mathcal{L}_2, \theta_2 \rangle$ be the tuple that is obtained by a use of this case. Towards exhibiting φ, we observe first that, for some $y \notin \mathcal{F}(\sigma)$, $\exists y \sigma([y/x]G)$ is an alphabetic variant of $\sigma(\exists x G)$. Now, there is, by assumption, a proof for $\sigma(\mathcal{P}) \longrightarrow \sigma(\exists x G)$. Using Theorems 1 and 5, it follows that there is a term t such that $\sigma(\mathcal{P}) \longrightarrow [t/y]\sigma([y/x]G)$ also has a proof and, in fact, one that is shorter than any proof for the previous sequent. Further, using Theorem 3, we may assume that no parameter occurs in t that does not already occur in $\sigma(\mathcal{P})$ or in $\sigma([y/x]G)$. Finally, we let φ be the substitution like σ except that, for the w chosen in the application of step (iii), $\varphi(w) = t$.

It remains to be verified that the requirements of the lemma are met by the indicated substitution and tuple for the case considered above. Clearly φ is proper with respect to $\mathcal{L}_2$; the only case where this might be in question pertains to the substitution for w, but $\mathcal{L}_2(w) = I$ and t was chosen such that the labels of the constants appearing in it are bounded by I. The second condition follows by noting that $\theta_2 = \emptyset$ and σ and φ agree on $\mathcal{V}_1$. The choice of w and the properness of $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1 \rangle$ ensure that $\sigma(G'') = \varphi(G'')$ and $\sigma(\mathcal{P}') = \varphi(\mathcal{P}')$ for all the tuples $\langle G'', \mathcal{P}', I' \rangle$ in $\mathcal{G}_2$ that are distinct from $\langle [w/x]G', \mathcal{P}, I \rangle$. Thus, the third requirement is satisfied with regard to these tuples. For the remaining case, the choice of w, y and φ ensures that $\varphi([w/x]G')$ is an alphabetic variant of $[t/y]\sigma([y/x]G')$ as also is $\varphi(\mathcal{P})$ of $\sigma(\mathcal{P})$. Thus $\varphi(\mathcal{P}) \vdash_I \varphi([w/x]G')$ and in fact

$\pi(\varphi(\mathcal{P}), \varphi([w/x]G')) < \pi(\sigma(\mathcal{P}), \sigma(\exists x G'))$. Thus (iii) holds and the last observation also verifies (iv).

Consider now the case when G is of the form $\forall x G'$. By assumption, $\sigma(\mathcal{P}) \longrightarrow \sigma(\forall x G')$ has a proof. Now, for any $y \notin \mathcal{F}(\sigma)$, $\forall y \sigma([y/x]G')$ is an alphabetic variant of $\sigma(\forall x G')$. By Corollary 1, for any constant c, $\sigma(\mathcal{P}) \longrightarrow [c/y]\sigma([y/x]G')$ has a proof that is shorter than any proof for $\sigma(\mathcal{P}) \longrightarrow \sigma(\forall x G')$. But $[c/y]\sigma([y/x]G')$ is actually an alphabetic variant of $\sigma([c/x]G')$. Letting φ be identical to σ and using these observations, the lemma is easily verified for this case.

The only remaining case is that when G is an atomic formula, say A. Now, $\sigma(\mathcal{P}) \longrightarrow \sigma(A)$ has a proof by assumption. Hence, by virtue of Theorem 5, one of two situations must hold:

(a) $\sigma(A)$ must be identical to an instance of a formula in $\sigma(\mathcal{P})$, or

(b) some formula in $\sigma(\mathcal{P})$ must have as instance a formula of the form $G' \supset \sigma(A)$ where G' is such that $\sigma(\mathcal{P}) \longrightarrow G'$ has a proof that is shorter than any proof for $\sigma(\mathcal{P}) \longrightarrow \sigma(A)$.

We verify the requirements of the lemma only when the latter case holds, the argument when the former is true being similar but simpler.

In the case being examined, there must be a formula F of the form $\forall x_1 \ldots \forall x_m (G'' \supset A'')$ in $elab(\mathcal{P})$, that is such that $G' \supset \sigma(A)$ is an instance of $\sigma(F)$. We eventually consider the application of (vii) in Definition 6 with regard to this formula. However some prior analysis is necessary to ensure this step can be applied and to also make it possible to identify the substitution φ. Let $w_1, \ldots, w_m \notin \mathcal{V}_1$ be the variables that might be chosen in the step under consideration for renaming the quantified variables in F, let $\mathcal{L}'$ be the labelling function obtained by modifying $\mathcal{L}_1$ as required for these "new" variables and let $\gamma_1 = \{\langle x_i, w_i\rangle | 1 \le i \le m\}$ be the renaming substitution. Now, the w_i variables may be free in σ, so we have to consider a further renaming step in order to exhibit an instance of $\sigma(F)$ in a form useful for further analysis. Specifically, let $y_1, \ldots, y_m$ be distinct variables not in $\mathcal{F}(\sigma) \cup \mathcal{V}_1 \cup \{w_1, \ldots, w_m\}$ and let $\gamma_2 = \{\langle w_i, y_i\rangle | 1 \le i \le m\}$. Now, letting σ' be σ restricted to $\mathcal{V}_1$, it is easily seen that there is a substitution θ with domain $\{y_1, \ldots, y_m\}$ such that

$$\theta \circ \sigma' \circ \gamma_2 \circ \gamma_1(A'') = \sigma(A) \text{ and } \theta \circ \sigma' \circ \gamma_2 \circ \gamma_1(G'') = G'.$$

Finally this θ can be transformed into a substitution θ' that satisfies the following properties:

(1) $\theta' \circ \sigma' \circ \gamma_2$ is proper with respect to $\mathcal{L}'$,

(2) $\theta' \circ \sigma' \circ \gamma_2 \circ \gamma_1(A'') = \sigma(A)$, and

(3) there is a proof for $\mathcal{P} \longrightarrow \theta' \circ \gamma' \circ \gamma_2 \circ \gamma_1(G'')$ that is shorter than any for $\sigma(\mathcal{P}) \longrightarrow \sigma(A)$.

In essence, we obtain θ' from θ by replacing constants that have labels greater than I with ones that have labels bounded by I. This transformation is designed to make (1) true: if $\theta \circ \sigma' \circ \gamma_2$ itself is not satisfactory, it is only because the substitutions for some of the w_i variables contain constants with label greater than I. Since $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1\rangle$ is proper and σ is proper relative to $\mathcal{L}_1$, the labels on the constants appearing in $\sigma(A)$ and $\sigma(\mathcal{P})$ are bounded by I. Thus, (2) follows from $\theta \circ \sigma' \circ \gamma_2 \circ \gamma_1(A'') = \sigma(A)$ by noting that nothing is replaced in $\sigma(A)$ by the transformation just described. Finally (3) follows from the assumption that $\sigma(\mathcal{P}) \longrightarrow G'$ has a proof that is shorter than any proof for $\sigma(\mathcal{P}) \longrightarrow \sigma(A)$ with a possible recourse to Theorem 3; the latter may be needed because some constants in G', *i.e.*, in $\theta \circ \sigma' \circ \gamma_2 \circ \gamma_1(G'')$, may have to be renamed to get $\theta' \circ \sigma' \circ \gamma_2 \circ \gamma_1(G'')$.

Now, it is easily seen that $\theta' \circ \sigma' \circ \gamma_2(A) = \sigma(A)$. Thus, noting properties (1) and (2) above, it follows that $\theta' \circ \sigma' \circ \gamma_2$ is a unifier for A and $\gamma_1(A'')$ relative to $\mathcal{L}'$. Since these formulas have a unifier relative to $\mathcal{L}'$, they must, by Theorem 10, have a most general unifier. Let θ_2 be a most general unifier and let $\mathcal{L}_2$ be the labelling induced from $\mathcal{L}'$ by θ_2. Then there is a substitution φ that is proper with respect to $\mathcal{L}_2$ and such that $\theta' \circ \sigma' \circ \gamma_2 = \varphi \circ \theta_2$. Let

$$\mathcal{G}_2 = \theta_2((\mathcal{G}_1 - \{\langle A, \mathcal{P}, I\rangle\}) \cup \{\langle \gamma_1(G''), \mathcal{P}, I\rangle\}),$$

$\mathcal{V}_2 = \mathcal{V}_1 \cup \{w_1, \ldots, w_m\}$ and $\mathcal{C}_2 = \mathcal{C}_1$. Clearly $\langle \mathcal{G}_2, \mathcal{C}_2, \mathcal{V}_2, \mathcal{L}_2, \theta_2\rangle$ is derived from $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1\rangle$. We claim that φ and $\langle \mathcal{G}_2, \mathcal{C}_2, \mathcal{V}_2, \mathcal{L}_2, \theta_2\rangle$ satisfy the requirements of the lemma. The first requirement is true by construction. The second follows from observing that σ and $\theta' \circ \sigma' \circ \gamma_2$ agree on $\mathcal{V}_1$. For the third requirement, we observe first that the free variables of G_1 and $\mathcal{P}'$ for every tuple $\langle G_1, \mathcal{P}', I'\rangle$ in $\mathcal{G}_1$ are contained in $\mathcal{V}_1$ and hence $\varphi(\theta_2(G_1))$ and $\sigma(G_1)$ are alphabetic variants as also are $\varphi(\theta_2(\mathcal{P}'))$ and $\sigma(\mathcal{P}')$. Thus this requirement follows from the assumptions for each tuple in $\mathcal{G}_2$ that is obtained by applying the substitution θ_2 to a tuple in $\mathcal{G}_1$. For the only other tuple, *i.e.*, for $\langle \theta_2(\gamma_1(G'')), \theta_2(\mathcal{P}), I\rangle$, we have observed that there is a proof for $\sigma(\mathcal{P}) \longrightarrow \theta' \circ \sigma' \circ \gamma_2 \circ \gamma_1(G'')$ that is shorter than any for $\sigma(\mathcal{P}) \longrightarrow \sigma(A)$. But $\varphi(\theta_2(\mathcal{P}))$ and $\sigma(\mathcal{P})$ are alphabetic variants and so are $\varphi(\theta_2(\gamma_1(G'')))$ and $\theta' \circ \sigma' \circ \gamma_2 \circ \gamma_1(G'')$. Thus the third requirement holds for this case as well and, the additional information concerning the lengths of proofs actually ensures that the fourth requirement is also met.

□

We now use the above lemma to conclude, in the manner outlined earlier, the proof of completeness of our procedure.

Theorem 15 *Let $\mathcal{P}$ be a finite set of closed program clauses and let G be a goal formula. Further, let σ be a substitution that is proper with respect to the labelling function that is 0 valued on variables and such that $\mathcal{P} \vdash_I \sigma(G)$. Then there is a derivation of G from $\mathcal{P}$ with an answer substitution δ that can be composed with another substitution to yield the restriction of σ to the free variables of G. Further, such a derivation and such an answer substitution exists regardless of the element acted upon at each stage in constructing a derivation sequence.*

Proof. Let $\mathcal{S} = \langle \mathcal{G}, \mathcal{C}, \mathcal{V}, \mathcal{L}, \theta\rangle$ be a proper tuple and let φ be a substitution that is proper with respect to $\mathcal{L}$ and such that $\varphi(\mathcal{P}) \vdash_I \varphi(G)$ for every $\langle G, \mathcal{P}, I\rangle \in \mathcal{G}$. We associate the following measure with such a tuple and substitution:

$$\mu(\mathcal{S}, \varphi) = \sum_{\langle G, \mathcal{P}, I\rangle \in \mathcal{G}} \pi(\varphi(\mathcal{P}), \varphi(G)).$$

Given such a tuple and substitution, using Lemma 14 in conjunction with the measure just defined, a derivation sequence $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1\rangle, \ldots, \langle \mathcal{G}_n, \mathcal{C}_n, \mathcal{V}_n, \mathcal{L}_n, \theta_n\rangle$ and an associated sequence of substitutions $\varphi_1, \ldots, \varphi_n$ satisfying the following properties can be identified:

(a) $\langle \mathcal{G}_1, \mathcal{C}_1, \mathcal{V}_1, \mathcal{L}_1, \theta_1\rangle = \langle \mathcal{G}, \mathcal{C}, \mathcal{V}, \mathcal{L}, \theta\rangle$ and $\varphi_1 = \varphi$,

(b) the derivation sequence terminates successfully, and

(c) for $1 \leq i < n$, φ_i and $\varphi_{i+1} \circ \theta_{i+1}$ agree on $\mathcal{V}_{i+1}$.

Now, let $\mathcal{G}$ be $\{\langle G, \mathcal{P}, 0\rangle\}$, let $\mathcal{C}$ and $\mathcal{V}$ be, respectively, the set of constants and the set of free variables in $\{G\} \cup \mathcal{P}$, let $\mathcal{L}$ be the constant 0 valued function over $\mathcal{V}$ and let θ be the empty substitution. Further, let φ be σ. From the assumptions in the theorem, these assignments ensure that the requirements of $\langle \mathcal{G}, \mathcal{C}, \mathcal{V}, \mathcal{L}, \theta\rangle$ and φ are satisfied. But then the indicated derivation sequence is really a derivation of G from $\mathcal{P}$. Further, using an induction on the length of the sequences together with (c) and the observations that $\theta_1 = \emptyset$ and, for $1 \leq i < n$, $\mathcal{V}_i \subseteq \mathcal{V}_{i+1}$, it can be seen that there is a substitution γ such that φ and $\gamma \circ \theta_n \circ \cdots \circ \theta_1$ agree on $\mathcal{V}_1$. But then it follows that σ agrees with $\gamma \circ \delta$ on the free variables in G, where δ is the answer substitution corresponding to the derivation under consideration.

We have thus exhibited a derivation of G from $\mathcal{P}$ with an answer substitution satisfying the requirements of the theorem. Lemma 14 guarantees that, in constructing this derivation and the associated sequence of substitutions, an arbitrary element of $\mathcal{G}_i$ can be used to generate the $(i+i)$th items in the sequences. Thus, the final requirement of the theorem is seen to be true.

□

6 Extension to Higher-Order Formulas

The propositional and quantifier structure of the higher-order goal formulas and program clauses bears a close similarity to the first-order versions. One distinction is that the higher-order formulas are typed. Typing is necessary to ensure the consistency of the underlying logic. In the discussions here we implicitly assume the presence of types. Another difference, introduced only for technical reasons, is that we include in the vocabulary of our logic the symbol $\top$ to denote the tautologous proposition and we consider this to be an acceptable goal formula. The final, and most significant difference is that first-order terms are replaced by the terms of a (simply typed) lambda calculus.

The lambda terms in higher-order logic can generally contain within them arbitrary quantifiers and connectives. However, we shall only use terms that do not contain the symbols $\supset$ and $\sim$. These terms are referred to as positive terms and the restriction to them is necessary for reasons explained in [16]. A (positive) atomic formula is then a formula of the form $(P\ t_1\ \ldots\ t_n)$ where P is a predicate name or variable and, for $1 \leq i \leq n$, t_i is a positive term. We refer to such an atomic formula as a rigid one in the case that P is a constant and as a flexible one otherwise. Using the symbol A_r to represent a rigid atomic formula and A to denote an arbitrary atomic formula, the higher-order versions of goal formulas and program clauses are given by the following syntax rules:

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x G \mid D \supset G \mid \forall x G,$$

$$D ::= A_r \mid G \supset A_r \mid D \wedge D \mid \forall x D.$$

In formalizing the notion of intuitionistic provability for our higher-order logic, a sequent calculus very similar to the one presented in Section 2 may be used. There are in fact only two changes that need to be made. First, we permit leaves in proofs to be labelled with sequents of the form $\Delta \longrightarrow \top$. Second, we allow the inference rules generated from the following schema to be used:

$$\frac{\Delta' \longrightarrow \Theta'}{\Delta \longrightarrow \Theta}\,\lambda$$

where Δ' and Θ' are obtained from Δ and Θ by replacing some formulas by ones that can be obtained from them via λ-conversion (specifically, α-, β- and η-conversion) rules. A point to note is that the operation of substitution needs to be more carefully defined in the higher-order context because of the presence of abstractions in terms. However, there is a simple, and standard, way of doing this using λ-conversion. We assume such a definition here; the reader unfamiliar with this formalization of substitution may look, for example, at [21].

As in the first-order context, the idea of programming can be thought of as asking if a proof exists for a goal formula from a set of program clauses. Now, a property very similar to that presented in Theorem 5 holds in the higher-order context as well and this is once again useful in designing a procedure for determining the existence of a proof. This property is stated in the following theorem.

Theorem 16 *Let $\mathcal{P}$ be a finite set of higher-order program clauses and let G be a higher-order goal formula such that $\mathcal{P} \longrightarrow G$ has a derivation of length l. Then one of the following holds:*

(1) *G is $\top$.*

(2) *G is an atomic formula and it is identical to an instance of a formula in $\mathcal{P}$ or there is an instance $G' \supset G$ of some formula in $\mathcal{P}$ such that $\mathcal{P} \longrightarrow G'$ has a derivation of length less than l.*

(3) *G is $G_1 \wedge G_2$ and $\mathcal{P} \longrightarrow G_1$ and $\mathcal{P} \longrightarrow G_2$ have derivations of length less than l.*

(4) *G is $G_1 \vee G_2$ and, for $i = 1$ or $i = 2$, the sequent $\mathcal{P} \longrightarrow G_i$ has a derivation of length less than l.*

(5) *G is $\exists x G_1$ and there is a positive term t such that $\mathcal{P} \longrightarrow [t/x]G_1$ has a derivation of length less than l.*

(6) *G is $D \supset G_1$ and $D, \mathcal{P} \longrightarrow G_1$ has a derivation of length less than l.*

(7) *G is $\forall x G_1$ and, for some constant c not appearing in $\mathcal{P}$ or in G_1, $\mathcal{P} \longrightarrow [c/x]G_1$ has a derivation of length less than l.*

The proof of this theorem is not provided here, but it may be found (in essence) in [16]. The critical step is in showing that the restriction to positive instantiation terms in (5) is possible. Once this fact is shown, arguments similar to those employed in Section 3 can be used to reach the desired conclusion.

Given this theorem, the discussion at the end of Section 3 becomes relevant to the design of a proof procedure in the higher-order context as well. One point to note is that quantifications over higher-order variables is permitted in the new context. Our labelling scheme will therefore have to be extended to apply to constants and variables of function type as well. A second point to note is that in the course of solving goals, it is possible that we encounter flexible atomic formulas. The analysis in the proof of Theorem 16 shows that solutions of such goal formulas can be delayed till no other goals are left to be solved. At this stage a simple solution can be provided. This solution effectively consists of substituting the universal relation of appropriate type — *i.e.*, the predicate term $\lambda x_1 \ldots \lambda x_n \top$, where the number of abstractions and the type of each abstraction depends on the type of the variable being substituted for — for the predicate variables that are the "heads" of the atomic formulas.

There is, however, one significant difference between the proof procedures for the first- and higher-order formulas: the notion of unification in the higher-order context must incorporate an equality relation on terms that is based on λ-conversion. The problem of unifying terms under this extended notion of equality differs in several respects from the first-order unification problem: the higher-order unification problem is an undecidable one in general and most general unifiers might not exist even when there are unifiers for given terms. There is, nevertheless, a procedure that can be used to find unifiers for terms whenever they exist, and this procedure is described in [7]. This procedure can be factored into the repeated application of certain simple steps, and this permits its amalgamation into a notion of derivation akin to the one described in Section 5. Such an amalgamation is described explicitly for a higher-order version of Horn clauses in [21], and a similar process can be used in the case of hereditary Harrop formulas. The one difference is that substitutions that are suggested for the purpose of unification must respect the constraints imposed by labels on symbols. As in the first-order case, this can be ensured by incorporating checks into the generation of substitutions. In particular, substitutions are suggested when an attempt is made to unify a pair of terms of the form $\lambda x_1 \ldots \lambda x_n(f\ t_1 \ldots t_p)$ and $\lambda y_1 \ldots \lambda y_m(c\ s_1 \ldots s_q)$, where f is a variable and c is a constant or one of the variables $y_1, \ldots, y_m$. Two kinds of substitutions are considered here for f:

(1) If c is a constant, then f might be made to "imitate" the head of the other term. Specifically, a substitution of the form

$$\lambda w_1 \ldots \lambda w_p(c\ (h_1\ w_1 \ldots w_p) \ldots (h_q\ w_1 \ldots w_p))$$

where $h_1, \ldots, h_q$ are new variables is considered for f.

(2) The "projection" of f onto one of its arguments might be attempted. In this case, substitutions of the form

$$\lambda w_1 \ldots \lambda w_p(w_i\ (h_1\ w_1 \ldots w_p) \ldots (h_j\ w_1 \ldots w_p)),$$

where $1 \leq i \leq m$ and $h_1, \ldots h_j$ are new variables, are considered for f; certain typing constraints have to be satisfied by w_i for these substitutions to be actually used and the number of arguments in the substitution term then depends on the type of w_i.

Given the overall structure of the proof procedure for hereditary Harrop formulas, we see that an additional constraint has to be satisfied for the imitation substitution to be a possibility: the label of c would have to be less than the label of f. If this condition is satisfied, the substitution may be generated, but the labels associated with $h_1, \ldots, h_q$ must be made identical to that associated with f. Intuitively, this is necessary for ensuring that later instantiations of these variables do not violate the constraint on substitution terms for f. As for the projection substitutions, these continue to be possibilities. However, once again the labelling constraint on substitutions for f will have to be passed on to the variables $h_i, \ldots, h_j$, *i.e.*, their label values become that of f.

The ideas described above can be used to detail a satisfactory proof procedure for the logic of higher-order hereditary Harrop formulas. A proof of correctness for this procedure can also be provided. In outline, this proof would amalgamate the arguments in [21] showing that all the possible substitutions are considered with the arguments in Section 5 showing that the substitutions that are considered respect the necessary constraints. The detailed presentation of this procedure and its correctness proof is somewhat tedious and is therefore not undertaken in this paper.

7 Conclusion

We have described a proof procedure in this paper for the logic of hereditary Harrop formulas. The procedure exploits the possibility of conducting a search directed by the logical structure of the goal formula. Further, it uses unification in order to control the search for instantiations for existentially quantified goal formulas. The formulas for which proofs are sought may have appearances of universal and existential quantifiers in mixed order and this necessitates a careful use of unification. This problem is dealt with in our proof procedure by a numeric labelling of variables and constants coupled with an occurs-check based on these labels. The overall scheme is discussed in a comprehensive fashion for the first-order case and a proof of correctness is provided. We have also indicated the manner in which this scheme can be extended to the higher-order case.

The proof procedure that we have presented for first-order goal formulas is, we believe, amenable to efficient implementation. The labels associated with constants and variables can be incorporated as an additional component in the representation of these objects and would then be readily available when the "consistency" check has to be done. Labels for (logic) variables can be generated by using a global register that is initially set to 0 and is incremented within the scope of a universal goal. Several aspects of the procedure can also be compiled. Techniques used in conjunction with Horn clauses can be employed in compiling the search specified by $\vee$, $\wedge$ and $\exists$. The first of these symbols is still a source of non-determinism, but the usual depth-first search with backtracking can be used to implement it. A significant portion of unification can also be compiled. In this mode, the process of label checking either becomes redundant or reduces to setting up labels that will be used in the interpretive part of unification. Universal quantifiers can be compiled into the operation of incrementing the "label" register and generating a new constant with a label identical to the value of this register. With regard to implications, our presentation of the proof procedure makes it appear as though each goal must carry its own "program" context. However, this is unnecessary and a stack based approach can be used to update programs. Thus, a goal such as $(D_1 \supset G_1) \wedge (D_2 \supset G_2)$ can be solved by adding D_1 to the existing program and solving G_1 and then removing D_1 and adding D_2 to solve G_2. The possibility of backtracking complicates the situation (*e.g.*, consider what must be done upon failure in solving G_2 in the goal above), but a bookkeeping mechanism can be integrated into the basic scheme to deal with this. This scheme also supports the compilation of the actions that need to be carried out when an implication is encountered. Finally, it is also possible to compile the program clauses that appear on the left of implications. We have in general to consider program clauses containing variables that might be further instantiated (*e.g.*, consider solving the goal $\exists x(P(x) \supset G(x)))$ but a notion similar to the closures used in functional programming can be employed for this purpose.

The various ideas outlined above are also useful in implementing the appropriate proof procedure for the higher-order case. However, there are substantial additional problems that must be dealt with in providing an implementation of this procedure that is of acceptable efficiency. One of these problems concerns the representation of lambda terms and the implementation of operations such as beta reduction on these terms. This issue has been considered in the past, especially in the realm of functional programming. However, the particular use that is made of these terms in our context requires a solution to this problem that is of a somewhat different nature. Specifically, unification requires the comparison of terms and therefore makes it necessary to examine the structure of a term, perhaps even parts of it embedded under abstractions. We have examined this problem in some detail in [22] and we believe that the notation for lambda terms developed there provides the basis for a satisfactory solution. Another problem concerns the implementation of higher-order unification itself. One issue here is whether any aspect of this operation can be compiled. A start in this direction has been made in [18] — for example, it is shown there that some of the "first-order" aspects of this operation can be compiled — but there is clearly much more that can be done. Another issue is that of dealing with the possibility for branching within the unification process. Mechanisms for accommodating this possibility have been suggested in [18], but we suspect that these can be bettered, especially after experience is gained with an actual implementation of these mechanisms. Finally it is of interest to examine whether a recognition of special kinds of unification problems can be built into the unification procedure of [7] to improve its behavior in practical situations. This aspect has been studied in [14] and [13], but, once again, this is a topic that can benefit from additional research.

As we have mentioned already, the proof procedure presented in this paper and the ideas concerning its implementation are of immediate practical utility: they can be used in a realization of λProlog, a logic programming language based on hereditary Harrop formulas. We have, in fact, used them in this capacity towards an implementation of a first-order version of λProlog [8, 19], and the additional machinery needed to extend this implementation to the full language is the subject of a forthcoming paper.

Acknowledgements

This paper has benefitted greatly from comments that were provided by Dale Miller on an earlier draft. Suggestions from the referees have also contributed to improvements in presentation. This work has been supported by NSF Grant CCR-89-05825.

References

[1] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Implemented as part of the CMU ERGO project, May 1989.

[2] 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.

[3] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *Ninth International Conference on Automated Deduction*, pages 61–80, Argonne, IL, May 1988. Springer-Verlag.

[4] Melvin Fitting. *First-order logic and automated theorem proving.* Springer-Verlag, 1990.

[5] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North Holland Publishing Co., 1969.

[6] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. PhD thesis, University of Pennsylvania, August 1990.

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

[8] Bharat Jayaraman and Gopalan Nadathur. Implementation techniques for scoping constructs in logic programming. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 871–886, Paris, France, June 1991. MIT Press.

[9] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Lanuages and Systems*, 4(2):258–282, April 1982.

[10] L. Thorne McCarty. Clausal intuitionistic logic II. Tableau proof procedures. *Journal of Logic Programming*, 5:93–132, 1988.

[11] Dale Miller. Hereditary Harrop formulas and logic programming. In *Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science*, pages 153–156, Moscow, August 1987.

[12] Dale Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

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

[14] Dale Miller. Unification under a mixed prefix. Technical Report MS-CIS-91-81, Computer Science Department, University of Pennsylvania, October 1991. To appear in the *Journal of Symbolic Computation.*

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

[16] 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.

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

[18] 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, Cleveland, Ohio, October 1989.

[19] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Submitted, May 1992.

[20] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

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

[22] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 341–348. ACM Press, 1990.

[23] Lawrence R. Paulson. The representation of logics in higher-order logic. Technical Report Number 113, University of Cambridge, Computer Laboratory, August 1987.

[24] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 153–163, 1988.

[25] Natarajan Shankar. Proof search in the intuitionistic sequent calculus. In Deepak Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction – CADE-11*, pages 522–536. Springer Verlag, June 1992.

[26] M. H. van Emden and R. H. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[27] Lincoln A. Wallen. *Automated Proof Search in Non-Classical Logics: Efficient Matrix Proof Methods for Modal and Intuitionistic Logics.* MIT Press, 1990.