A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification

Dale Miller
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389 USA
dale@cis.upenn.edu

Abstract: It has been argued elsewhere that a logic programming language with function variables and λ-abstractions within terms makes a very good meta-programming language, especially when an object language contains notions of bound variables and scope. The λProlog logic programming language and the closely related Elf and Isabelle systems provide meta-programs with both function variables and λ-abstractions by containing implementations of higher-order unification. In this paper, we present a logic programming language, called L_λ, that also contains both function variables and λ-abstractions, but certain restriction are placed on occurrences of function variables. As a result, an implementation of L_λ does not need to implement full higher-order unification. Instead, an extension to first-order unification that respects bound variable names and scopes is all that is required. Such unification problems are shown to be decidable and to possess most general unifiers when unifiers exist. A unification algorithm and logic programming interpreter are described and proved correct. Several examples of using L_λ as a meta-programming language are presented.

1. Introduction

A meta-programming language must be able to represent and manipulate such syntactic structures as programs, formulas, types, and proofs. A common characteristic of all these structures is that they involve notions of abstractions, scope, bound and free variables, substitution instances, and equality up to alphabetic changes of bound variables. Although the data types available in most computer programming languages are, of course, rich enough to represent all these kinds of structures, such data types do not have direct support for these common characteristics. For example, although it is trivial to represent first-order formulas in Lisp, it is a more complex matter to write Lisp programs that correctly substitute a term into a formulas (being careful not to capture bound variables), to test for the equality of formulas up to alphabetic variation, and to determine if a certain variable's occurrence is free or bound. This situation is the same when structures like programs or (natural deduction) proofs are to be manipulated and if other programming languages, such as Pascal, Prolog, and ML, replace Lisp.

It is desirable for a meta-programming language to have language-level support for these various aspects of formulas, proofs, types, and programs. What is a common framework for representing these structures? Early work by Church, Curry, Howard, Martin-Löf, Scott, Strachey, Tait, and others concluded that typed and untyped λ-calculi provide a common syntactic representation for all these structures. Thus a meta-programming language that is able to represent terms in such λ-calculi directly could be used to represent these structures using the techniques described by these authors.

One problem with designing a data type for λ-terms is that methods for destructuring them should be invariant under the intended notions of equality of λ-terms, which usually include α-conversion. Thus, destructuring the λ-term $\lambda x.fxx$ into its bound variable x and body fxx is not invariant under α-conversion: this term is α-convertible to $\lambda y.fyy$ but the results of destructuring this equal term does not yield equal answers. Although the use of nameless dummies [1] can help simplify this one problem since both of these terms are represented by the same structure $\lambda(f11)$, that representation still requires fairly complex manipulations to represent the full range of desired operations on λ-terms. A more high-level approach to the manipulation of λ-terms modulo α and β-conversion has been the use of unification of simply typed λ-terms [9, 15, 24, 26]. Huet and Lang [10] describe how such a technique, when restricted to second-order matching, could be used to analyze and manipulate simple functional and imperative programs. Their reliance on unification, which solved equations up to α, β, and η-conversion, made their meta programs elegant, simple to write, and easy to prove correct. They chose second-order matching because it was strong enough to implement a certain collection of template matching program transformations and it was decidable. The general problem of the unification of simply typed λ-terms of order 2 and higher is undecidable [11].

This use of unification on λ-terms has been extended in several recent papers. In [4, 5, 6, 16] various meta programs, including theorem provers and program transformers, were written in the logic programming language λProlog [18] that contains unification of simply typed λ-terms. Paulson [20, 21] exploits such unification in the theorem proving system Isabelle. Pfenning and Elliot [22] argue that product types are also of use. Elliot [3] has studied unification in a dependent type framework and Pfenning [23] developed a logic programming language, Elf, that incorporates that unification process. That programming language can be used to provide a direct implementation of signatures written in the LF type specification language [7].

In this paper, we shall focus on a particularly simple logic programming language, called L_λ, that is completely contained within λProlog and admits a very natural implementation of the data type of λ-terms. The term language of L_λ is the simply typed λ-calculus with equality modulo α, β, and η-conversion. The "β-aspects" of L_λ are, however, greatly restricted and, as a result, unification in this language will resemble first-order unification – the main difference being that λ-abstractions are handled directly.

The structure of L_λ is motivated in Section 2 and formally defined in Section 3. An interperter and unification algorithm for L_λ are presented in Sections 4 and 5, respectively. The unification algorithm is proved correct in Section 6 and the interpreter is proved correct in Section 7. Finally, several examples of L_λ programs are presented in Section 8.

2. Two motivations

There are at least two motivations for studying the logic L_λ. The first is based on experience with using stronger logics for the specification of meta-programs. The second is based on seeing how L_λ can be thought of as a kind of "closure" of a first-order logic programming language.

2.1. Past experience. Both the Isabelle theorem prover and λProlog contain simply typed λ-terms, β-conversion, and quantification of variables at all functional orders. These systems have been used to specify and implement a large number of meta-programming tasks, including theorem proving, type checking, and program transformation, interpretation, and compilation. An examination of the structure of those specifications and implementations revealed two interesting facts. First, free or "logic" variables of functional type were often applied only to distinct λ-bound variables. For example, the free functional variable M might appear in the following context:

$$\lambda x \ldots \lambda y \ldots (Myx) \ldots.$$

When such free variables were instantiated, the only new β-redexes that arise are those involving distinct λ-bound variables. For example, if M above is instantiated with a λ-terms, say $\lambda u \lambda v.t$, then the only new β-redex formed is $((\lambda u \lambda v.t)yx)$. This is reduced to normal form simply by renaming the variables u and v to y and x — a very simple computation. Second, in the cases where free variables of functional type were applied to general terms, meta-level β-reduction was invoked to simply perform object-level substitution.

The logic L_λ is designed to permit the first kind of β-redex but not the second. As a result, implementations of this logic can make use of a very simple kind of unification. The fact that object-level substitution is not automatically available can be fixed, however, since object-level substitution can be written very simply as L_λ programs. We illustrate this for a simple, first-order, object logic in Section 8. Thus, L_λ requires that some of the functionality of β-conversion be moved from the term level to the logic level. The result can be more complex logic programs but with simpler unification problems. This seems like a trade-off worth investigating.

Another characteristic of most meta programs written in Isabelle and λProlog is that they quantify over at most second-order functional types. Despite this observation, we shall present an ω-order version of L_λ here. As we shall see, however, in Section 5, the unification procedure of L_λ is not dependent on types and, hence, not on order.

2.2. Discharging constants from terms. Consider a first-order logic whose logical connectives are $\wedge$ (conjunction), $\supset$ (implication), and $\forall$ (universal quantification). Let A be a syntactic variable that ranges over atomic formulas, and let D and G range over formulas defined by the following grammar:

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

It has been argued in various places (for example, [14, 17]) that the intuitionistic theory of these formulas provides a foundation for logic programming if we permit D-formulas to be program clauses and G-formulas as goals or queries asked of them. As a logic programming language, it forms a rich extension to Horn clauses and still retains several important properties that make it suitable for program specification and implementation.

One of those important properties is that a theorem prover having a simple operational description is sound and non-deterministically complete with respect to intuitionistic logic. This operational behavior can be described as follows. Let Σ be a first-order signature (set of constants), let $\mathcal{P}$ be a finite set of closed D-formulas, and let G be a closed G-formula, both over Σ (*i.e.*, all of whose constants are from Σ). Intuitionistic provability of G from Σ and $\mathcal{P}$, written as $\Sigma; \mathcal{P} \vdash_I B$ can be characterized using the following search operations:
AND: $\Sigma; \mathcal{P} \vdash_I G_1 \wedge G_2$ if $\Sigma; \mathcal{P} \vdash_I G_1$ and $\Sigma; \mathcal{P} \vdash_I G_2$.
AUGMENT: $\Sigma; \mathcal{P} \vdash_I D \supset G$ if $\Sigma; \mathcal{P} \cup \{D\} \vdash_I G$.
GENERIC: $\Sigma; \mathcal{P} \vdash_I \forall x.G$ if $\Sigma \cup \{c\}; \mathcal{P} \vdash_I [x \mapsto c]G$, provided that c is not in Σ.
BACKCHAIN: $\Sigma; \mathcal{P} \vdash_I A$ if there is a formula $D \in \mathcal{P}$ whose universal instantiation with closed terms over Σ is A or is $G \supset A$ and $\Sigma; \mathcal{P} \vdash_I G$.

Clearly, this characterization of intuitionistic provability can be molded into a simple theorem proving mechanism. Such a mechanism using unification and a depth-first searching discipline can be used to give a logic programming interpretation to this logic. Notice that both components to the left of the turnstile may vary within the search for a proof. For example, the terms used to instantiate the universal quantifiers mentioned in the BACKCHAIN rule can be taken from different signatures at different parts of a proof.

While this logic has its uses (for example, see [12, 13, 14]), there is a kind of incompleteness in its space of values. Consider the following example. Let Σ_0 be a signature containing at least the constants *append*, *cons*, *nil*, *a*, *b* and let $\mathcal{P}_0$ contain just the two clauses

$$\forall x \forall l \forall k \forall m (\mathit{append}\ l\ k\ m \supset \mathit{append}\ (\mathit{cons}\ x\ l)\ k\ (\mathit{cons}\ x\ m))$$
$$\forall k (\mathit{append}\ \mathit{nil}\ k\ k).$$

Now, consider the problem of finding a substitution term over Σ_0 for the free variable X so that the goal formula $\forall y (\mathit{append}\ (\mathit{cons}\ a\ (\mathit{cons}\ b\ \mathit{nil}))\ y\ X)$ is provable. Proving this goal can be reduced to finding an instantiation of X so that

$$(\mathit{append}\ (\mathit{cons}\ a\ (\mathit{cons}\ b\ \mathit{nil}))\ k\ X)$$

is provable, where k is not a member of Σ_0. Using BACKCHAIN twice, this goal is provable if and only if X can be instantiated with $(\mathit{cons}\ a\ (\mathit{cons}\ b\ k))$. This is not possible, however, since X can be instantiated with terms over Σ_0 but not over $\Sigma_0 \cup \{k\}$. Such a failure here is quite sensible since the value of X should be independent of the choice of the constant used to instantiate $\forall y$. It might be very desirable, however, to have this computation succeed if we could, in some sense, abstract away this particular choice of constant. That is, an interesting value is computed here, but it cannot be used since it is not well defined. If we are willing to admit some forms of λ-abstraction into our logic, this value can be represented.

Consider, for example, proving the goal $\forall y\ (append\ (cons\ a\ (cons\ b\ nil))\ y\ (H\ y))$ where H is a functional variable that may be instantiated with a λ-term whose constants are again from the set Σ_0. Assume that we instantiate $\forall y$ again with the constant k. This time we need $(H\ k)$ to be equal to $(cons\ a\ (cons\ b\ k))$. There are two simply-typed λ-terms (up to λ-conversion) that when substituted for H into $(H\ k)$ and then λ-normalized yield $(cons\ a\ (cons\ b\ k))$, namely, the terms $\lambda w\ (cons\ a\ (cons\ b\ k))$ and $\lambda w\ (cons\ a\ (cons\ b\ w))$. Since H cannot contain k free, only the second of these possible substitutions will succeed in being a legal solution for this goal. Notice that the choice of constant to instantiate the universal quantifier in this goal is not reflected in this answer substitution since λ-terms obey α-conversion. In a sense, the λ-term $\lambda w\ (cons\ a\ (cons\ b\ w))$ is the result of *discharging* the constant k from the term $(cons\ a\ (cons\ b\ k))$. Notice, however, that discharging a first-order constant from a first-order term is now a "second-order" term: it can be used to instantiate a function variable.

The higher-order variable H in the above example is restricted in such a way that when it is involved in a unification problem, there is a single, most general unifier for it. We shall define L_λ in such a way that this is the only kind of "higher-order" unification problem that can occur. All such uses of a higher-order variable will be associated with discharging a constant from a term. Term models for β-reduction of the simply typed λ-calculus interpret a λ-term, say $\lambda x.t$ of type $\tau \rightarrow \sigma$, as a mapping from λ-equivalence classes of type τ to such equivalence classes of type σ. In L_λ, this functional interpretation must be restricted greatly: $\lambda x.t$ can be though of as a function that carries an increment in a signature to a term over that signature.

The reader who is comfortable with the above discussion and who is familiar with basic notions of logic programming and simply typed λ-terms may wish to read next Section 8 where several examples of L_λ programs are given and discussed. The following five sections are rather formal and address technical aspects of a unification algorithm and interpreter for L_λ.

3. Definition of L_λ

We assume that the reader is familiar with the basic properties of λ-terms and λ-conversion. Below we review some definitions and properties. See [8] for a more complete presentation.

Let S be a fixed, finite set of *primitive types* (also called *sorts*). The set of *types* is the smallest set of expressions that contains the primitive types and is closed under the construction of function types, denoted by the binary, infix symbol $\rightarrow$. This arrow associates to the right: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. The Greek letters τ and σ are used as syntactic variables ranging over types.

Let τ be the type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $\tau_0 \in S$ and $n \geq 0$. (By convention, if $n = 0$ then τ is simply the type τ_0.) The types $\tau_1, \ldots, \tau_n$ are the *argument types of* τ while the type τ_0 is the *target type of* τ. The order of a type τ is defined as follows: If $\tau \in S$ then τ has order 0; otherwise, the order of τ is one greater than the maximum order of the argument types of τ. Thus, τ has order 1 exactly when τ is of the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $n \geq 1$ and $\{\tau_0, \tau_1, \ldots, \tau_n\} \subseteq S$.

A *signature (over S)* is a (possibly infinite) set Σ of pairs of tokens and types that satisfies the usual functionality condition: a given token is associated with at most one type in a given signature. We shall also assume that there are denumerably many tokens that are not mentioned in a given signature. We often enumerate signatures by listing their pairs as $a : \tau$. A signature is of order n if all its tokens have types of order n or less. The expression $\Sigma + c : \tau$ is legal if c is not assigned by Σ, in which case, it is equal to $\Sigma \cup \{c : \tau\}$.

A *Σ-term of type* τ is defined by the proof system in Figure 1. If the sequent $\Sigma \xrightarrow{stt} t : \tau$ is provable from those rules, we write $\Sigma \vdash_{stt} t : \tau$ and say that t is a Σ-term of type τ.

$$\Sigma + c : \tau \xrightarrow{stt} c : \tau$$

$$\frac{\Sigma \xrightarrow{stt} t : \tau \to \sigma \qquad \Sigma \xrightarrow{stt} s : \tau}{\Sigma \xrightarrow{stt} (ts) : \sigma}$$

$$\frac{\Sigma + d : \tau \xrightarrow{stt} [c \mapsto d]t : \sigma}{\Sigma \xrightarrow{stt} \lambda c.t : \tau \to \sigma}$$

Figure 1: Proof rules for the syntax of simply typed λ-terms

The proof rules of Figure 1 show that tokens can change status from being a constant (*i.e.* a member of signature) to being a bound variable. Context will make it clear when a given token is considered a constant or a bound variable.

Substitution and the rules of α, β, and η conversion are defined as usual. Expressions of the form $\lambda x\ (t\ x)$ are called η-redexes (provided x is not free in t) while expressions of the form $(\lambda x\ t)s$ are called β-redexes. A term is *λ-normal* if it contains no β or η-redexes. The binary relation $\lambda conv$, denoting *λ-conversion*, is defined so that $t\ \lambda conv\ s$ if there is a list of terms $t_1, \ldots, t_n$, with $n \geq 1$, t equal to t_1, s equal to t_n, and for $i = 1, \ldots, n-1$, either t_i converts to t_{i+1} or t_{i+1} converts to t_i by α, β, or η. Every term can be converted to a λ-normal term, and that normal term is unique up to the name of bound variables. We shall consider two terms, say t and s, identical if they are α-convertible to one another and in such a case we shall write $t = s$. The operator $[x_1 \mapsto s_1, \ldots, x_n \mapsto s_n]$ denotes the operation of simultaneous substitution, systematically changing bound variables in order to avoid variable capture, and then returning the λ-normalized result.

Following Church [2], we introduce logic over these terms by assuming that the primitive type o, meant to denote propositions, is always given as a member of S. Predicate types are type expressions of the form

$$\tau_1 \to \cdots \to \tau_n \to o \quad (n \geq 0)$$

where the type expressions $\tau_1, \ldots, \tau_n$ do not contain o. Signatures are constrained so that if a type in it contains o, that type must be a predicate type. If Σ assigns a token a predicate type, that token is called a predicate (via Σ). Equality can be admitted as a special predicate, but it shall not be used in this paper. The following defines the class of *Σ-formulas.*

- If A is a λ-normal Σ-term of type o then A is an atomic Σ-formula.
- If B and C are Σ-formulas then $B \wedge C$ and $B \supset C$ are Σ-formulas.
- If $[c \mapsto d]B$ is a Σ+$d{:}\tau$-formula then $\forall_\tau c.B$ is a Σ-formula. In this paper we shall also assume the additional restriction that the type τ does not contain the primitive type o. Thus, predicate quantification is not permitted in this logic. There are various ways to allow forms of predicate quantification in this setting: one approach is described in [17, 19] and another is described in [13]. The kinds of meta-programs that we discuss here do not require any forms of predicate quantification.

These formulas will now be restricted using the proof system in Figure 2. Let $\mathcal{Q}$ denote a list of universal and existential quantifiers in which the quantified variables are all distinct. We write $\mathcal{Q} \vdash_T t : \tau$ if the sequent $\mathcal{Q} \xrightarrow{T} t : \tau$ is provable, $\mathcal{Q} \vdash_D B$ if the sequent $\mathcal{Q} \xrightarrow{D} B$ is provable, and $\mathcal{Q} \vdash_G B$ if the sequent $\mathcal{Q} \xrightarrow{G} B$ is provable. This proof system has four provisos. The first two, (α) and (†), deal with only bound variable names and hence are not significant restrictions. The remaining two restrictions are of more consequence.

(α) The terms t and t' are related by α-conversion.
(†) The variable x does not occur in $\mathcal{Q}$.
(‡) The quantifier prefix $\mathcal{Q}$ contains $\forall h$ where the type on the quantifier is $\tau_1 \to \cdots \to \tau_n \to \tau$ $(n \geq 0)$.
(♯) The variable y is existentially quantified in $\mathcal{Q}$ with type $\tau_1 \to \cdots \to \tau_n \to \tau$ to the left of where the distinct variables $x_1, \ldots, x_n$ are universally quantified with types $\tau_1, \ldots, \tau_n$, respectively $(n \geq 0)$.

Let Σ be a given signature and let $\mathcal{Q}_\Sigma$ be the prefix that is an enumeration of the quantifiers $\forall_\tau x$, for each pair $x{:}\tau \in \Sigma$, in some arbitrary but fixed order. A *goal formula* of L_λ is a Σ-formula G so that $\mathcal{Q}_\Sigma \vdash_G G$. A *definite clause* (or *program clause*) of L_λ is a Σ-formula D so that $\mathcal{Q}_\Sigma \vdash_D D$.

The restrictions that these additional syntax rules provide are of two kinds. First is the mild restriction that if a definite clause is an implication, its consequence must be atomic. Given the intuitionistic equivalences

$$\begin{aligned} B_1 \supset (B_2 \wedge B_3) &\equiv (B_1 \supset B_2) \wedge (B_1 \supset B_3) \\ B_1 \supset (B_2 \supset B_3) &\equiv (B_1 \wedge B_2) \supset B_3 \\ B_1 \supset \forall_\tau x B_2 &\equiv \forall_\tau x (B_1 \supset B_2), \end{aligned}$$

(provided in the last case that x is not free in B_1) such a restriction is largely superficial: proofs involving formulas on either side of $\equiv$ differ only trivially. Much more restrictive are the kinds of applications that can be formed using variables bound at the top-level of a definite clause. In particular, if $\forall_\tau y D$ is a definite formula, then the only occurrences of y in D are in subterms of the form $y x_1 \cdots x_n$, where $n \geq 0$ and $x_1, \ldots, x_n$ are distinct tokens bound by either negative universal quantifiers occurrences in D or by internal λ-abstractions.

$$\frac{Q \xrightarrow{T} t : \tau}{Q \xrightarrow{T} t' : \tau}\,\alpha \qquad \frac{Q \xrightarrow{G} t}{Q \xrightarrow{G} t'}\,\alpha \qquad \frac{Q \xrightarrow{D} t}{Q \xrightarrow{D} t'}\,\alpha$$

$$\frac{Q\forall_\tau x \xrightarrow{T} t : \sigma}{Q \xrightarrow{T} \lambda x.t : \tau \rightarrow \sigma}\,\dagger \qquad \frac{}{Q \xrightarrow{T} y x_1 \cdots x_n : \tau}\,\sharp \qquad \frac{Q \xrightarrow{T} t_1 : \tau_1 \quad \ldots \quad Q \xrightarrow{T} t_n : \tau_n}{Q \xrightarrow{T} h t_1 \cdots t_n : \tau}\,\ddagger$$

$$\frac{Q\forall_\tau x \xrightarrow{G} G}{Q \xrightarrow{G} \forall_\tau x.G}\,\dagger \qquad \frac{Q \xrightarrow{G} G_1 \qquad Q \xrightarrow{G} G_2}{Q \xrightarrow{G} G_1 \wedge G_2} \qquad \frac{Q \xrightarrow{D} D \qquad Q \xrightarrow{G} G}{Q \xrightarrow{G} D \supset G} \qquad \frac{Q \xrightarrow{T} A : o}{Q \xrightarrow{G} A}$$

$$\frac{Q\exists_\tau x \xrightarrow{D} D}{Q \xrightarrow{D} \forall_\tau x.D}\,\dagger \qquad \frac{Q \xrightarrow{D} D_1 \qquad Q \xrightarrow{D} D_2}{Q \xrightarrow{D} D_1 \wedge D_2} \qquad \frac{Q \xrightarrow{G} G \qquad Q \xrightarrow{T} A : o}{Q \xrightarrow{D} G \supset A} \qquad \frac{Q \xrightarrow{T} A : o}{Q \xrightarrow{D} A}$$

Figure 2: Proof rules for the syntax of L_λ

All first-order positive Horn clauses are legal definite clauses and goal formulas. Let Σ_1 be the signature $\{p : (i \rightarrow i) \rightarrow i \rightarrow i \rightarrow o, f : (i \rightarrow i) \rightarrow i\}$. The following Σ_1-formula is an example of a definite clause:

$$\forall_i t \forall_{i \rightarrow i} n \forall_{i \rightarrow i \rightarrow i} m[\forall_i y(p\ (\lambda x.y)\ t\ y \supset p\ (\lambda x.m\ x\ y)\ t\ (n\ y)) \supset p\ (\lambda x.f(m\ x))\ t\ (f\ n)].$$

Section 8 contains several examples of L_λ programs.

Before we present a proof system for L_λ, we need the following *elaboration* function that maps definite formulas to slightly more restricted definite formulas:

- elab$(A) = \{A\}$,
- elab$(G \supset A) = \{G \supset A\}$,
- elab$(D_1 \wedge D_2) =$ elab$(D_1) \cup$ elab(D_2), and
- elab$(\forall x.D) = \{\forall x.D' \mid D' \in$ elab$(D)\}$.

Elaboration essentially breaks a D-formula into its conjuncts. If a conjunction occurs in a formula of elab(D), that occurrence is in the scope of an implication. The conjunction of all the formulas in elab(D) is intuitionistically equivalent to D.

A sequent (for L_λ) is a triple $\Sigma\ ;\ \mathcal{P} \longrightarrow G$ where $Q_\Sigma \vdash_G G$ and for all $D \in \mathcal{P}$, $Q_\Sigma \vdash_D D$ and all occurrences of conjunctions in D are in the scope of an implication. A sequent-style proof system for L_λ is given in Figure 3. The proviso $\dagger$ states that there must be some formula

$$\forall_{\tau_1} x_1 \ldots \forall_{\tau_n} x_n (G' \supset A') \in \mathcal{P} \quad (n \geq 0)$$

and terms t_i so that $Q_\Sigma \vdash_T t_i : \tau_i$ $(i = 1, \ldots, n)$ and $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]G' = G$ and $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]A' = A$. A sequent is *initial* if it is of the form $\Sigma\ ;\ \mathcal{P} \longrightarrow A$ for atomic A such that there is some formula $\forall_{\tau_1} x_1 \ldots \forall_{\tau_n} x_n.A' \in \mathcal{P}$ and terms t_i so that $\Sigma \vdash_T t_i : \tau_i$ and $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]A' = A$. Proof trees built in the usual way from these initial sequents using the rules in Figure 3 will be called *goal-directed* proofs. As the following theorem states, restricting to goal-directed proofs for L_λ yields a logic that is sound and complete for intuitionistic logic (formulated to permit empty types).

$$\frac{\Sigma\,;\,\mathcal{P} \longrightarrow G_1 \qquad \Sigma\,;\,\mathcal{P} \longrightarrow G_2}{\Sigma\,;\,\mathcal{P} \longrightarrow G_1 \wedge G_2}\ \wedge\text{-R}$$

$$\frac{\Sigma\,;\,\mathrm{elab}(D),\mathcal{P} \longrightarrow G}{\Sigma\,;\,\mathcal{P} \longrightarrow D \supset G}\ \supset\text{-R} \qquad \frac{\Sigma + c:\tau\,;\,\mathcal{P} \longrightarrow [x \mapsto c]B}{\Sigma\,;\,\mathcal{P} \longrightarrow \forall_\tau x\, B}\ \forall\text{-R}$$

$$\frac{\Sigma\,;\,\mathcal{P} \longrightarrow G}{\Sigma\,;\,\mathcal{P} \longrightarrow A}\ \mathrm{BC}^\dagger$$

Figure 3: Proof rules for L_λ

Theorem 3.1. *Let $\Sigma\,;\,\mathcal{P} \longrightarrow G$ be a sequent. The formula G follows from Σ and $\mathcal{P}$ intuitionistically if and only if this sequent has a goal-directed proof.*
Proof. Proofs in [12] and in [17] can be easily modified to prove this theorem. It can be proved directly by transforming a given cut-free proof into a goal-directed proof by permuting inference rules in a straightforward fashion. QED

As a result of this theorem, the operational interpretation of provability presented in the previous section can be applied to L_λ.

4. An interpreter for L_λ

An interpreter for L_λ can be implemented to search for goal-directed proofs by attempting to build them in a bottom-up fashion. The BACKCHAIN step is, of course, the most difficult since it requires chosing a clause and terms to substitute into that clause. We shall present a standard technique using "free" variables and unification to help in chosing such substitution terms.

In such an interpreter, it is necessary, in some fashion, to keep track of notions such as the "current goal," the "current program," the "current set of constants," and restrictions on free variables. Interpreters for Horn clauses need to keep track of only the first of these: there the current program and set of constants remains unchanged during a computation, and restrictions on free variables do not need to be made. In the description of an interpreter for L_λ given below, a *quantifier prefix* is used to encode both the current set of constants and the restrictions on free variables, and a *sequent* is used to connect a program to a goal.

Let $\mathcal{Q}$ be a quantifier prefix, *i.e.* a list of universally and existentially quantified distinct tokens, and let Σ be a signature. Signatures and prefixes are similar in their use below, although quantifier prefixes have more structure. In particular, we have already defined $\mathcal{Q}_\Sigma$ as a way to map a signature into a prefix. Similarly, we define $\Sigma_\mathcal{Q}$ to be the signature that contains the pair $x : \tau$ if $\forall_\tau x$ is in $\mathcal{Q}$ (existential quantifiers are ignored).

A *$\mathcal{Q}$-substitution* is a substitution that contains a substitution term for precisely the existentially quantified variables in $\mathcal{Q}$. Furthermore, if $\mathcal{Q}$ is of the form $\mathcal{Q}'\exists_\tau x\mathcal{Q}''$ and the pair $x \mapsto t$ is in such a substitution, then t is λ-normal and $\Sigma, \Sigma_{\mathcal{Q}'} \vdash_{stt} t : \tau$. In other words, the only tokens free in t are either in Σ or are universally quantified to the left of x. In this sense, a $\mathcal{Q}$-substitution is a closed substitution; that is, substitution terms do not contain existentially quantified variables. Two $\mathcal{Q}$-substitutions, say φ and ψ, are equal if for each $\exists x$ in $\mathcal{Q}$, $\varphi x = \psi x$. In that case, we write $\varphi = \psi$.

It is possible that for a given Σ and $\mathcal{Q}$, there may not be any $\mathcal{Q}$-substitutions. For example, if Σ is the signature $\{f : i \to i, g : i \to i \to i\}$ and $\mathcal{Q}$ is the prefix $\exists_i x$, there is no $\mathcal{Q}$-substitution since there is no λ-term whose only free tokens are f and g and which has type i. That is, the type i is, in a sense, empty. Since the problem of determining if there is a $\mathcal{Q}$-substitution for a given $\mathcal{Q}$ and Σ reduces to proving theorems in the implicational fragment of intuitionistic logic, this problem is decidable [25] (this assumes that there is only a finite set of distinct types in Σ).

A $\mathcal{Q}$-sequent is a pair $\mathcal{P} \longrightarrow G$ where $\mathcal{Q}_\Sigma\mathcal{Q} \vdash_G G$ and where for every D in the finite set $\mathcal{P}$, $\mathcal{Q}_\Sigma\mathcal{Q} \vdash_D D$. Furthermore, we assume that D is the result of an elaboration; that is, every occurrence of a conjunction in D is in the scope of an implication. A *state* (of the interpreter) is a triple $\langle\theta, \mathcal{Q}, \mathcal{S}\rangle$ where θ is a substitution, $\mathcal{Q}$ is a quantifier prefix, and $\mathcal{S}$ is a finite set of $\mathcal{Q}$-sequents. These sequents specify what remains to be proved for the interpretation to be successful. A *success state* is a state in which the set of sequents is empty.

A substitution φ *satisfies* a prefix $\mathcal{Q}$ and a set of $\mathcal{Q}$-sequents $\mathcal{S}$ if φ is a $\mathcal{Q}$-substitution such that for every sequent $\mathcal{P} \longrightarrow G$ in $\mathcal{S}$ it is the case that $\Sigma \cup \Sigma_\mathcal{Q}, \varphi\mathcal{P} \vdash_I \varphi G$. If the set $\mathcal{S}$ is empty, this condition reduces to requiring that φ is just a $\mathcal{Q}$-substitution. The purpose of an interpreter is to search for the existence of satisfying substitutions for the prefix and sequents of its state. Checking satisfiability is, of course, not decidable.

The following transition rules describe the heart of a non-deterministic interpreter. The operation $\uplus$ denotes disjoint union.

AND: Given the state $\langle\theta, \mathcal{Q}, \{\mathcal{P} \longrightarrow G_1 \wedge G_2\} \uplus \mathcal{S}\rangle$, change to the state

$$\langle\theta, \mathcal{Q}, \{\mathcal{P} \longrightarrow G_1, \mathcal{P} \longrightarrow G_2\} \cup \mathcal{S}\rangle.$$

This transition simply translates the logical connective $\wedge$ into an AND-node in the interpreter's search space.

AUGMENT: Given the state $\langle\theta, \mathcal{Q}, \{\mathcal{P} \longrightarrow D \supset G\} \uplus \mathcal{S}\rangle$, change to the state

$$\langle\theta, \mathcal{Q}, \{\text{elab}(D) \cup \mathcal{P} \longrightarrow G\} \cup \mathcal{S}\rangle.$$

An implication in a goal is thus an instruction to augment the program with the (elaboration of the) antecedent of the implication.

GENERIC: Given the state $\langle\theta, \mathcal{Q}, \{\mathcal{P} \longrightarrow \forall_\tau x.G\} \uplus \mathcal{S}\rangle$, change to the state

$$\langle\theta, \mathcal{Q}\forall_\tau y, \{\mathcal{P} \longrightarrow [x \mapsto y]G\} \cup \mathcal{S}\rangle,$$

where y is a token not bound in $\mathcal{Q}_\Sigma\mathcal{Q}$. A universal quantifier in a goal causes a new constant to be added to the current signature.

The following two transitions assume the existence of a unification algorithm. This will be described in the next section. The algorithm is called by unify$(\theta, \mathcal{Q}P)$, where θ is a substitution that is being accumulated, $\mathcal{Q}$ is a quantifier prefix, and P is a list of typed equations between terms. The algorithm returns either *fail* or a pair, $\langle\theta', \mathcal{Q}'\rangle$, where $\mathcal{Q}'$ is a quantifier prefix and θ' is the new accumulated substitution. The symbol $\emptyset$ will denote the empty substitution.

BACKCHAIN: Consider the state $\langle\theta, \mathcal{Q}, \{\mathcal{P} \longrightarrow A'\} \uplus \mathcal{S}\rangle$ where

$$\forall x_1 \ldots \forall x_n (G \supset A) \in \mathcal{P} \quad (n \geq 0)$$

and A and A' are atomic formulas. Using α-conversion, we may assume that $x_1, \ldots, x_n$ are not bound in $\mathcal{Q}_\Sigma\mathcal{Q}$. If unify$(\emptyset, \mathcal{Q}\exists x_1 \ldots \exists x_n [A \doteq A']) = \langle\rho, \mathcal{Q}'\rangle$ then the interpreter can make a transition to the state

$$\langle\theta \circ \rho, \mathcal{Q}', \rho(\{\mathcal{P} \longrightarrow G\} \cup \mathcal{S}'\rangle).$$

(Composition is defined as $(\theta \circ \rho)(x) = \rho(\theta(x))$. The application of ρ to a set of sequents is the set of sequents resulting from applying ρ to all formulas in all the sequents.)

CLOSE: Consider the state $\langle\theta, \mathcal{Q}, \mathcal{S}\rangle$ where $\mathcal{S}$ is the set

$$\{\{\forall x_1 \ldots \forall x_n . A\} \cup \mathcal{P} \longrightarrow A'\} \uplus \mathcal{S}'$$

for some set $\mathcal{S}'$ and for $n \geq 0$ and for atomic formulas A and A'. Using α-conversion, we may assume that $x_1, \ldots, x_n$ are not bound in $\mathcal{Q}_\Sigma\mathcal{Q}$. If

$$\text{unify}(\emptyset, \mathcal{Q}\exists x_1 \ldots \exists x_n [A \doteq A']) = \langle\rho, \mathcal{Q}'\rangle$$

then the interpreter can make a transition to the state $\langle\theta \circ \rho, \mathcal{Q}', \rho\mathcal{S}'\rangle$.

No transition can be applied to a success state.

The formal correctness properties for this interpreter will be proved in Section 7. We can now describe a simple, depth-first, deterministic interpreter for L_λ. First, we must consider the third component of a state and the antecedent of sequents as lists instead of sets. AUGMENT concatenates elaborated clauses to the front of an antecedent. When given a non-success state, the first sequent is used to determine which transition to consider. If the succedent of that sequent is an implication, apply AUGMENT; if it is a conjunction, apply AND; if it is universally quantified, apply GENERIC. The choice of new token used in GENERIC is immaterial. Finally, if the succedent is an atom, then we need to use BACKCHAIN or CLOSE. Here, we select a D-formula from the antecedent in a left-to-right order and see if either of these transitions can be applied. The only backtrack points we must store are those involved with the selection of this clause: these backtrack points will be returned to in a depth-first fashion. This style of search, although incomplete, is similar to the ones used in Prolog and λProlog. In this paper, we shall be mostly interested in the non-deterministic version of this interpreter.

5. A unification algorithm for L_λ

In order to complete the interpreter for L_λ, we need to describe a unification algorithm. That is, given a quantifier prefix $\mathcal{Q}$ and a finite collection of typed equalities, $t_1 \overset{\tau_1}{=} s_1, \ldots, t_n \overset{\tau_n}{=} s_n$, so that $\mathcal{Q}_\Sigma\mathcal{Q} \vdash_T t_i : \tau_i$ and $\mathcal{Q}_\Sigma\mathcal{Q} \vdash_T s_i : \tau_i$ for each $i = 1, \ldots, n$, we wish to determine whether or not there is a $\mathcal{Q}$-substitution φ such that for $i = 1, \ldots, n$, $\varphi t_i = \varphi s_i$. Furthermore, we shall need to characterize all such φ if they exist. The BACKCHAIN and CLOSE transitions will call this algorithm with an initial quantifier prefix and a single, typed equation $A \overset{o}{=} A'$.

Let Σ be a fixed signature. A *unification problem* is a quantified list of equations, written

$$\mathcal{Q}[t_1 \overset{\tau_1}{=} s_1, \ldots, t_n \overset{\tau_n}{=} s_n], \quad (n \geq 0) \tag{$*$}$$

where $\mathcal{Q}$ is a quantifier prefix, and $\mathcal{Q}_\Sigma\mathcal{Q} \vdash_T t_i{:}\tau_i$ and $\mathcal{Q}_\Sigma\mathcal{Q} \vdash_T s_i{:}\tau_i$ for each $i = 1, \ldots, n$. (These unification problems are examples of the *variable-defining* unification problems described in [15].) If n is zero, we shall write [] to denote the empty list. A *solution* to ($*$) is a $\mathcal{Q}$-substitution φ such that for $i = 1, \ldots, n$, φt_i and φs_i are λ-convertible.

In describing a unification algorithm below, we shall drop most mentioning of types since the equations we need to unify can be solved over λ-normal, untyped terms. That is, if we delete all references to types in ($*$) above, we can decide whether or not that unification problem has a solution. We can also produce a substitution, which is unique up to λ-conversion and variable renamings, so that all instances of it that are $\mathcal{Q}$-substitutions are precisely the set of solutions. This substitution essentially corresponds to a *most general unifier*, a technical term we shall not need here. In this setting, the untyped and typed calculus differ only on the typing restriction in η-conversion. The only care that we must exercise below is to use η-expansion only in situations where, in the typed setting, the expansion is performed on a term that would have to have functional type. Otherwise, types play no role in this unification algorithm, something that is not true for the problem of unifying unrestricted simply typed λ-terms [9].

Types do play a role, however, in the interpretation of the results of the unification algorithm. If unify$(\emptyset, \mathcal{Q}P) = \langle\theta, \mathcal{Q}'\rangle$ then (as stated in Proposition 6.4) $\mathcal{Q}'$-substitutions and the solutions to $\mathcal{Q}P$ can be placed in a one-to-one correspondence. While this is true of both the typed and untyped cases, in the untyped case, there are always $\mathcal{Q}'$-substitutions for all $\mathcal{Q}'$ while in the typed case, there may not be $\mathcal{Q}'$-substitutions. Thus, in the typed case, the unification algorithm may return a non-failure value and there still may not be solutions to the given unification problem.

In the remainder of this paper, the expression $t = s$ can mean either the proposition that t and s are α-convertible or the pair of terms t and s is a member of a unification problem. The choice of these two interpretations should always be clear from context. The use of a bar over a letter, *e.g.* $\bar{x}$, will be used as shorthand to refer to some list, such as $x_1, \ldots, x_n$, for some index n that is determined from context or whose value is not needed.

The state of the unification algorithm is a pair, $\langle\theta, \mathcal{Q}P\rangle$, where θ is a substitution that is being accumulated and $\mathcal{Q}P$ is a unification problem that needs to be solved. The algorithm is a loop out of which it is possible to fail. In that case, the algorithm returns the keyword *fail*. Otherwise, the algorithm returns a pair, such as $\langle\theta', \mathcal{Q}'\rangle$, where θ' is a substitution and $\mathcal{Q}'$ is a quantifier prefix. The expression unify$(\theta, \mathcal{Q}P)$ denotes the value the unification algorithm returns when applied to the state $\langle\theta, \mathcal{Q}P\rangle$. The algorithm's description is given below. There is an initial check for an immediate, successful termination. If that is not possible, then three preprocessing steps — simplification, raising, and pruning — are applied to the first equation in the unification problem. After that preprocessing, the first equation of the processed unification problem is either solved completely or causes a failure. The algorithm then repeats all these steps. It is convenient to introduce the following two terms: if t is a term in a unification problem with prefix $\mathcal{Q}$, t is *flexible* if its head is existentially quantified in $\mathcal{Q}_\Sigma\mathcal{Q}$ and is *rigid* otherwise, *i.e.* its head is universally quantified in $\mathcal{Q}_\Sigma\mathcal{Q}$.

Successful termination. Given a state of the form $\langle\theta, \mathcal{Q}[]\rangle$ return $\langle\theta, \mathcal{Q}\rangle$.

If this step is not applicable, then the state is of the form $\langle\theta, \mathcal{Q}[t = s, P]\rangle$, where P is a list of all the equations after the first (P may be empty). There are two kinds of simplifications that are applied to this state. Repeatedly apply both of these two simplifications to the state until they return a failure or are both no longer applicable.

Simplification step 1. We can write t as $\lambda\bar{x}.t'$ and s as $\lambda\bar{y}.s'$, where t' and s' are not themselves abstractions. If the lists of binders $\lambda\bar{x}$ and $\lambda\bar{y}$ are not of equal length, then use η-expansions to increase the length of the shorter binder until they are of the same length. Notice that in the typed case, if it is $\lambda\bar{x}$ that is the shorter, then t' would be of functional type, and such an η-expansion would be permitted. Using α-conversion, we may assume that these two binders are the same, that is, $t = s$ can be written as $\lambda\bar{w}.t'' = \lambda\bar{w}.s''$, and that the variables in $\bar{w}$ are not bound in $\mathcal{Q}_\Sigma\mathcal{Q}$. Then change to the state $\langle\theta, \mathcal{Q}\forall\bar{w}[t'' = s'', P]\rangle$.

Simplification step 2. If the first equation $t = s$ has the form $ht_1 \ldots t_n = hs_1 \ldots s_n$, where h is universally quantified in $\mathcal{Q}_\Sigma\mathcal{Q}$, replace the equation $t = s$ with the equations $t_1 = s_1, \ldots t_n = s_n$. If the first equation $t = s$ has the form $ht_1 \ldots t_n = ks_1 \ldots s_m$, where h and k are different universally quantified variables in $\mathcal{Q}_\Sigma\mathcal{Q}$, then return *fail.*

If we have reached this point and neither of these rules are applicable, then at least one of the terms in the first equation is flexible. We may assume, therefore, that the first term is flexible, since otherwise we simply switch around the equation. Thus, the unification algorithm has a state of the form

$$\langle\theta, \mathcal{Q}[vy_1 \ldots y_n = t, P]\rangle, \qquad (**)$$

where v is existentially quantified in $\mathcal{Q}$, the distinct variables $y_1, \ldots, y_n$ are universally quantified to the right of v in $\mathcal{Q}$, and t is some term (either flexible or rigid). We next perform the raising and pruning steps. These steps test for and simplify the interdependencies of universally and existentially quantified variables.

Raising step. Is there an existential variable u free in t and such that the prefix is of the form $\mathcal{Q}_1\exists v\mathcal{Q}_2\exists u\mathcal{Q}_3$ where $\mathcal{Q}_2$ has at least one universally quantified variable? If not, go to the next step. Otherwise, let $\bar{w}$ be the list of universally quantified tokens that occur in $\mathcal{Q}_2$. Set $\rho = [u \mapsto u'\bar{w}]$, where u' is not bound in $\mathcal{Q}_\Sigma\mathcal{Q}$, and change state to $\langle\theta \circ \rho, \mathcal{Q}_1\exists v\exists u'\mathcal{Q}_2\mathcal{Q}_3[v\bar{y} = \rho t, \rho P]\rangle$. Repeat this step until it no longer applies. Notice that the fact that the tokens in $\bar{w}$ may appear in substitution terms for u is made explicit by replacing u with a "higher-type" token u', which may not be instantiated with a term containing those tokens, applied explicitly to $\bar{w}$. Repeat this step until it is no longer applicable.

Pruning step. Is there a universal variable z of $\mathcal{Q}$ that has a free occurrence in t, is to the right of v, and is not in the list $\bar{y}$? If not, go to the next step. If so and if that occurrence in t is not in the scope of an existentially quantified variable, then return *fail.* Otherwise, z occurs in a subformula occurrence of the form $u\bar{w}_1 z\bar{w}_2$ of t where u is existentially quantified in $\mathcal{Q}$ and $\bar{w}_1$ and $\bar{w}_2$ are lists of universally quantified variables to the right of u. We must prune away u's dependency on the argument occupied by z. This is done by setting ρ to the substitution $[u \mapsto \lambda\bar{w}_1\lambda z\lambda\bar{w}_2.u'\bar{w}_1\bar{w}_2]$, where u' is not bound in $\mathcal{Q}_\Sigma\mathcal{Q}$. Update the state to be

$$\langle \theta \circ \rho, \mathcal{Q}'[\rho(vy_1 \cdots y_n) = \rho t, \rho P]\rangle,$$

where $\mathcal{Q}'$ is the same as $\mathcal{Q}$ except that $\exists u'$ replaces $\exists u$. Repeat this step until it is no longer applicable.

If we reach here, it must be the case that the unification problem is still of the form (**) and that all the universally quantified tokens to the right of v that are free in t are in the list $\bar{y}$, and that a universally quantified token to the left of an existentially quantified variable free in t is also to the left of v.

Flexible-flexible step. First, we deal with the case that the term t is also flexible; that is, it has the form $uz_1 \ldots z_p$ where $z_1, \ldots, z_p$ are distinct universally quantified variables bound to the right of where u is existentially bound. It must be the case that the variables in $\bar{z}$ are contained in the list $\bar{y}$ and that there are no universal quantifiers to the right of $\exists v$ which are also to the left of $\exists u$ in the prefix. We distinguish two cases.

Case 1. Assume that v and u are different. Set ρ to the substitution $[v \mapsto \lambda\bar{y}.u\bar{z}]$. Change state to $\langle \theta \circ \rho, \mathcal{Q}'\rho P\rangle$, where $\mathcal{Q}'$ is the result of deleting $\exists v$ from $\mathcal{Q}$.

Case 2. Assume that v and u are equal. Then the pair is of the form $vy_1 \ldots y_n = vz_1 \ldots z_n$ where $y_1 \ldots y_n$ is a permutation of $z_1 \ldots z_n$. Let $\bar{w}$ be the enumeration of the set $\{y_i \mid y_i = z_i, i \in \{1, \ldots, n\}\}$ which orders variables the same as they are ordered in $\bar{y}$ (the choice of this particular ordering is not important). Set ρ to the substitution $[v \mapsto \lambda\bar{y}.v'\bar{w}]$ (notice that this is the same via α-conversion to $[v \mapsto \lambda\bar{z}.v'\bar{w}]$), where v' is not quantified in $\mathcal{Q}_\Sigma\mathcal{Q}$. Change state to $\langle \theta \circ \rho, \mathcal{Q}'\rho P\rangle$, where $\mathcal{Q}'$ is the result of replacing $\exists v$ with $\exists v'$.

Occurrence check step. The only remaining case occurs when the term t is rigid. If the variable v occurs free in t then return *fail.* Otherwise, set ρ to the substitution $[v \mapsto \lambda\bar{y}.t]$ and set the state to $\langle \theta \circ \rho, \mathcal{Q}'\rho P\rangle$, where $\mathcal{Q}'$ is the result of deleting $\exists v$ from $\mathcal{Q}$.

This completes the description of the unification algorithm. In the next section we prove its formal correctness. It is worth making the following simple observations about this unification algorithm here.

(1) The cases above are exhaustive. That is, no matter what equation occurs first in the given unification problem, some step of the algorithm can be applied to it.

(2) There are four places where this unification algorithm is nondeterministic: the choice of the name of binders in the first simplification step; the choice of which order to raise existential variables; the choice of which order to prune existential variables; and the choice of what name to give the new variables. In each case, the resulting unification problems differ either up to α-conversion or in the order of existential variables within a sequence of existential variables. Neither difference is significant for our purposes here, and we shall assume that unification problems

are equal modulo such differences. Thus, if the unification algorithm terminates, its value is uniquely determined.

(3) The substitutions ρ that are constructed and applied to terms in unification problems are such that one existentially quantified variable is substituted with a term and the result of performing that substitution and subsequent λ-normalization yields another unification problem; that is, occurrences of existentially quantified variables in the resulting problem are properly restricted.

(4) When a substitution is applied, the only new β-redexes are those of the form $(\lambda x.t)y$ where y is a token that is either universally quantified or is λ-bound and is such that y is not free in $\lambda x.t$. That is, full β-conversion is not required. In particular, let β_0 be the equation $(\lambda x.t)x = t$; that is, we can only perform a β-reduction if it involves a token that is not free in the abstraction. The restrictions on terms in L_λ are such that the equality theory that is being considered is only that of α, β_0, and η.

(5) The unification procedure described in [9] can be applied to unification problems with unrestricted function variable application and with purely existentially quantified prefixes. As described in [15], it is straightforward to extend Huet's procedure to the mixed quantifier prefix case described here. The process in [9], however, computes pre-unifiers instead of unifiers; that is, it finds substitutions that reduce unification problems into just problems that contain flexible-flexible equations. In the general, unrestricted setting, computing unifiers for flexible-flexible equations is a very unconstrained and undirected procedure, so it is often best avoided. In the L_λ case, however, flexible-flexible equations are so simple that their solutions can be completely characterized. Thus, if Huet's procedure is modified to handle a mixed quantifier prefix and to solve those flexible-flexible equations as describe above, then it would also serve as a complete unification procedure for L_λ.

6. Correctness of the unification algorithm

We first show that the unification algorithm terminates. For this, we need some measures on unification problems. If t is a λ-normal term all of whose free variables are contained in the quantifier prefix $\mathcal{Q}$, the measure $|t|$ counts the number of occurrences of applications in t that are not in the scope of existentially quantified variables of $\mathcal{Q}$. (Of course, $|t|$ also has $\mathcal{Q}$ as an argument, but its value will always be clear from context.) That is, $|t|$ is defined by

$$|\lambda\bar{x}(ht_1 \dots t_n)| = \begin{cases} 0 & \text{if } h \text{ is existentially quantified in } \mathcal{Q} \\ n + \sum_{i=1}^{n} |t_i| & \text{if } h \text{ is universally quantified in } \mathcal{Q} \end{cases} \quad (n \geq 0).$$

Let $\mathcal{Q}P$ be a unification problem where P is the list $[t_1 = s_1, \dots, t_n = s_n]$ and $\mathcal{Q}$ contains m existentially quantified variables. The measure associated to this unification problem is defined by the triple

$$|\mathcal{Q}P| = \langle m, \sum_{i=1}^{n} |t_i| + |s_i|, n \rangle.$$

Triples are ordered lexicographically.

Theorem 6.1. *The unification algorithm always terminates.*
Proof. The unification algorithm is a looping program that first applies the simplification, raising, and pruning steps, and then attempts to construct a substitution that solves the first equation in a most general fashion. We shall show that the measure of a unification problem does not rise during the simplification, raising, and pruning steps, while it decreases when an equation is removed.

The simplification step must terminate since it decreases the total number of applications and abstractions that occur in the unification problem. If $Q'P'$ arises from applying either simplification step to QP, then $|Q'P'| \leq |QP|$.

The raising step terminates since there are only a finite number of existential quantifiers and these can only be moved a finite number of times to the left. Raising may cause the number of applications in a unification problem to increase, but all new occurrences of applications are in the scope of an existentially quantified variable. Hence, if $Q'P'$ arises from applying the raising step to QP, then $|Q'P'| = |QP|$.

The pruning step terminates since there are only a finite number of universal quantifiers that can be pruned. Pruning will cause the number of applications in a unification problem to decrease, but all deleted occurrences of applications are in the scope of an existentially quantified variable. Hence, if $Q'P'$ arises from applying the pruning step to QP, then $|Q'P'| = |QP|$.

We shall now show that after every successful application of a remaining step, the resulting unification problem has a measure strictly less than the original unification problem.

In the first flexible-flexible step, the number of existentially quantified variables decreases by one. Hence, the overall measure decreases. In the second case, the number of existentially quantified variables and the number of occurrences of applications not in the scope of existentially quantified variables remain the same. Since the number of equations decrease, the overall measure decreases.

Finally, in the flexible-rigid case, if the occurrence check succeeds, then the number of existentially quantified variables decreases by one so the overall measure decreases.

Thus, the main loop of the unification algorithm either returns *fail* or decreases the measure on a unification problem. As a result, the algorithm must terminate. QED

The following series of propositions and lemmas show that the unification algorithm can be used to determine whether or not solutions exist and to characterize all of them if they do exists.

Lemma 6.2. *Assume that the unification algorithm makes a single transition from the state $\langle\theta, QP\rangle$ to the state $\langle\theta \circ \rho, Q'P'\rangle$. The solutions to QP can be put into one-to-one correspondence with the solution to $Q'P'$ so that if the solution φ for QP corresponds to the solution φ' for $Q'P'$ then $\rho \circ \varphi' = \varphi$.*
Proof. We consider each case in which a transition can occur. The result is immediate if the transition was caused by either simplification step. In those two steps, ρ is the empty substitution and the set of solutions do not change.

Assume that the transition was caused by the raising step. That is, the state changed from $\langle\theta, Q_1\exists v Q_2 \exists u Q_3, [v\bar{y} = t, P]\rangle$ to the state $\langle\theta \circ \rho, Q_1\exists v\exists u' Q_2 Q_3, [v\bar{y} = \rho t, \rho P]\rangle$, where ρ is the substitution $[u \mapsto u'\bar{w}]$ and where $\bar{w}$ is the list of universally quantified variables in Q_2. The correspondence of solutions is given by either letting φ'

be the result of replacing $u \mapsto s$ in φ with $u' \mapsto \lambda\bar{w}.s$, or conversely, letting φ be the result of replacing $u' \mapsto r$ in φ' with $u \mapsto r\bar{w}$. Via the rules of λ-conversion, these two descriptions are inverses. The fact that the resulting φ and φ' are substitution for their respective prefixes is easy to check. Since φ and φ' differ only on u and u' and since $(\rho \circ \varphi')u = \varphi'(u'\bar{w}) = (\lambda\bar{w}.s)\bar{w}\ \lambda conv\ s = \varphi u$, it follows that $\rho \circ \varphi' = \varphi$. Notice that raising is a general transition for unification problems: it is dependent only on the prefix of unification problems and not on the actual list of equality pairs. A fuller description of this transition is presented in [15].

Assume that the transition was caused by the pruning step. That is, the state changed from $\langle\theta, \mathcal{Q}_1\exists u\mathcal{Q}_2, [v\bar{y} = t, P]\rangle$ to the state $\langle\theta \circ \rho, \mathcal{Q}_1\exists u'\mathcal{Q}_2, [\rho(v\bar{y}) = \rho t, \rho P]\rangle$, where ρ is the substitution $[u \mapsto \lambda\bar{w}_1\lambda z\lambda\bar{w}_2.u'\bar{w}_1\bar{w}_2]$ and where z is a universal variable in $\mathcal{Q}$ that has a free occurrence in t, is to the right of v, and is not in the list $\bar{y}$, and where $u\bar{w}_1z\bar{w}_2$ is a subterm of t. Thus z is not free in $\varphi(v\bar{y})$ for any solution φ to the first unification problem. Hence, if we write φu as $\lambda\bar{w}_1\lambda z\lambda\bar{w}_2.s$ it must be the case that z is also not free in s, since otherwise, it would be free in φt and hence in $\varphi(v\bar{y})$. Thus, the correspondence of solutions is given by either letting φ' be the result of replacing $u \mapsto \lambda\bar{w}_1\lambda z\lambda\bar{w}_2.s$ in φ with $u' \mapsto \lambda\bar{w}_1\lambda\bar{w}_2.s$, or conversely, letting φ be the result of replacing $u' \mapsto \lambda\bar{w}_1\lambda\bar{w}_2.s$ in φ' with $u \mapsto \lambda\bar{w}_1\lambda z\lambda\bar{w}_2.s$. Since φ and φ' differ only on u and u' and since $(\rho \circ \varphi')u = \varphi'(\lambda\bar{w}_1\lambda z\lambda\bar{w}_2.u'\bar{w}_1\bar{w}_2) = (\lambda\bar{w}_1\lambda z\lambda\bar{w}_2.(\lambda\bar{w}_1\lambda\bar{w}_2.s)\bar{w}_1\bar{w}_2)\ \lambda conv\ \lambda\bar{w}_1\lambda z\lambda\bar{w}_2.s = \varphi u$, it follows that $\rho \circ \varphi' = \varphi$.

Assume that the transition was caused by the first of the flexible-flexible steps. That is, there is a transition from the state $\langle\theta, \mathcal{Q}_1\exists v\mathcal{Q}_2, [v\bar{y} = u\bar{z}, P]\rangle$ to the state $\langle\theta \circ \rho, \mathcal{Q}_1\mathcal{Q}_2, \rho P\rangle$, where the variables of $\bar{z}$ are contained in the list $\bar{y}$, where there are no universal quantifiers to the right of $\exists v$ which are also to the left of $\exists u$ in the prefix, and where ρ is the substitution $[v \mapsto \lambda\bar{y}.u\bar{z}]$. Let φ be a solution to the unification problem in the first state. Then φv is $\lambda\bar{y}.t$ and φu is $\lambda\bar{z}.t$ for some t. Let φ' be the result of deleting the substitution pair for v from φ. (Conversely, from φ' we can insert the substitution term $\lambda\bar{y}.t$ for v given that $\varphi'u$ is $\lambda\bar{z}.t$.) Since $(\rho \circ \varphi')v = \varphi'(\lambda\bar{y}.u\bar{z}) = \lambda\bar{y}.(\lambda\bar{z}.t)\bar{z}\ \lambda conv\ \lambda\bar{y}.t = \varphi v$, we have $\rho \circ \varphi' = \varphi$.

Assume that the transition was caused by the second of the flexible-flexible steps. That is, there is a transition from the state $\langle\theta, \mathcal{Q}_1\exists v\mathcal{Q}_2, [v\bar{y} = v\bar{z}, P]\rangle$ to the state $\langle\theta \circ \rho, \mathcal{Q}_1\exists v'\mathcal{Q}_2, \rho P\rangle$, where $\bar{y}$ is a permutation of $\bar{z}$, ρ is the substitution $[v \mapsto \lambda\bar{y}.v'\bar{w}]$, and $\bar{w}$ is the enumeration of the set $\{y_i \mid y_i = z_i, i = 1, \ldots, n\}$. Let φ be a solution to the unification problem in the first state and let φv be $\lambda\bar{y}.t$, for some term t. Thus, applying φ to the first equation, we have $t = (\lambda\bar{y}.t)\bar{z}$. It is easy to show by induction on the structure of t that if y_i and z_i are not the same token, then y_i cannot be free in t. Thus, only the variables in $\bar{w}$ can be free in t: the others can be pruned. Hence, set φ' to the result of replacing $v \mapsto \lambda\bar{y}.t$ with $v' \mapsto \lambda\bar{w}.t$. (The reverse construction of φ from φ' is immediate.) Given $(\rho \circ \varphi')v = \varphi'(\lambda\bar{y}.v'\bar{w}) = \lambda\bar{y}.(\lambda\bar{w}.t)\bar{w}\ \lambda conv\ \lambda\bar{y}.t = \varphi v$, we again have $\rho \circ \varphi' = \varphi$.

The final case to consider is the flexible-rigid case, that is, the unification algorithm makes a transition from the state $\langle\theta, \mathcal{Q}_1\exists v\mathcal{Q}_2, [v\bar{y} = t, P]\rangle$ to the state $\langle\theta \circ \rho, \mathcal{Q}_1\mathcal{Q}_2, \rho P\rangle$, where v is not free in t and ρ is $[v \mapsto \lambda\bar{y}.t]$. Let φ be a solution to the unification problem in the first state. Thus, φv is some term $\lambda\bar{y}.s$ where s is φt. Let φ' be the substitution resulting from deleting the substitution term for v in φ. Then $(\rho \circ \varphi')v = \varphi'(\lambda\bar{y}.t)$. Since v is not free in t, this latter term is also equal to $\varphi(\lambda\bar{y}.t)$. As a result of raising, φ

does not substitute into any existentially quantified tokens in t terms containing tokens in $\bar{y}$. Thus, $\varphi(\lambda\bar{y}.t)$ is also equal to $\lambda\bar{y}.\varphi t = \lambda\bar{y}.s = \varphi v$. Again we have $\rho \circ \varphi' = \varphi$. QED

Proposition 6.3. *If unify($\theta, \mathcal{Q}P$) = fail then $\mathcal{Q}P$ has no solution.*

Proof. There are three places where the unification algorithm can return *fail*: step 2 of simplification, pruning, and the occurrence check. We show that in each of these cases, the unification problem that caused the problem had no solution.

In the second step of simplification, a failure occurs if the first term is of the form $ht_1 \ldots t_n = ks_1 \ldots s_m$, where h and k are different universally quantified variables in $\mathcal{Q}_\Sigma\mathcal{Q}$. Since instances of these two terms will always have different top-level structure, they can never be made equal and hence a unification problem containing them cannot have a solution.

In the pruning step, assume that the first equation is of the form $v\bar{y} = t$ and that there is a universal variable z of $\mathcal{Q}$ that has a free occurrence in t, is to the right of v in $\mathcal{Q}$, is not in the list $\bar{y}$, and it has an occurrence in t that is not in the scope of an existentially quantified variable. Thus, z occurs free in t and in every substitution instance of t. However, z will not have an occurrence in $\varphi(v\bar{y})$ for any $\mathcal{Q}$-substitution φ. Thus, there can be no solution.

Finally, in the occurrence check step, the first equation is $v\bar{y} = t$ where v occurs in t. Assume that φ is a solution and let φv be $\lambda\bar{y}.s$. Clearly, $|\varphi t| > |s|$ since φt contains a proper subterm that is an alphabetic variant of s. Thus, no occurrence of $v\bar{y}$ can be equal to the same instance of t. QED

Proposition 6.4. *If unify($\emptyset, \mathcal{Q}P$) = $\langle\theta, \mathcal{Q}'\rangle$ then the solutions to $\mathcal{Q}P$ can be put into one-to-one correspondence with $\mathcal{Q}'$-substitutions so that if the solution φ for $\mathcal{Q}P$ corresponds to the $\mathcal{Q}'$-substitution φ' then $\theta \circ \varphi' = \varphi$.*

Proof. Assume that the unification algorithm makes a series of transitions through the states

$$\langle\emptyset, \mathcal{Q}_0P_0\rangle, \langle\rho_1, \mathcal{Q}_1P_1\rangle, \langle\rho_1 \circ \rho_2, \mathcal{Q}_2P_2\rangle, \ldots, \langle\rho_1 \circ \cdots \circ \rho_n, \mathcal{Q}_nP_n\rangle,$$

where $n \geq 1$. Using Lemma 6.2, it is possible to place solutions of $\mathcal{Q}_iP_i$ $(i = 1, \ldots, n)$ in one-to-one correspondence so that, if for $i = 1, \ldots, n$, the selection φ_i as a solution for $\mathcal{Q}_iP_i$ is in such a correspondence, we have

$$\varphi_0 = \rho_1 \circ \varphi_1, \ \ldots, \ \varphi_{n-1} = \rho_n \circ \varphi_n.$$

Thus, $\varphi_0 = \rho_1 \circ \cdots \circ \rho_n \circ \varphi_n$. If unify($\emptyset, \mathcal{Q}P$) = $\langle\theta, \mathcal{Q}'\rangle$ then a series of transitions were made where $\mathcal{Q}_0P_0$ is equal to $\mathcal{Q}P$, $\theta = \rho_1 \circ \cdots \circ \rho_n$, the prefix $\mathcal{Q}'$ is equal to $\mathcal{Q}_n$, and P_n is the empty list. Thus $\mathcal{Q}'$-substitutions, since they are solutions to $\mathcal{Q}_nP_n$, can be placed in one-to-one correspondence with solutions to $\mathcal{Q}P$. If φ' is a $\mathcal{Q}'$-substitution and φ is the corresponding solution to $\mathcal{Q}P$, then $\varphi = \theta \circ \varphi'$. QED

The following proposition follows immediately from the previous propositions.

Proposition 6.5. *The unification problem $\mathcal{Q}P$ has no solution if and only if either unify($\emptyset, \mathcal{Q}P$) = fail or unify($\emptyset, \mathcal{Q}P$) = $\langle\theta, \mathcal{Q}'\rangle$ and there are no $\mathcal{Q}'$-substitutions.*

As we mentioned earlier, one difference between the typed and untyped versions of the unification problems is in determining whether or not there is a $\mathcal{Q}$-substitution for a given prefix $\mathcal{Q}$. For the untyped case, such substitutions always exist. For the typed case, this is not always the case. Hence, in the typed case the fact that the unification algorithm does not return *fail* is not enough to guarantee that there exist solutions.

7. Correctness of interpretation

We can now prove the correctness of the interpreter described in Section 4.

If $\mathcal{Q}\mathcal{Q}'$ is a quantifier prefix and if φ is a $\mathcal{Q}\mathcal{Q}'$-substitution, then $\varphi \downarrow \mathcal{Q}$ is the $\mathcal{Q}$-substitution that results from deleting from φ those pairs $x \mapsto t$ where x is existentially quantified in $\mathcal{Q}'$.

Lemma 7.1. *If the interpreter makes a single transition from the state $\langle\theta, \mathcal{Q}, \mathcal{S}\rangle$ to the state $\langle\theta \circ \rho, \mathcal{Q}', \mathcal{S}'\rangle$ and if φ satisfies $\mathcal{Q}'$ and $\mathcal{S}'$ then $(\rho \circ \varphi) \downarrow \mathcal{Q}$ satisfies $\mathcal{Q}$ and $\mathcal{S}$.*

Proof. We proceed by considering the cases that can cause a transition in the interperter.

Assume that the transition was caused by BACKCHAIN. That is, the state changed from $\langle\theta, \mathcal{Q}, \{\mathcal{P} \longrightarrow A'\} \uplus \mathcal{S}\rangle$ to the state $\langle\theta \circ \rho, \mathcal{Q}', \rho(\{\mathcal{P} \longrightarrow G\} \cup \mathcal{S})\rangle$, where $\mathcal{P}$ contains $\forall\bar{x}(G \supset A)$ (and $\bar{x}$ are tokens not bound in $\mathcal{Q}_\Sigma\mathcal{Q}$), and

$$\langle\rho, \mathcal{Q}'\rangle = \text{unify}(\emptyset, \mathcal{Q}\exists\bar{x}[A \overset{\circ}{=} A']).$$

Also assume that φ satisfies $\mathcal{Q}'$ and $\rho(\{\mathcal{P} \longrightarrow G\} \cup \mathcal{S})$. Thus, there is a goal-directed proof of the sequent

$$\Sigma, \Sigma_{\mathcal{Q}'}\ ;\ (\rho \circ \varphi)\mathcal{P} \longrightarrow (\rho \circ \varphi)G.$$

By Proposition 6.4, $\rho \circ \varphi$ is a $\mathcal{Q}\exists\bar{x}$-substitution such that $(\rho \circ \varphi)A = (\rho \circ \varphi)A'$. Let ψ be $(\rho \circ \varphi) \downarrow \mathcal{Q}$. Since the tokens in $\bar{x}$ are not free in $\mathcal{P}$ or in A', and since $\Sigma_{\mathcal{Q}}$ and $\Sigma_{\mathcal{Q}'}$ are the same, $(\rho \circ \varphi)A = \psi A'$ and this sequent can be written as simply

$$\Sigma, \Sigma_{\mathcal{Q}}\ ;\ \psi\mathcal{P} \longrightarrow (\rho \circ \varphi)G$$

Using the BC inference rule, we can build a proof of the sequent $\Sigma, \Sigma_{\mathcal{Q}}\ ;\ \psi\mathcal{P} \longrightarrow \psi A'$. Since the tokens in $\bar{x}$ are not free in the sequents in $\mathcal{S}$, it then follows that ψ satisfies $\mathcal{Q}$ and $\{\mathcal{P} \longrightarrow A'\} \uplus \mathcal{S}$.

The case when the transition was caused by CLOSE is similar and simpler.

Assume that the transition was caused by GENERIC. That is, the state changed from $\langle\theta, \mathcal{Q}, \{\mathcal{P} \longrightarrow \forall_\tau x G\} \uplus \mathcal{S}\rangle$ to the state $\langle\theta \circ \rho, \mathcal{Q}\forall_\tau x, \{\mathcal{P} \longrightarrow G\} \cup \mathcal{S}\rangle$, where x is not bound in $\mathcal{Q}_\Sigma\mathcal{Q}$ and where ρ is the empty substitution. Also assume that φ satisfies $\mathcal{Q}\forall_\tau x$ and $\{\mathcal{P} \longrightarrow G\} \cup \mathcal{S}$. Thus, $\rho \circ \varphi = \varphi$ and this is also a $\mathcal{Q}$-substitution. Since $\Sigma, \Sigma_{\mathcal{Q}}, y : \tau\ ;\ \varphi\mathcal{P} \longrightarrow \varphi G$ has a sequent proof and since y is not free in $\varphi\mathcal{P}$, using $\forall$-R we can build a proof for $\Sigma, \Sigma_{\mathcal{Q}}\ ;\ \varphi\mathcal{P} \longrightarrow \forall_\tau y.\varphi G$. Since no term in the range of φ contains y free, this sequent is the same as $\Sigma, \Sigma_{\mathcal{Q}}\ ;\ \varphi\mathcal{P} \longrightarrow \varphi(\forall_\tau y.G)$. Thus, φ satisfies $\mathcal{Q}$ and $\mathcal{S}$.

If the transition was the result of AND, the result is immediate: simply use $\wedge$-R to put the two proofs guaranteed by induction together. If the transition was the result of AUGMENT, build the new proof using the $\supset$-R rule. QED

Theorem 7.2. *There is a substitution φ that satisfies $\mathcal{Q}$ and $\mathcal{S}$ if and only if there is a series of transitions that carries the state $\langle\emptyset, \mathcal{Q}, \mathcal{S}\rangle$ to the success state $\langle\theta, \mathcal{Q}', \emptyset\rangle$ such that there is a $\mathcal{Q}'$-substitution φ' so that $\varphi = (\theta \circ \varphi') \downarrow \mathcal{Q}$.*

Proof. Assume that the interpreter makes a series of transitions

$$\langle \emptyset, \mathcal{Q}_0, \mathcal{S}_0 \rangle, \langle \rho_1, \mathcal{Q}_1, \mathcal{S}_1 \rangle, \langle \rho_1 \circ \rho_2, \mathcal{Q}_2, \mathcal{S}_2 \rangle, \ldots, \langle \rho_1 \circ \cdots \circ \rho_n, \mathcal{Q}_n, \mathcal{S}_n \rangle,$$

where $n \geq 1$. Let φ_n satisfy $\mathcal{Q}_n$ and $\mathcal{S}_n$. Using Lemma 7.1, there is a sequence of substitutions $\varphi_1, \ldots, \varphi_{n-1}$ such that for $i = 1, \ldots, n-1$, φ_i satisfies $\mathcal{Q}_i$ and $\mathcal{S}_i$ and $\varphi_i = (\rho_{i+1} \circ \varphi_{i+1}) \downarrow \mathcal{Q}_i$. Thus, $(\rho_1 \circ \cdots \circ \rho_n \circ \varphi_n) \downarrow \mathcal{Q}_0$ satisfies $\mathcal{Q}_0$ and $\mathcal{S}_0$. If $\mathcal{S}_n$ is empty, $\mathcal{Q}_n$ is $\mathcal{Q}'$, and $\varphi' = \varphi_n$ then setting θ to the substitution $\rho_1 \circ \cdots \circ \rho_n$ completes this part of the proof.

Now assume that the interpreter's state is $\langle \theta, \mathcal{Q}, \mathcal{S} \rangle$ and that there is a φ that satisfies $\mathcal{Q}$ and $\mathcal{S}$. Thus, every sequent in $\varphi\mathcal{S}$ has a goal-directed proof. We proceed by induction on the sum of the number of inference rule occurrences in those goal-directed proofs. If this number is zero, then $\mathcal{S}$ is empty and we are finished. Otherwise, let $\mathcal{S}$ be written as $\{\mathcal{P} \longrightarrow G\} \uplus \mathcal{S}'$. The structure of a goal-directed proof of $\Sigma_{\mathcal{Q}}\ ;\ \varphi\mathcal{P} \longrightarrow \varphi G$ dictates which transition can be performed. In particular, if the proof is an initial sequent, then use CLOSE. Otherwise, if the last inference rule in that proof is $\wedge$-R, use AND; if it is $\supset$-R, use AUGMENT; if it is $\forall$-L, use GENERIC; if it is BC, use BACKCHAIN. We illustrate this final case in more detail since it is the hardest.

Since the last rule is BC, there is a $\forall\bar{x}(G' \supset A') \in \varphi\mathcal{P}$ and a list of $\Sigma, \Sigma_{\mathcal{Q}}$-terms $\bar{t}$ so that $[\bar{x} \mapsto \bar{t}]A' = \varphi A$ and the sequent $\Sigma, \Sigma_{\mathcal{Q}}\ ;\ \varphi\mathcal{P} \longrightarrow [\bar{x} \mapsto \bar{t}]G'$ is provable ($[\bar{x} \mapsto \bar{t}]$ is an abbreviation for $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$). Since the variables in $\bar{x}$ can be picked to be new, there is a $\forall\bar{x}(G'' \supset A'') \in \mathcal{P}$ so that $\varphi G'' = G'$ and $\varphi A'' = A'$. Set ψ to be the substitution $\varphi \circ [\bar{x} \mapsto \bar{t}]$. Thus, ψ is a $\mathcal{Q}\exists\bar{x}$-substitution that unifies A and A'' and is such that $\Sigma, \Sigma_{\mathcal{Q}}\ ;\ \psi\mathcal{P} \longrightarrow \psi G''$ is provable. Set $\langle \mathcal{Q}', \rho \rangle = \mathrm{unify}(\emptyset, \mathcal{Q}\exists\bar{x}[A = A''])$. Since unification does not delete universal quantifiers, $\Sigma_{\mathcal{Q}} \subseteq \Sigma_{\mathcal{Q}'}$. The result of the BACKCHAIN transition is then the state $\langle \theta \circ \rho, \mathcal{Q}', \rho(\{\mathcal{P} \longrightarrow G''\} \cup \mathcal{S}') \rangle$. By Lemma 6.4, there is a $\mathcal{Q}'$-substitution φ' so that $\psi = \rho \circ \varphi'$. Thus, φ' satisfies $\mathcal{Q}'$ and $\rho(\{\mathcal{P} \longrightarrow G\} \cup \mathcal{S}')$, and the proof of the inductive case is complete. QED

The nondeterministic interpreter for L_λ described above can be thought of doing computation in the following fashion. Let $\mathcal{Q}$ be a purely existential prefix and let $\mathcal{P} \longrightarrow G$ be a $\mathcal{Q}$-sequent. Here, $\mathcal{P}$ is considered to be a logic program and G a query to be proved. The existential variables in $\mathcal{Q}$ are essentially *logic variables* that the interpreter can instantiate as it needs in order to find a proof. If the interpreter makes a transition from the initial state $\langle \emptyset, \mathcal{Q}, \{\mathcal{P} \longrightarrow G\} \rangle$ to a success state $\langle \theta, \mathcal{Q}', \emptyset \rangle$, then for every $\mathcal{Q}'$-substitution φ', the substitution $\theta \circ \varphi'$ restricted to the variables in $\mathcal{Q}$ is a solution or answer to this computation.

8. Examples of L_λ programs

Below we present several examples of programs written using L_λ. Since the logic programming language λProlog [18] fully implements L_λ (and more), we shall use the syntax of λProlog to present L_λ programs. The symbol `=>` denotes $\supset$, `:-` denotes its converse, a comma denotes conjunction, an infix occurrence of backslash `\` denotes λ-abstraction, and `pi` along with a λ-abstraction denotes universal quantification. Tokens with an uppercase initial letter are assumed to be universally quantified variables with outermost scope. The piece of syntax

```
kind i        type.
type sterile  i -> o.
type bug      i -> o.
type in       i -> i -> o.
type dead     i -> o.
sterile J :- pi b\((bug b, in b J) => dead b).
```

declares `i` to be a primitive type, declares the type for four predicate constants, and presents one definite clause, which could be written as

$$\forall J(\forall b((bug\ b \wedge in\ b\ J) \supset dead\ b) \supset sterile\ J).$$

In all the examples given in this section, once types are given for tokens that are to be taken as constants, the type of bound variables can easily be inferred from their context.

8.1. Specifying an object logic.

Three meta-programs — substitution, Horn clause interpretation, and the computation of prenex normal forms — are presented in this section. All there programs will compute with the same first-order, object logic, which contains universal and existential quantification and implication and conjunction. These are introduced by the syntax

```
kind  term   type.
kind  form   type.
type  all    (term -> form) -> form.
type  some   (term -> form) -> form.
type  and    form -> form -> form.
type  imp    form -> form -> form.
```

The first two lines declare the tokens `term` and `form` as primitive types.

The object logic will contain just five non-logical constants: an individual constant, a function symbol of one argument and another of two arguments, and a predicate symbol of one argument and another of two arguments. Their types are declared with the following lines.

```
type  a     term.
type  f     term -> term.
type  g     term -> term -> term.
type  p     term -> form.
type  q     term -> term -> form.
```

Terms over this signature of type `form` denote object logic formulas and of type `term` denote object logic terms. We shall need to lift this typing information more directly into the meta language by introducing the following two meta-level predicates and clauses. These clauses are obviously derived directly from the above signature. (The token `term` is used as both predicate symbol and type symbol.)

```
type term   term -> o.
type atom   form -> o.

term a.
term (f X)   :- term X.
term (g X Y) :- term X, term Y.
atom (p X)   :- term X.
atom (q X Y) :- term X, term Y.
```

Various other meta-predicates over object formulas are easy to write. For example, the following defines a predicate that determines whether or not its argument is a quantifier-free object-level formula.

```
type    quant_free      form -> o.

quant_free A :- atom A.
quant_free (and B C) :- quant_free B, quant_free C.
quant_free (imp B C) :- quant_free B, quant_free C.
```

This predicate is used in the definition of prenex normal formulas below. The following code describes how to determine if a term of type `form` encodes a Horn clause or a conjunction of atomic formulas.

```
type  hornc  form -> o.
type  conj   form -> o.

hornc (all C) :- pi x\(term x => hornc (C x)).
hornc (imp G A) :- atom A, conj G.
hornc A :- atom A.

conj (and B C) :- conj B, conj C.
conj A :- atom A.
```

Notice the structure of the first of the `hornc` clauses above. It involves a second-order variable `C` of type `term -> form` as well as an implication and universal quantifier in the clause's body. The variable `C` will get bound to an abstraction over an object-level formula. For example, if the goal

```
hornc (all u\(all v\(imp (p u) (and (q v a) (q a u)))))
```

is attempted, the variable `C` will get bound to the λ-abstraction

```
u\(all v\(imp (p u) (and (q v a) (q a u)))).
```

The intended processing of this λ-abstraction can be described by the following set of operations. Via the universally quantified goal, a new constants is picked. This new constant will play the role of a name for the bound variable x. Since this new constant is now temporarily part of the object logic, program clauses that were determined from the signature of the object logic may need to be extended. Thus, the definition of the `term` predicate needs to be extended with the fact that this new constant is a term. Thus, when `hornc` subsequently calls `atom`, the latter predicate will succeed for formulas containing this new constant. Finally, the application `(C x)` represents the body of the object-level abstraction with the new constant substituted for the abstracted variable. Thus, if the new constant picked by an L_λ interpreter is `d`, then the next goal to be attempted will be

```
hornc (all v\(imp (p d) (and (q v a) (q a d))))
```

with the additional assumption `(term d)` added to the program.

8.2. Implementing object-level substitution.

Substitution at the object-level can be implemented by first specifying the following copy-clauses.

```
type   copyterm    term -> term -> o.
type   copyform    form -> form -> o.

copyterm a a.
copyterm (f X) (f U)      :- copyterm X U.
copyterm (g X Y) (g U V) :- copyterm X U, copyterm Y V.
copyform (p X) (p U)      :- copyterm X U.
copyform (q X Y) (q U V) :- copyterm X U, copyterm Y V.
copyform (and X Y) (and U V) :- copyform X U, copyform Y V.
copyform (imp X Y) (imp U V) :- copyform X U, copyform Y V.
copyform (all X) (all U)     :-
   pi y\(pi z\(copyterm y z => copyform (X y) (U z))).
copyform (some X) (some U)   :-
   pi y\(pi z\(copyterm y z => copyform (X y) (U z))).
```

These clauses can be derived directly from the signature of the constants they are based on by using the following function. Let $[\![t, s : \tau]\!]$ be a formula defined by recursion on the structure of the type τ, which is assumed to be built only from the base types `term` and `form`, with the following clauses:

$$\begin{aligned} [\![t, s : \texttt{term}]\!] &= \texttt{copyterm}\ t\ s \\ [\![t, s : \texttt{form}]\!] &= \texttt{copyform}\ t\ s \\ [\![t, s : \tau \texttt{ -> } \sigma]\!] &= \forall x \forall y ([\![x, y : \tau]\!] \supset [\![t\ x, s\ y : \sigma]\!]) \end{aligned}$$

The copy-clauses displayed above are essentially those clauses that are equal to $[\![c, c : \tau]\!]$ where the signature for representing the object logic contains $c : \tau$.

The extension of these copy-clauses is exactly the same as that for equality. That is, (`copyterm` t s) is provable from these clauses if and only if t and s are the same term. A similar statement is true for `copyform`. Now consider adding a new constant, say `c`, of type `term`, and adding the clause `(copy c (f a))`. Given this extended set of copy-clauses, (`copyterm` t s) is provable if and only if s is the result of replacing every occurrence of `c` in t with `(f a)`; that is, s is $[\mathtt{c} \mapsto \mathtt{(f\ a)}]t$. This can be formalized using the following code.

```
type subst  (term -> form) -> term -> form -> o.

subst M T N :- pi c\(copyterm c T => copyform (M c) N).
```

Here, the first argument of `subst` is an abstraction over formulas. The second argument is then substituted into that abstraction to get the third argument. To instantiate a universal quantifier with a given term, the following code could be used.

```
type uni_instan  form -> term -> form -> o.

uni_instan (all B) T C :- subst B T C.
```

Consider the somewhat simpler clause for implementing `subst`:

```
    subst M T (M T).
```

This clause is not a legal L_λ D-formula since the second occurrence of `M` is applied to another positively quantified universally variable. This clause correctly specifies substitution if the meta-level contains the full theory of β-conversion for simply typed λ-terms. Such a clause is available in λProlog and the higher-order logic programming languages described in [17] and [19]. These languages have a much richer unification problem than L_λ.

8.3. Implementing a simple higher-order unification problem.

The restriction on functional variables in L_λ ensures that it is never the case that a term, such as `(F a)` (for function variable `F`) is unified with a term such as `(g a a)` (here, `g` and `a` are as declared in Subsection 8.1). Such a unification problem, however, is permitted in the more general setting explored in [9]. While this is not a permissible unification problem in L_λ, it is very easy to solve this problem in L_λ using the `subst` program written above. In particular, the set of substitutions for `F` that unify `(F a)` and `(g a a)` is exactly the set of substitutions for `F` that makes the goal

```
subst F a (g a a)
```

provable. In particular, an L_λ interpreter should return the following four substitutions for `F`:

```
w\(g w w)   w\(g w a)   w\(g a w)   w\(g a a).
```

These are exactly the unifiers for this more general unification problem. Arbitrary higher-order unification problems can be encoded into L_λ using various calls to predicates like `subst` defined above, although the translation is often more complex than the simple example illustrated here.

8.4. Interpretation of first-order Horn clauses.

Since object-level logic programs will be denoted by lists of formulas, we first specify the data type of formula lists and a simple membership program. These are supplied by the following code.

```
kind  lst    type.

type  nil    lst.
type  cons   form -> lst -> lst.
type  memb   form -> lst -> o.

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

It is possible in λProlog to specify lists and list operations that are polymorphic; such lists, however, are not required for this example.

The following code describes the interpreter for Horn clauses.

```
type  interp      lst -> form -> o.
type  instan      form -> form -> o.
type  backchain   lst -> form -> form -> o.

interp Cs (and B C) :- interp Cs B, interp Cs C.
interp Cs A :- memb D Cs, instan D E, backchain Cs E A.

instan (all A) B :- pi x\(copyterm x T => instan (A x) B).
instan B C :- copyform B C.

backchain Cs A A.
backchain Cs (imp G A) A :- interp Cs G.
```

The `backchain` clause preforms operations similar to those called BACKCHAIN and CLOSE in Section 4. The `instan` predicate implements substitution as describe above. Operationally, its function can be thought of as stripping off the universal quantifiers on a Horn clause by instantiating them with unspecified terms. Subsequent actions of the `interp` program and meta-level unification will further specify those terms.

8.5. Computing prenex-normal forms.

Our last example of a meta-program on our small object logic is the computation of prenex-normal forms. Our goal is to write a set of clauses so that the formula (`prenex` B C) is provable if and only if C is a prenex-normal form of B. This relationship is not functional: there are possibly many prenex-normal formulas that can arise from moving embedded quantifiers into a prefix. The following code correctly captures this full relation. To define `prenex`, an auxillary predicate `merge` is used.

```
type    prenex          form -> form -> o.
type    merge           form -> form -> o.

prenex B B :- atom B.
prenex (and B C) D :- prenex B U, prenex C V, merge (and U V) D.
prenex (imp B C) D :- prenex B U, prenex C V, merge (imp U V) D.
prenex (all B)  (all D)  :- pi x\(term x => prenex (B x) (D x)).
prenex (some B) (some D) :- pi x\(term x => prenex (B x) (D x)).
merge (and (all B) (all C)) (all D) :-
   pi x\(term x => merge (and (B x) (C x)) (D x)).
merge (and (all B) C) (all D) :-
   pi x\(term x => merge (and (B x) C) (D x)).
merge (and B (all C)) (all D) :-
   pi x\(term x => merge (and B (C x)) (D x)).
merge (and (some B) C) (some D) :-
   pi x\(term x => merge (and (B x) C) (D x)).
merge (and B (some C)) (some D) :-
   pi x\(term x => merge (and B (C x)) (D x)).
merge (imp (all B) (some C)) (some D) :-
   pi x\(term x => merge (imp (B x) (C x)) (D x)).
merge (imp (all B) C) (some D) :-
   pi x\(term x => merge (imp (B x) C) (D x)).
merge (imp B (some C)) (some D) :-
   pi x\(term x => merge (imp B (C x)) (D x)).
merge (imp (some B) C) (all D) :-
   pi x\(term x => merge (imp (B x) C) (D x)).
merge (imp B (all C)) (all D) :-
   pi x\(term x => merge (imp B (C x)) (D x)).
merge B B :- quant_free B.
```

The `merge` predicate is used to bring together two prenex normal formulas into a single prenex normal formula. Notice the nondeterminism in `merge`: there are three clauses that can be employed to solve a `merge`-goal whose first argument is of the form `(and (all B) (all C))`. These clauses represent the fact that the universal quantifiers can be jointly moved into the prefix or that one can be moved out before the other.

Given these clauses, there is a unique prenex-normal form for the formula

```
imp (all x\(and (p x) (and (all y\(q x y)) (p (f x))))) (p a),
```

which is the formula

```
some x\(some y\(imp (and (p x) (and (q x y) (p (f x)))) (p a))).
```

The formula `(and (all x\(q x x)) (all z\(all y\(q z y))))`, however, has the following five prenex-normal forms:

```
all z\(all y\(and (q z z) (q z y)))
all x\(all z\(all y\(and (q x x) (q z y))))
all z\(all x\(and (q x x) (q z x)))
all z\(all x\(all y\(and (q x x) (q z y))))
all z\(all y\(all x\(and (q x x) (q z y)))).
```

These results can be computed on a depth-first implementation of L_λ, such as λProlog, in the following fashion. Given the clauses presented above, λProlog can be asked to search for substitution instances of the variable P so that the atom

```
prenex (and (all x\(q x x)) (all z\(all y\(q z y)))) P
```

is provable. Using its depth-first search stradegy, λProlog will find five different proofs of this atom, each with a different instance of P (the five terms listed above, in that order). As written, however, the depth-first interpretation of this code cannot be used to determine the converse relation, namely, compute those formulas which have a given prenex-normal form, since it would start to generate object-level formulas in an undirected fashion and would not, in general, terminate. A breath-first search could, however, compute this converse.

9. Conclusion

Meta-programming systems need to be able to treat various structures that naturally contain notions of scope and bound variable. Conventional programming languages do not contain language-level support for such structures. Computation systems such as λProlog, Elf, and Isabelle do have such support since they contain typed λ-terms and implement the equations of α, β, and η. Such a treatment of λ-terms is, however, a complex operation since the unification of λ-terms modulo those equations is undecidable in general. Many uses of function variables and λ-terms in meta-programs can, however, be restricted to the point where unification over these same equations is a simple extension of first-order unification. Much of what is lost in restricting function variables can be regained by writing logic programs. This restriction on functional variables is integrated into logic programming yielding a language called L_λ. Unification for L_λ is decidable and generalizes first-order unification. A nondeterministic interpretation of L_λ is easily described by merging unification with a sequent-style theorem prover.

Acknowledgements. I am grateful to Amy Felty, Elsa Gunter, John Hannan, and Frank Pfenning for discussions related to the paper. An anonymous reviewer also made several helpful comments on an earlier draft of this paper. The work reported here has been supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018.

10. References

[1] de Bruijn, N. (1972), Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem, Indag. Math. (34:5), 381 – 392.

[2] Church, A. (1940), A Formulation of the Simple Theory of Types, Journal of Symbolic Logic **5**, 56 – 68.

[3] Elliott, C. (1989), Higher-Order Unification with Dependent Types, Proceedings of the 1989 Rewriting Techniques and Applications, Springer-Verlag LNCS.

[4] Felty, A. and Miller, D. (1988), Specifying Theorem Provers in a Higher-Order Logic Programming Language, Proceedings of the Ninth International Conference

on Automated Deduction, Argonne, IL, 23 – 26, eds. E. Lusk and R. Overbeek, Springer-Verlag Lecture Notes in Computer Science, Vol. 310, 61 – 80.

[5] Hannan, J. and Miller, D. (1988), Uses of Higher-Order Unification for Implementing Program Transformers, Fifth International Conference and Symposium on Logic Programming, ed. K. Bowen and R. Kowalski, MIT Press, 942 – 959.

[6] Hannan, J. and Miller, D. (1989), A Meta Language for Functional Programs, Chapter 24 of *Meta-Programming in Logic Programming*, eds. H. Rogers and H. Abramson, MIT Press, 453–476.

[7] Harper, R., Honsell, F. and Plotkin, G. (1987), A Framework for Defining Logics, Second Annual Symposium on Logic in Computer Science, Ithaca, NY, 194 – 204.

[8] Hindley, J. and Seldin, J. (1986), *Introduction to Combinators and λ-calculus*, Cambridge University Press.

[9] Huet, G. (1975), A Unification Algorithm for Typed λ-Calculus, Theoretical Computer Science **1**, 27 – 57.

[10] Huet, G. and Lang, B. (1978), Proving and Applying Program Transformations Expressed with Second-Order Logic, Acta Informatica **11**, 31 – 55.

[11] Goldfarb, W. (1981), The Undecidability of the Second-Order Unification Problem, Theoretical Computer Science **13**, 225 – 230.

[12] Miller, D. (1989), A Logical Analysis of Modules in Logic Programming, Journal of Logic Programming **6**, 79 – 108.

[13] Miller, D. (1989), Lexical Scoping as Universal Quantification, Sixth International Logic Programming Conference, Lisbon, eds. G. Levi and M. Martelli, MIT Press, 268 – 283.

[14] Miller, D. (1990), Abstractions in logic programming, in *Logic and Computer Science* edited by P. Odifreddi, Academic Press, 329 – 359.

[15] Miller, D., Unification under a Mixed Prefix, Journal of Symbolic Computation (to appear).

[16] Miller, D. and Nadathur, G. (1987), A Logic Programming Approach to Manipulating Formulas and Programs, Proceedings of the IEEE Fourth Symposium on Logic Programming, IEEE Press, 379 – 388.

[17] Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A., Uniform Proofs as a Foundation for Logic Programming, Annals of Pure and Applied Logic (to appear).

[18] Nadathur, G. and Miller, D. (1988), An Overview of λProlog, Fifth International Conference on Logic Programming, eds. R. Kowlaski and K. Bowen, MIT Press, 810 – 827.

[19] Nadathur, G. and Miller, D., Higher-Order Horn Clauses, Journal of the ACM (to appear).

[20] Paulson, L. (1986), Natural Deduction as Higher-Order Resolution, Journal of Logic Programming **3**, 237 – 258.

[21] Pauslon, L. (1989), The Foundation of a Generic Theorem Prover, Journal of Automated Reasoning **5**, 363 – 397.

[22] Pfenning, F. and Elliot, C. (1988), Higher-Order Abstract Syntax, Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 199 – 208.

[23] Pfenning, F. (1989), Elf: A Language for Logic Definition and Verified Metaprogramming, Fourth Annual Symposium on Logic in Computer Science, Monterey, CA, 313 - 321.

[24] Pietrzykowski, T. and Jensen, D. (1976), Mechanizing ω-Order Type Theory Through Unification, Theoretical Computer Science **3**, 123 - 171.

[25] Statman, R. (1979), Intuitionistic Propositional Logic is Polynomial-Space Complete, Theoretical Computer Science **9**, 67 - 72.

[26] Snyder, W. and Gallier, J. (1989), Higher-Order Unification Revisited: Complete Sets of Transformations, Journal of Symbolic Computation **8**, 101 - 140.