University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

August 1990

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

Dale Miller
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Dale Miller, "A Logic Programming Language With Lambda-Abstraction, Function Variables, and Simple Unification", . August 1990.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-54.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/576
For more information, please contact repository@pobox.upenn.edu.

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

Abstract

It has been argued elsewhere that a logic programming language with function variables and λ-abstractions within terms makes a good meta-programming language, especially when an object-language contains notions of bound variables and scope. The λProlog logic programming language and the related Elf and Isabelle systems provide meta-programs with both function .variables and λ-abstractions by containing implementations of higher-order unification. This paper presents a logic programming language, called $L\lambda$, that also contains both function variables and λ-abstractions, although certain restrictions are placed on occurrences of function variables. As a result of these restrictions, an implementation of $L\lambda$ 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 $L\lambda$ as a meta-programming language are presented.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-54.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/576

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

MS-CIS-90-54
LINC LAB 182

Dale Miller

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

August 1990

[[To appear in *Extensions of Logic Programming* edited by Peter Schroeder-Heister, Lecture Notes in Artificial Intelligence, Springer-Verlag. This paper has also appeared in the *Journal of Logic and Computation, 1991.* Supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018]]

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

Abstract: It has been argued elsewhere that a logic programming language with function variables and λ-abstractions within terms makes a good meta-programming language, especially when an object-language contains notions of bound variables and scope. The λProlog logic programming language and the related Elf and Isabelle systems provide meta-programs with both function variables and λ-abstractions by containing implementations of higher-order unification. This paper presents a logic programming language, called L_λ, that also contains both function variables and λ-abstractions, although certain restrictions are placed on occurrences of function variables. As a result of these restrictions, 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 should 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 change 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 or when 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 object-level syntax. 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 directly in such λ-calculi 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 notion of equality of λ-terms, which usually includes 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 do not yield equal answers. Although the use of nameless dummies [2] 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 [13, 20, 33, 34]. Huet and Lang [14] described how such an approach, when restricted to second-order matching, can be used to analyze and manipulate simple functional and imperative programs. Their reliance on unification modulo 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 is strong enough to implement a certain collection of template matching program transformations and it is decidable. The general problem of the unification of simply typed λ-terms of order 2 and higher is undecidable [8].

The use of λ-term unification in meta-programming has been extended in several recent papers and computer systems. In [6, 9, 10, 21] various meta-programs, including theorem provers and program transformers, were written in the logic programming language λProlog [24], which performs unification of simply typed λ-terms. Paulson [28, 29] exploited such unification in the theorem proving system Isabelle. Pfenning and Elliot [32] argued that product types are also of use. Elliot [4] studied unification in a dependent type framework and Pfenning [30] developed a logic programming language Elf, which incorporates that unification process. Elf can be used to provide a direct implementation of signatures written in the LF type specification language [11].

This paper presents a logic programming language, called L_λ, which 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 resembles 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 4 after some formal preliminaries are covered in Section 3. An interpreter and unification algorithm for L_λ are presented in Sections 5 and 6, respectively. The unification algorithm is proved correct in Section 7 and the interpreter is proved correct in Section 8. Various comments about unification and interpretation are made in Section 9. Finally, several examples of L_λ programs are presented in Section 10.

This paper is an expanded, reorganized, and corrected version of the paper [18].

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, $\beta\eta$-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 may appear in the following context:

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

When such free variables are 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-term, 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 in t 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 simply to perform object-level substitution. For example, an object-level universal quantifier can be specified using the symbol *all* of second-order type $(term \to formula) \to formula$. The binary predicate that relates a universally quantified formula to the result of instantiating it with some term can be coded simply by the following meta-level axiom

$$\forall B \forall T(instan\ (all\ B)\ (B\ T)),$$

where B and T are typed as $term \to formula$ and $term$, respectively. At the object-level, this predicate relates the formulas $\forall x.B$ and $[x \mapsto T]B$: object-level substitution is expressed at the meta-level using β-conversion.

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. Although object-level substitution is not automatically available, it can be specified naturally as an L_λ program. We illustrate this for a simple, first-order, object-logic in Section 10. 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, the ω-order version of L_λ is presented here since, as is shown in Section 9, 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, [17, 22]) that the intuitionistic theory of these formulas provides a foundation for logic programming if programs are identified with collections of D-formulas and goals or queries with G-formulas. 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 simple operational interpretation of the logical connectives is sound and non-deterministically complete with respect to intuitionistic logic. This operational interpretation 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 non-logical constants are from Σ). Intuitionistic provability of G from Σ and $\mathcal{P}$, written as $\Sigma; \mathcal{P} \vdash_I G$, 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 shaped into a simple theorem proving mechanism. Such a mechanism using unification and a depth-first searching discipline can be used to give a Prolog-style implementation of 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 [15, 16, 17]), 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 formulas

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

Now, consider the problem of finding a substitution term over Σ_0 for the 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

$$(append\ (cons\ a\ (cons\ b\ nil))\ k\ X)$$

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 $(cons\ a\ (cons\ b\ k))$. 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 desirable, however, to have this computation succeed if this particular choice of constant could be abstracted away. That is, an interesting value is computed here, but it cannot be used since it is not well defined. Admitting λ-abstraction into this logic provides a representation of such a value.

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 $\forall y$ is again instantiated with the constant k. This time, $(H\ k)$ must equal $(cons\ a\ (cons\ b\ k))$. 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. 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 solvable 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 thought of as a function that carries an increment in a signature to a term over that increment.

The reader who is comfortable with the above discussion may wish to read Section 10 next where several examples of L_λ programs are given and discussed.

3. Logical Preliminaries

We assume that the reader is familiar with the basic properties of λ-terms, λ-conversion, and logic built on top of simply typed λ-terms. Some definitions and properties are reviewed below. See [1, 3, 12] for more complete presentations.

Untyped λ-terms are built up from a set of tokens and from application and abstraction in the usual way. Occurrence of tokens in terms are classified as either free or bound occurrences. 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 expression $t = s$ means that t and s are α-convertible. The term r *β-reduces* to the term r' if r has an occurrence of a β-redex, say $(\lambda x\ t)s$, and r' is the result of replacing that redex with the result of substituting s for x in t (changing bound variable names to avoid variable capture). The term r *η-reduces* to the term r' if r has an occurrence of an η-redex, say $\lambda x\ (t\ x)$, and r' is the result of replacing that redex with t. 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 relates to t_{i+1} or t_{i+1} relates to t_i by α-conversion or by by β or η-reduction. If a term can be converted to a λ-normal term, that normal term is unique up to the name of bound variables. If t is a λ-term then $\lambda\text{norm}(t)$ denotes its λ-normal form. Since not all untyped λ-terms have λ-normal forms, this function is partial. When applied to simply typed versions of λ-terms (as below), λ-normal forms always exist, and this function is then total. A λ-normal term that is not a top-level abstraction is of the form $(ht_1 \ldots t_n)$ where h is a token. This token is the *head* of this term.

Substitutions are finite association lists written as $[x_1 \mapsto s_1, \ldots, x_n \mapsto s_n]$, where the variables $x_1, \ldots, x_n$ are all distinct. The list $x_1, \ldots, x_n$ is the domain of this substitution. If $n = 0$, this substitution is the empty substitution. When a substitution is applied to a term, it denotes the operation of simultaneous substitution, systematically changing bound variables in order to avoid variable capture. If T is a set of terms and φ is a substitution, then $\varphi T = \{\varphi t \mid t \in T\}$. Two substitutions, φ and ψ, are equal if their domains are equal and if whenever $x \mapsto t \in \varphi$ and $x \mapsto s \in \psi$ then $s\ \lambda conv\ t$. The notation $\varphi \circ \psi$ denotes the composition of two substitutions. Functionally $(\varphi \circ \psi)(t) = \psi(\varphi t)$ and as an association list, $z \mapsto t \in \varphi \circ \psi$ if z is in the domain of φ and t is $\psi(\varphi z)$ or z is in the domain of ψ and not in the domain of φ and t is ψz.

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, built using 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 finite 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. Signatures are often presented by listing their pairs as $a:\tau$. A signature is of order n if all its tokens have types of order n or less and at least one token has a type of order n. The expression $\Sigma + c:\tau$ is legal if c is not assigned by Σ, in which case, it is equal to $\Sigma \cup \{c:\tau\}$.

Signatures can be used as type assignments in the following way. Let Σ be a signature. A λ-normal, untyped λ-term t is a Σ-*term of type* τ if all free tokens in t are members of Σ and if the term t can be given the type τ using the type assignment Σ. We shall think of signatures as "local declaration" of which tokens should be considered constants. Tokens in t are thus either bound variables or free tokens that appear in Σ, in which case we shall call them constants. It is very natural, however, for bound variables to change status to constants by a change in signature. For example, if x is not a token in Σ then $\lambda x.t$ is a Σ-term of type $\tau \to \sigma$ if and only if t is a $\Sigma \cup \{x:\tau\}$-term of type σ.

As in [3], logic over terms is introduced 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). The following defines the class of Σ-*formulas*.

- If A is a 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 $[x \mapsto y]B$ is a $\Sigma + y:\tau$-formula then $\forall_\tau x.B$ is a Σ-formula (x and y are tokens).

This paper assumes the additional restriction that if a quantified variable is of type τ then 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 [22, 25] and another is described in [16]. The kinds of meta-programs that we discuss here do not require any forms of predicate quantification.

A sequent calculus is used to define intuitionistic provability over these formulas. A *sequent* is a triple, written $\Sigma\ ;\ \Gamma \longrightarrow B$, where Σ is a signature, Γ is a finite (possibly empty) set of Σ-formulas, and B is a Σ-formula. The set Γ is the *antecedent* and B is the *succedent* of this sequent. Intuitionistic provability is given by the sequent proof system $\mathcal{I}$ displayed in Figure 1. Since antecedents are sets of formulas, the structural rules of contraction and weakening are not needed. The notation Γ, B is short for $\Gamma \cup \{B\}$ and the notation Γ, Δ is short for $\Gamma \cup \Delta$. The two universal introduction rules have the following provisos: in $\forall$-L t must be a Σ-term of type τ; in $\forall$-R y must be a token that is not in Σ. A rule that permits formulas in the premise sequent to be replaced with α-convertible formulas in the conclusion is implicitly assumed to be available whenever it is needed. We write $\Sigma; \Gamma \vdash B$ to mean that the sequent $\Sigma\ ;\ \Gamma \longrightarrow B$ has a sequent proof.

$$\frac{\Sigma\,;\, B, C, \Delta \longrightarrow E}{\Sigma\,;\, B \wedge C, \Delta \longrightarrow E}\ \wedge\text{-L} \qquad \frac{\Sigma\,;\, \Gamma \longrightarrow B \qquad \Sigma\,;\, \Gamma \longrightarrow C}{\Sigma\,;\, \Gamma \longrightarrow B \wedge C}\ \wedge\text{-R}$$

$$\frac{\Sigma\,;\, \Gamma \longrightarrow B \qquad \Sigma\,;\, C, \Gamma \longrightarrow E}{\Sigma\,;\, B \supset C, \Gamma \longrightarrow E}\ \supset\text{-L} \qquad \frac{\Sigma\,;\, B, \Gamma \longrightarrow C}{\Sigma\,;\, \Gamma \longrightarrow B \supset C}\ \supset\text{-R}$$

$$\frac{\Sigma\,;\, \lambda\mathrm{norm}([x \mapsto t]B), \Gamma \longrightarrow C}{\Sigma\,;\, \forall_\tau x.\, B, \Gamma \longrightarrow C}\ \forall\text{-L} \qquad \frac{\Sigma \cup \{y{:}\tau\}\,;\, \Gamma \longrightarrow [x \mapsto y]B}{\Sigma\,;\, \Gamma \longrightarrow \forall_\tau x.\, B}\ \forall\text{-R}$$

$$\frac{\Sigma\,;\, \Gamma \longrightarrow B \qquad \Sigma\,;\, B, \Delta \longrightarrow E}{\Sigma\,;\, \Gamma, \Delta \longrightarrow E}\ \mathrm{cut} \qquad \frac{}{\Sigma\,;\, \Gamma, B \longrightarrow B}\ \mathrm{initial}$$

Figure 1: $\mathcal{I}$: Inference rules for intuitionistic provability

Gentzen's cut-elimination theorem [7] can be used on $\mathcal{I}$ to prove that if a sequent is provable then it is provable without the cut rule. The rest of this paper considers only *cut-free* proofs.

4. Logic Programming

The proof system $\mathcal{I}$ can be used as the basis of a logic programming language since a goal-directed style of theorem proving is complete for it. Goal-directed provability can be formalized within general sequent calculus proof systems using the notion of *uniform proof* [22]: a cut-free sequent proof is *uniform* if whenever the succedent in an occurrence of a sequent is not atomic, that sequent occurrence is the conclusion of a right-introduction rule. Within $\mathcal{I}$, this means that if the occurrence of a sequent has a succedent that is a conjunction, implication, or universal quantifier, that sequent occurrence is the result of the $\wedge$-R, $\supset$-R, or $\forall$-R rules, respectively. The following proposition follows from considering permutations of inference rules in cut-free proofs. Stronger results are established in [22].

Proposition 4.1. *If the sequent* $\Sigma\,;\, \Gamma \longrightarrow B$ *has a proof in* $\mathcal{I}$*, it has a uniform proof. In other words, the following equivalences hold.*

- $\Sigma; \Gamma \vdash B_1 \wedge B_2$ *if and only if* $\Sigma; \Gamma \vdash B_1$ *and* $\Sigma; \Gamma \vdash B_2$.
- $\Sigma; \Gamma \vdash B_1 \supset B_2$ *if and only if* $\Sigma; \Gamma \cup \{B_1\} \vdash B_2$.
- $\Sigma; \Gamma \vdash \forall_\tau x.B$ *if and only if* $\Sigma \cup \{y : \tau\}; \Gamma \vdash [x \mapsto y]B$, *where* y *is a token that is not in* Σ.

The structure of proofs in $\mathcal{I}$ of sequents that have atomic succedent can be characterized by using a notion of *backchaining*. In Section 2 we considered backchaining in the simple setting where program formulas are of the form $\forall\bar{x}.A$ and $\forall\bar{x}.(G \supset A)$,

where A is atomic and $\forall\bar{x}$ is some list of universally quantified variables. Backchaining will be described below in a setting where this restriction on program formulas is not assumed. This is not problematic because formulas built freely from $\wedge$, $\supset$, and $\forall$ can be related directly to conjunctions of restricted clauses via the following simple intuitionistic equivalences:

$$B_1 \supset (B_2 \wedge B_3) \equiv (B_1 \supset B_2) \wedge (B_1 \supset B_3)$$
$$B_1 \supset (B_2 \supset B_3) \equiv (B_1 \wedge B_2) \supset B_3$$
$$B_1 \supset \forall_\tau x B_2 \equiv \forall_\tau x (B_1 \supset B_2)$$

(provided in the last case that x is not free in B_1). The following definitions are a simple way to capture these equivalences and incorporate them into a sequent proof system.

Let Γ be a finite set of Σ-formulas. The set of pairs $|\Gamma|_\Sigma$ is defined to be the smallest set such that

- if $D \in \Gamma$ then $\langle \emptyset, D\rangle \in |\Gamma|_\Sigma$,
- if $\langle \Gamma, D_1 \wedge D_2\rangle \in |\Gamma|_\Sigma$ then $|\Gamma|_\Sigma \in \langle \Gamma, D_1\rangle$ and $|\Gamma|_\Sigma \in \langle \Gamma, D_2\rangle$,
- if $\langle \Gamma, G \supset D\rangle \in |\Gamma|_\Sigma$ then $\langle \Gamma \cup \{G\}, D\rangle \in |\Gamma|_\Sigma$, and
- if $\langle \Gamma, \forall_\tau x\, D\rangle \in |\Gamma|_\Sigma$ and t is a Σ-term, then $\langle \Gamma, \lambda\mathrm{norm}([x \mapsto t]D)\rangle \in |\Gamma|_\Sigma$.

Notice that in general, $|\Gamma|_\Sigma$ is an infinite set of pairs. Referring to $|\Gamma|_\Sigma$ within a sequent calculus eliminates the need to have all three left-introduction rules, $\forall$-L, $\wedge$-L, $\supset$-L, as well as the initial rule. Thus, consider the proof system $\mathcal{I}'$ that is the result of deleting the cut and initial rules and the three left-introduction rules from $\mathcal{I}$ and replacing them with the *BC rule* (for backchaining) given in Figure 2. It is worth noting that, in general, applicability of BC is difficult to check: it is equivalent to doing "higher-order matching," which is not known to be decidable. We shall only be interested in using this inference rule in the restricted setting of L_λ, and there (as a consequence of Proposition 7.3) determining the applicability of BC will be decidable. The following proposition follows from results in [22].

$$\frac{\{\Sigma\,;\,\Gamma \longrightarrow G\}_{G\in\Delta}}{\Sigma\,;\,\Gamma \longrightarrow A}\ \mathrm{BC}$$

provided that A is atomic and $\langle \Delta, A\rangle \in |\Gamma|_\Sigma$. If Δ is empty, then no premises appear and the sequent is treated as an initial sequent.

Figure 2: Backchaining as an inference rule

Proposition 4.2. *Let Σ be a signature, Γ a set of closed Σ-formulas, and B a closed Σ-formula. Then $\Sigma;\Gamma \vdash B$ if and only if the sequent $\Sigma\,;\,\Gamma \longrightarrow B$ is provable in $\mathcal{I}'$.*

This proposition can be used to describe a non-deterministic interpreter that first decomposes goal formulas using right-introduction rules and then attempts to backchain to prove atomic goals. Moving from this style of non-deterministic interpreter to an actual deterministic interpreter is a difficult task. Various aspects of implementing such an interpreter are considered in [5, 24, 26]. In order to motivate the introduction of the restrictions defining L_λ, it is important to note that the above non-deterministic

interpreter will need to perform β-reductions while looking for proofs. That is, although programs and goals start out in λ-normal form (by the definition of Σ-formulas), substitutions may cause them to become non-normal. Thus, the use of the λnorm() function in the definition of $|\Gamma|_\Sigma$ is necessary in general. Unification in this setting is complex because β-conversion can cause significant changes to a term. L_λ will be restricted in such a way that only a very simple fragment of general β-conversion is required in the interpreter. As a result, unification in that language will be much simpler than for the full, unrestricted logic.

As we motivated in Section 2, we wish to restrict β-reductions that need to be performed within a theorem prover for this logic. The only place where λnorm() is used in the description of the proof system $\mathcal{I}'$ is within the definition of $|\Gamma|_\Sigma$ used in backchaining. In order to restrict the formation of β-redexes in proofs, we need to restrict occurrences of those universally quantified variables for formulas that can be instantiated in the definition of $|\Gamma|_\Sigma$. Such quantifier instances are those that can appear at the top-level of a formula in the antecedent. A universal quantifiers that can appear at the top-level of the succedent do not need to be restricted since they are instantiated by only new constants and not general terms. Thus we must make a distinction between formulas that can occur in antecedents and those that can occur in succedents. Using the operational reading of such formulas in logic programming, we shall informally refer to formulas that can appear in the antecedent as *program formulas*, *definite formulas*, or just *D-formulas*. Formulas that can appear in succedents will be called *queries*, *goals*, or *G-formulas*. We now motivate our eventual definitions of both G and D-formulas.

Let Σ be a signature and let B be a Σ-formula. If B is to be considered a G-formula, then we classify bound variables in B as follows: A bound variable occurrence in B is *essentially universal* if it is bound by a positive occurrence of a universal quantifier or by a (term-level) λ-abstraction in B; otherwise, it is *essentially existential*; that is, it is bound by a negative universal quantifier in G. Dually, if B is to be considered a D-formula, then a bound variable occurrence in B is *essentially universal* if it is bound by a negative occurrence of a universal quantifier or by a (term-level) λ-abstraction in B; otherwise, it is *essentially existential*; that is, it is bound by a positive universal quantifier in G.

The central restriction in L_λ is that for every subterm in B of the form $(x\ y_1\ \ldots\ y_n)$ $(n \geq 0)$ where x is essentially existentially quantified in B, it must be the case that $y_1, \ldots, y_n$ is a list of distinct variables that are essentially universally quantified within the scope of the binding for x. This restriction ensures that if x is ever instantiated by some term, say t, then the only β-redexes that appear after that substitution are of the form $(ty_1 \ldots y_n)$ where the variables $y_1, \ldots, y_n$ are not free in t. Using α and η-conversions, we can assume that t is of the form $\lambda y_1 \ldots \lambda y_n.t'$. Thus, β-reduction simply reduces $(\lambda y_1 \ldots \lambda y_n.t')y_1 \ldots y_n$ to t'. Let β_0-conversion be that subcase of β-conversion that relates redexes of the form $(\lambda x.s)x$ to s. As is mentioned in Section 9, the equational theory of L_λ is only that of α, β_0, and η-conversions.

This restriction on G and D-formulas can be described more formally using the proof system in Figure 3. Let $\mathcal{Q}$ denote a *quantifier prefix*, that is, a list of universal and existential quantifiers in which the quantified variables are all distinct. Quantifier in

prefixes are slightly richer than those in Σ-formulas; in particular, universal quantifiers of predicate types and existential quantifiers (of non-predicate types) are allowed. We write $\mathcal{Q} \vdash_0 t : \tau$ if the sequent $\mathcal{Q} \xrightarrow{0} t : \tau$ is provable, $\mathcal{Q} \vdash_- B$ if the sequent $\mathcal{Q} \xrightarrow{-} B$ is provable, and $\mathcal{Q} \vdash_+ B$ if the sequent $\mathcal{Q} \xrightarrow{+} B$ 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 term t (resp., the formula G, D) is α-convertible to t' (G', D').

(†) The variable x does not occur in $\mathcal{Q}$.

(‡) $\mathcal{Q}$ contains $\forall h$ where the type on the quantifier is $\tau_1 \to \cdots \to \tau_n \to \tau$ ($n \geq 0$).

(♯) The variable x is existentially quantified in $\mathcal{Q}$ to the left of where the distinct variables $y_1, \ldots, y_n$ ($n \geq 0$) are universally quantified. The quantifier for x has type $\tau_1 \to \cdots \to \tau_n \to \tau$ while the quantifiers for $y_1, \ldots, y_n$ have type $\tau_1, \ldots, \tau_n$, respectively.

$$\frac{\mathcal{Q} \xrightarrow{0} t : \tau}{\mathcal{Q} \xrightarrow{0} t' : \tau}\,\alpha \qquad \frac{\mathcal{Q} \xrightarrow{+} G}{\mathcal{Q} \xrightarrow{+} G'}\,\alpha \qquad \frac{\mathcal{Q} \xrightarrow{-} D}{\mathcal{Q} \xrightarrow{-} D'}\,\alpha$$

$$\frac{\mathcal{Q}\forall_\tau x \xrightarrow{0} t : \sigma}{\mathcal{Q} \xrightarrow{0} \lambda x.t : \tau \to \sigma}\,\dagger \qquad \frac{}{\mathcal{Q} \xrightarrow{0} x y_1 \cdots y_n : \tau}\,\sharp \qquad \frac{\mathcal{Q} \xrightarrow{0} t_1 : \tau_1 \;\cdots\; \mathcal{Q} \xrightarrow{0} t_n : \tau_n}{\mathcal{Q} \xrightarrow{0} h t_1 \cdots t_n : \tau}\,\ddagger$$

$$\frac{\mathcal{Q}\forall_\tau x \xrightarrow{+} G}{\mathcal{Q} \xrightarrow{+} \forall_\tau x.G}\,\dagger \qquad \frac{\mathcal{Q} \xrightarrow{+} G_1 \qquad \mathcal{Q} \xrightarrow{+} G_2}{\mathcal{Q} \xrightarrow{+} G_1 \wedge G_2} \qquad \frac{\mathcal{Q} \xrightarrow{-} D \qquad \mathcal{Q} \xrightarrow{+} G}{\mathcal{Q} \xrightarrow{+} D \supset G} \qquad \frac{\mathcal{Q} \xrightarrow{0} A : o}{\mathcal{Q} \xrightarrow{+} A}$$

$$\frac{\mathcal{Q}\exists_\tau x \xrightarrow{-} D}{\mathcal{Q} \xrightarrow{-} \forall_\tau x.D}\,\dagger \qquad \frac{\mathcal{Q} \xrightarrow{-} D_1 \qquad \mathcal{Q} \xrightarrow{-} D_2}{\mathcal{Q} \xrightarrow{-} D_1 \wedge D_2} \qquad \frac{\mathcal{Q} \xrightarrow{+} G \qquad \mathcal{Q} \xrightarrow{-} D}{\mathcal{Q} \xrightarrow{-} G \supset D} \qquad \frac{\mathcal{Q} \xrightarrow{0} A : o}{\mathcal{Q} \xrightarrow{-} A}$$

Figure 3: Proof rules for the syntax of L_λ

Let Σ be a 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* or *G-formula* of L_λ is a Σ-formula G so that $\mathcal{Q}_\Sigma \vdash_+ G$. A *definite formula* or *D-formula* of L_λ is a Σ-formula D so that $\mathcal{Q}_\Sigma \vdash_- D$.

All first-order positive Horn clauses are both G and D-formulas. If the constant p has type $i \to o$ and f has type $i \to i$ then the formula

$$\forall_{i \to i} x \forall_i y (p\ (x\ y) \supset p\ (f\ y))$$

is an example of a G-formula but not a D-formula. As a D-formula of L_λ, it has a subterm occurrence $(x\ y)$ where both x and y are essentially existential, and this is ruled out by proviso (♯). Section 10 contains several examples of G and D-formulas.

5. An interpreter for L_λ

Interpretation of L_λ can be described as a bottom-up search for goal-directed proofs. The BACKCHAIN step is, of course, the most difficult to implement since it requires chosing a D-formula and terms to substitute into that formula. The interpreter described below uses unification to discover what instances of D-formulas lead to successful BACKCHAINing steps.

In such an interpreter, it is necessary to keep track of notions such as the "current goal," the "current program," the "current signature," and restrictions on free variables. Interpreters for Horn clauses need to keep track of only the first of these: there the current program and signature remain 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, explicit meta-level quantification is used to encode both the current signature and the restrictions on free variables, and sequents are used to connect programs to goals.

Consider the simple meta-logic that contains the logical constants $\wedge$, $\top$ (true), $\bot$ (false), $\forall_\tau$ (τ ranges over all types, including predicate types), and $\exists_\tau$ (τ ranges over all non-predicate types). The reuse of the object-level logical constants $\forall_\tau$ and $\wedge$ should not lead to confusion. Meta-level atomic propositions, called *judgements*, are of four kinds: the two constants, $\top$ and $\bot$, the equality judgement $t \stackrel{\tau}{=} s$, and the sequent judgement $\mathcal{P} \longrightarrow G$. A formula of the meta-logic denotes a *state formula* if (a) all tokens that are not bound in object-level formulas are bound by meta-level quantifiers and (b) names of all meta-level bound variables within a given state formula are distinct. The first condition implies that state formulas are closed; the second condition is a convenience. No separate signature is assumed: instead of having a signature Σ and a state formula $\mathcal{S}$, consider only the state formula $\mathcal{Q}_\Sigma \mathcal{S}$.

Let $\mathcal{S}$ be a state formula. A substitution θ is an *$\mathcal{S}$-substitution* if the domain of θ does not contain any meta-level, universally quantified variables but does contain all of the meta-level, existentially quantified variables of $\mathcal{S}$. Also, let $\exists_\tau x$ occur in $\mathcal{S}$ and let Σ be the set of typed universally quantified variables in which $\exists_\tau x$ is in the scope. Then φx must be a Σ-term of type τ. In this sense, an $\mathcal{S}$-substitution is a closed substitution; that is, its substitution terms do not contain existentially quantified variables. It is for convenience that variables other than existentially quantified variables are permit in the domain of such $\mathcal{S}$-substitutions: since such variables are neither free nor quantified in the meta-level of $\mathcal{S}$, they shall play no role in the interpreter. Two $\mathcal{S}$-substitutions, say φ and ψ, are equal if for each $\exists_\tau x$ in $\mathcal{S}$, $\varphi x = \psi x$. In that case, we write $\varphi = \psi \pmod{\mathcal{S}}$.

It is possible that for a given $\mathcal{S}$, there may not be any $\mathcal{S}$-substitutions. For example, if Σ is the signature $\{f : i \to i, g : i \to i \to i\}$ and $\mathcal{S}$ is $\mathcal{Q}_\Sigma \exists_i x(x \stackrel{i}{=} x)$, there is no $\mathcal{S}$-substitution since there is no λ-term of type i whose only free tokens are f and g. That

is, the type i is, in a sense, empty. Since the problem of determining if there is an $\mathcal{S}$-substitution for a given $\mathcal{S}$ reduces to proving theorems in the implicational fragment of intuitionistic logic, this problem is decidable [35].

An $\mathcal{S}$-substitution φ *satisfies* $\mathcal{S}$ if (i) $\mathcal{S}$ does not contain $\bot$, (ii) for every equation $t \stackrel{\tau}{=} s$ in $\mathcal{S}$, φt $\lambda conv$ φs, and (iii) for every sequent judgement $\mathcal{P} \longrightarrow G$ in $\mathcal{S}$, the sequent

$$\Sigma \,;\, \lambda\text{norm}(\varphi\mathcal{P}) \longrightarrow \lambda\text{norm}(\varphi G)$$

has a proof in $\mathcal{I}'$ (where Σ is the set of typed universal variables in which this sequent is in the scope). A *solution* to $\mathcal{S}$ is an $\mathcal{S}$-substitution that satisfies $\mathcal{S}$. By definition, a state formula containing $\bot$ has no solutions. The purpose of an interpreter is to search for solutions to a state formula. Checking satisfiability is, of course, not decidable in general.

The BACKCHAIN transition below requires the following *elaboration function*, which is related to the $|\Gamma|_\Sigma$ function defined in Section 4 except that it does not choose substitution terms. Let $\mathcal{S}$ be a state formula and let $\mathcal{P}$ be a finite set of D-formulas. Then elab$(\mathcal{S},\mathcal{P})$ is defined to be the smallest set of triples such that

- if $D \in \mathcal{P}$ then $\langle\langle\rangle, \emptyset, D\rangle \in \text{elab}(\mathcal{S},\mathcal{P})$ ($\langle\rangle$ denotes the empty list of quantifiers),
- if $\langle\mathcal{Q},\mathcal{G}, D_1 \wedge D_2\rangle \in \text{elab}(\mathcal{S},\mathcal{P})$ then $\langle\mathcal{Q},\mathcal{G}, D_1\rangle \in \text{elab}(\mathcal{S},\mathcal{P})$ and $\langle\mathcal{Q},\mathcal{G}, D_2\rangle \in \text{elab}(\mathcal{S},\mathcal{P})$.
- if $\langle\mathcal{Q},\mathcal{G}, G \supset D\rangle \in \text{elab}(\mathcal{S},\mathcal{P})$ then $\langle\mathcal{Q},\mathcal{G}\cup\{G\}, D\rangle \in \text{elab}(\mathcal{S},\mathcal{P})$, and
- if $\langle\mathcal{Q},\mathcal{G}, \forall_\tau x\, D\rangle \in \text{elab}(\mathcal{S},\mathcal{P})$ and y is the first token (in some ordering of tokens) that is not bound in $\mathcal{S}$, then $\langle\mathcal{Q}\exists_\tau y, \mathcal{G}, ([x \mapsto y]D)\rangle \in \text{elab}(\mathcal{S},\mathcal{P})$.

If $\langle\mathcal{Q},\mathcal{G}, D\rangle \in \text{elab}(\mathcal{S},\mathcal{P})$, then $\mathcal{P}$ can be used to show that for suitable substitutions φ, if each of the formulas $G \in \varphi\mathcal{G}$ is provable, then φD is provable. The BACKCHAIN step below uses members of this elaboration only when D is atomic.

Interpretation and unification are presented as collections of non-deterministic, labeled transitions $\mathcal{S} \stackrel{\rho}{\Longrightarrow} \mathcal{S}'$ where $\mathcal{S}$ and $\mathcal{S}'$ are state formulas and ρ is a substitution. Generally, ρ is neither an $\mathcal{S}$ nor an $\mathcal{S}'$-substitution: instead, composing it with an $\mathcal{S}'$-substitution yields an $\mathcal{S}$-substitution.

The following four transition rules describe the heart of a non-deterministic interpreter. Each of these transitions describes how to make a labeled transition, where the label ρ is the empty substitution. In each case, $\mathcal{S}'$ is built by replacing a sequent judgement in $\mathcal{S}$ by a formula.

AND step. Replace a sequent of the form $\mathcal{P} \longrightarrow G_1 \wedge G_2$ with the conjunction

$$(\mathcal{P} \longrightarrow G_1) \wedge (\mathcal{P} \longrightarrow G_2).$$

AUGMENT step. Replace a sequent of the form $\mathcal{P} \longrightarrow D \supset G$ with the sequent $D \cup \mathcal{P} \longrightarrow G$.

GENERIC step. Replace a sequent of the form $\mathcal{P} \longrightarrow \forall_\tau x.G$ with the quantified formula $\forall_\tau y(\mathcal{P} \longrightarrow [x \mapsto y]G)$, where y is a token not in $\mathcal{S}$.

BACKCHAIN step. Replace a sequent of the form $\mathcal{P} \longrightarrow A$ with

$$\mathcal{Q}(A \stackrel{o}{=} A' \wedge (\mathcal{P} \longrightarrow G_1) \wedge \ldots \wedge (\mathcal{P} \longrightarrow G_n)),$$

where A and A' are atomic formulas, $\langle \mathcal{Q}, \{G_1, \ldots, G_n\}, A' \rangle \in \text{elab}(\mathcal{S}, \mathcal{P})$, and $n \geq 0$. If $n = 0$ then the above displayed formula is simply $\mathcal{Q}(A \stackrel{o}{=} A')$. If there is no such member of $\text{elab}(\mathcal{S}, \mathcal{P})$ (that is, $\mathcal{P}$ is empty), then replace that sequent with $\bot$.

If D-formulas were restricted to Horn clauses and G-formulas to conjunctions of atoms, then the structure of these transitions could be greatly simplified. In particular, the GENERIC and AUGMENT transition steps would not be needed; the antecedent of all sequents in state formulas would be the same; and meta-level quantification would simply be outermost universal variables and inner-most existential quantifiers (no quantifier alternations), in which case the notion of $\mathcal{S}$-substitution simplifies to the notion of substitution.

6. A unification algorithm for L_λ

Let $t \stackrel{\tau}{=} s$ be an equational judgement in the state formula $\mathcal{S}$. If t is not a top-level abstraction, then t is *flexible* if its head is existentially quantified in $\mathcal{S}$ and is *rigid* otherwise; that is, its head is universally quantified in $\mathcal{S}$. Flexible and rigid can similarly be applied to s. The head of a rigid term is invariant under $\mathcal{S}$-substitutions.

To illustrate some of the features of unifying λ-terms in state formulas, consider the example

$$\mathcal{Q}\exists_{\iota\to\iota\to\iota\to\iota} u \exists_{\iota\to\iota\to\iota} v \forall_\iota y [f(\lambda x.f(uxy)) \stackrel{\iota}{=} f(\lambda w.vy)],$$

where the quantifier $\forall_{(\iota\to\iota)\to\iota} f$ occurs in the quantifier list $\mathcal{Q}$. This problem can be simplified to the formula

$$\mathcal{Q}\exists u \exists v \forall y [\lambda x.f(uxy) \stackrel{\iota\to\iota}{=} \lambda w.vy]$$

(dropping types from quantifiers). Such transitions are done by the rigid-rigid step below. Using the equivalence between $\lambda x.t = \lambda x.s$ and the quantified equation $\forall x.t = s$ (via the ξ inference rule [12]), this formula can be simplified to

$$\mathcal{Q}\exists u \exists v \forall y \forall x [f(uxy) \stackrel{\iota}{=} vy].$$

Such transitions are done by the ξ step below. At this point, the substitution $[v \mapsto \lambda y.f(uxy)]$ could be suggested except that quantification rules out substituting v with a term that contains x free. It is possible, however, to solve this state formula if the x can be removed from the left-hand of the equation: this is possible only if u is vacuous in its first argument. Thus, apply the substitution $[u \mapsto \lambda x \lambda y.u'y]$ and make the transition to the state formula

$$\mathcal{Q}\exists u' \exists v \forall y \forall x [f(u'y) \stackrel{\iota}{=} vy],$$

where u' has type $\iota \to \iota \to \iota$. Such transitions are done by the pruning step below. This state formula can be solved by substituting $[v \mapsto \lambda y.f(u'y)]$ and making the transition to the formula

$$\mathcal{Q}\exists u' \forall y \forall x [\top].$$

This last transition is done by the flexible-rigid step below. This final formula arises from the original state formula via the substitution $[u \mapsto \lambda x \lambda y.u'y, v \mapsto \lambda y.f(u'y)]$.

Thus, composing this substitution with one for u' (that is, a $\mathcal{Q}\exists u'\forall y\forall x[\top]$-substitution) yields solutions to the original state formula.

Let $\mathcal{S}$ be a state formula that contains at least one equation, say $t \stackrel{\tau}{=} s$. Each of the following steps produces a transition $\mathcal{S} \stackrel{\rho}{\Longrightarrow} \mathcal{S}'$ by describing how to compute ρ and $\mathcal{S}'$. In those cases when ρ is applied to the judgements in $\mathcal{S}$ to form judgements in $\mathcal{S}'$, the resulting judgements are assumed to be placed in λ-normal form. For convenience we shall stop writing type information explicitly in quantifiers and in equational judgements. In all cases, the type information is easy to determine and insert. Also, types do not play a critical role in unification: in Section 9 an untyped version of unification is outlined.

Raising step. Let u be an existentially quantified variable free in t and let v be a different existentially quantified variable free in s. One of these variables must be quantified in the scope of the other. Assume that the scope of v contains the scope of u (otherwise switch the role of u and v below). Let $\bar{w}$ be the list of universally quantified variables that are quantified in the scope of $\exists v$ and that contain the scope of $\exists u$. If the list $\bar{w}$ is empty then the raising step is not applicable to this pair of variables. Set $\rho = [u \mapsto u'\bar{w}]$, where u' is not bound in $\mathcal{S}$. Build $\mathcal{S}'$ from $\mathcal{S}$ by dropping the quantifier $\exists u$, replacing the one quantifier $\exists v$ with the two quantifiers $\exists v\exists u'$, and applying ρ to all the judgements in $\mathcal{S}$. 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 $\bar{w}$.

ξ step. Assume that t is of the form $\lambda\bar{x}.t'$ and s is of the form $\lambda\bar{y}.s'$, where t' and s' are not themselves abstractions and where at least one of the lists of variables, $\bar{x}$ or $\bar{y}$, is not empty. 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. Using α-conversion, we may assume that these two binders are the same; that is, $t = s$ can be written as $\lambda\bar{w}.t'' = \lambda\bar{w}.s''$ where the variables in $\bar{w}$ are not bound in $\mathcal{S}$. Then replace the equation $t = s$ in $\mathcal{S}$ with $\forall\bar{w}[t'' = s'']$ to form $\mathcal{S}'$. The substitution ρ is empty.

Rigid-rigid step. If the equation $t = s$ has the form $ht_1 \ldots t_n = hs_1 \ldots s_n$, where $n \geq 0$ and h is universally quantified in $\mathcal{S}$, replace the equation $t = s$ with the conjunction $t_1 = s_1 \wedge \ldots \wedge t_n = s_n$ to form $\mathcal{S}'$. If $n = 0$ then simply replace with $\top$. If the 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 $\mathcal{S}$, then replace the equation with $\bot$. In either case, ρ is empty.

Pruning step. Given an equation of the form $vy_1 \ldots y_n = r$, let z be a meta-level, universally bound variable of $\mathcal{S}$ that has a free occurrence in r, is bound in the scope of $\exists v$, and is not in the list $\bar{y}$. If no such z occurs, then the pruning step is not applicable. Otherwise, if z has an occurrence in r that is not in the scope of an existentially quantified variable, then replace that equation with $\bot$ and let ρ be the empty substitution. Otherwise, z occurs in a subterm $u\bar{w}_1 z\bar{w}_2$ of r where u is existentially quantified in $\mathcal{S}$ and $\bar{w}_1$ and $\bar{w}_2$ are lists of either λ-bound variables or universally quantified variables bound in the scope of $\exists u$. The dependency of u on the argument occupied by z is removed by setting $\rho = [u \mapsto \lambda\bar{w}_1\lambda z\lambda\bar{w}_2.u'\bar{w}_1\bar{w}_2]$, where u'

is not bound in $\mathcal{S}$. The formula $\mathcal{S}'$ is the result of replacing $\exists u$ with $\exists u'$ and applying ρ to all judgements in $\mathcal{S}$.

Flexible-flexible step. Assume that the equation $t = s$ is flexible-flexible; that is, it is of the form $vy_1 \ldots y_n = uz_1 \ldots z_p$ where $n, p \geq 0$, $y_1, \ldots, y_p$ are distinct universally quantified variables bound in the scope of $\exists v$, and $z_1, \ldots, z_p$ are distinct universally quantified variables bound in the scope of $\exists u$. There are two cases.

Case 1. Assume that v and u are different and that there is no universal variable bound between the binding occurrences of v and u (otherwise the raising step can be first performed). We may also assume that the lists $\bar{x}$ and $\bar{y}$ are permutations of each other (otherwise the pruning step can be first performed). Set $\rho = [v \mapsto \lambda \bar{y}.u\bar{z}]$ and form $\mathcal{S}'$ by replacing $t = s$ with $\top$, by deleting $\exists v$, and by applying ρ to all remaining judgements in $\mathcal{S}$.

Case 2. Assume that v and u are equal; that is, the pair is of the form $vy_1 \ldots y_n = vz_1 \ldots z_n$. Let $\bar{w}$ be the enumeration of the set $\{y_i \mid y_i = z_i, i \in \{1, \ldots, n\}\}$ that orders variables the same way as they are ordered in $\bar{y}$ (the choice of this particular ordering is not important). Set $\rho = [v \mapsto \lambda \bar{y}.v'\bar{w}]$ (notice that this is the same via α-conversion to $[v \mapsto \lambda \bar{z}.v'\bar{w}]$), where v' is not quantified in $\mathcal{S}$. Form $\mathcal{S}'$ by replacing $t = s$ with $\top$ and $\exists v$ with $\exists v'$ and by applying ρ to all remaining judgements in $\mathcal{S}$.

Flexible-rigid step. Assume that the flexible-rigid equation $t = s$ in $\mathcal{S}$ is of the form $vy_1 \ldots y_n = r$. (Of course, if s is flexible and t is rigid, then switch this equation around first.) Given that the raising and pruning steps are available, we can also assume that (a) if an existentially quantified variable, say u, appears free in r then there is no universally quantified variable in the scope of $\exists v$ that also contains the scope of $\exists u$, and (b) if a universally quantified variable is free in r and is bound in the scope of $\exists v$, then that variable is in the list $y_1 \ldots y_n$. Given these constraints, all that remains in addressing this equation is to do the *occurrence-check*: if v is free in r then replace this equation by $\bot$ and set ρ to the empty substitution. Otherwise, set $\rho = [v \mapsto \lambda y_1 \ldots \lambda y_n.r]$ and build $\mathcal{S}'$ by replacing this equation with $\top$, dropping $\exists v$, and applying ρ to all remaining judgements in $\mathcal{S}$.

These transitions are organized into a deterministic algorithm below. We shall leave unspecified those choices that could give rise to α-convertible state formulas or to different orderings on existential (or universal) variables in a sequence of existential (or universal) variables. Such differences are inconsequential and can be fixed largely arbitrarily.

Unification Algorithm. To solve the equations in a given initial state formula $\mathcal{S}_0$, order the choice of transitions using the following three steps. These choices are made until there are no equations left or until $\bot$ appears in a state formula. This gives rise to a series of transitions

$$\mathcal{S}_0 \stackrel{\rho_1}{\Longrightarrow} \cdots \stackrel{\rho_n}{\Longrightarrow} \mathcal{S}_n \quad (n \geq 0).$$

The result of the unification algorithm is the pair $\langle \rho_1 \circ \cdots \circ \rho_n, \mathcal{S}_n \rangle$.

(1) Apply either the ξ or the rigid-rigid step to the first applicable equation found in a left-to-right transversal of the state formula. If neither of these steps applies, move to the next step.

(2) Select the first flexible-flexible or flexible-rigid equation in a left-to-right order. Apply the raising and then the pruning steps to that equation and its converse until these transitions can no longer be applied: then move on to the next step. The exact order in which the various raising steps or various pruning steps are applied can be specified arbitrarily.

(3) Apply as appropriate either the flexible-flexible or flexible-rigid step to the resulting selected equation.

Several optimizations of this algorithm are, of course, possible. For example, it is not necessary to prune and raise prior to applying the second flexible-flexible step as this algorithm would do. ∎

The substitutions, named ρ above, generated by individual transitions are of two kinds. Those generated by the flexible-rigid step are of the form $[v \mapsto \lambda\bar{y}.t]$ where t can be a complex term. All the other substitutions have the much simpler form $[v \mapsto \lambda\bar{y}.v'\bar{w}]$, where v' is a "new" or existing existentially quantified variable. The application of ρ and λ-normalization to a state formula returns another state formula; that is, occurrences of existentially quantified variables in the resulting state formula are properly restricted.

7. Correctness of the unification transitions

We first show that there can be no infinite series of unification transitions. For this, we need a measure on equations in state formulas. If t is a λ-normal term all of whose free tokens are quantified at the meta-level in $\mathcal{S}$, the measure $|t|$ counts the number of occurrences of abstractions and applications in t that are not in the scope of existentially quantified variables of $\mathcal{S}$. That is, $|t|$ is defined by

$$|\lambda x_1 \ldots \lambda x_k(ht_1 \ldots t_n)| = \begin{cases} k & h \text{ existentially quantified in } \mathcal{S} \\ k + n + \sum_{i=1}^{n} |t_i| & h \text{ universally quantified in } \mathcal{S} \end{cases} \quad (k, n \geq 0).$$

(Of course, $|t|$ also has $\mathcal{S}$ as an argument, but its value will always be clear from context.) The *weight* of a meta-level, universal quantifier is the number of occurrences of meta-level, existential quantifiers in its scope. A universally quantified variable z of $\mathcal{S}$ is ***possibly prunable*** from an equational judgement $t = s$ of $\mathcal{S}$ if z occurs free in either t or s but not both and if all existentially quantified variables of $\mathcal{S}$ that are free in the term in which z is not free contain $\forall z$ in their scope. Thus if z is possibly prunable from $t = s$ and z occurs free in s then no $\mathcal{S}$-substitution instance of t contains z free.

Let $t_1 = s_1, \ldots, t_n = s_n$ be the list of equations that occur in $\mathcal{S}$ and let m be the number of existentially quantified variables in $\mathcal{S}$. The measure associated to $\mathcal{S}$ is defined by the quintuple

$$|\mathcal{S}| = \langle m, \sum_{i=1}^{n} |t_i| + |s_i|, n, w, p \rangle,$$

where w is the sum of the weights of all meta-level, universal quantifiers in $\mathcal{S}$, and p is the total number of occurrences of variables in the equations $t_1 = s_1, \ldots, t_n = s_n$

that are possibly prunable from the equation containing that occurrence. Quintuples are ordered lexicographically.

Theorem 7.1. *There is no infinite series of unification transitions.*

Proof. Let $\mathcal{S} \stackrel{\rho}{\Longrightarrow} \mathcal{S}'$ via a unification transition step. We should that for each unification step $|\mathcal{S}'| < |\mathcal{S}|$.

If the transition is the raising step, then the weight of at least one meta-level, universal quantifier in $\mathcal{S}$ decreases in $\mathcal{S}'$. Although the number of applications in the state formula may have increased, all new applications are in the scope of existentially quantified variables and are therefore not counted by the $|\ |$-measure. Since the number of equations and number of existentially quantified variables have not changed, $|\mathcal{S}'| < |\mathcal{S}|$.

If the transition is the ξ step, the number of abstractions in equations decreases. If the transition is the rigid-rigid step, then either the number of applications decreases or the number of equations decreases. Thus in either of these cases, $|\mathcal{S}'| < |\mathcal{S}|$.

If the transition is the pruning step, either the number of equations is reduced by one or all components of the measure are unchanged except for the last, which gets strictly smaller. Thus $|\mathcal{S}'| < |\mathcal{S}|$.

In the first case of the 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 decreases, the overall measure decreases.

Finally, if $\mathcal{S}'$ arises from $\mathcal{S}$ by applying the flexible-rigid case, the number of equations reduces by one and an existentially quantified variable from $\mathcal{S}$ may also be deleted. Thus, again $|\mathcal{S}'| < |\mathcal{S}|$. ∎

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

Lemma 7.2. *Assume that $\mathcal{S} \stackrel{\rho}{\Longrightarrow} \mathcal{S}'$ is a unification transition. The solutions to $\mathcal{S}$ can be put into one-to-one correspondence with the solution to $\mathcal{S}'$ so that if the solution φ for $\mathcal{S}$ corresponds to the solution φ' for $\mathcal{S}'$ then $\rho \circ \varphi' = \varphi \pmod{\mathcal{S}}$.*

Proof. Assume that the transition is the raising step. That is, the state formula changed by lifting $\exists u$ up over the universally quantified variables in $\bar{w}$ to get the quantifier $\exists u'$ and $\rho = [u \mapsto u'\bar{w}]$ is applied to all judgements. 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 $u \mapsto \lambda\text{norm}(r\bar{w})$. 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 $\rho \circ \varphi' = \varphi \pmod{\mathcal{S}}$. Notice that raising is a general transition for unification problems: it is dependent only on the scope of quantifiers and not on the judgements of the state formula. A fuller description of this transition is presented in [20].

If the transition is the ξ step, the result follows immediately since ρ is the empty substitution and the set of solutions does not change.

Assume that the transition is the rigid-rigid step. If the equation replaced with this step is $ht_1 \ldots t_n = hs_1 \ldots s_n$, a substitution makes these terms λ-convertible if and only if it makes t_i λ-convertible s_i, for $i = 1, \ldots, n$. Thus, $\mathcal{S}$ and $\mathcal{S}'$ have the same solutions. If the equation replaced with this step is $ht_1 \ldots t_n = ks_1 \ldots s_m$, where h and k are different universally quantified variables in $\mathcal{S}$, then this equation cannot be made equal and $\mathcal{S}$ has no solutions. Neither does $\mathcal{S}'$ since it contains $\bot$.

Assume that the transition is the pruning of the equation $vy_1 \ldots y_n = r$. Let z be a universal variable of $\mathcal{S}$ that occurs free in r, is not in the list $y_1, \ldots, y_n$, and is bound in the scope of $\exists v$. Assume that the occurrence of z in r is not in the scope of an existentially quantified variable. Thus all instances of r contain z free. Since no $\mathcal{S}$-substitution instance of $vy_1 \ldots y_n$ contains z free, $\mathcal{S}$ has no solution. In this case, neither does $\mathcal{S}'$ since it contains $\bot$. Assume that the occurrence of z in r is in the scope of an existentially quantified variable, say in the expression $u\bar{w}_1 z \bar{w}_2$. A solution φ for $\mathcal{S}$ must substitute for u a term of the form $\lambda\bar{w}_1 \lambda z \lambda \bar{w}_2.t$ where z is not free in t. The corresponding substitution φ' for $\mathcal{S}'$ is given by substituting $\lambda\bar{w}_1 \lambda\bar{w}_2.t$ for u'. Conversely, let φ be the result of replacing $u' \mapsto \lambda\bar{w}_1 \lambda\bar{w}_2.t$ in φ' with $u \mapsto \lambda\bar{w}_1 \lambda z \lambda\bar{w}_2.t$ in φ. Given that $(\rho \circ \varphi')u = \varphi'(\lambda\bar{w}_1 \lambda z \lambda\bar{w}_2.u'\bar{w}_1\bar{w}_2) = \lambda\bar{w}_1 \lambda z \lambda\bar{w}_2.(\lambda\bar{w}_1 \lambda\bar{w}_2.t)\bar{w}_1\bar{w}_2$ $\lambda conv$ $\lambda\bar{w}_1 \lambda z \lambda\bar{w}_2.t = \varphi v$, we again have $\rho \circ \varphi' = \varphi \pmod{\mathcal{S}}$.

Assume that the transition is the first case of the flexible-flexible step. That is, the equation $v\bar{y} = u\bar{z}$ in $\mathcal{S}$ is replaced with $\top$ and $\rho = [v \mapsto \lambda\bar{y}.u\bar{z}]$. Let φ be a solution to the unification problem in the first state. Thus, modulo α and η-conversions, 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, given 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 $\rho \circ \varphi' = \varphi \pmod{\mathcal{S}}$.

Assume that the transition is the second case of the flexible-flexible step. That is, the equation $v\bar{y} = v\bar{z}$ in $\mathcal{S}$ is replaced with $\top$, $\rho = [v \mapsto \lambda\bar{y}.v'\bar{w}]$, and $\bar{w}$ is an 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. 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 that $(\rho \circ \varphi')v = \varphi'(\lambda\bar{y}.v'\bar{w}) = \lambda\bar{y}.(\lambda\bar{w}.t)\bar{w}$ $\lambda conv$ $\lambda\bar{y}.t = \varphi v$, we again have $\rho \circ \varphi' = \varphi \pmod{\mathcal{S}}$.

The final case to consider is the flexible-rigid step; that is, either the equation $v\bar{y} = r$ is replaced with $\bot$ and ρ is empty or it is replaced with $\top$ and $\rho = [v \mapsto \lambda\bar{y}.r]$. The first case arises if v has a free occurrence in r. Assume that $\mathcal{S}$ has a solution φ. Let $\#(s)$ be the number of occurrences in s of meta-level, universally quantified variables of $\mathcal{S}$ that contain $\exists v$ in their scope. Thus, $\#(\lambda\text{norm}(\varphi(v\bar{y}))) < \#(\lambda\text{norm}(\varphi r))$ since the latter count includes $\#(r)$ and the occurrence of the head of $\lambda\text{norm}(\varphi r)$, which is also the head of the rigid term r. Thus, $\mathcal{S}$ has no solution. In this case, neither does $\mathcal{S}'$ since it contains $\bot$. On the other hand, assume that the above equation is replaced with $\top$ and $\rho = [v \mapsto \lambda\bar{y}.r]$. 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 φr. Let φ' be the substitution resulting from deleting the substitution term for v in φ (φ arises from φ' by adding that substitution term). Then $(\rho \circ \varphi')v = \varphi'(\lambda\bar{y}.r)$. Since v is not free in r, this latter term is also equal to $\varphi(\lambda\bar{y}.r)$. As a result of raising, φ does not substitute into any existentially quantified tokens in r terms containing tokens in $\bar{y}$. Thus, $\varphi(\lambda\bar{y}.r)$ is also equal to $\lambda\bar{y}.\varphi r = \lambda\bar{y}.s = \varphi v$. Again we have $\rho \circ \varphi' = \varphi \pmod{\mathcal{S}}$. ∎

Proposition 7.3. *If the unification algorithm is applied to the state formula $\mathcal{S}$, it terminates with a result, say $\langle\theta, \mathcal{S}'\rangle$. If $\mathcal{S}'$ contains $\perp$, then $\mathcal{S}$ has no solutions. Otherwise, $\mathcal{S}'$ contains no equational judgements and the solutions to $\mathcal{S}$ and $\mathcal{S}'$ can be placed in one-to-one correspondence so that if the solution φ for $\mathcal{S}$ corresponds to the solution φ' for $\mathcal{S}'$ then $\theta \circ \varphi' = \varphi \pmod{\mathcal{S}}$.*

Proof. The fact that the unification algorithm terminates is an immediate consequence of Theorem 7.1. Assume that the unification algorithm makes the series of transitions

$$\mathcal{S} = \mathcal{S}_0 \stackrel{\rho_1}{\Longrightarrow} \cdots \stackrel{\rho_n}{\Longrightarrow} \mathcal{S}_n = \mathcal{S}' \quad (n \geq 0),$$

where $\theta = \rho_1 \circ \cdots \circ \rho_n$ (if $n = 0$ then θ is the empty substitution). Now $\mathcal{S}'$ either contains $\perp$ or contains no equations (that is, there is a unification transition available for every possible equation). In the first case, it follows immediately from Lemma 7.2 that none of the state formulas $\mathcal{S}_0, \ldots, \mathcal{S}_n$ can have a solution. In the second case, again using Lemma 7.2, it is possible to place solutions of $\mathcal{S}_i$ $(i = 0, \ldots, n)$ in one-to-one correspondence so that, if φ_i as a solution for $\mathcal{S}_i$ $(i = 0, \ldots, n)$ is in such a correspondence, we have

$$\rho_1 \circ \varphi_1 = \varphi_0 \pmod{\mathcal{S}_0}, \ \ldots, \ \rho_n \circ \varphi_n = \varphi_{n-1} \pmod{\mathcal{S}_n}.$$

Thus, $\rho_1 \circ \cdots \circ \rho_n \circ \varphi_n = \varphi_0 \pmod{\mathcal{S}}$. Therefore, solutions φ' to $\mathcal{S}'$ can be placed in one-to-one correspondence with solutions φ of $\mathcal{S}$ so that $\theta \circ \varphi' = \varphi \pmod{\mathcal{S}}$. ∎

A *unification problem* is a state formula that does not contain any sequent judgements. The following theorem follows immediately from the previous proposition.

Theorem 7.4. *Let $\mathcal{S}$ be a unification problem without the $\perp$ judgement. Assume that the unification algorithm returns $\langle\theta, \mathcal{S}'\rangle$ when applied to $\mathcal{S}$. Then $\mathcal{S}$ has no solution (i.e. unifier) if and only if $\mathcal{S}'$ contains $\perp$ or there are no $\mathcal{S}'$-substitutions. If $\mathcal{S}'$ does not contain $\perp$, the substitution θ represents the most general unifier of $\mathcal{S}$ in the sense that the set of solutions to $\mathcal{S}$ is exactly the set of substitutions $\theta \circ \varphi'$ where φ' ranges over $\mathcal{S}'$-substitutions.*

8. Correctness of interpretation

We can now prove the correctness of the interpreter described in Section 5.

Lemma 8.1. *If the interpreter makes a single transition* $\mathcal{S} \stackrel{\rho}{\Longrightarrow} \mathcal{S}'$ *and if* φ *satisfies* $\mathcal{S}'$ *then* $\rho \circ \varphi$ *satisfies* $\mathcal{S}$.

Proof. We proceed by considering the cases that can cause a transition in the interpreter. The cases when this transition is a unification transition follow from Lemma 7.2. In all the other cases, ρ is empty so we simply need to show that a solution φ to $\mathcal{S}'$ is a solution to $\mathcal{S}$. To do this, we need to show that if J is a judgement in $\mathcal{S}$, then either φJ is $\top$, an equation between λ-convertible terms, or a sequent that has an $\mathcal{I}'$-proof. Since $\mathcal{S}'$ arises from changing one judgement of $\mathcal{S}$, we simply need to show that that one judgement has this property. Let $\hat{\varphi}$ be defined as $\hat{\varphi}t = \lambda\text{norm}(\varphi t)$.

Assume that the transition is caused by the BACKCHAIN step. That is, the state changed by replacing a sequent $\mathcal{P} \longrightarrow A$ with the conjunction

$$\mathcal{Q}(A \stackrel{o}{=} A' \wedge \mathcal{P} \longrightarrow G_1 \wedge \ldots \wedge \mathcal{P} \longrightarrow G_n),$$

where A and A' are atomic formulas, $\langle \mathcal{Q}, \{G_1, \ldots, G_n\}, A' \rangle \in \text{elab}(\mathcal{S}, \mathcal{P})$, and $n \geq 0$. Let φ satisfy $\mathcal{S}'$ and let Σ be the set of universally quantified variables of $\mathcal{S}$ with $\mathcal{P} \longrightarrow A$ in their scope. To show that φ also satisfies $\mathcal{S}$, it is necessary to show that if $\hat{\varphi}A = \hat{\varphi}A'$ and for every $i = 1, \ldots, n$, Σ ; $\hat{\varphi}\mathcal{P} \longrightarrow \hat{\varphi}G_i$ has an $\mathcal{I}'$-proof, then Σ ; $\hat{\varphi}\mathcal{P} \longrightarrow \hat{\varphi}G$ has an $\mathcal{I}'$-proof. This follows immediately if it is the case that $\langle \{\hat{\varphi}G_1, \ldots, \hat{\varphi}G_n\}, \hat{\varphi}A' \rangle \in |\hat{\varphi}\mathcal{P}|_\Sigma$, which follows by a simple induction on the definition of elab given the fact that $\langle \mathcal{Q}, \{G_1, \ldots, G_n\}, A' \rangle \in \text{elab}(\mathcal{S}, \mathcal{P})$.

Assume that the transition is caused by the GENERIC step. That is, $\mathcal{S}'$ arises by replacing a sequent occurrence $\mathcal{P} \longrightarrow \forall_\tau x.G$ in $\mathcal{S}$ with $\forall_\tau y(\mathcal{P} \longrightarrow [x \mapsto y]G)$, where y is not bound in $\mathcal{S}$. Assume that φ satisfies $\mathcal{S}'$. If Σ is the set of meta-level, universally quantified variables of $\mathcal{S}$ in which this sequent is in the scope, then $\Sigma, y : \tau$; $\hat{\varphi}\mathcal{P} \longrightarrow \hat{\varphi}[x \mapsto y]G$ has an $\mathcal{I}'$-proof and since $y \notin \Sigma$, the inference rule $\forall$-R yields a proof of Σ ; $\hat{\varphi}\mathcal{P} \longrightarrow \forall_\tau y.\hat{\varphi}[x \mapsto y]G$. Since no term in the range of φ contains y free, this sequent is the same as Σ ; $\hat{\varphi}\mathcal{P} \longrightarrow \hat{\varphi}(\forall_\tau y.[x \mapsto y]G)$, which is α-convertible to the sequent Σ ; $\hat{\varphi}\mathcal{P} \longrightarrow \hat{\varphi}(\forall_\tau x.G)$. Thus, φ satisfies $\mathcal{S}$.

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

The notation $\mathcal{S} \stackrel{\theta}{\Longrightarrow}{}^{*} \mathcal{S}'$ means that there exists a series of transitions

$$\mathcal{S} = \mathcal{S}_0 \stackrel{\rho_1}{\Longrightarrow} \cdots \stackrel{\rho_n}{\Longrightarrow} \mathcal{S}_n = \mathcal{S}' \quad (n \geq 0)$$

where θ is $\rho_1 \circ \cdots \circ \rho_n$ if $n > 0$ and empty if $n = 0$.

Theorem 8.2. *The state formula* $\mathcal{S}$ *has a solution* φ *if and only if* $\mathcal{S} \stackrel{\theta}{\Longrightarrow}{}^{*} \mathcal{S}'$ *where the only judgements in* $\mathcal{S}'$ *are occurrences of* $\top$ *and where there is a* $\mathcal{S}'$*-substitution* φ' *so that* $\theta \circ \varphi' = \varphi \pmod{\mathcal{S}}$.

Proof. The only-if part of this theorem follows by induction and Lemma 8.1. Assume that φ satisfies $\mathcal{S}$. Thus, for every judgement J of $\mathcal{S}$,either J is $\top$, φJ is an equation

between λ-convertible terms, or φJ specifies a sequent with an $\mathcal{I}'$-proof. Define the measure $\|\mathcal{S}\|$ to be the pair $\langle n, m\rangle$ where m is the number of equational judgements in $\mathcal{S}$ and n is the sum of the number of inference rules in minimal $\mathcal{I}'$-proofs proving all the sequent judgements in $\mathcal{S}$. Here, "minimal" is with respect to the number of occurrences of inference rules in a proof. These pairs are ordered lexicographically. The proof is completed by induction on the measure $\|\mathcal{S}\|$.

If $\mathcal{S}$ contains any equality judgements, apply the unification algorithm of Section 7 and make the transition $\mathcal{S} \stackrel{\theta'}{\Longrightarrow}{}^{*} \mathcal{S}'$. By Proposition 7.3, there is a solution φ' for $\mathcal{S}'$ so that $\theta' \circ \varphi' = \varphi \pmod{\mathcal{S}}$. Since $\|\mathcal{S}'\| < \|\mathcal{S}\|$, the inductive hypothesis provides a transition $\mathcal{S}' \stackrel{\theta''}{\Longrightarrow}{}^{*} \mathcal{S}''$ where the only judgements in $\mathcal{S}''$ are $\top$ and an $\mathcal{S}''$-substitution φ'' so that $\theta' \circ \varphi'' = \varphi' \pmod{\mathcal{S}'}$. Thus, setting θ to $\theta' \circ \theta''$, we have $\mathcal{S} \stackrel{\theta}{\Longrightarrow}{}^{*} \mathcal{S}''$ and $\theta \circ \varphi'' = \varphi \pmod{\mathcal{S}}$.

If $\mathcal{S}$ has some sequent judgement, say $\mathcal{P} \longrightarrow G$, then let Σ be the list of typed, universally quantified variables in which this sequent is in the scope. The structure of a minimal $\mathcal{I}'$-proof of $\Sigma\ ;\ \varphi\mathcal{P} \longrightarrow \varphi G$ dictates which transition can be performed. In particular, if the last inference rule in such a proof is $\wedge$-R, use the AND step; if it is $\supset$-R, use the AUGMENT step; if it is $\forall$-R, use the GENERIC step; if it is BC, use the BACKCHAIN step. We illustrate this final case in more detail since it is the hardest. Again, let $\hat{\varphi}$ be defined as $\hat{\varphi}t = \lambda\text{norm}(\varphi t)$.

Since the last rule is BC, G is atomic and there is a $\langle \Delta, \hat{\varphi}G\rangle \in |\hat{\varphi}\mathcal{P}|_{\Sigma}$ such that for every $H \in \Delta$, $\Sigma\ ;\ \varphi\mathcal{P} \longrightarrow H$ has an $\mathcal{I}'$-proof. By induction on the definition of elaboration, there is a triple $\langle \exists x_1 \ldots \exists x_m, \{G_1, \ldots, G_n\}, A\rangle \in \text{elab}(\mathcal{S}, \mathcal{P})$ $(m, n \geq 0)$ and a substitution $\psi = [x_1 \mapsto t_1, \ldots, x_m \mapsto t_m]$, where for $i = 1, \ldots, m$, t_i is a Σ-term, so that $(\varphi \circ \psi)A = \hat{\varphi}G$ and $(\varphi \circ \psi)\{G_1, \ldots, G_n\} = \Delta$. Use the BACKCHAIN step to yield the state formula $\mathcal{S}'$ where $\mathcal{P} \longrightarrow G$ is replaced with

$$\exists x_1 \ldots \exists x_m (G \stackrel{o}{=} A \wedge \mathcal{P} \longrightarrow G_1 \wedge \ldots \wedge \mathcal{P} \longrightarrow G_n).$$

Clearly, $\|\mathcal{S}'\| < \|\mathcal{S}\|$ and $\varphi \circ \psi$ is a solution to $\mathcal{S}''$. The proof of this case now follows by induction. ∎

The non-deterministic interpreter for L_λ described in Section 5 can be thought of doing computation in the following fashion. Let $\mathcal{S}$ be $\mathcal{Q}_\Sigma \exists \bar{x}(\mathcal{P} \longrightarrow G)$, for some signature Σ. Here, $\mathcal{P}$ is considered to be a logic program and G a query to be proved. The existential variables $\bar{x}$ are *logic variables* that the interpreter can instantiate as it needs in order to find a proof. Theorem 8.2 states that if the interpreter makes a transition $\mathcal{S} \stackrel{\theta}{\Longrightarrow}{}^{*} \mathcal{S}'$ where all of the judgements of $\mathcal{S}'$ are $\top$, then for every $\mathcal{S}'$-substitution φ', the substitution $\theta \circ \varphi'$ restricted to the variables in $\bar{x}$ is a solution or *answer substitution* to this computation. Theorem 8.2 also states that if there is a solution to the initial state then there is a series of transitions in the interpreter that yields a state formula whose only judgements are $\top$.

A simple, depth-first, deterministic interpreter for L_λ can be described as follows. First, we must consider the antecedent of sequents as lists instead of sets. The AUGMENT step concatenates formulas to the front of an antecedent. Elaboration, elab,

must take a state formula and a list of D-formulas and return a list of triples in which the second component is a list. The only backtrack points that must be remembered are those arising from the BACKCHAIN step. When given a state containing $\bot$, backtrack in a depth-first manner. When given a state containing an equational judgement, apply the unification algorithm. Otherwise, the given state formula contains only $\top$ and sequent judgements. If there are no sequent judgements, then make no further transitions: this represents a success. If there are a sequent judgements, select the first such judgement in a left-to-right order. If the succedent of that sequent is a conjunction, implication, or universal quantifier, then apply the AND, AUGMENT, or GENERIC step, respectively. If the succedent is an atomic formula, then select the first triple in the elaboration of the antecedent on which to backchain, leaving all the other members of the elaboration for subsequent backtracking. This style of search, although incomplete, is similar to the ones used in Prolog and λProlog.

9. Some Observations

Below are a few observations about unification and interpretation in L_λ.

9.1. Restricted β-conversion. When forming the transition $\mathcal{S} \stackrel{\rho}{\Longrightarrow} \mathcal{S}'$, ρ is applied to some of the judgements in $\mathcal{S}$ and the resulting λ-normal judgements are placed in $\mathcal{S}'$. Given the restrictions on meta-level existentially quantified variables within terms and formulas, only very weak instances of β-conversion are needed to compute these λ-normal forms. In particular, the only new β-redexes are those of the form $(\lambda x.t)y$ where y is a token that is not free in $\lambda x.t$ and is either universally quantified or λ-bound. Let β_0-conversion be the restriction to β-conversion where the only redexes considered are of the form $(\lambda x.t)x$. 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. For this reason, we shall refer to unification in L_λ as *$\beta_0\eta$-unification.*

A state formula is a $\forall\exists\forall$-state formula if there are no meta-level, universal quantifiers that are in the scope of an existential quantifier and themselves contain an existential quantifier in their scope. Huet's unification procedure [13] deals with $\beta\eta$-unification, sometimes called "higher-order" unification, for $\forall\exists\forall$-unification problems. As the author shows in [20], Huet's procedure can be extended to the case where the meta-level quantification is not so restricted. Applying this extended version to the unification problems considered in this paper results in the reduction of the unification problem to problems that contain only flexible-flexible equational judgements. In the general, unrestricted setting, computing unifiers for flexible-flexible equations is very unconstrained and undirected, so it is often best avoided. In the L_λ case, however, flexible-flexible equations are simple enough that their solutions can be completely characterized. Generalizations of the raising, pruning, and flexible-flexible steps described in Section 6 to $\beta\eta$-unification can be found in [20]. The algorithm presented here can be derived directly from that paper.

While $\beta_0\eta$-unification is much weaker than $\beta\eta$-unification, it is possible to specify declaratively $\beta\eta$-unification problems as logic programs within L_λ. Section 10 presents aspects of this specification and [19] describes the full translation.

9.2. $\forall\exists\forall$-Quantification. If a unification problem has the $\forall\exists\forall$-quantification structure, then any transition on such a problem yields a problem which is also $\forall\exists\forall$. On such problems, the raising step is never applicable although raising can be used to transform any unification problem into a $\forall\exists\forall$-unification problem. Thus, for the considerations of just unification, only $\forall\exists\forall$-unification problems are needed. Nipkow in [27] presents a version of the L_λ unification algorithm that works essentially on unification problems with $\forall\exists\forall$ quantification only. When considering the problem of interpreting L_λ, however, a transition from a state formula that has the $\forall\exists\forall$ form does not necessarily yield a similarly restricted state formula. Thus, after applying a BACKCHAINING step to a $\forall\exists\forall$-state formula, it might be necessary to apply the raising step several times to yield a $\forall\exists\forall$-state formula. Pairing of the raising step with backchaining is essentially the same as $\forall$-lifting in [29].

Let $\mathcal{S}$ be a unification problem and let $\mathcal{S} \stackrel{\theta_1 *}{\Longrightarrow} \mathcal{S}_1$ and $\mathcal{S} \stackrel{\theta_2 *}{\Longrightarrow} \mathcal{S}_2$ where $\mathcal{S}_1$ and $\mathcal{S}_2$ contain neither equations nor $\bot$. Is it possible to compare θ_1 and θ_2? As is shown in Section 7, these substitutions correspond to most general unifiers. In the first-order setting, two such most general unifiers differ only in the name of (existentially bound) variables. Given that the meta-level, quantificational structure of $\mathcal{S}_1$ and $\mathcal{S}_2$ can differ, it is not possible to so simply characterize such a relation between θ_1 and θ_2. If, however, $\mathcal{S}$ is a $\forall\exists\forall$-unification problem, then so too are $\mathcal{S}_1$ and $\mathcal{S}_2$. In this case, we can describe a simple relationship between θ_1 and θ_2. We first need the following lemma that holds for general state formulas. Its proof is immediate.

Lemma 9.1. *Let $\mathcal{S}$ be a unification problem and let $\mathcal{S} \stackrel{\theta *}{\Longrightarrow} \mathcal{S}'$.*

(i) The solutions to $\mathcal{S}$ can be put into one-to-one correspondence with the solutions to $\mathcal{S}'$ so that if the solution φ for $\mathcal{S}$ corresponds to the solution φ' for $\mathcal{S}'$ then $\theta \circ \varphi' = \varphi$.

(ii) Let $x \mapsto t \in \theta$ where x is existentially bound in $\mathcal{S}$. If an existentially quantified variable u of $\mathcal{S}'$ has an occurrence in t, then that occurrence is in a subterm of the form $u\bar{w}$ where $\bar{w}$ is a list of distinct variables that are either λ-bound in t or are universally quantified in $\mathcal{S}'$ in the scope of $\exists u$. If $\mathcal{S}$ is a $\forall\exists\forall$-state formula, then the list $\bar{w}$ consists of only λ-bound variables of t.

(iii) If u is not bound in $\mathcal{S}$, $\mathcal{S}'$, nor any state formula involved in this transition then $\forall_\tau u\mathcal{S} \stackrel{\theta *}{\Longrightarrow} \forall_\tau u\mathcal{S}'$.

Let $\mathcal{S}$ and $\mathcal{S}'$ be two $\forall\exists\forall$-state formulas each with $n \geq 0$ existentially quantified variables, namely $x_1, \ldots, x_n$ in $\mathcal{S}$ and $y_1, \ldots, y_n$ in $\mathcal{S}'$. A *variable renaming substitution from $\mathcal{S}$ to $\mathcal{S}'$* is a substitution φ such that for $i = 1, \ldots, n$, $\varphi x_i = \lambda\bar{w}.y_{\kappa i}\bar{v}$, where κ is a permutation of $\{1, \ldots, n\}$, $\bar{w}$ is some list of tokens (not including $y_{\kappa i}$), and $\bar{v}$ is a permutation of $\bar{w}$. The inverse of φ is the substitution $\varphi^{-1} = \{y \mapsto \lambda\bar{v}.x\bar{w} |\ x \mapsto \lambda\bar{w}.y\bar{v} \in \varphi\}$ and it is a variable renaming substitution from $\mathcal{S}'$ to $\mathcal{S}$.

Proposition 9.2. *Let $\mathcal{S}$ be a $\forall\exists\forall$-unification problem and let $\mathcal{S} \stackrel{\theta_1 *}{\Longrightarrow} \mathcal{S}_1$ and $\mathcal{S} \stackrel{\theta_2 *}{\Longrightarrow} \mathcal{S}_2$ where $\mathcal{S}_1$ and $\mathcal{S}_2$ contain neither equations nor $\bot$. Then there is a variable renaming substitution ρ such that $\theta_1 = \theta_2 \circ \rho$.*

Proof. Let the existentially quantified variables of $\mathcal{S}_1$ be $\bar{x} = x_1, \ldots, x_n$ $(n \geq 0)$ and let the existentially quantified variables of $\mathcal{S}_2$ be $\bar{y} = y_1, \ldots, y_m$ $(m \geq 0)$. There are

sequences of transitions from $\mathcal{S}$ to both $\mathcal{S}_1$ and $\mathcal{S}_2$. A *new* variable in this context is a variable that is not bound in $\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2$, nor any state formula in either of these sequences. Let $\bar{c} = c_1, \ldots, c_n$ be a list of distinct new variables and let $\forall\bar{c}$ be $\forall_{\tau_1} c_1 \ldots \forall_{\tau_n} c_n$ where τ_i is the type given to x_i in $\mathcal{S}_1$ $(i = 1, \ldots, n)$. By Lemma 9.1 *(iii)*, $\forall\bar{c}\mathcal{S} \stackrel{\theta_1 *}{\Longrightarrow} \forall\bar{c}\mathcal{S}_1$ and $\forall\bar{c}\mathcal{S} \stackrel{\theta_2 *}{\Longrightarrow} \forall\bar{c}\mathcal{S}_2$. Since $[\bar{x} \mapsto \bar{c}] = [x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]$ is $\forall\bar{c}\mathcal{S}_1$-substitution that satisfies $\forall\bar{c}\mathcal{S}_1$, $\theta_1 \circ [\bar{x} \mapsto \bar{c}]$ satisfies $\forall\bar{c}\mathcal{S}$, by Lemma 9.1 *(i)*. By the same lemma, there is a $\forall\bar{c}\mathcal{S}_2$-substitution φ so that $\theta_1 \circ [\bar{x} \mapsto \bar{c}] = \theta_2 \circ \varphi \pmod{\mathcal{S}}$. By applying $[\bar{c} \mapsto \bar{x}]$ to both sides of this equation and setting $\rho_1 = \varphi \circ [\bar{c} \mapsto \bar{x}]$, we get $\theta_1 = \theta_2 \circ \rho_1$. A dual argument yields a substitution ρ_2 so that $\theta_2 = \theta_1 \circ \rho_2$. Since $\theta_1 \circ \rho_2 = \theta_2 \circ \rho_1 \circ \rho_2$, the substitution $\rho_1 \circ \rho_2$ is the identity on the variables $\bar{y}$. A similar argument establishes $\rho_2 \circ \rho_1$ as the identity on the variables $\bar{x}$.

Let $j \in \{1, \ldots, m\}$. The only variables that can be free in $\rho_1 y_j$ are either the $\bar{x}$ variables or outer-most, universally quantified variables of $\mathcal{S}$. The latter case is not possible, however, since such a variable would also need to appear free in $\rho_2(\rho_1 y_j)$, but this is simply y_j. By Lemma 9.1 *(ii)*, if x_i occurs in $\rho_1 y_j$, for some $i = 1, \ldots, n$, then that occurrence of x_i is such that it is applied to only λ-bound variables of $\rho_1 y_j$. Thus, the only possible structure for $\rho_1 y_j$ is a term of the form $\lambda\bar{w}.x_i\bar{v}$ for some $i = 1, \ldots, n$ and where $\bar{v}$ is a list of distinct variables taken from the list $\bar{w}$. A similar observation holds for the term $\rho_2 x_i$ for $i \in \{1, \ldots, n\}$. Finally, $y_j = \rho_2(\rho_1 y_j) = \rho_2(\lambda\bar{w}.x_i\bar{v}) = \lambda\bar{w}.(\rho_2 x_i)\bar{v}$ which is only possible if $\rho_2 x_i$ $\lambda conv$ $\lambda\bar{v}.y_j\bar{w}$. Thus, $\bar{v}$ and $\bar{w}$ are permutations of each other. Since the converse relation between x_i and y_j must also hold, the list $\bar{x}$ and $\bar{y}$ must be of equal lengths $(n = m)$ and the connection between index i and index j is a permutation. Thus, both ρ_1 and ρ_2 are variable renaming substitutions. ∎

9.3. Untyped versions of unification and interpretation.

Type information was used in very few places in the description of the unification and interpreter transitions. It is possible, in fact, to describe an untyped version of L_λ and of the unification and interpreter transitions. The fact that $\beta_0\eta$-unification is independent of types means that it can be used in various different typed λ-calculi. For example, Pfenning in [31] uses a variation of L_λ-unification in the Calculus of Constructions. This situation is different from that of the full $\beta\eta$-theory of equality: types play a significant role in the search for solutions in the procedures given in [13] and in [19].

An untyped version of terms and G and D-formulas arises by simply deleting the typing information from inference rules in Figure 3. In that system, essentially existential variables can occur in predicate positions. Logic programming languages with such possibilities have been analyzed elsewhere [22, 25]. Here we shall assume that the two inference rules that permit the inference of an atomic D-formula and of an atomic G-formula are modified as in Figure 4. There the proviso (§) is that $\mathcal{Q}$ contains $\forall h$ and that $n \geq 0$. This restriction ensures that the resulting language is first-order in the sense that predicate substitutions never need to be considered. Notice that in this

$$\frac{\mathcal{Q} \xrightarrow{0} t_1 \quad \ldots \quad \mathcal{Q} \xrightarrow{0} t_n}{\mathcal{Q} \xrightarrow{+} ht_1 \cdots t_n}\ \S \qquad \frac{\mathcal{Q} \xrightarrow{0} t_1 \quad \ldots \quad \mathcal{Q} \xrightarrow{0} t_n}{\mathcal{Q} \xrightarrow{-} ht_1 \cdots t_n}\ \S$$

Figure 4: Two modified proof rules for the syntax of an untyped of version of L_λ

language, it is possible for the self-application of essentially universal variables but not for essentially existential variables.

To obtain the untyped version of the unification transitions requires only a small change to the rigid-rigid step and to the second case of the flexible-flexible step. In the rigid-rigid step, it is possible to have an equation of the form $ht_1 \ldots t_n = hs_1 \ldots s_p$ where $n \neq p$. Similarly, in the second case of the flexible-flexible step, it is possible to have the equation $vy_1 \ldots y_n = vz_1 \ldots z_p$ where $n \neq p$. In both of these cases, replace this equation with $\bot$. These transitions are correct since no substitution instance of these equations contain λ-convertible terms.

It should be noticed that the ξ step uses η-expansion, and that η-expansion can differ between the typed calculus (where it can only be used on terms of functional type) and the untyped calculus (where there is no such restriction). As this step is described, however, η-expansion is only used on terms which, if typed, must have functional type. Hence, no modification to this step is necessary.

The results in Section 7 and 8 can be established for the untyped case. As was mentioned in Section 5, there may be no $\mathcal{S}$-substitutions in the typed setting for a given $\mathcal{S}$: in the typed case the fact that the unification algorithm returns a state formula not containing $\bot$ is not enough to guarantee that there exist solutions. In the untyped case, there are always $\mathcal{S}$-substitutions for every untyped state formula $\mathcal{S}$. Thus, in the untyped setting, a unification problem has no solutions if and only if the unification algorithm returns a state formula containing $\bot$.

10. Examples of L_λ programs

The logic programming language λProlog [24] fully implements L_λ as well as the more general class of *higher-order hereditary Harrop formulas* [22]: the author has no experience in using an interpreter designed only to handle the L_λ subset. See [5] for a functional programming implementations of interpreters for languages such as L_λ and λProlog; see [23, 26] for discussions concerning the compilation of these languages.

Below we present several examples of L_λ programs written using the syntax of λProlog. 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 upper case 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 (the type of propositions is the built-in type `o`), and presents one D-formula, 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 constants, the type of bound variables can easily be inferred from their context.

Much of the formal and technical detail of the preceding several sections was caused by the difficulty of keeping track of bound variable names and scope. Since all these details have now formally been incorporated inside L_λ, programs written using L_λ should be relieved of some of the need to deal with these details. The following examples attempt to illustrate this point.

10.1. Specifying an object-logic.

Three meta-programs — substitution, Horn clause interpretation, and the computation of prenex normal forms — are presented in this section and all compute with the same first-order, object-logic. This object-logic contains universal and existential quantification and implication and conjunction. These are declared 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 contains 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 formulas. These formulas 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-logic 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 Horn clause interpreter and in the computation 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.
```

The first D-formula above is not a (meta-level) first-order Horn clause since it involves a variable `C` of functional type `term -> form` and since its body contains an implication and universal quantifier. 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 (and (q v a) (q a u)) (p u))))
```

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

```
u\(all v\(imp (and (q v a) (q a u)) (p 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 (modeled as a new universal quantifier in a state formula). 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, D-formulas 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 (via β_0-reduction) 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 (and (q v a) (q a d)) (p d)))
```

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

10.2. Implementing object-level substitution.

Equality and 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 object-level signature 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:

$$[\![t, s : \texttt{term}]\!] = \texttt{copyterm}\ t\ s$$

$$[\![t, s : \texttt{form}]\!] = \texttt{copyform}\ t\ s$$

$$[\![t, s : \tau \texttt{ -> } \sigma]\!] = \forall x \forall y([\![x, y : \tau]\!] \supset [\![t\ x, s\ y : \sigma]\!])$$

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, $(\texttt{copyterm}\ t\ s)$ 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 formula `(copyterm 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 $[\texttt{c} \mapsto \texttt{(f a)}]t$. 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 formula for implementing `subst`:

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

This formula is not a legal L_λ D-formula since the second occurrence of `M` is applied to another positively quantified universal variable. This formula correctly specifies substitution if the meta-level contains the full theory of β-conversion for simply typed λ-terms. Such a D-formula is available in λProlog and the higher-order logic programming languages described in [22] and [25]. These languages have a much richer unification problem than L_λ.

Similarly, consider the following λProlog code that makes use of full β-conversion.

```
type double (term -> term) -> term -> term -> o.
double F X (F (F X)).

kind termlist type.
type nil      termlist.
type cons     term -> termlist -> termlist.
type mapfun   (term -> term) -> termlist -> termlist -> o.

mapfun F nil nil.
mapfun F (cons X L) (cons (F X) K) :- mapfun F L K.
```

Such specifications cannot be written so directly in L_λ, but it is easy to see how they can be translated into L_λ: namely, find all instances of where an essentially existential variable is applied to arguments that are not distinct essentially universal variables and use a call to a `subst`-like predicate. In both of the examples above, we need to substitute at the type `term -> term`, so we need to introduce a substitution predicate at this type. The code below is the specification of that substitution predicate and the rewriting of two clauses above.

```
type substterm  (term -> term) -> term -> term -> o.
substterm M T N :- pi c\(copyterm c T => copyterm (M c) N).
```

```
double F X S :- substterm F X T, substterm F T S.
mapfun F (cons X L) (cons T K) :- substterm F X T, mapfun F L K.
```

Of course, the notions of doubling and mapping can be applied to more than just the type `term`: if difference types are used, simply use the appropriate `copy`-clauses and `subst`-predicate at those types.

10.3. 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 10.1). Such a unification problem, however, is permitted in the more general setting explored in [13]. While this is not a permissible unification problem in L_λ, it is very easy to solve this problem in L_λ using the `substterm` program written above. In particular, the set of substitutions for `F` that unifies `(F a)` and `(g a a)` is exactly the set of substitutions for `F` that makes the goal

```
substterm 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` and `substterm` defined above, although the translation is often more complex than the simple example illustrated here (see [19]).

10.4. Interpretation of first-order Horn clauses.

Object-level logic programs are represented by lists of formulas. The data type of formula lists and a simple membership program are specified by the following code.

```
kind  formlist    type.
type  nil     formlist.
type  cons    form -> formlist -> formlist.
type  memb    form -> formlist -> o.
memb X (cons X L).
memb X (cons Y L) :- memb X L.
```

The declarations above for `cons` and `nil` are intended to replace the declarations for them given in Subsection 10.2. It is possible in λProlog to specify lists and list operations that are polymorphic; in that case, one declaration could have been used in both of these settings.

The following code describes an interpreter for Horn clauses.

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

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

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

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

The `backchain` formula performs operations similar to those done by the BACKCHAIN transition presented in Section 5. 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.

10.5. 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 D-formulas so that the goal (**prenex** B C) is provable from them 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 auxiliary 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 non-determinism in **merge**: there are three ways to solve a **merge**-goal whose first argument is of the form `(and (all B) (all C))`. These formulas 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 formulas, 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 by a depth-first implementation of L_λ, such as λProlog, in the following fashion. Given the specification of **prenex** 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 strategy, 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 breadth-first search could, however, compute this converse.

11. Conclusion

Meta-programming systems need to be able to treat structures that contain notions of scope and bound variable. Conventional programming languages do not have 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. 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 non-deterministic interpretation of L_λ is described by merging unification with a sequent-style theorem prover. Several examples of L_λ programs are presented to show how it can be used to do simple meta-programming tasks.

Acknowledgements. I am grateful to Amy Felty, Elsa Gunter, John Hannan, Eva Ma, Daniel Nesmith, Tobias Nipkow, and Frank Pfenning for discussions and comments on this paper. The *Journal* reviewers also made several very helpful comments for improving the readability 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. The final draft of this paper was prepared while I was visiting LFCS, University of Edinburgh where I have been supported by SERC Grant No. GR/E 78487 "The Logical Framework" and ESPRIT Basic Research Action No. 3245 "Logical Frameworks: Design, Implementation, and Experiment."

12. References

[1] P. Andrews (1986). *An Introduction to Mathematical Logic and Type Theory*, Academic Press.

[2] N. de Bruijn (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.

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

[4] C. Elliott (1989). Higher-Order Unification with Dependent Types, Proceedings of the 1989 Rewriting Techniques and Applications, Springer-Verlag Lecture Notes in Computer Science, Vol. 355, 121 – 136.

[5] C. Elliott and F. Pfenning (1991). A Semi-Functional Implementation of a Higher-Order Logic Programming Language, in *Topics in Advanced Language Implementation*, edited by Peter Lee, MIT Press.

[6] A. Felty and D. Miller (1988). Specifying Theorem Provers in a Higher-Order Logic Programming Language, Ninth International Conference on Automated Deduction, Argonne, IL, 23 – 26, edited by E. Lusk and R. Overbeek, Springer-Verlag Lecture Notes in Computer Science, Vol. 310, 61 – 80.

[7] G. Gentzen (1935). Investigations into Logical Deductions, in *The Collected Papers of Gerhard Gentzen*, edited by M. E. Szabo, North-Holland Publishing Co., 1969, 68 – 131.

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

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

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

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

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

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

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

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

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

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

[18] D. Miller (1991). A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification, in *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, edited by P. Schroeder-Heister, Lecture Notes in Artificial Intelligence 475, Springer-Verlag, 253 – 281.

[19] D. Miller (1991). Unification of Simply Typed Lambda-Terms as Logic Programming, Eight International Logic Programming Conference, Paris, edited by Koichi Furukawa, MIT Press.

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

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

[22] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov (1991). Uniform Proofs as a Foundation for Logic Programming, *Annals of Pure and Applied Logic* **51**, 125 – 157.

[23] G. Nadathur and B. Jayaraman (1989). Towards a WAM Model for λProlog, North American Conference on Logic Programming, Cleveland, Ohio, edited by Ewing Lusk and Ross Overbeek, 1180 – 1198.

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

[25] G. Nadathur and D. Miller (1990). Higher-Order Horn Clauses, *Journal of the ACM* **37** (4), 777 – 814.

[26] G. Nadathur and D. Wilson (1990). A Representation of lambda terms suitable for operations on their intensions, ACM Conference on Lisp and Functional Programming, edited by M. Wand, ACM Press, 341 – 348.

[27] T. Nipkow (1991). Higher-Order Critical Pairs, Sixth Annual IEEE Symposium on Logic in Computer Science, Amsterdam, edited by G. Kahn.

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

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

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

[31] F. Pfenning (1991). Unification and Anti-Unification in the Calculus of Constructions, Sixth Annual IEEE Symposium on Logic in Computer Science, Amsterdam, edited by G. Kahn.

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

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

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

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