A LOGIC PROGRAMMING LANGUAGE WITH LAMBDA-ABSTRACTION, FUNCTION VARIABLES, AND SIMPLE UNIFICATION:

Extended Abstract

Draft, 20 September 1989

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

1. Introduction

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

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

One problem of designing a data type for λ-terms is that methods for destructuring them should be invariant under the intended notions of equality of λ-terms, which usually 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 does not yield equal answers. Although the use of deBruijn's representation of terms (deBruijn, 1972), where both of these terms would be represented by the same structure $\lambda(f11)$, can help simplify this one problem, 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 was the use of unification of simply typed λ-terms [Pietrzykowski & Jensen, 1976; Huet 1975]. Huet and Lang (1978) describe how such a technique, when restricted to second-order matching, could be used to analyze and manipulate simple functional and imperative programs. Their reliance on unification, which solved equations up to α, β, and η-conversion, made their meta programs elegant, simple to write, and easy to prove correct. They choose second-order matching because it was strong enough to implement a certain collection of template matching program transformations and it was decidable. The general problem of the unification of simply typed λ-terms of order 2 is undecidable (Goldfarb, 1981).

This use of unification on λ-terms has been extended in several recent papers. In Miller and Nadathur (1987), Felty and Miller (1988), and Miller and Hannan (1988a, 1988b), various meta programs, including theorem provers and program transformers, were written in the logic programming λProlog (Nadathur & Miller, 1988) that contains an implementation of such unification. Paulson (1986, 1988) exploits such unification in the theorem proving system Isabelle. All of these papers used unification in simply typed systems. Pfenning and Elliot (1988) argue that product types are also of use. Elliot (1989) has studied unification in a dependent type framework and Pfenning (1989) developed a logic programming language, Elf, that incorporates that unification process. That programming language can be used to provide direct implementation of signatures written in the LF type specification language (Harper, Honsel, & Plotkin, 1987).

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

In Section 2 we present a simple class of second-order unification problems that are at the core of L_λ. In Section 3 we generalize this unification problem slightly and in Section 4 we define L_λ. Finally, in Section 5 we present several examples of L_λ programs and show how to write arbitrary second-order unification problems as simple L_λ programs.

In this paper we shall explore only certain aspects of second-order quantification in logic programming. Much of what is presented here can be extended to third and higher-order. See the sequel to this paper for details. Although L_λ only marginally extends the first-order quantification nature of such logic programming languages as first-order

Horn clauses, L_λ has a much richer propositional nature than Horn clauses. This richer propositional aspects of the language make it possible to provide the logic programming paradigm with notions of modular programming (Miller, 1989a) and lexical scoping (Miller, 1989b), both of which are not supported directly by Horn clause programs.

2. A Simple Class of Second-Order Unification Problems

We assume that the reader is familiar with the basic properties of λ-terms and λ-conversion. Below we review some definitions and properties. See (Hindley & Seldin, 1986) for a more complete presentation.

Let S be a fixed, finite set of *primitive types* (also called *sorts*). The set of *types* is the smallest set of expressions that contains the primitive types and is closed under the construction of function types, denoted by the binary, infix symbol $\rightarrow$, which 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 $\{\tau_0, \tau_1, \ldots, \tau_n\} \subseteq S$.

For each type τ, we assume that there are denumerably many constants and variables of that type. Constants and variables do not overlap and if two constants (or variables) have different types, they are different constants (or variables).

If $\mathbf{x}$ and $\mathbf{s}$ are terms of the same type then $[\mathbf{x} \mapsto \mathbf{s}]$ denotes the operation of substituting $\mathbf{s}$ for all free occurrences of $\mathbf{x}$, systematically changing bound variables in order to avoid variable capture.

The rules of α, β, and η conversion are defined as usual. The binary relation *conv*, denoting λ-*conversion*, is defined so that $\mathbf{t}$ *conv* $\mathbf{s}$ if there is a list of terms $\mathbf{t}_1, \ldots, \mathbf{t}_n$, with $n \geq 1$, $\mathbf{t}$ equal to $\mathbf{t}_1$, $\mathbf{s}$ equal to $\mathbf{t}_n$, and for $i = 1, \ldots, n-1$, either $\mathbf{t}_i$ converts to $\mathbf{t}_{i+1}$ or $\mathbf{t}_{i+1}$ converts to $\mathbf{t}_i$ by α, β, or η. Expressions of the form $\lambda\mathbf{x}\ (\mathbf{t}\ \mathbf{x})$ are called η-redexes (provide $\mathbf{x}$ is not free in $\mathbf{t}$) while expressions of the form $(\lambda\mathbf{x}\ \mathbf{t})\mathbf{s}$ are called β-redexes. A term is in λ-normal form if it contains no β or η-redexes. Every term can be converted to a λ-normal term, and that normal term is unique up to the name of bound variables.

A *signature (over S)* is a finite set Σ of constants. We often enumerate signatures by listing their members as pairs, written $a{:}\,\tau$, where a is a constant of type τ. Although attaching a type in this way is redundant, it makes reading signatures easier. A signature is *second-order* if all its constants have type 0, 1, or 2. A term is a Σ-*term* if all of its constants are members of Σ.

Let Σ be a second-order signature. A *second-order unification problem (over* Σ*)* is a

finite list of *disagreement pairs*, pairs of λ-normal terms of the same type, such that each λ-term in each disagreement pair is a Σ-term and the free variables of each of those terms are of order 0 or 1.

2.1. Example. Let S be the set of primitive types $\{i\}$. Let Σ be the signature $\{a{:}\, i, b{:}\, i, f{:}\, i \to i, g{:}\, i \to i \to i\}$. The following are second-order unification problems (assuming that the type for the free variables F, G, H are first-order):

$$[\langle Fa, gab\rangle, \langle Fb, gbb\rangle] \tag{1}$$
$$[\langle \lambda x.Fxx, \lambda y.Gy\rangle] \tag{2}$$
$$[\langle \lambda x.g(Hx)(f(Fx)), \lambda y.g(fy)(fy)\rangle] \tag{3}$$
$$[\langle \lambda x \lambda y.f(gxy), \lambda u \lambda v.Huu\rangle] \tag{4}$$

∎

Standard operations on substitutions are assumed: that they can be represented as finite association lists between variables and terms of the same types, they can be considered as functions from terms to terms, that application of a substitution φ to a term t is denoted by φt, substitutions can be composed, and they can be ordered by subsumption (deriving the notion of *more general than*).

A *solution* (also, *unifier*) for a given unification problem P over signature Σ is a substitution φ that has all the free variables of P as a domain and closed Σ-terms as a range and is such that φt λ-converts to φs for every disagreement pair $\langle t, s\rangle$ in P.

2.2. Example. In the example above, (4) does not have a solution. Solutions for the other unification problems there are:

$$\{\langle F, \lambda w.gwb\rangle\} \tag{1}$$
$$\{\langle F, \lambda u \lambda v.g(fu)v\rangle, \langle G, \lambda w.g(fw)w\rangle\} \tag{2}$$
$$\{\langle F, \lambda u \lambda v.a\rangle, \langle G, \lambda w.a\rangle\} \tag{2}$$
$$\{\langle F, \lambda u.u\rangle, \langle H, f\rangle\} \tag{3}$$

Notice, that for examples (1) and (3), all solutions are equal (up to λ-conversion) to the displayed solutions. For example (2), there are an infinite number of solutions, all of which are the result of composing with the substitution $\{\langle G, \lambda w.Fww\rangle\}$. For technical reasons, we desire that this substitution, which has an open term in its range is not actually a solution to (2), although all its closed instances are. ∎

A second-order unification problem is *variable-defining* if it satisfies the following two restrictions: (1) every occurrence of a subformula of the form $(Ft_1 \ldots t_n)$, where F is a free variables and $n \geq 0$, is such that the terms $t_1 \ldots t_n$ are distinct λ-bound variables, and (2) all internally bound variables are of primitive type. In the examples above, only (3) is

a variable-defining unification problem. All first-order unification problems are variable-defining.

We have the following theorem regarding this class of unification problems.

2.3. Theorem. *Let P be a variable-defining unification problem. It is decidable whether or not P has a solution. Furthermore, if P has a solution, it is possible to find a substitution θ such that all solutions to P are instances of θ. (The substitution θ plays the role of a most general unifier.)*

Outline of Proof: We shall write a bar over a letter, for example, $\bar{x}$, to denote the (possibly empty) list of *distinct* variables and will write $\lambda\bar{x}$ to denote the abstraction of those variables. By using α and η-conversions, all disagreement pairs can be put into the form

$$\langle \lambda\bar{x}.ht_1 \dots t_n, \lambda\bar{x}.ks_1 \dots s_m \rangle,$$

where $m, n \geq 0$ and h and k are either constants or variables. Such a disagreement pair is *simplified* if h and k are distinct and a unification problem is simplified if all its disagreement pairs are simplified. Every unification problem can be put into an equivalent (i.e., set of solutions is the same) simplified unification problem by the following process: repeatedly replace a disagreement pair of the form $\langle \lambda\bar{x}.ht_1 \dots t_n, \lambda\bar{x}.hs_1 \dots s_n \rangle$ with the n disagreement pairs

$$\langle \lambda\bar{x}.t_1, \lambda\bar{x}.s_1 \rangle, \dots, \langle \lambda\bar{x}.t_n, \lambda\bar{x}.s_n \rangle.$$

Unification problems have essentially unique simplified forms. A terms in a simplified disagreement pair is of one of the following forms:

(i) $\lambda\bar{x}.ht_1 \dots t_n$ where h is a constant in the signature Σ,

(ii) $\lambda\bar{x}.y$ where y is some variable in the list $\bar{x}$, or

(iii) $\lambda\bar{x}.F\bar{y}$ where $\bar{y}$ is a list of distinct variables in the list $\bar{x}$ and F is free.

We now present a non-deterministic procedure that takes a simplified unification problem and either signals an failure or completes the construction of a sequence of unification problems in which the last one is empty. In that case, a substitution θ can be extracted from the construction of this sequence. It will then be the case that a substitution is a solution to P if and only if it is a closed instance of θ.

Pick some disagreement pair from the given unification problem P. If both terms of this pair are of the form (i) or (ii), then signal an error: P cannot have a solution. Thus, assume that one of the terms is of form (iii), namely, of the form $\lambda x_1 \dots \lambda x_n.Fy_1 \dots y_m$. Consider the following three case depending on the structure of the other term in the selected pair.

Case 1: Let the other term be $\lambda\bar{x}.kt_1 \dots t_p$. If F occurs free in this term, then signal a failure and stop. Otherwise, apply to the entire unification problem the substitution $\{\langle F, \lambda w_1 \dots \lambda w_n.k(H_1\bar{w}) \dots (H_p\bar{w}) \rangle\}$ and call this process again on the simplified form of the resulting unification problem. Here, the variables $H_1, \dots H_p$ are "new" (possessing no

occurrences in this or previous unification problems or substitutions) and of appropriate types.

Case 2: Let the other term be $\lambda\bar{x}.y$. If y is not in the list $y_1 \dots y_m$ then signal an error and stop. Otherwise, apply to the entire unification problem the substitution $\{\langle F, \lambda y_1 \dots \lambda y_m.y\rangle\}$ and call this process again on the simplified form of the resulting unification problem.

Case 3: Otherwise the given disagreement must be of the form

$$\langle \lambda x_1 \dots \lambda x_n.Fy_1 \dots y_m, \lambda x_1 \dots \lambda x_n.Gz_1 \dots z_p\rangle.$$

If F and G are distinct then let $w_1, \dots, w_q$ be an enumeration of the intersection of the lists $y_1 \dots y_m$ and $z_1 \dots z_p$. If F and G are the same, then let $w_1, \dots, w_q$ be an enumeration of the set $\{y_i \mid y_i = z_i, i = 1, \dots, m\}$. In either case, apply the substitution

$$\{\langle F, \lambda y_1 \dots \lambda y_m.Hw_1 \dots w_q)\rangle, \langle G, \lambda z_1 \dots \lambda z_p.Hw_1 \dots w_q)\rangle\},$$

and call this process again on the simplified form of the resulting unification problem. Here, the variable H is new and of appropriate type.

If this process does not signal an error, it will terminate only with an empty unification problem. At that point, let θ be the composite substitution that records the substitutions applied at each step in this procedure. A simple induction now establishes the desired property of θ: the set of all solutions to the original problem is the set of closed instances of θ.

3. Being Explicit and More Flexible with Quantification

Unification problems can naturally be considered closed formulas if disagreement pairs, written as equations, are conjoined together and if its free variables are existentially quantified. (Empty conjunctions will be written as $\top$.) Similarly, it is possible to replace top-level λ-abstractions in disagreement pairs with universal quantification: the disagreement pair $\langle \lambda\bar{x}.t, \lambda\bar{x}.s\rangle$ can be written as the formula $\forall\bar{x}.t = s$. For example, the unification problem

$$[\langle \lambda x.g(Hx)(f(Fx)), \lambda y.g(fy)(fy)\rangle, \langle \lambda x.gxL, \lambda x.gLL\rangle]$$

can be written as the quantified formula

$$\exists F \exists H \exists L \forall x \forall y.g(Hx)(f(Fx)) = g(fx)(fx) \wedge (gyL) = (gLL).$$

Thus, all unification problems can be written as prenex normal conjunctions of equations in which the quantifier prefix is of the form $\exists\forall$. Clearly, the unification process implicit in the proof of Theorem 2.3 can be generalized to this representation of unification problems.

Here, existentially quantified variables are of order 0 or 1 while universally quantified variables of order 0.

To define L_λ most naturally in the next section, it will be useful to generalize unification problems from having simple $\exists\forall$-prefixes to having richer quantifier alternation in the prefix. Given our second-order setting, this generalization is very easy to accommodate because the existence of second-order variables permits any quantifier prefix to be simplified to a $\exists\forall$-prefix. This is achieved by using *raising*, a process that is essentially the dual of Skolemization. In particular, let P be a unification problem of the form $\mathcal{Q}_1\forall x\exists h\mathcal{Q}_2 D$, where $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are strings of quantifiers, x is a variable of some primitive type τ and h is some variable of type $\tau_1 \to \ldots \to \tau_n \to \tau_0$ $(n \geq 0)$ where $\tau_0, \ldots, \tau_n$ are primitive types. The the result of raising P at $\exists h$ is the unification problem

$$\mathcal{Q}_1\exists h'\forall x\mathcal{Q}_2\theta D$$

where θ is the substitution $\{\langle h, h'x\rangle\}$ and h' is a variable or type $\tau \to \tau_1 \to \cdots \to \tau_n \to \tau_0$ that does not occur in P. By repeatedly raising a unification problem on all existential variables that are to the right of a universal variable, the problem can be simplified to a unification problem with a $\exists\forall$ prefix.

Provability of such quantified formulas in classical or intuitionistic settings where equality is interpreted by $\beta\eta$-conversion can be completely characterized as follows. Let P be such a formula all of whose non-logical constants are from Σ. P is provable if and only if there is a substitution for all the quantified variables in the prefix of P that has the following properties: first, the image of the universal variables are distinct constants that are not members of Σ, and second, the image of an existentially quantified variable x is a $\Sigma \cup \Sigma'$-term where Σ' is the set of constants instantiating universal variables quantified to the left of x in the prefix of P. (Actually, to get an exact match with certain versions of classical or intuitionistic logics, it is necessary to ensure that all types in S are inhabited, an assumption not enforced here.)

3.1. Example. Consider the quantified formula

$$\forall x\exists F.Fx = gxx$$

over the signature $\{g : i \to i \to i\}$. This formula can be raised at $\exists F$ to yield the formula

$$\exists F'\forall x.F'xx = gxx.$$

This is the formula version of the unification problem $[\langle \lambda x.F'xx, \lambda x.gxx\rangle]$. This unification problem has four solutions, namely, $\{\langle F', \lambda u\lambda v.guv\rangle\}$, $\{\langle F', \lambda u\lambda v.guu\rangle\}$, $\{\langle F', \lambda u\lambda v.gvv\rangle\}$, and $\{\langle F', \lambda u\lambda v.gvu\rangle\}$. These four solutions yield the four solutions to the original, unraised problem: $\{\langle x, a\rangle, \langle F, \lambda v.gav\rangle\}$, $\{\langle x, a\rangle, \langle F, \lambda v.gaa\rangle\}$, $\{\langle x, a\rangle, \langle F, \lambda v.gvv\rangle\}$, and $\{\langle x, a\rangle, \langle F, \lambda v.gva\rangle\}$. ∎

In order to lift the definition of variable-defining to this more general setting, we introduce a binary judgment $\vdash_T$ between quantifier prefixes and λ-terms. This judgment is defined by the provability rules given in Figure 1. The three rules have the following provisos.

α The terms t and t' are related by α-conversion.

$\dagger$ The variable x does not occur in $\mathcal{Q}$. Otherwise, use the rule of α-conversion to change the bound variable.

$\ddagger$ The symbol h is either a constant or a universally quantified variable in $\mathcal{Q}$.

$\sharp$ The variable F is existentially quantified to the left of where the distinct variables $x_1, \ldots, x_n$ are universally quantified.

Now, we shall consider a quantified conjunctions of equation $\mathcal{Q}D$ to be a variable-defining unification problem if for every term t in each of its equations, it is the case that the judgement $\mathcal{Q} \vdash_T t$ is provable from the rules in Figure 1. The justification for this extension to the definition of variable-defining is that the result of completely raising any such formula results in a formula that can be identified with such unification problems as defined in the previous section.

$$\frac{\mathcal{Q} \vdash_T t}{\mathcal{Q} \vdash_T t'}\ \alpha \qquad \frac{\mathcal{Q}\forall x \vdash_T t}{\mathcal{Q} \vdash_T \lambda x.t}\ \dagger \qquad \frac{\mathcal{Q} \vdash_T t_1 \quad \ldots \quad \mathcal{Q} \vdash_T t_n}{\mathcal{Q} \vdash_T h t_1 \cdots t_n}\ \ddagger$$

$$\mathcal{Q} \vdash_T F x_1 \cdots x_n\ \sharp$$

Figure 1: Term syntax rules

The procedure of taking a quantified conjunction of equations from this now extended class of variable-defining unification problems, raising to get a $\exists\forall$ prefix, and applying the procedure of Theorem 2.3 to the result, is a complete and sound theorem proving method for such quantified formulas. Obviously, since raising is such a simple device, it is straightforward to describe a unification procedure that can handle quantifier alternation directly without using raising as a preprocessing step.

4. The Logic Programming Language L_λ

We shall now present a logic programming language whose implementation only requires considering variable-defining unification problems under a mixed prefix. In order to introduce formulas we use the technique of Church (1940) of assuming that the type symbol o is a primitive symbol, that is, is always a member of S. The logical constants are given the following types: $\wedge, \supset$ are all of type $o \to o \to o$; $\top$ is of type o; and $\forall$ is of type

$(\tau \to o) \to o$, for all types τ that are of order 1 or 0 and do not contain the type symbol o. The binary constants $\wedge, \supset$ will be written with infix notation and the expression $\forall \lambda x.B$ will be abbreviated to $\forall x.B$. A formula will be identified as a term of type o. The fact that type o is not permitted in the types of quantified variables means that this language is higher-order in a very weak sense: predicate quantification, the difficult aspect of higher-order logics, is not permitted in this version of L_λ. Inclusion of predicate quantification can be accommodated along the lines that have been studied for higher-order Horn clauses (Nadathur & Miller, TA) and higher-order hereditary Harrop formulas (Miller, Nadathur, Pfenning, & Scedrov, TA).

Following the lines outlined in (Miller, 1989a) and (Miller, Nadathur, Pfenning, & Scedrov, TA), a logic programming language is a specification of a set of *goal formulas*, a set of *program* or *definite clauses*, and a provability relation that satisfies a goal directed interpretations of logical constants in non-atomic goals. Intuitionistic or minimal logic will serve as the provability relation for L_λ. We shall define both the class of goal formulas and definite clauses by a proof system which extends the one for terms that was presented in Figure 1. That is done by introducing two new binary judgments, $\vdash_G$ and $\vdash_D$ that relate quantifier strings with formulas. The inference rules for these two judgments are given in Figure 2. Several examples of goals and definite clauses are given in the next section. For now, note that first-order Horn clauses can be identified with subsets of both goal formulas and definite clauses.

$$\frac{\mathcal{Q}\forall x \vdash_G G}{\mathcal{Q} \vdash_G \forall x.G}\ \dagger,\sharp \qquad \frac{\mathcal{Q} \vdash_G G_1 \qquad \mathcal{Q} \vdash_G G_2}{\mathcal{Q} \vdash_G G_1 \wedge G_2} \qquad \frac{\mathcal{Q} \vdash_D D \qquad \mathcal{Q} \vdash_G G}{\mathcal{Q} \vdash_G D \supset G} \qquad \frac{\mathcal{Q} \vdash_T A}{\mathcal{Q} \vdash_G A}\ \ddagger$$

$$\frac{\mathcal{Q}\exists x \vdash_D D}{\mathcal{Q} \vdash_D \forall x.D}\ \dagger \qquad \frac{\mathcal{Q} \vdash_D D_1 \qquad \mathcal{Q} \vdash_D D_2}{\mathcal{Q} \vdash_D D_1 \wedge D_2} \qquad \frac{\mathcal{Q} \vdash_G G \qquad \mathcal{Q} \vdash_D D}{\mathcal{Q} \vdash_D G \supset D} \qquad \frac{\mathcal{Q} \vdash_T A}{\mathcal{Q} \vdash_D A}\ \ddagger$$

Figure 2: Goal and clause syntax rules

The inference rules in Figure 2 have the following provisos:

† The variable x is does not occur in $\mathcal{Q}$. Otherwise, use the rule of α-conversion to change the bound variable (assuming that there is an α-rule for both $\vdash_G$ and $\vdash_D$ just as there is one for $\vdash_T$).

‡ The formula A is atomic, that is, its top-level symbol is not a logical connective.

♯ The quantified variable x is of primitive, non-propositional type.

In the rest of this section we outline how intuitionistic theorem proving of goal formulas from finite sets of definite clauses can be implemented. From this sketch, it will be clear that only variable-defining unification problems need to be solved.

Unification problems contain one primitive proposition, that of equality. To specify

a theorem prover, we introduce a second judgment $\Longrightarrow$ to denote sequent provability. A sequent judgment will be between lists of formulas and formulas. Lists will be built by using two new constants, @ for "cons" and *nil*. Both @ and $\Longrightarrow$ will be written with infix notation and @ associates to the right. An *equality proposition* is any expression of the form $t = s$. A *sequent proposition* is any expression of the form $\mathcal{P} \Longrightarrow L$. A *deduction problem* is a quantified conjunction of equality and sequent judgments P. Thus, a deduction problem without any sequent propositions is a unification problem. Let G be a goal formula. The *deduction problem for* G is the formula $\mathcal{Q}[nil \Longrightarrow G']$ where G' is the λ-normal form of G and $\mathcal{Q}$ is the purely existential quantifier prefix with a quantifier for each free variable in G'.

Let P be a deduction problem with prefix $\mathcal{Q}$ and some sequent proposition. Below are five rewriting rules for such deduction problems. The first three do not change the prefix of a deduction problem while the last two add an innermost quantifier to the prefix. In all five cases, a sequent proposition is replaced by either an equality proposition or one or two sequent propositions.

init: Replace $\mathcal{P} \Longrightarrow A$ in P with $A = A'$ provided A' is a member of $\mathcal{P}$ and A and A' are atomic.

$\supset$ -R: Replace $\mathcal{P} \Longrightarrow D \supset G$ with $D \mathbin{@} \mathcal{P} \Longrightarrow G$.

$\supset$ -L: Replace $\mathcal{P} \Longrightarrow G_1$ with $(D \mathbin{@} \mathcal{P} \Longrightarrow G_1) \wedge (\mathcal{P} \Longrightarrow G_2)$ provided $G_2 \supset D$ is member of $\mathcal{P}$.

$\forall$-R: Replace $\mathcal{P} \Longrightarrow \forall x.G$ with $\mathcal{P} \Longrightarrow G$ and x is a variable not present in $\mathcal{Q}$ (otherwise, change its name first). The new prefix is $\mathcal{Q}\forall x$.

$\forall$-L: Replace $\mathcal{P} \Longrightarrow G$ with $D \mathbin{@} \mathcal{P} \Longrightarrow G$ where $\forall x.D$ is a member of $\mathcal{P}$ and x is a variable not present in $\mathcal{Q}$ (otherwise, change its name first). The new prefix is $\mathcal{Q}\exists x$.

A deduction problem P can be *rewritten* to a deduction problem P' if there is a sequence of zero or more deductions problems $P = P_1, \ldots, P_n = P'$ such that for $i = 1, \ldots, n-1$, P_{i+1} results from P_i by one of the above five rules.

A proof of the following theorem can be found in (Miller, drafts).

4.1. Theorem. *Let $\mathcal{P}$ be a list of closed definite clauses and let G be a goal formula all of whose free variables are in the list $\bar{x}$. The deduction problem $\exists \bar{x}[\mathcal{P} \Longrightarrow G]$ can be rewritten to a solvable unification problem if and only if there is some instance of G which is intuitionistically entailed by $\mathcal{P}$.*

The theorem proving process implicit in this theorem places all the work of unification at the end of the search for a proof. This is clearly undesirable for most practical considerations. Fortunately, it is easy to describe a modificated deduction rewriting scheme which attempts to solve equations when they first appear. Since all these equations are variable-defining, it is possible to completely solve them.

For the sake of presenting examples in the next section, we provide an operational description of a non-deterministic interpreter.

Let A be a syntactic variable for atomic formulas, G be a syntactic variable for goal formulas, and D a syntactic variable for definite clauses. Let $\mathcal{P}$ be a finite set of definite clauses. We define $\mathcal{P}_\Sigma$ to be the smallest set of pairs $\langle D, \mathcal{G}\rangle$ where D is a closed definite formula and $\mathcal{G}$ is a finite set of closed goal formulas satisfying the following conditions.

(i) If $D \in \mathcal{P}$ then $\langle D, \emptyset\rangle \in \mathcal{P}_\Sigma$.

(ii) If $\langle D_1 \wedge D_2, \mathcal{G}\rangle \in \mathcal{P}_\Sigma$ then $\langle D_1, \mathcal{G}\rangle \in \mathcal{P}_\Sigma$ and $\langle D_2, \mathcal{G}\rangle \in \mathcal{P}_\Sigma$.

(iii) If $\langle \forall x.D, \mathcal{G}\rangle \in \mathcal{P}_\Sigma$ then $\langle [x \mapsto t]D, \mathcal{G}\rangle \in \mathcal{P}_\Sigma$ for every closed term t over signature Σ.

(iv) If $\langle G \supset D, \mathcal{G}\rangle \in \mathcal{P}_\Sigma$ then $\langle D, \{G\} \cup \mathcal{G}\rangle \in \mathcal{P}_\Sigma$.

We can now describe a nondeterministic interpreter for L_λ. Let Σ be a given second-order signature and let $\mathcal{P}$ be a finite set of closed definite clauses and let G be a closed goal formula, both over Σ. Intuitionistic provability of G from $\mathcal{P}$, written as $\Sigma; \mathcal{P} \vdash_I G$ can be characterized as follows:

AND: $\Sigma; \mathcal{P} \vdash_I G_1 \wedge G_2$ if and only 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 and only if $\Sigma; \mathcal{P} \cup \{D\} \vdash_I G$.

GENERIC: $\Sigma; \mathcal{P} \vdash_I \forall x.G$ if and only if $\Sigma \cup \{c\}; \mathcal{P} \vdash_I [x \mapsto c]G$, provided that c is a constant of the same type as x and that c does not occur in Σ.

BACKCHAIN: $\Sigma; \mathcal{P} \vdash_I A$ if and only if there is a pair $\langle A, \mathcal{G}\rangle \in \mathcal{P}_\Sigma$ such that for all $G \in \mathcal{G}$, $\Sigma; \mathcal{P} \vdash_I G$.

In this description of a theorem prover, details of unification have been suppressed. When free variables are allowed in the formulas of $\mathcal{P}$ and in G, the condition that c be a "new" constant in GENERIC must mean that any variables free when c was introduced must be restricted so that future instantiations of them cannot be to terms containing c. This restriction is enforced in our earlier unification-based description of theorem proving in L_λ by the alternation of quantifiers.

5. Examples of L_λ programs

We have used prefixed conjunctions over equations and sequents as a meta language for L_λ. We shall now use L_λ as a meta language for a few simple applications. The hope is that since we have addressed the nature of α-conversion and scope in the meta theory of L_λ, the applications written in L_λ need not address these issues directly.

For our first example, we show how to code a predicate that will determine if its one argument represents a tail recursive procedure in a simple functional programming language. To this end, we shall assume that we have the following signature available. Here we list types and the constants that are of that type.

$$\begin{aligned} i &: 0, 1, 2, \ldots \\ i \to i \to i &: app, equal, plus, times, minus \\ i \to i \to i \to i &: if \\ (i \to i) \to i &: abs, fix \end{aligned}$$

Terms of type i will be identified with the representation of simple functional programs. The constant *app* is used to form curried function application, *abs* forms abstractions, and fix forms recursive definitions from abstractions. An example of a program (term of type i) is

$$(fix\lambda f(abs\lambda x(abs\lambda y(if\ (equal\ 0\ x)\ y\ (app\ (app\ f\ (minus\ x\ 1))\ (times\ x\ y))))))$$

which defines a tail recursive form of the factorial program: the value of $(fact\ n\ 1)$ is intended to be $n!$ where $fact$ denotes the displayed formulas above.

Before presenting the tail recursion example, consider specifying the one place predicate *vacuous* of type $i \rightarrow o$ such that $(vacuous\ B)$ is provable if and only if B is a vacuously defined recursive function. In L_λ, specifying this predicate is trivial and is given by the following universally quantified clause:

$$\forall H.vacuous\ (fix\ \lambda F.H).$$

Any atom of the form $(vacuous\ B)$ that is provable from this clause must be a term that is a top-level fix which is applied to a vacuous abstraction.

Now consider adding the following predicate constants to the signature displayed above:

$$i \rightarrow o : trfun$$
$$(i \rightarrow i) \rightarrow o : trabs, trcond, tratom$$

To make the specification of these predicates easier to read and more in line with standard logic programming conventions, when displaying definite clauses we shall use :- as the converse of $\supset$ and assume that free variables are universally quantified around the scope of each clause. Consider the following definite clauses.

$trfun\ (fix\ B)$:- $trabs\ B$.
$trabs\ \lambda f(abs\ (B\ f))$:- $\forall x.trabs\ \lambda f(Bfx)$.
$trabs\ C$:- $trcond\ C$.
$trcond\ \lambda f(if\ C\ (B_1\ f)\ (B_2\ f))$:- $trcond\ B_1 \wedge trcond\ B_2$.
$trcond\ B$:- $tratom\ B$.
$tratom\ \lambda f\ H$.
$tratom\ \lambda f\ f$.
$tratom\ \lambda f(app\ (H\ f)\ T)$:- $tratom\ H$.

In the three predicates $trabs, trcond$, and $tratom$, a fragment of the programs is analyzed. In each of these cases, the recursive function abstraction is maintained around that program fragment. This permits unification to determine whether or not that recursive call occurs in parts of the recursion, a key part of the analysis of tail recursion.

For our final example of L_λ programs, we show how to reduce arbitrary second-order unification problems to simple L_λ programs for computing such unifiers. First, by a second-order unification problem we mean one with free variables of order 0 or 1 and internal abstractions of order 0. Second, we need to fix a signature Σ over which we plan to encode second-order unification. For this example, we choose Σ to be

$$\begin{aligned} i &: a, b \\ i \to i &: f \\ i \to i \to i &: g \\ (i \to i) \to i &: h \end{aligned}$$

Given this signature, we develop the following copying program.

$$\begin{aligned} &\mathit{copy}\ a\ a. \\ &\mathit{copy}\ b\ b. \\ &\mathit{copy}\ (f\ X)\ (f\ Y) \text{ :- } \mathit{copy}\ X\ Y. \\ &\mathit{copy}\ (g\ X\ Y)\ (g\ U\ V) \text{ :- } \mathit{copy}\ X\ U \wedge \mathit{copy}\ Y\ V. \\ &\mathit{copy}\ (h\ F)\ (h\ G) \text{ :- } \forall x \forall y (\mathit{copy}\ x\ y \supset \mathit{copy}\ (F\ x)\ (G\ y)). \end{aligned}$$

A closed goal $(\mathit{copy}\ t\ s)$ is provable if t and s are the same term. (Calling this identity predicate "copy" suits its operational behavior.) Clearly, *copy* has type $i \to i \to o$. We shall also need a have the denumerable class of programs, of which we present only the first two.

$$\begin{aligned} &\mathit{subst}_1\ T_1\ M\ P \text{ :- } \forall x_1\ (\mathit{copy}\ x_1\ T_1 \supset \mathit{copy}\ (M\ x_1)\ P). \\ &\mathit{subst}_2\ T_1\ T_2\ M\ P \text{ :- } \forall x_1\ (\mathit{copy}\ x_1\ T_1 \supset \forall x_2 (\mathit{copy}\ x_2\ T_2 \supset \mathit{copy}\ (M\ x_1\ x_2)\ P)). \end{aligned}$$

The type of subst_1 is $i \to (i \to i) \to i \to o$ and the type of subst_2 is $i \to i \to (i \to i \to i) \to i \to o$. In general, subst_n describes how to simultaneously substitute n terms for n abstractions. Given any one second-order unification problem, only a finite number of these subst_n programs are needed. Since L_λ does not permit interesting β-redexes, substitutions of this kind are not done at the meta level.

5.1. Example. Consider the problem of find a closed term to substitute for the variable F so that (Fa) λ-converts to $(g\ a\ a)$. The redex (Fa) is not variable-restricted. The β-reduction that would substitution a into F can be transferred to the program subst_1. That is, the closed terms that solve this unification program are exactly the closed terms that solve the open goal $(\mathit{subst}_1\ a\ F\ (g\ a\ a))$. ∎

More complex unification problems can be encoded using this technique. New free variables may have to be introduced, however.

5.2. Example. Consider the problem of finding closed Σ-terms to substitute for F and X so that the two terms $(F\ (F\ a))$ and $(f\ (f\ X))$ are λ-convertible. This can be encoded into L_λ as the open goal

$$subst_1\ a\ F\ H \wedge subst_1\ H\ F\ (f\ (f\ X)).$$

These two problems share the same closed instances of F and X. For another example, consider the unifying $\lambda x(G\ x\ x)$ with $\lambda x(f\ x)$. This can be translated into the problem of solving the open L_λ goal

$$\forall x.copy\ x\ x \supset subst_2\ x\ x\ G\ (f\ x).$$

∎

The general translation procedure can be described as follows: Take an initial second-order unification problem $t = s$ and set that equal to the goal G_0. Now, given G_i, if it is not an L_λ goal then we construct G_{i+1} as follows. G_i is of the form $t' = s' \wedge G'$ where G' is an L_λ goal containing calls to $subst_1$, $subst_2$, etc. Since G_i is not an L_λ goal, select an innermost subterm of $t' = s'$ that is of the form $(Ft_1 \cdots t_m)$ where F is a free (existential) variable and were this subterm is not variable-defining. This subterm may appear in the scope of λ-abstractions. Say that those variables are the list $\bar{x} = x_1, \ldots, x_n$. Then replace the occurrence $(Ft_1 \cdots t_m)$ with the expression $(G\bar{x})$ for a new free variable G of suitable type. This replacement yields a new equation $t'' = s''$ (only one of the terms will actually be modified). The goal G_{i+1} is then set equal to

$$t'' = s'' \wedge G' \wedge \forall \bar{x}[copy\ x_1\ x_1 \supset \cdots \supset copy\ x_n\ x_n \supset subst_m\ t_1\ \cdots\ t_m\ F\ (G\ x_1 \cdots x_n)].$$

The resulting goal has one fewer non-variable-restricted subterms so this process will terminate with an L_λ goal. Since solutions to each successive goal is are equivalent on the variables that they share, the final L_λ goal does correctly represent the original unification problem.

Acknowledgements. I am grateful to Amy Felty, Elsa Gunter, John Hannan, and Frank Pfenning for discussions related to the 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.

6. References

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

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

Elliott, C. (1988), "Some Extensions and Applications of Higher-Order Unification," Ergo Report 88-061, CMU.

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

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

Hannan, J. and Miller, D. (1988b), A Meta Language for Functional Programs, Proceedings of the 1988 Workshop on Meta Programming, Bristol, UK, eds. H. Rogers and H. Abramson, MIT Press (in press).

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

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

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

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

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

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

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

Miller, D. (drafts), two papers: Solutions to λ-Term Equations under a Mixed Prefix, and Unification under a Mixed Prefix.

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

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

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

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

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

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

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

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

Pietrzykowski, T., and Jensen, D. (1976), "Mechanizing ω-Order Type Theory Through Unification," TCS 3 123-171.