Unification of Simply Typed Lambda-Terms as Logic Programming

Dale Miller*
Laboratory for the Foundation of Computer Science
University of Edinburgh, and
Computer Science Department
University of Pennsylvania

March 1991

Abstract

The unification of simply typed λ-terms modulo the rules of β- and η-conversions is often called "higher-order" unification because of the possible presence of variables of functional type. This kind of unification is undecidable in general and if unifiers exist, most general unifiers may not exist. In this paper, we show that such unification problems can be coded as a query of the logic programming language L_λ in a natural and clear fashion. In a sense, the translation only involves explicitly axiomatizing in L_λ the notions of equality and substitution of the simply typed λ-calculus: the rest of the unification process can be viewed as simply an interpreter of L_λ searching for proofs using those axioms.

1 Introduction

Various recent computer systems require typed λ-terms to be unified. For example, the theorem proving systems TPS [1] and Isabelle [14] and the logic programming language λProlog [13] all require unification of simply typed λ-terms. The logic programming language Elf [15], based on the type system LF [5], requires a similar operation for dependent typed λ-terms. Flexible implementations of type systems will probably need to employ various aspects of such unification.

In order to avoid using the very vague and over used adjective "higher-order," we shall refer to the problem of unifying simply typed λ-terms modulo β- and η-conversion as $\beta\eta$-unification. There have been several presentations of $\beta\eta$-unification. One of the first to have been implemented in numerous systems was given by Huet in [7]. Snyder and Gallier in [16] and the author in [11] follow Huet's presentation closely except that details of the search for unifiers are made more declarative using notions similar to the transition systems found in [8].

The presentation given here will depart significantly from those found in these other papers, although interesting connections between these presentations can be made. The most significant departure is that the logic programming language L_λ [10] is employed to assist in specifying

*Appears in the Proceedings of the 1991 International Conference on Logic Programming, edited by Koichi Furukawa, June 1991.

$\beta\eta$-unification. To a certain extent, the transition systems used in [8, 11, 16] could be formalized using first-order Horn clauses. The logic L_λ is more expressive than Horn clauses because it contains constructs for the scoped introduction of program clauses and local constants. These scoping constructs are used to address problems in handling the scopes and names of bound variables in λ-terms.

The logic L_λ is a weak subset of the logic underlying λProlog: it is weaker in that an implementation of L_λ would only need to contain a kind of first-order unification while λProlog needs full higher-order unification. This paper is an attempt to understand exactly this gap and to show that the gap can be bridged completely within the weaker language in a very direct and declarative way.

This paper is divided into the following sections. The next section motivates a style of specification used throughout the paper. Section 3 describes some basic aspects of the simply typed λ-calculus and Section 4 presents two logic programming languages, hh^ω and L_λ, that we use for specification. Equality and substitution are given an L_λ specification in Section 5. The non-deterministic specification of $\beta\eta$-unification is completed in Section 6. Some considerations for producing a deterministic implementation of this specification are given in Section 7. We briefly conclude in Section 8.

2 Motivations

Consider a simple, multi-sorted first-order logic that consists of the primitive types (sorts) $S = \{i, j\}$ and signature (i.e., the set of constants)

$$\Sigma_0 = \{a : i,\ b : j,\ f : i \to j,\ g : j \to i \to i\}.$$

For example, the term $(g\ (f\ a)\ (g\ b\ a))$ is a closed, first-order terms of type i over Σ_0. Let $copy_i$ and $copy_j$ be the binary equality predicates for these two types (the reason for chosing the root word "copy" instead of, say, "equal" will be apparent later). The provable instances of equality for types i and j can be axiomatized using the two clauses

$$\forall_i x.copy_i\ x\ x, \quad \forall_j x.copy_j\ x\ x.$$

(The subscript on a quantifier indicates the type the quantified variable assumes in its scope.) Of course, this description of equality does not provide any information about how equality is checked. It is a convenient specification, however, since it is actually independent of the signature used to build terms of these two sorts. A more detailed specification for these predicates given the signature above would be the clauses $\mathcal{C}_0$ listed below:

$$\begin{gathered} copy_i\ a\ a, \qquad copy_j\ b\ b, \\ \forall_i x \forall_i u (copy_i\ x\ u \supset copy_j\ (f\ x)\ (f\ u)), \\ \forall_j x \forall_j u (copy_j\ x\ u \supset \forall_i y \forall_i v (copy_i\ y\ v \supset copy_i\ (g\ x\ y)\ (g\ u\ v))). \end{gathered}$$

It is a simple matter to prove that if t and s are two closed terms, then $copy_i\ t\ s$ is provable from these formulas if and only if t and s are equal terms of type i over the signature Σ_0. All the clauses in $\mathcal{C}_0$ are essentially first-order Horn clauses.

Given this formulation of equality, it is very simple to specify substitution in the following fashion. Let $x : i$ be a "new" constant (chosen so as not to be in Σ_0), let t be some closed term of type i over Σ_0, and let s be some term over $\Sigma_0 \cup \{x : i\}$. Then it is again an easy matter to show that the atom $copy_i\ s\ r$ is provable from $\mathcal{C}_0$ augmented with the clause $copy_i\ x\ t$ if and only if r is the result of substituting t for x in s; that is, $\mathcal{C}_0 \cup \{copy_i\ x\ t\} \vdash copy_i\ s\ r$ if and only if $r = [x \mapsto t]s$. This simple device of augmenting equality programs will be used frequently to encode substitution. Since *copy* will sometimes indicate equality and sometimes substitution, depending on the context, it was named for a more operational and neutral concept.

Finally, notice that the structure of the signature of Σ_0 gives rise immediately to the structure of the program clauses in $\mathcal{C}_0$. Thus, for each functional arrow $\rightarrow$ in the type of a constant, there corresponds two universal quantifiers and an implication in the program. Following this observation, it seems clear how to incorporate a constant of second order type into the specification of equality. For example, let $\Sigma_1 = \Sigma_0 \cup \{h : (i \rightarrow j) \rightarrow i\}$. The λ-term $h\ (\lambda w.f\ (g\ b\ w))$, for example, is a Σ_1-term of type i. Following the example above, the clause for describing equality for terms containing h should be written as

$$\forall_{i\rightarrow j} x \forall_{i\rightarrow j} u (\forall_i y \forall_i v (copy_i\ y\ v \supset copy_j\ (x\ y)\ (u\ v)) \supset copy_j\ (h\ x)\ (h\ u)).$$

As we shall see later in Section 5, this is the correct axiomatization of equality with respect to the constant h. This clause, however, is clearly not a first-order Horn clause since it contains an implication and universal quantifier in its body and because it uses quantification of the second-order variables x and u.

The material in the next two sections provide a formal background by which the above observations can be made precise and generalized.

3 Simply Typed λ-Calculus

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, using the binary, infix symbol $\rightarrow$. The Greek letters τ and σ are used as syntactic variables ranging over types. The type constructor $\rightarrow$ associates to the right: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. 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). 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 constant or variable of type τ is a term of type τ. If t is a term of type $\tau \rightarrow \sigma$ and s is a term of type τ, then the application $(t\ s)$ is a term of type σ. Application associates to the left; that is, the expression $(t_1\ t_2\ t_3)$ is read as $((t_1\ t_2)\ t_3)$. Finally, if x is a variable of type τ and t is a term of type σ, then the abstraction $\lambda x\ t$ is a term of type $\tau \rightarrow \sigma$. If Σ is a signature and t is a closed term all of whose constants are members of Σ, then t is a Σ-*term*.

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

Terms are related to other terms by the following conversion rules.

- The term s α-converts to the term s' if s contains a subformula occurrence of the form $\lambda x\ t$ and s' arises from replacing that subformula occurrence with $\lambda y\ [x \mapsto y]t$, provided y is not free in t.

- The term s β-converts to the term s' if s contains a subformula occurrence of the form $(\lambda x\ t)t'$ and s' arises from replacing that subformula occurrence with $[x \mapsto t']t$.

- The term s η-converts to s' if s contains a subformula occurrence of the form $\lambda x\ (t\ x)$, where x is not free in t, and s' arises from replacing that subformula occurrence with t.

The binary relation of λ-*conversion* is defined so that t λ-converts to s if there is a list of terms $t_1, \ldots, t_n$, with $n \geq 1$, t equal to t_1, s equal to t_n, and for $i = 1, \ldots, n-1$, either t_i converts to t_{i+1} or t_{i+1} converts to t_i by α, β, or η. 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 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. The expression $\lambda\mathrm{norm}(t)$ denotes the λ-normal form of t. See [6] for a fuller discussion of these basic properties of the simply typed λ-calculus.

To define formulas, we shall now consider the following extension to terms. Let o be the type of propositions, where o is assumed not to be a member of S. A constant of type $\tau_1 \to \cdots \to \tau_n \to o$ will be used to denote predicates; that is, a predicate is denoted by a functional expression that takes its arguments to a proposition. The logical constants are given the following types: $\wedge, \vee, \supset$ are all of type $o \to o \to o$; and $\forall_\tau$ and $\exists_\tau$ are of type $(\tau \to o) \to o$, for all types τ. We shall rule out quantification over predicates by restricting the type τ in $\forall_\tau$ and $\exists_\tau$ not to contain the type symbol o. We shall assume that the logical constants are not members of any signature. A formula is a term of type o. The logical constants $\wedge, \vee, \supset$ are written in the familiar infix form. The expressions $\forall_\tau(\lambda z\ t)$ and $\exists_\tau(\lambda z\ t)$ are written simply as $\forall_\tau z\ t$ and $\exists_\tau z\ t$. A closed formula is a Σ-*formula* if all of its non-logical constants are members of Σ. The substitution operation and conversion relations on terms immediately extend to formulas.

4 Two Logic Programming Languages

4.1 Hereditary Harrop formulas: hh^ω

Our first logic programming language, called hh^ω, is based on two sets of closed, λ-normal formulas: $\mathcal{D}$, which can be used as program clauses, and $\mathcal{G}$, which can be used as goals or queries. The formulas in $\mathcal{D}$, denoted by the syntactic variable D, are those that do not have any positive occurrence of a disjunction or existential quantifier, while formulas in $\mathcal{G}$, denoted by the syntactic variable G, are their dual; that is, formulas in $\mathcal{G}$ cannot have any negative occurrence of a disjunction or existential quantifier.

In order to formalize a notion of backchaining over clauses of this general form, we need the following definition. Let $\mathcal{P}$ be a finite subset of $\mathcal{D}$. The set of pairs $|\mathcal{P}|_\Sigma$ is defined to be the smallest set such that

- if $D \in \mathcal{P}$ then $\langle \emptyset, D \rangle \in |\mathcal{P}|_\Sigma$,

- if $\langle \Gamma, D_1 \wedge D_2 \rangle \in |\mathcal{P}|_\Sigma$ then $\langle \Gamma, D_1 \rangle$ and $\langle \Gamma, D_2 \rangle$ are members of $|\mathcal{P}|_\Sigma$,

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

The following proposition has been used elsewhere to justify calling hh^ω a logic programming language [9, 12].

Proposition 1 *Let Σ be a signature, let $\mathcal{P}$ be a finite subset of $\mathcal{D}$, let $\{G_1, G_2, \exists_\tau x.G, \forall_\tau x.G\} \subseteq \mathcal{G}$, and let $D \in \mathcal{D}$. Then the following holds for intuitionistic provability $\vdash$. (When we write $\Sigma; \mathcal{P} \vdash G$ we assume that $\mathcal{P} \cup \{G\}$ is a set of Σ-formulas.)*

- $\Sigma; \mathcal{P} \vdash G_1 \wedge G_2$ *if and only if* $\Sigma; \mathcal{P} \vdash G_1$ *and* $\Sigma; \mathcal{P} \vdash G_2$.
- $\Sigma; \mathcal{P} \vdash G_1 \vee G_2$ *if and only if* $\Sigma; \mathcal{P} \vdash G_1$ *or* $\Sigma; \mathcal{P} \vdash G_2$.
- $\Sigma; \mathcal{P} \vdash \exists_\tau x\, G$ *if and only if there is a Σ-term t of type τ such that* $\Sigma; \mathcal{P} \vdash \lambda norm([x \mapsto t]G)$.
- $\Sigma; \mathcal{P} \vdash D \supset G_1$ *if and only if* $\Sigma; \mathcal{P} \cup \{D\} \vdash G_1$.
- $\Sigma; \mathcal{P} \vdash \forall_\tau x.G$ *if and only if* $\Sigma \cup \{c : \tau\}; \mathcal{P} \vdash [x \mapsto c]G$, *where c is a constant of type τ that is not in Σ.*
- *If A is atomic, then $\Sigma; \mathcal{P} \vdash A$ if and only if for some Γ, $\langle \Gamma, A \rangle \in |\mathcal{P}|_\Sigma$ and for every $G \in \Gamma$, $\Sigma; \mathcal{P} \vdash G$.*

This proposition in fact describes a non-deterministic interpreter for hh^ω. Moving from this proposition to an actual deterministic interpreter for hh^ω is a difficult task. Various aspects of implementing a language like hh^ω have been considered in [3, 13]. We mention a couple aspects in Section 7. In order to motivate introducing the next logic programming language, it is important to mention here that an interpreter for hh^ω will need to perform β-reductions while looking for proofs. That is, although programs and goals start out in λ-normal form, substitutions may cause them to become non-normal. Thus, references to the $\lambda\mathrm{norm}()$ function in Proposition 1 and in the definition of $|\mathcal{P}|_\Sigma$ are necessary in general. It is because β-conversion can cause significant changes to a term that unification in this setting is very hard. The next language we introduce 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 particularly simple.

4.2 The sublanguage: L_λ

A bound variable occurrence in a formula $G \in \mathcal{G}$ is *essentially universal* if it is bound by a positive occurrence of a universal quantifier or by a (term-level) λ-abstraction in G; it is *essentially existential* if it is bound by either a positive existential or a negative universal quantifier in G. Dually: a bound variable occurrence in a formula $D \in \mathcal{D}$ is *essentially existential* if it is bound by a positive occurrence of a universal or negative occurrence of an existential quantifier in D; it is *essentially universal* if it is bound by a negative universal quantifier or a (term-level) λ-abstraction in D. In the running of the non-deterministic interpreter described above, the essentially existential variables can get instantiated with general terms; it is via substitutions for these variables that new β-redexes can appear.

Our second logic programming language, called L_λ, is based on two sets of λ-normal formulas: $\mathcal{D}' \subseteq \mathcal{D}$, which can be used as program clauses, and $\mathcal{G}' \subseteq \mathcal{G}$, which can be used as goals or queries.

The restriction to determine the subsets intended is the following in both cases: whenever any formula in these sets has an essentially existential bound variable occurrence, say x, appearing as the head of an expression of the form $(x\ t_1\ \dots\ t_n)$ $(n \geq 1)$ then $t_1, \dots, t_n$ is a list of distinct variables that are essentially universally quantified within the scope of the binding for x.

For example, if predicate p has type $j \to o$ then the formula

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

is an example of both a goal and program clause for hh^ω; it is only a legal goal in L_λ. As a clause of L_λ, it has a subterm occurrence $(x\ y)$ where both x and y are essentially existential. Such a subterm is not premitted. All the formulas in Section 2 are in L_λ. Of course, first-order Horn clauses are both goals and clauses in L_λ.

Given this restriction to the syntax of program clauses and goals, the only β-redexes that must be computed from within an interpreter for L_λ are those of the form $(\lambda x.M)y$ where y is a bound variable that is not free in $\lambda x.M$. Such β-redexes are very simple to reduce: just change free occurrences of x in M to y. Given that α-conversion is available, this can be stated even more simply: a term t is related to s by *β_0-conversion* if one is gotten from the other by replacing a *β_0-redex* $(\lambda x.M)x$ in one with M in the other. If the interpreter for hh^ω is given a program and goal of the restricted language L_λ, the only β-redexes that need to be reduced are β_0-redexes.

It is proved in [10] that the unification problems that arise from writing an interpreter for L_λ, say $\beta_0\eta$-unification problems, are decidable and most general unifiers exists when unifiers exist. It is argued in that paper that the unification needed for L_λ is the weakest extensions to first-order unification that treats bound variables directly.

5 Specifying Equality and Substitution

Let t and s be two λ-normal terms of type σ. Define the following function by induction on the structure of simple types.

$$[\![t, s : \sigma]\!] = \begin{cases} \lambda\mathrm{norm}(copy_\sigma\ t\ s) & \text{if } \sigma \text{ is primitive} \\ \forall_{\sigma_1} x \forall_{\sigma_1} u([\![x, u : \sigma_1]\!] \supset [\![(t\ x), (s\ u) : \sigma_2]\!]) & \text{if } \sigma \text{ is } \sigma_1 \to \sigma_2. \end{cases}$$

(This recursive definition is similar to that used in [4] to code a dependent typed λ-calculus into hh^ω.) For example, the expression $[\![h, \lambda x.(g\ (x\ a)\ a) : (i \to j) \to i]\!]$ yields the formula

$$\forall_{i \to j} x \forall_{i \to j} u (\forall_i y \forall_i v [copy_i\ y\ v \supset copy_j\ (x\ y)\ (u\ v)] \supset copy_i\ (h\ x)\ (g\ (u\ a)\ a)).$$

Notice that the clauses given in Section 2 are exactly the clauses

$$[\![a, a : i]\!],\ [\![b, b : j]\!],\ [\![f, f : i \to j]\!],\ [\![g, g : j \to i \to i]\!],\ [\![h, h : (i \to j) \to i]\!]$$

It is an easy matter to show that such a formula is always both a goal and a clause for L_λ and that the formula $[\![t, s : \sigma]\!]$ is a Horn clause if and only if σ is of order 0 or 1. The following proposition is stated here without proof. Its proof is a straightforward induction on the structure of proofs (which mirrors the structure of $\beta\eta$-long normal forms [6]).

Proposition 2 *Let Σ contain at least the distinct constants $c_1 : \sigma_1, \dots, c_n : \sigma_n$ $(n \geq 0)$. Let $t_1, \dots, t_n$ be Σ-terms of type $\sigma_1, \dots, \sigma_n$, respectively, and let $\mathcal{C}$ be the set $\{[\![c_i, t_i : \sigma_i]\!] \mid i = 1, \dots, n\}$. Finally, let M and N be Σ-terms of type τ. Then $\Sigma; \mathcal{C} \vdash [\![M, N : \tau]\!]$ if and only if M is a $\{c_1, \dots, c_n\}$-term and $(\lambda c_1 \dots \lambda c_n.M)t_1 \dots t_n$ $\beta\eta$-converts to N.*

From this proposition, the following corollary follows immediately.

Corollary 1 *Let Σ be a signature and let $\mathcal{C}_\Sigma$ be the set $\{[\![c, c : \sigma]\!] \mid c : \sigma \in \Sigma\}$.*

- *If M and N are Σ-terms of type τ, then $\Sigma; \mathcal{C}_\Sigma \vdash [\![M, N : \tau]\!]$ if and only if M $\beta\eta$-converts to N.*
- *If M and N are Σ-terms of type $\sigma \to \tau$ and τ, respectively, then $\Sigma; \mathcal{C}_\Sigma \vdash \forall_\sigma x([\![x, t : \sigma]\!] \supset [\![M\ x, N : \tau]\!])$ if and only if $(M\ t)$ $\beta\eta$-converts to N.*

Thus, it is possible to use the formulas $[\![c, c : \sigma]\!]$ to help specify both equality (that is, $\beta\eta$-conversion) and substitution. To illustrate how substitution can be axiomatized, consider the following L_λ clause

$$\forall_i x(copy_i\ x\ T \supset copy_i\ (M\ x)\ S) \supset subst_{i \to i}\ M\ T\ S.$$

(Here, we shall start adopting the (familiar Prolog) convention that essentially existential variables will be capitalized letters and that if any variable is not explicitly quantified, it is assumed to be universally quantified or existentially quantified with outermost scope depending on whether or not the formula is intended to be a program clause or a goal.) The type of $subst_{i \to i}$ is $(i \to i) \to i \to i \to o$.

Assume for the moment that we have a Prolog-like interpreter for L_λ, and consider attempting to find a substitution term for the (essentially existential) variable F so that the goal $subst_{i \to i}\ F\ a\ (g\ b\ a)$ is provable from this clause and the clauses in Section 1. Backchaining would cause this goal to be reduced to

$$\forall_i x(copy_i\ x\ a \supset copy_i\ (F\ x)\ (g\ b\ a)).$$

This goal is then reduced by introducing a new constant, say $c : i$, and then adding the clause $copy_i\ c\ a$ to the program before attempting to prove the goal $copy_i\ (F\ c)\ (g\ b\ a)$. Notice that since c was introduced after the "logic" variable F was introduced, a correct interpreter for L_λ would need to make certain that F is not instantiated with a term that contains c. There is only one clause, namely $[\![g, g : j \to i \to i]\!]$, on which to backchain to prove this goal. Doing so reduces this goal to the two goals $copy_j\ F_1\ b$ and $copy_i\ F_2\ a$, where the disagreement pair $F\ c = g\ F_1\ F_2$ must still be solved. The first of these two goals has exactly one solution, namely $F_1 \mapsto b$, gotten by backchaining on $copy_j\ b\ b$. The second goal, however, can be proved two different ways: by backchaining over either $copy_i\ a\ a$, yielding $F_2 \mapsto a$ or $copy_i\ c\ a$ yielding $F_2 \mapsto c$. Putting these substitutions back together, we get two different solutions to the original goal: namely $F \mapsto \lambda w.(g\ b\ a)$ by solving the disagreement pair $F\ c = g\ b\ a$, and $F \mapsto \lambda w.(g\ b\ w)$ by solving the disagreement pair $F\ c = g\ b\ c$. (The possible solution $F \mapsto \lambda w.g\ b\ c$ is ruled out since c is not permitted to occur free in the substitution term of F.) Notice, that these two substitutions are exactly the two solutions to the $\beta\eta$-unification problem $\exists_{i \to i} F.\ F\ a = g\ b\ a$.

Substitution can be axiomatized in a general fashion by extending the example above. Assume that we have the predicates

$$subst_{\tau \to \sigma} : (\tau \to \sigma) \to \tau \to \sigma \to o$$

for each pair of types τ and σ. These predicates are then axiomatized by the following clause scheme:

$$\forall_\tau x([\![x, T : \tau]\!] \supset [\![(M\ x), S : \sigma]\!]) \supset subst_{\tau \to \sigma}\ M\ T\ S.$$

It follows immediately from the corollary above, that, when used in conjunction with the clauses $\{[\![c, c : \sigma]\!] \mid c : \sigma \in \Sigma\}$, these clauses prove $(subst_{\tau\to\sigma}\ M\ T\ S)$ if and only if $(M\ T)$ is $\beta\eta$-convertible to S. Now assume that M is of the form $\lambda x.M'$. The computation of the λ-normal form of $(M\ T)$ happens in two steps. First, T is substituted for free occurrences of x in M'. It is this step that the logical structure of the *subst* clause makes explicit. The second step requires any newly introduced β-redexes to be reduced. This step is not made explicit in the code above: if it needs to happen, the meta-level proof operation must perform those reductions. Thus, the *subst*-clauses cannot generally be L_λ program clauses. For example, the clause specifying substitution at type $(i \to j) \to i$ is

$$\forall_{i\to j} x(\forall_i Y \forall_i V(copy_i\ Y\ V \supset copy_j\ (x\ Y)\ (T\ V)) \supset copy_i\ (M\ x)\ S) \supset subst_{(i\to j)\to i}\ M\ T\ S.$$

This clause is not an L_λ program clause because the essentially existential variable T has an occurrence $(T\ V)$ where it is applied to another essentially existential variable. New β-redexes (which are not β_0-redexes) can be introduced at this point. In defining $subst_{\tau\to\sigma}$, if the type τ is primitive, then no new β-redexes will appear and the corresponding *subst* clause is, in fact, in L_λ (see the axiomatization of $subst_{i\to i}$ above).

It is possible to axiomatize *subst* completely in L_λ. Since it can be determined statically where β-reductions will need to be performed within the computation of a *subst* goal, it is possible to replace the β-redex with an explicit call to *subst*, this time at a lower type. In particular, if τ is functional, then $subst_{\tau\to\sigma}$ will need to call $subst_\tau$. For example, the following is an L_λ specification of $subst_{(i\to j)\to i}$:

$$\begin{aligned}\forall_{i\to j} x(\forall_i Y \forall_i V(copy_i\ Y\ V \supset \forall_j U[subst_{i\to j}\ T\ V\ U \supset copy_j\ (x\ Y)\ U])\\ \supset copy_i\ (M\ x)\ S) \supset subst_{(i\to j)\to i}\ M\ T\ S.\end{aligned}$$

Here, the positively occurring atom $copy_j\ (x\ Y)\ (T\ V)$ is replaced with

$$\forall_j U[subst_{i\to j}\ T\ V\ U \supset copy_j\ (x\ Y)\ U]:$$

the β-reduction needed to simplify $(T\ V)$ is made explicit by the call to $subst_{i\to j}$.

The two implementations of *subst* prove the same goals. In this fashion, we shall assume that the predicates $subst_{\tau\to\sigma}$ are all axiomatized completely in L_λ. The translation of a clause of hh^ω into a clause in L_λ given by this example can be generalized. We present a general translation in the next section.

6 Transforming hh^ω Goals into L_λ Goals

It is possible to systematically translate a goal in hh^ω into a goal in L_λ so that the proofs of the goal in hh^ω differ from the proofs in L_λ only in that additional *subst* and *copy* goals need to be established. Otherwise, all substitutions made in these proofs are identical. Since $\Sigma; \{D_1, \ldots, D_n\} \vdash G$ is equivalent to $\Sigma; \emptyset \vdash D_1 \supset \cdots \supset D_n \supset G$, it is enough to restrict this translation to goal formulas only: it dualizes immediately for program clauses.

For convenience, we axiomatize the predicates

$$subst^n_{\tau_1\to\cdots\to\tau_n\to\sigma} : (\tau_1 \to \cdots \to \tau_n \to \sigma) \to \tau_1 \to \cdots \to \tau_n \to \sigma \to o$$

to do an n-fold substitution in the following way:

$$\begin{aligned}\forall_{\tau_1} x_1([\![x_1, T_1 : \tau_1]\!] \supset \cdots \supset \forall_{\tau_n} x_n([\![x_n, T_n : \tau_n]\!] \supset [\![M x_1 \cdots x_n, S : \sigma]\!]) \ldots)\\ \supset subst^n\ M\ T_1 \ldots T_n\ S.\end{aligned}$$

As before, these clauses can be adjusted so that they are actually L_λ program clauses.

A subterm of a goal formula G will be called a non-L_λ subterm if it is of the form $(X\ t_1 \cdots t_n)$ where $n \geq 1$, X is essentially existential in G, and the terms $t_1, \ldots, t_n$ do not satisfy the restrictions defining L_λ. The translation from hh^ω goals to L_λ goals is by induction on the number of non-L_λ subterm occurrences. Let $G \in \mathcal{G}$. If G is not an L_λ goal, then there is an occurrence of an atomic formula A in G which has a subterm of the form $(X\ t_1 \cdots t_n)$ where n, X and $t_1, \ldots, t_n$ are as above. Let $y_1, \ldots, y_m$ $(m \geq 0)$ be the list of essentially universal variables that are bound in the scope of X's binding occurrence and that also contain $(X\ t_1 \cdots t_n)$ in their scope. Let $\sigma_1, \ldots, \sigma_m$ be the types of $y_1, \ldots, y_m$, respectively. Let H be a variable not occurring free in A and let A' be the result of replacing the occurrence of $(X\ t_1 \cdots t_n)$ with $(H\ y_1 \ldots y_m)$. Let G' be the result of replacing A with either the expression

$$\forall H([\forall_{\sigma_1} y_1([\![y_1, y_1 : \sigma_1]\!] \supset \cdots \supset \forall_{\sigma_m} y_m([\![y_m, y_m : \sigma_m]\!] \supset \\ subst^n\ X\ t_1 \ldots t_n\ (H\ y_1 \ldots y_m)) \cdots)] \supset A')$$

if A occurs negatively in G, or

$$\exists H([\forall_{\sigma_1} y_1([\![y_1, y_1 : \sigma_1]\!] \supset \cdots \supset \forall_{\sigma_m} y_m([\![y_m, y_m : \sigma_m]\!] \supset \\ subst^n\ X\ t_1 \cdots t_n\ (H\ y_1 \ldots y_m)) \ldots)] \wedge A')$$

if A occurs positively in G. The resulting formula now has one fewer non-L_λ subterms. If we repeat this process until all such subterms are removed, the result will be an L_λ goal formula. The following proposition establishes a simple correctness property for this translation.

Proposition 3 *Let Σ be a signature, let $\mathcal{C}_\Sigma$ be the set $\{[\![c, c : \sigma]\!] \mid c : \sigma \in \Sigma\}$, and let G be an hh^ω goal formula that does not contain occurrences of any* copy *or* subst *predicates. Let G'' be the L_λ goal that results from performing the above mentioned translations to the hh^ω goal formula G. Then*

$$\Sigma; \emptyset \vdash G \quad \textit{if and only if} \quad \Sigma; \mathcal{C}_\Sigma \vdash G''.$$

Furthermore, if G is of the form $\exists_\tau X.H$ then G'' is of the form $\exists_\tau X.H''$, and for all Σ-terms t of type τ,

$$\Sigma; \emptyset \vdash \lambda norm([X \mapsto t]H) \quad \textit{if and only if} \quad \Sigma; \mathcal{C}_\Sigma \vdash \lambda norm([X \mapsto t]H'').$$

In other words, answers substitutions are the same between these two goals.

Consider the $\beta\eta$-unification problem

$$\exists_{i\to i} X[k(\lambda v(m\ (X\ v))) = k(\lambda y(X\ (m\ y)))],$$

over the signature $\{m : i \to i,\ k : (i \to i) \to i\}$. This can be written as the hh^ω goal

$$\forall_i Z(eq\ Z\ Z) \supset \exists_{i\to i} X[eq\ (k(\lambda v(m\ (X\ v))))\ (k(\lambda y(X\ (m\ y))))].$$

This query has one non-L_λ subterm, namely, $(X\ (m\ y))$. Using the above mentioned transformation leads to the L_λ goal

$$\forall_i Z(eq\ Z\ Z) \supset \exists_{i\to i} X \exists_{i\to i} H[\forall_i y(copy_i\ y\ y \supset subst_{i\to i}\ X\ (m\ y)\ (H\ y)) \wedge \\ eq\ (k(\lambda v(m\ (X\ v))))\ (k(\lambda y(H\ y)))].$$

Solving this query using the *copy*-clauses for the constants m and k, we find that there are an infinite number of proofs, yielding a sequence of substitution terms for X, namely, $\lambda w.w$, $\lambda w.(f\ w)$, $\lambda w.(f(f\ w))$, etc. These are thus the unifiers for the original $\beta\eta$-unification problem.

7 Some Implementation Considerations

The presentation of $\beta\eta$-unification given so far is rather simple and declarative. This is partly due to pushing lots of details concerning bound variables and search into an imaginary interpreter for L_λ, where, presumably, all these details must be carefully addressed. The eLP implementation of λProlog [2] provides an interpreter for both hh^ω and L_λ. Many details regarding how such an interperter can be built are given in the paper [3].

There are only two special implementation considerations that we would like to address here. Assume that we are designing an interpreter that uses logic variables and unification in the usual way to postpone determining substitution terms for essentially existential variables. Let t be a λ-normal term of primitive type. We say t is *flexible* if its head symbol is a logic variable; otherwise, it is *rigid*. If σ is a primitive type, a goal of the form $copy_\sigma\ t\ s$ is classified as a rigid-rigid, flexible-rigid, rigid-flexible, or flexible-flexible *copy*-goal depending on the status of the two terms t and s. Given that *copy* is axiomatized only in the forms used in this paper, that is, as axiomatizations of equality and substitution, then we can conclude the following behavior for proving *copy*-goals. A rigid-rigid or rigid-flexible *copy*-goal can be used to backchain over at most one clause. These are therefore "deterministic" goals. A flexible-rigid goal may have several clauses to backchain over. Consider the case where the top-level constant of s is, say, $f : i \rightarrow i$. Thus, it will be possible to backchain using the clause $[\![f, f : i \rightarrow i]\!]$. There may have also been extensions to the program via *subst*-goals in which clauses of the form $copy_i\ x\ (f\ u)$, where x is some "new" constant. It is then possible to backchain over these additional clauses. Thus, flexible-rigid *copy*-goals give rise to the searching that goes on in $\beta\eta$-unification: it is this non-determinancy that is reflected in the MATCH procedure of [7]. A sensible interpretation of such a goal is to order these backchaining choices and to try one after the other using some search discipline (such as depth-first search).

Consider proving a flexible-flexible goal $copy_\sigma\ t\ s$. Every $copy_\sigma$-clause in the program could be used to backchain on this goal: there may be a large number of such clauses. Also, backchaining over a clause encoding a constant of non-primitive type will generate more flexible-flexible *copy*-goals and these may also be used to backchain over a large number of clauses. In certain cases, say when the only clauses for $copy_\sigma$ are for constants that are of primitive type or when there are, in fact, no clauses for $copy_\sigma$ (σ is an empty type), proving such flexible-flexible *copy*-goals may not lead to an explosion in the search space. In general, however, a sensible approach to proving flexible-flexible goals would be to suspend them (using delay mechanisms such as in NU-Prolog [17]) and attempt to prove other goals in the hope that the flexible heads will be instantiated to make them rigid, at which point they could be resumed. Suspending such flexible-flexible *copy*-goals is similar to advice given in [7] and to the treatment of $\beta\eta$-unification in λProlog [13] and its current implementations [2, 3].

As a final comment on an implementation, consider the occurrence-check in the usual first-order unification algorithm. This does not generalize directly when variables of higher-order type are present. For example, the unification problem $\exists_i X \exists_{i\rightarrow i} F.\ X = F\ X$ has answer substitutions as long as there are terms of type i. For example, if a is of type i, then set F to $\lambda w.w$ or to $\lambda w.a$ and set X to a. In our setting, however, it is possible to generalize the occurrence-check by a check on ancestor goals. That is, if the current *copy*-goal is subsumed by an ancestor goal, then it is possible to fail the current goal without loss of completeness. In the first order setting where the clauses $[\![t, s : \sigma]\!]$ are just Horn clauses, this is equivalent to the occurrence-check. In the general case, however, the ancestor check is still a legitimate step although a simple occurrence-check is not. Thus, a reasonable implementation of L_λ when employed to deal with $\beta\eta$-unification might well have such an ancestor check to stop this simple kind of infinite branch. For more explicit information on generalizations of the occurrence-check to $\beta\eta$-unification, see [7, 11].

8 Conclusion

We have presented a specification of $\beta\eta$-unification using the logic programming language L_λ. This specification approach simplifies the presentation of an implementation of $\beta\eta$-unification by allowing us to focus on the simple declarative aspects of equality and substitution in isolation from details of search and the complex, low-level syntax of λ-abstractions. These latter details are addressed by an implementation of L_λ. Fortunately, a great many techniques and ideas from the implementation of various other logic programming languages can be used to build such an implementation. Of course, L_λ is of greater interest than as just the basis for implementing $\beta\eta$-unification. The translation given in Section 6 shows that many λProlog programs can be translated to L_λ programs in a very direct fashion. Thus, an L_λ interpreter could be used as the core of a λProlog interpreter.

9 Acknowledgements

I am grateful to Bob Constable, Fernando Pereira, and the conference reviewers for their comments and suggestions on an earlier draft of this paper. At the University of Edinburgh, this work has 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." At the University of Pennsylvania, from where the author is on a one year leave, this work has been supported by ONR N00014-88-K-0633 and NSF CCR-87-05596.

References

[1] Peter B. Andrews, Eve Longini Cohen, Dale Miller, and Frank Pfenning. Automating higher order logic. In *Automated Theorem Proving: After 25 Years*, pages 169–192. American Mathematical Society, 1984.

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

[3] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*. MIT Press, 1990. (in press).

[4] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark Stickel, editor, *Proceedings of the 1990 Conference on Automated Deduction*, volume 449, pages 221–235. Springer Lecture Notes in Artificial Intelligence, 1990.

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

[6] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.

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

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

[9] Dale Miller. Abstractions in logic programming. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329 – 359. Academic Press, 1990.

[10] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*. Springer Lecture Notes in Artificial Intelligence, 1990.

[11] Dale Miller. Unification under a mixed prefix. To appear in the *Journal of Symbolic Computation*.

[12] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. To appear in the *Annals of Pure and Applied Logic*.

[13] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[14] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361 – 386. Academic Press, 1990.

[15] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.

[16] Wayne Snyder and Jean H. Gallier. Higher order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1–2):101–140, 1989.

[17] James Thom and Justin Zobel. NU-Prolog reference manual, version 1.3. Technical report, Department of Computer Science, University of Melbourne, Australia, 1988.