# Lambda Unification

Michael Beeson[*]

August 8, 2006

**Abstract**

We give a formal description of an algorithm for *lambda unification*. The input to the algorithm consists of two terms $t$ and $s$ of lambda logic. The purpose of the algorithm is to find (sometimes) a substitution $\sigma$ such that $t\sigma = s\sigma$ is provable in lambda logic. In general such unifiers are not unique. Lambda logic itself is defined in [2]. We prove some basic metatheorems about lambda unification, and compare it to the previously known notion of "higher order unification." Examples of the use of lambda unification to find proofs by mathematical induction, using its implementation in the theorem prover Otter-lambda, are given in [4].

## Introduction

Lambda logic is the logical system one obtains by adding lambda calculus to first order logic. This system was formulated, and some fundamental metatheorems were proved, in an earlier publication [3]. The appropriate generalization of unification to lambda logic is this notion: two terms are said to be *lambda unified* by substitution $\sigma$ if $t\sigma = s\sigma$ is provable in lambda logic. More generally, a multiset $S$ of equations is lambda unified by $\sigma$ if each of its member equations is lambda unified by $\sigma$. *Lambda unification* is an algorithm for producing lambda unifying substitutions. In Otter-$\lambda$, lambda unification is used, instead of only first-order unification, in the inference rules of resolution, factoring, paramodulation, and demodulation.

In computer files meant for use with implementations, we write $lambda(x, t)$ for $\lambda x.\, t$, and we write $Ap(t, s)$ for $t$ applied to $s$, which is often abbreviated in technical papers to $t(s)$ or even $ts$. In this paper, $Ap$ will always be written explicitly, to avoid confusion between $Ap(f, x)$ and $f(x)$, which are different in lambda logic. In [4], we also wrote out *lambda* to make it easy to compare computer-produced proofs and algorithm definitions. Since this is primarily a theoretical paper, we use the notation $\lambda x.\, t$ instead of writing *lambda* out explicitly, except when quoting from the algorithmic definition of lambda unification given in [4].

[*]Research supported by NSF grant number CCR-0204362.

As we define it here, lambda unification is a non-deterministic algorithm: it can return, in general, many different unifying substitutions for two given input terms. The input to the lambda-unification algorithm, like the input to ordinary unification, is two terms $t$ and $s$ (this time terms of lambda logic). The output, if the algorithm succeeds, is a substitution $\sigma$ such that $t\sigma = s\sigma$ is provable in lambda logic.

Although the lambda unification algorithm has been described in [3], [2], and [4], this paper is intended to supply a more formal definition of the algorithm, in the style used by researchers in unification theory [1], [7], and to prove some basic theorems about the algorithm. For examples of the use of lambda unification, see [4] and the Otter-lambda website [6].

## 1 Definition of a lambda unification algorithm

This section repeats the definition of the algorithm as given in [4]. We first give the relatively simple clauses in the definition. These have to do with first-order unification, alpha-conversion, and beta-reduction. The rule related to first-order unification just says that we try that first; for example $Ap(x, y)$ unifies with $Ap(a, b)$ directly in a first-order way. However, the usual recursive calls in first-order unification now become recursive calls to lambda unification. In other words: to unify $f(t_1, \ldots, t_n)$ with $g(s_1, \ldots, s_m)$ (according to this rule) we must have $f = g$ and $n = m$; in that case we do the following:

```
$\sigma$ =identity substitution;
for $i = 1$ to $n$ {
    $\tau$ = unify($t_i, s_i$);
    if ($\tau$ = failure)
        return failure;
    $\sigma = \sigma \circ \tau$; }
return $\sigma$;
```

Here the call to `unify` is a recursive call to the algorithm being defined. Since the algorithm is non-deterministic, there are choices to be made for each argument. For example, if there are two substitutions $\sigma_i$ that unify $a$ and $c$, and two ways to unify $b\sigma_i$ and $d\sigma_i$, then there will be four ways to unify $f(a, b)$ with $f(c, d)$.

To unify a variable $x$ with a term $t$, return the substitution $x := t$ if $t$ is identical to $x$ or $x$ is not bound and $x$ does not occur in $t$.

The rule related to alpha-conversion says that, if we want to unify $lambda(z, t)$ with $lambda(x, s)$, first rename bound variables if necessary to ensure that $x$ does not occur in $t$ and $z$ does not occur in $s$. Then let $\tau$ be the substitution $z := x$ and unify $t\tau$ with $s$, rejecting any substitution that assigns a value depending on $x$ or a value to $x$.[1] If this unification succeeds with substitution $\sigma$, return $\sigma$.

[1]Care is called for in this clause, as illustrated by the following example: Unify $lambda(x, y)$ with $lambda(x, f(x))$. The "solution" $y = f(x)$ is wrong, since substituting $y = f(x)$ in $lambda(x, y)$ gives $lambda(z, f(x))$, because the bound variable is renamed to avoid capture.

The rule related to beta-reduction says that, to unify $Ap(lambda(z,s),q)$ with $t$, we first beta-reduce and then unify. That is, we unify $s[z := q]$ with $t$ and return the result.

Lambda unification's most interesting instructions tell how to unify $Ap(x,w)$ with a term $t$, where $t$ may contain the variable $x$, and $t$ does not have main symbol $Ap$. Note that the occurs check of first-order unification does not apply in this case. The term $w$, however, is not allowed to contain $x$. In this case lambda unification is given by the following non-deterministic algorithm:

1. Pick a *masking subterm* $q$ of $t$. That means a subterm $q$ such that every occurrence of $x$ in $t$ is contained in some occurrence of $q$ in $t$. (So $q$ "masks" the occurrences of $x$; if there are no occurrences of $x$ in $t$, then $q$ can be any subterm of $t$, but see the next step.)

2. Call lambda unification to unify $w$ with $q$. Let $\sigma$ be the resulting substitution. If this unification fails, or assigns any value other than a variable to $x$, return failure. If it assigns a variable to $x$, say $x := y$ reverse the assignment to $y := x$ so that $x$ remains unassigned.

3. If $q\sigma$ occurs more than once in $t\sigma$, then pick a set $S$ of its occurrences. If $q$ contains $x$ then $S$ must be the set of *all* occurrences of $q\sigma$ in $t$. Let $z$ be a fresh variable and let $r$ be the result of substituting $z$ in $t\sigma$ for each occurrence of $q\sigma$ in the set $S$.

4. Append the substitution $x := \lambda z.\, r$ to $\sigma$ and return the result.

There are two sources of non-determinism in the above, namely in steps 1 and 3. Otter-$\lambda$ has a parameter `max_unifiers`, that can be set in the input file by a command like `assign(max_unifiers,9)`. In that case, lambda unification will backtrack over different selections of a masking subterm and set $S$, up to the maximum number of unifiers specified (per lambda unification). The default value of this parameter is one, in which case there is no backtracking, i.e. a deterministic selection is made. Even if backtracking is allowed, Otter-$\lambda$ still attempts to pick "good" masking subterms according to some heuristics. Here are some of the heuristics used: in step 1, if $x$ occurs in $t$, we prefer the smallest masking subterm $q$ that occurs as a second argument of $Ap$.[2] If $x$ occurs in $t$, but no masking subterm occurs as a second argument of $Ap$, we prefer the smallest masking subterm[3] If $x$ does not occur in $t$, we pick a constant that occurs in $t$, or more generally a constant subterm of $t$; if there is none, we fail. Which constant subterm we pick is determined by some heuristics that seem to work well in the examples we have tried. In step 3, if $q$ does not contain $x$, then an important application of this choice is to proofs by mathematical induction, where the choice of $q$ corresponds to choosing a constant $n$, replacing some of the occurrences of $n$ by a variable, and deciding to prove the theorem by induction on that variable. Therefore the choice of $S$ is determined by heuristics that prove

[2]The point of this choice is that, if we want the proof to be implicitly typable, then $q$ should be chosen to have the same type as $w$, and $w$ is a second argument of $Ap$.

[3]This will not be done if the input file contains `set(types)`, because it might result in mis-typings; unless, of course, the input file also provides a `list(types)` that can be used to check the type of the masking subterm.

useful in this case. In particular, when proving equations by induction, we pick a constant that occurs on both sides of the equation, but not necessarily when proving non-equations. If there is a constant term of weight 1 that occurs on both sides of the equation, that term is used instead of a constant—this allows Otter-$\lambda$ to "generalize" a goal, and since weight templates can be specified in the input file, it also gives the user some control over what terms can be selected as masking subterms. Our present heuristics call for never choosing a term of weight greater than 1; but weights can be set by the user in the input file, if it should be necessary.

Finally, lambda unification needs some rules for unifying $Ap(r, w)$ with $t$, when $r$ is not a variable. The rule is this: create a fresh variable $X$, unify $Ap(X, w)$ with $t$ generating substitution $\sigma$, then unify $X\sigma$ with $r\sigma$, generating substitution $\tau$; if this succeeds return $\sigma\tau$, or rather, the substitution that agrees with $\sigma\tau$ but is not defined on $X$, since $X$ does not occur in the original unification problem.

*Example.* Unify $Ap(Ap(x, y), z)$ with 3. Choose fresh $X$, unify $Ap(X, z)$ with 3, getting $z := 3$ and $X := lambda(u, u)$. Now unify $lambda(u, u)$ with $Ap(x, y)$, getting $y := lambda(u, u)$ and $x := lambda(v, v)$. So the final answer is $x := lambda(v, v)$, $y := lambda(u, u)$, $z := 3$. We can check that this really is a correct lambda unifier as follows:

$$\begin{aligned} Ap(Ap(x, y), z) &= Ap(Ap(lambda(u, u), lambda(v, v)), 3) \\ &= Ap(lambda(v, v), 3) \\ &= 3. \end{aligned}$$

Note that failure of the occurs check in first order unification does not cause failure of lambda unification. For example, we can unify $x = f(x)$ by taking $x\sigma$to be $Ap(\omega, \omega)$ where $\omega = \lambda x.\, f(Ap(x, x))$. On this example, the occurs check fails, but lambda unification does not fail, because after the occurs check fails, we go on to try the other rules.

## An inference system for lambda unification

The standard way of proving theorems about unification algorithms begins with replacing an algorithmic definition of unification by a system of inference rules. Experts have urged me to present lambda unification in that style, and I hereby oblige.

There are two standard references for inference systems for unification: [1] for first order unification and [7] for higher-order unification. These two references differ in their setup, both notationally and in substance. In addition, an inference system for lambda unification needs one more feature not included in either cited reference. We therefore carefully discuss the general characteristics of these inference systems before giving the specific system for lambda unification.

The premises and conclusions of the rules in both cited references are either the special sign $\perp$ (read *fail*), or multisets of equations $t = s$, where $t$ and $s$ are terms of lambda logic. In case $t$ is a variable that does not occur free in $s$ then the equation $t = s$ is said to be *solved*.

The differences between the two cited references are as follows. First there is the trivial difference that [1] uses $\Rightarrow$ while [7] uses a horizontal line (as we shall do). Second, [1] uses *two* multisets of equations instead of just one, distinguishing the "solved" equations from the rest. We shall use just one multiset, defining an equation to be *solved* if it has the form $x = t$, where $x$ is a variable and $x$ does not occur free in $t$. Instead, [7] applies substitution implicitly, so solved equations become identities, which he calls "trivial". Thus, the goal in the system of [7] is to derive a trivial system, while [1] tries to derive a solved system. It is in that sense that we follow [1] more closely than [7], though superficially our notation looks more like [7]. The reason why [7] does it this way is that his system is only meant to establish *unifiability* rather than actually return a substitution in every case.

The additional feature that we will need is some way to express the concept "$x$ is forbidden to $y$", which means that variable $y$ cannot be assigned a value depending on $x$. In the implementation, each variable is associated with a (dynamic) list of variables forbidden to it. In an inference system, we will represent this by making the premises and conclusions of the rules pairs. One member is a multiset $S$ of equations, as in the cited references. The other member is an "environment", which specifies which variables are forbidden to which other variables. The relation "$x$ is forbidden to $y$ in $E$" is transitive: if $x$ is forbidden to $y$ and $y$ is forbidden to $z$ then $x$ is forbidden to $z$. Whether this transitivity is explicitly represented in $E$ or not is an implementation issue. Since this is a theoretical paper, for definiteness we may suppose that an environment $E$ is (represented by) a matrix with a boolean entry for each pair of variables occurring free in $S$; each non-false entry represents the "restriction" $F[x, y]$, meaning "$x$ is forbidden to $y$". Since what we want to indicate in the rules is a *change* in the relevant environment, we use the notation $F[x, y], E$ to indicate the new environment resulting from $E$ by adding the restriction $F[x, y]$ to the environment $E$ and then forming the transitive closure. We use a colon to separate a multiset of equations from the relevant environment, so the premises and conclusions of our rules will have the form $S : E$.

We have defined an equation $x = t$ to be "solved" if $x$ does not occur free in $t$, and as already explained, the goal should be to derive a multiset of solved equations; but now that our conclusions also include an environment, the goal should be to derive a multiset of solved equations compatible with the (restrictions in the) environment. We say that a solved equation $x = t$ is *compatible with $E$"* (where $E$ is an environment) if $E$ does not forbid $x$ to $y$ for any variable $y$ occurring free in $t$.

Since only one rule changes the environment, most of the rules will have the same $E$ above and below the inference line. In that case there is no point in writing it, so we omit it, but it is officially there. In the following system, we follow the notation of [7] (a horizontal "inference line") but we follow [7] more closely in substance, retaining solved equations so that the goal is to derive a multiset consisting only of solved equations, which thus determines a substitution. When writing multisets in these rules, we omit the set brackets and the union symbol, so that for example $\{t = s\} \cup \Gamma$ becomes $t = s, \Gamma$. In these rules, $x$ and $z$ stand for variables, and $t$, $s$, and $q$ stand for any terms. To say that a variable occurring in the conclusion of a rule is "fresh" means that it does not occur in the premises of the rule.

First we give the inference rules corresponding to first order unification. We give these rules the same names as in [1].

$$\frac{\bot, \Gamma}{\bot} \quad \textbf{Failure}$$

$$\frac{t = t, \Gamma}{\Gamma} \quad \textbf{Trivial}$$

$$\frac{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n), \Gamma}{t_1 = s_1, \ldots, s_1 = s_n, \Gamma} \quad \textbf{Decomposition}$$

Technically, in lamba logic constants are not regarded as just 0-ary function symbols, so the previous two rules do not apply to constants. We therefore need two rules for constants:

$$\frac{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_n), \Gamma}{\bot} \quad \textbf{Symbol Clash} \text{ if } f \neq g \text{ and } f \neq Ap \text{ and } g \neq Ap$$

$$\frac{a = b, \Gamma}{\bot} \quad \textbf{Different Constants} \qquad \text{if } a \text{ and } b \text{ are distinct constants}$$

$$\frac{a = a, \Gamma}{\Gamma} \quad \textbf{Constant} \qquad \text{if } a \text{ is a constant}$$

$$\frac{t = x, \Gamma}{x = t, \Gamma} \quad \textbf{Orient}, \text{ if } t \text{ is not a variable}$$

The conditions for the following rule are
(i) $x$ is not free in $t$
(ii) $x$ occurs free in $\Gamma$
(iii) $E$ does not contain $F[x, y]$ for any variable $y$ occurring free in $t$. (When doing only first-order unification, $E$ is always empty, but when these rules are used as part of lambda unification, this condition prevents useless derivations.)

$$\frac{x = t, \Gamma : E}{x = t, \Gamma[x := t] : E} \quad \textbf{Variable Elimination}$$

That completes the inference rules for first order unification. Note that the occurs check has been omitted as a inference rule, since if it is included in an inference system it causes failure, while (as discussed above)in lambda unification, we go on to try other rules. Instead, the occurs check appears as a condition in the substitution rule.

Now we come to the rules for lambda unification.

$$\frac{\lambda x.\, t = s, \Gamma}{\lambda z.\ t[x := z] = s, \Gamma} \quad \textbf{Alpha} \quad \text{if } z \text{ is not free in } t$$

$$\frac{s = \lambda x.\, t, \Gamma}{s = \lambda z.\ t[x := z], \Gamma} \quad \textbf{Alpha} \quad \text{if } z \text{ is not free in } t$$

The following rule is the one that changes the environment. Therefore, the environment is explicitly shown. See the discusssion above. The conditions for the following rule are

(i) $x$ does not occur free in $\Gamma$
(ii) $y_1, \ldots, y_n$ is a complete list of all variables free in $t$, $s$, or $\Gamma$.

$$\frac{\lambda x.\, t = \lambda x.\, s, \Gamma : E}{Ap(t, x) = Ap(s, x), \Gamma : F[x, y_1], \ldots, F[x, y_n], E} \quad \textbf{Lambda}$$

Under the same conditions we could also consider the following rule, which we do not, however, include in this inference system:

$$\frac{\lambda x.\, t = \lambda x.\, s, \Gamma : E}{t = s, \Gamma : F[x, y_1], \ldots, F[x, y_n], E} \quad \textbf{Weak Lambda}$$

It is correct to call this "**Weak Lambda**" rather than "**Strong Lambda**", because the conclusion of this rule implies the conclusion of the **Lambda** rule, and the hypotheses are the same.

The environment is used to block certain unifications. That is expressed in the following rule:

$$\frac{x = t : E}{\bot} \quad \textbf{Forbidden} \quad \text{if for some variable } y \text{ free in } t,\ E \text{ contains } F[x, y]$$

$$\frac{Ap(\lambda x.\, t, q) = s, \Gamma}{t[x := q] = s, \Gamma} \quad \textbf{Beta}$$

The conditions for the following rule are:
(i) $q$ contains every free occurrence of $x$ in $t$; more precisely, $t[q := z]$ does not contain $x$ free. (There could be more than one occurrence of $q$, and all of them together contain every free occurrence of $x$.)
(ii) $t$ has at least one free occurrence of $x$
(iii) None of the occurrences of the free variables of $q$ is bound in $t$.
(iv) $z$ is fresh, i.e. does not occur in the premises.

$$\frac{Ap(x, r) = t, \Gamma}{r = q, Ap(x, z) = t[q := z], z = q, \Gamma} \quad \textbf{Masking Term}$$

The conditions for the following rule are
(i) $x$ occurs on the left of a solved equation in $\Gamma$.
(ii) $y$ does not occur on the left of any solved equation in $\Gamma$.

$$\frac{Ap(x, q) = t, x = y, \Gamma}{Ap(y, q) = t, x = y, \Gamma} \quad \textbf{Switch}$$

The conditions for the following rule are:
(i) $x$ does not occur free in $t$
(ii) $x$ does not occur on the left of any equation in $\Gamma$
(iii) $z$ does not occur in $t$ or $\Gamma$
(iv) $q$ is any subterm of $t$, none of whose free variables is bound in $t$.[4]
(v) $t'$ is the result of substituting $z$ for some (but not necessarily all) occurrences of $q$ as a subterm of $t$.

$$\frac{Ap(x,r) = t, \Gamma}{r = q, Ap(x,z) = t', z = q, \Gamma} \qquad \textbf{Substitution}$$

In case $r$ is the variable $z$ then one of the two identical equations $r = q$ and $z = q$ can be omitted in the conclusion. In case $r$, $q$, and $z$ are identical, then both of them can be omitted.

$$\frac{Ap(x,z) = t, \Gamma}{x = \lambda z.\, t, \Gamma} \qquad \textbf{Function Definition}$$

The following rule gets its name from the example given at the end of the last section–it permits unification to deal with "Curried" functions of several variables.

$$\frac{Ap(r,s) = t, \Gamma}{z = r, Ap(z,s) = t, \Gamma} \qquad \textbf{Curry} \qquad \text{if } r \text{ is not a variable and } z \text{ is fresh}$$

## 2 Relation of the algorithm and inference system

The inference system given above permits the arbitrary renaming of bound variables. The algorithm renames bound variables only as required. If we use the inference system to define a (non-deterministic) algorithm in the style of [1] and [7], it will permit non-deterministic renamings of bound variables; so it can be thought of as an algorithm on terms modulo alpha-equivalence. We say that an inference rule applies modulo ($\alpha$) if it applies after a renaming of bound variables.

The algorithm can be recovered from the inference system as follows: Begin with multiset $S$ containing just one equation, representing the input to the algorithm. Repeat the following step as long as possible: find the leftmost element of $S$ such that some inference rule (other than an **Alpha** rule) applies with that element of $S$ as the principal premise (the one explicitly shown in the list of inference rules). Apply the first such rule (if necessary first applying

[4]We could require only the slightly weaker condition that none of the occurrences of the free variables of $q$ be within the scope of a $\lambda$-binding in $t$. The condition as stated also rules out the harmless case that elsewhere in $t$ there occurs a bound occurrence of a variable that occurs free in $q$, but $q$ isn't in the scope of that binding. It is harmless to rule that case out too since the bound variable could be renamed and then the stated condition would be satisfied; and the condition as stated simplifies the proofs below.

**Alpha** rules until the rule is literally applicable), using the rest of $S$ as $\Gamma$, and replace $S$ with the resulting conclusion of the rule. This iteration continues until no rule (other than an **Alpha** rule) applies. The algorithm is nondeterministic, because of the selection of the masking term $q$ in rules **Masking Term** and **Substitution** and the selection of the subset of occurrences of $q$ in condition (v) of rule **Substitution**.

*Remark.* As implemented in the prover Otter-$\lambda$, the algorithm does not actually backtrack over the selection of term $q$ in rule **Masking Term**; it makes only one selection, preferring a term $q$ occurring as a second argument of $Ap$. However, provided the input file contains suitable commands, it *will* backtrack over the selection of term $q$ in rule **Substitution** and the selection of the subset of occurrences of $q$ in condition (v) of rule **Substitution**. It is this backtracking (over different subsets of occurrences) that is needed to try different possible choices of the induction variable in proofs by mathematical induction. Backtracking over different choices of $q$ is needed to allow the prover to generalize the theorem before attacking it by induction. We purposely limited the selection to terms $q$ of weight 1 or 2 only in the **Substitution** rule, since that works for examples of interest and cuts down on the number of unifiers. In this note, however, we are discussing the full (theoretical, non-deterministic) algorithm without these limitations.

*Remark.* To recover the algorithm published in [4] and implemented in Otter-lambda, we need only the **Weak Lambda** rule. The algorithm defined using the **Lambda Rule** may find more unifications; evidently those unifications were never needed in the examples in [4]. Of course as explained in the previous remark, there are other respects as well in which the implemented algorithm purposely does not search all the possibilities allowed by the inference system.

## 3 Correctness of Lambda Unification

The main lemma about the inference system for first order unification is that the rules do not change the set of unifiers ([1], p. 457). We first investigate whether the same is true for the inference system given above for lambda unification. Some adjustments to the statement are necessary because we now have environments to account for in the inference system.

**Definition 1** *If $E$ is an environment, we say that substitution $\sigma$ is* compatible *with $E$ if and only if, for each variable $x$ in the domain of $\sigma$, it is not the case that $E$ contains $F(x, y)$ for any variable $y$ occurring free in $x\sigma$.*

Our first lemma shows that one direction works, in the following sense.

**Lemma 1** *For any rule in the inference system for lambda unification, a substitution $\theta$ that unifies the equations in the conclusion, compatible with the conclusion's environment, also unifies the equations in the premise, and is consistent with the premise's environment, provided no variables in the domain of $\theta$ are bound in the premises.*

*Proof.* We go through the rules one by one. In the proof we will use the completeness and soundness of lambda logic [2]. We will show that for each rule, and any substitution $\sigma$, and any model $M$ of lambda logic, if $M$ satisfies $C\sigma$, where $C$ is the multiset of equations in the conclusion of the rule, and $\sigma$ is consistent with the environment $E$ of the conclusion, then $M$ satisfies $P\sigma$, where $P$ is the multiset of equations in the premise of the rule, and $\sigma$ is consistent with the environment of the premise. This implies the lemma as follows: Suppose $C\sigma$ is provable in lambda logic, but $P\sigma$ is not. Then by completeness, there is a model $M$ where $P\sigma$ is not satisfied. Hence $C\sigma$ is not satisfied; but this is a contradiction, since $C\sigma$ is provable and $M$ is a model of lambda logic.

For the **Symbol Clash** rule, the conclusion cannot be unified, so we have to show that lambda logic cannot prove $f(t_1,\ldots,t_n)\sigma = g(t_1,\ldots,t_n)\sigma$ unless $f$ and $g$ are the same function symbol. Take a model in which $f$ and $g$ are interpreted as constant functions with different values. For the **Different Constants** rule, take a model in which $a$ and $b$ have different interpretations. The **Constant**, **Orient**, and **Variable Elimination** rules have the property that instances of their premises and conclusions by the same substitution $\sigma$ are satisfied in the same models. The same is true for the two **Alpha** rules, since models of lambda logic satisfy the $(\alpha)$ axiom of lambda logic, and for the **Beta** rule, because models of lambda logic satisfy the $(\beta)$ axiom, and for the **Switch** and **Curry** rules, because models of lambda logic satisfy the equality axioms.

Consider the **Lambda** rule, subject to the condition that the variables in the domain of $\sigma$ must not be bound in the premises. That means that $\sigma$ does not assign a value to $x$. Also, $\sigma$ is assumed consistent with the environment of the conclusion; that environment forbids $x$ to all the free variables of $t$ and $s$. Since $\sigma$ is consistent with this environment, $x$ does not occur free in $t\sigma$ or $s\sigma$.

Hence if $M$ satisfies $Ap(t,x)\sigma = Ap(s,x)\sigma$, then it satisfies $Ap(t\sigma),x) = Ap(s\sigma,x)$. Hence by the weak extensionality axiom $(\xi)$ of lambda logic, $M$ satisfies $\lambda x.\,t\sigma = \lambda x.\,s\sigma$. That completes the treatment of the **Lambda** rule.

However, the rules **Substitution** and **Function Definition** apparently only satisfy one implication: if an instance $C\sigma$ of the conclusion is satisfied in a model $M$ of lambda logic, so is the corresponding instance $P\sigma$ of the premise. We first take up the **Substitution** rule. Suppose that an instance of the conclusion is satisfied in a model $M$. Say the instance is obtained by applying some substitution $\sigma$ to $r = q, Ap(x,z) = t', z = q, \Gamma$. Because of condition (iv) on this rule, we can assume that $\sigma$ does not assign values to an variable that is bound in $t$. We have to show that $M$ satisfies $Ap(x\sigma, r\sigma) = t\sigma, \Gamma\sigma$. Since $M$ satisfies $r\sigma = q\sigma$ and $z\sigma = q\sigma$, it follows that $M$ satisfies $r\sigma = q\sigma$. Therefore $M$ satisfies $Ap(x\sigma, r\sigma) = t'\sigma$. Therefore it suffices to show that $M$ satisfies $t\sigma = t'\sigma$. Now $t' = t[q := z]$ (where possibly only some occurrences of $q$ are replaced by $z$). Since $q\sigma = z\sigma$, it follows that $M$ satisfies $t'\sigma = t\sigma$. This uses the fact that $\sigma$ does not assign values to any bound variable of $t$. That completes the treatment of the **Substitution** rule.

Now we treat the **Function Definition** rule. Suppose that for some substitution $\sigma$, $M$ satisfies $x\sigma = (\lambda z.\,t)\sigma, \Gamma\sigma$. Since $(\lambda z.\ t)\sigma = \lambda z.,\ t\sigma$, $M$ satisfies

$x\sigma = \lambda z.\, t\sigma$. Then in $M$ we have

$$\begin{array}{rcl} Ap(x\sigma, z\sigma) & = & Ap(\lambda z.\, t\sigma, z\sigma) \\ & = & t\sigma \end{array}$$

But that shows that $M$ satisfies the instance of the hypothesis of the rule by $\sigma$, which was what we had to show. That completes the proof of the lemma.

**Theorem 1 (Correctness)** *Suppose there is a chain of inferences in the above inference system starting from $S : E$, where $S$ is a multiset of equations consistent with environment $E$, and ending in a $S' : E'$, where $S$ is a set of solved equations consistent with $E'$. Let $\sigma$ be the substitution defined by this set of solved equations. Then $\sigma$ is a lambda unifier for $S$ and $\sigma$ is consistent with $E$. In particular each of the equations in $S$ is provable in lambda logic.*

*Remark.* Normally the initial environment $E$ would be empty.
*Proof.* By definition of $\sigma$, the subsitution $\sigma$ unifies the last multiset in the chain of inferences. Repeatedly applying the lemma, $\sigma$ unifies each multiset in the chain, and is consistent with its environment. But this assertion, specialized to the first member in the chain of inferences, is the theorem. That completes the proof.

## 4 Incompleteness

The inference system (and algorithm) for lambda unification given above is incomplete. Consider the fixed-point example:

$$x = f(x).$$

This has a unifier, namely the substitution $\sigma$ that assigns

$$x := Ap(\lambda y.\, f(Ap(y, y)), \lambda y.\, f(Ap(y, y))).$$

One $\beta$-reduction converts $x\sigma$ to $f(x\sigma)$, as in the usual proof of the fixed point theorem. Yet consider: none of the rules in the inference system apply to $x = f(x)$. The **Variable Elimination** rule does not apply because of the failure of the occurs check: $x$ occurs on both sides of the equation. No other rule has a correct syntactic form to be applicable.

This leaves open the possibility of extending the lambda unification algorithm and inference system given above.

Note that if we set $\omega = \lambda y.\, f(Ap(y, y))$ then we can indeed lambda-unify $Ap(\omega, \omega)$ with $f(Ap(\omega, \omega))$. Thus lambda unification can verify fixed points, but not find them.

## 5 Undecidability

Like higher-order unification, the existence of lambda unifiers is undecidable. An easy proof of this result for higher-order unification in finite type theory is given in [7], p. 1025 (along with references to the original proofs given in 1972). That proof depends on the undecidability of Hilbert's tenth problem and the fact that polynomials (on the Church numerals) are representable by lambda terms. To complete the proof (for type theory or for lambda logic) we only need to show that the Church numerals themselves are equationally definable. The following is an untyped version of Proposition 3.4 on page 1025 of [7].

**Lemma 2** *Suppose lambda logic proves*

$$\lambda z.\, Ap(Ap(t,z),\lambda y.\, y)) = \lambda z.\, z.$$

*Then $t$ has a normal form and that normal form is a Church numeral.*

*Remark.* Every Church numeral $t$ satisfies the equation of the lemma.

*Proof.* We remind the reader that the Church numerals are the terms

$$\bar{n} = \lambda x\, \lambda f.\, Ap(f, Ap(\ldots Ap(f,x)\ldots))$$

where there are $n$ occurrences of $f$. The lemma in [7] is for normal terms in finite type theory; here we start with an untyped term and conclude that it is normalizable, so there is (on the face of it) more to prove. Actually, there isn't much more to prove, but the one-line proof in [7] ("by induction on the structure of $t$") is here made more explicit. In view of the ($\xi$) axiom the equation of the lemma may as well be written without the initial $\lambda$'s:

$$Ap(Ap(t,z),\lambda y.\, y) = z. \tag{1}$$

Suppose this equation is provable in lambda logic. Then the left and right side have a common reduct (using the term model constructed in [2]), and that reduct must be $z$. Hence the left side reduces to $z$. In particular it has a normal form; we may then assume without loss of generality that $t$ is already in normal form. Since the left side reduces, $t$ must be a term beginning with $\lambda$, so for some term $A$ we have

$$t = \lambda z.\, A.$$

Since $t$ is in normal form, $A$ must also be in normal form. We have

$$\begin{aligned} Ap(Ap(t,z),\lambda y.\, y) &= Ap(Ap(\lambda z.\, A, z),\lambda y.\, y) \\ &= Ap(A, \lambda y.\, y) \end{aligned}$$

Since $A$ is in normal form, but this term reduces (to $z$), $A$ must begin with $\lambda$, say $A = \lambda f.\, B$, with $B$ normal. We have

$$\begin{aligned} z &= Ap(A, \lambda y.\, y) \\ &= Ap(\lambda f.\, B, \lambda y.\, y) \\ &= B[f := \lambda y.\, y] \end{aligned}$$

If this term is normal then it must be $z$. Otherwise, it must reduce. For this term to reduce, $B$ must contain $f$ in the context $Ap(f,r)$ for some term $r$. Let $r$ be the maximal such term in $B$. Then $B[f := \lambda y.\, y]$ will reduce to $B[Ap(f,r) := r]$. This term, call it $s$, has has fewer symbols than $B$, so $\lambda z \lambda f.\, s$ has fewer symbols than $t$. Yet it satisfies the same equation as $t$:

$$\begin{aligned}
Ap(Ap(\lambda z\lambda f.\, s), z), \lambda y.\, y) &= Ap(\lambda f.\, s, \lambda y.\, y)\\
&= Ap(\lambda f.\, B[Ap(f,r) := r], \lambda y.\, y)\\
&= Ap(\lambda f.\, B[f : \lambda y.\, y], \lambda y.\, y)\\
&= Ap(\lambda f.\, z, \lambda y., y) \qquad \text{since } B[f := \lambda y.\, y] = z\\
&= z
\end{aligned}$$

Hence, proceeding by induction on the number of symbols in $t$, the induction hypothesis tells us that $\lambda z \lambda f.\, s$ is a Church numeral. That is, $s$ has the form $Ap(f^{(n)}, z)$ for some natural number $n$, where the superscript means $n$ times iteration. Since $B[Ap(f,r) := r]$ is $s$, it follows that $B$ has the form $s[r := Ap(f,r)]$ for some subterm $r$ of $s$. Indeed when defining $s$ we chose $r$ to be the maximal such term, so $r$ is $Ap(f^{(n)}, z)$, that is, $r = s$, and $B$ is then $s[r := Ap(r,f)] = Ap(f,s) = Ap(f^{(n+1)}, z)$. Hence $t = \lambda z \lambda f.\, B$ is $\lambda z \lambda f.\, Ap(f^{(n+1)}, z)$, the Church numeral for $n+1$. That completes the proof of the lemma.

**Theorem 2** *There is no algorithm to decide if two terms of lambda logic are unifiable or not.*

*Proof.* As sketched above. The details are the same as given on p. 1025 of [7] for type theory; only the lemma supplied above is slightly different.

## 6 Lambda unification compared to higher-order unification

If $t$ is a term in simple type theory then we can simply "erase the types" to produce a term $t'$ in lambda logic. Similarly, we can erase the types in a substitution $\sigma$ to get $\sigma'$. If $t$ and $s$ are typed terms unifiable in type theory by a substitution $\sigma$ then $t'$ and $s'$ are unifiable in lambda logic by $\sigma'$.[5]

The "inverse" of the type-erase operation is called *typing.* A *type assignment* assigns types to variables and constants, and to each function symbol $f$ (of a given arity) it assigns a *prototype*, that is, it specifies what the types of the arguments must be and the type of the value of a term with main symbol $f$ must be. A type assignment $\pi$ extends to be defined on more terms than just variables and constants, as follows:

[5]We note that this works just as well if more than one atomic type is allowed, as in [7]; in this way we can simultaneously treat the case of multi-sorted first-order logic, which can be embedded in type theory by regarding the sorts as atomic types, and the function symbols as constant symbols of the appropriate type.

(i) If $\pi$ assigns types $\rho_1, \ldots \rho_n$ to $t_1, \ldots, t_n$, and those are compatible with the prototype $\pi$ assigns to $f$, then $\pi$ assigns $f(t_1, \ldots, t_n)$ the type that $\pi$ assigns as the value type of $f$.
(ii) $\pi$ assigns the term $\lambda x., t$ the type $A \to B$ if $\pi$ assigns $x$ the type $A$ and $t$ the type $B$.
(iii) If $\pi$ assigns $t$ the type $A \to B$ and $s$ the type $A$, then $\pi$ assigns the term $Ap(t, s)$ the type $B$.

The type assignment $\pi$ is defined on a term $t$ only if these conditions require it to be defined, i.e. it is the least solution of these inductive conditions. Since "$\pi$ assigns type $A$ to term $t$" is mentioned only positively in these conditions, this is a legitimate inductive definition of $\pi$.

A set of terms $E$ of lambda logic is *typeable* with respect to a type assignment $\pi$ if $\pi$ is defined on all the terms in $E$. A set of terms is *simultaneously typeable* if it is typeable with respect to some type assignment $\pi$. The following lemma is an immediate consequence of these definitions:

**Lemma 3** *If $E$ is a set of terms in type theory, and $E'$ the set of terms in lambda logic obtained by erasing types, then $E'$ is simultaneously typeable.*

Namely, the type assignment $\pi$ just assigns each term the type it originally had before the types were erased. Now, you might think that if $\pi$ is a type assignment and $t$ and $s$ are in its domain and $\sigma$ is a lambda unifier of $t$ and $s$ then $\pi$ could be extended to a type assignment defined on $t\sigma$ and $s\sigma$. But this is false:

*Example*: Take $t$ to be $Ap(X, u)$ and $s$ to be $Ap(c, u)$, and let $\sigma$ be

$$[X : \lambda z.\, Ap(z, c);\ u : c].$$

Then $t\sigma = Ap(\lambda z.\, Ap(z, c), c) = Ap(c, c)$ and $s\sigma = Ap(c, c)$, so $\sigma$ is indeed a lambda unifier, but there is no way to type $Ap(c, c)$ in simple type theory. Thus it is not the case that every lambda unifier of simultaneously typeable terms "lifts" to a typed unifier. However, lambda unification problems in general have many solutions; in this example, there is also the lambda unifier $[X : c]$ which *does* lift, i.e. is the type erasure of a substitution in the language of type theory.

The inference system given for lambda unification can also be used, if all the variables are typed, and function symbols omitted, and different $Ap$ symbols used for different types, to define a notion of *typed lambda unification.* In the **Masking Term** and **Substitution** rules, the term $q$ must have the same type as the second argument of $Ap$ on the left, in order that the conclusion be properly typed.

To recap: we now have *three* kinds of unification to consider. There is "higher-order unification", there is "typed lambda unification", and there is (untyped) "lambda unification". We have an inference system for each; incomplete for untyped lambda unification, complete for higher-order unification, and up to now we have not made a claim about the completeness or incompleteness of the inference system for typed lambda unification.

It will be instructive to compare higher-order unification with typed lambda unification using an example. Consider, for example, the problem of unifying $\lambda x.\, Ap(X, x)$ with $\lambda x.\, Ap(f, Ap(g, x))$. Here $f$ and $g$ are distinct constants. We shall first work this problem using higher-order unification, then again using lambda unification.

Here is the solution by higher-order unification: This is what is called a "flexible-rigid" equation, so the rule **Generate** ([7], p. 1031) will be applied. With higher-order unification we introduce a new variable $H$ (choosing any of infinitely many possible types for $H$) and we get two templates of possible unifiers:

$$X = \lambda y.\, Ap(f, Ap(H, y))$$

and

$$X = \lambda y.\, Ap(y, Ap(H, y))$$

and in addition we get the rest of the conclusion of the inference rule, the result of substituting that value of $X$ into the premise and $\beta$-reducing, respectively

$$\lambda x.\, Ap(f, Ap(H, x)) = \lambda x.\, Ap(f, Ap(g, x))$$

and

$$\lambda x.\, Ap(x, Ap(H, x)) = \lambda x.\, Ap(f, Ap(g, x)).$$

The first of these alternatives succeeds with $H = g$, yielding the final unifier

$$X = \lambda y.\, Ap(f, Ap(g, y)).$$

The second alternative fails since it is a rigid-rigid equation, since $x$ is bound and $f$ is constant.

Now consider how lambda-unification handles the same example. To unify $\lambda x.\, Ap(X, x)$ with $\lambda x.\, Ap(f, Ap(g, x))$, we "freeze" $x$, and then unify $Ap(X, x)$ with $Ap(f, Ap(g, x))$. Since $X$ doesn't occur on the right, we will use the **Substitution** rule rather than the **Masking Term** rule. The (finitely many) choices are:

$$\begin{aligned}
x &= f, X = \lambda z.\, Ap(z, Ap(g, x)) \\
x &= g, X = \lambda z.\, Ap(f, Ap(z, x)) \\
X &= \lambda z.\, Ap(f, Ap(g, z)) \\
x &= Ap(g, x), X = \lambda z.\, Ap(g, z))
\end{aligned}$$

But only the third alternative is legal, since it is the only one of the four that produces a substitution that does not assign a value to $x$, so it is the only one to which the **Lambda** rule can be applied (in reverse). The final answer is

$$X = \lambda z.\, Ap(f, Ap(g, z))$$

just as for higher-order unification. (We did not, in this example, need to reject any substitutions because they produced values depending on $x$.)

No extra variable $H$ of infinitely many possible types was introduced, as in higher-order unification. What happened to that new variable $H$ introduced in higher-order unification? Eventually it got instantiated, essentially to the term used in the **Substitution** rule in lambda unification.

## References

[1] Baader, F., and Snyder, W., Unification Theory, in Robinson, A., and Voronkov, A. (eds.) *Handbook of Automated Deduction, Vol. 1*, pp. 445–530. Elsevier, Amsterdam (2001).

[2] Beeson, M., Lambda Logic, revised and expanded version of [3], available at www.michaelbeeson.com/research/papers/LambdaLogic.pdf.

[3] Beeson, M., Lambda Logic, in Basin, David; Rusinowitch, Michael (eds.) Automated Reasoning: Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings. Lecture Notes in Artificial Intelligence 3097, pp. 460–474, Springer (2004).

[4] Beeson, M., Mathematical induction in Otter-lambda, accepted for publication in the *Journal of Automated Reasoning* (to appear in late 2006). Meantime, available at www.michaelbeeson.com/research/papers/induction.pdf.

[5] Beeson, M., Implicit Typing in Lambda Logic, available at www.michaelbeeson.com/research/papers/ImplicitAndExplicitTyping.pdf.

[6] Beeson, M. The Otter-$\lambda$ website:

`www.MichaelBeeson.com/research/Otter-lambda/index.php`

[7] Dowek, G., Higher-order unification and matching, in Robinson, A., and Voronkov, A. (eds.) *Handbook of Automated Deduction, Vol. 2*, pp. 1009–1060. Elsevier, Amsterdam (2001).