Unification in Lambda-Calculus

Michael Beeson

Department of Mathematics and Computer Science
San Jose State University
San Jose, California 95192, USA
email: beeson@mathcs.sjsu.edu

Abstract. A new unification algorithm is introduced, which (unlike previous algorithms for unification in λ-calculus) shares the pleasant properties of first-order unification. Proofs of these properties are given, in particular uniqueness of the answer and the most-general-unifier property. This unification algorithm can be used to generalize first-order proof-search algorithms to second-order logic, making possible for example a straighforward treatment of McCarthy's circumscription schema.

1 Introduction

The origins of this work lie partly in an attempt to build a theorem-prover capable of implementing John McCarthy's work on circumscription. Circumscription is a second-order axiom schema, and cannot, in its full generality, be reduced to first-order. Indeed, it does not seem any easier to build a theorem-prover to handle all cases of circumscription than to handle second-order logic in general.

Second-order logic uses λ-terms to define predicates, and hence any attempt to mechanize second-order logic necessarily will involve unification of λ-terms. Huet [Huet 1975] introduced an algorithm for unifying terms in λ-calculus, and Miller and Nadathur [Miller and Nadathur] introduced an extension of Prolog to a fragment of second-order logic (in fact higher-order logic) based on a similar unification algorithm. However, it turned out that Huet's algorithm isn't enough to handle circumscription. The difficulty is not hard to understand: Suppose we want $Q(1) = 1$ and $Q(2) = 0$ (Q can be thought of as a predicate by treating 1 as true and 0 as false). Huet's unification, given the problem $Q(1) = 1$, is only going to produce $\lambda x.1$ and $\lambda x.x$ as possible values of Q, neither one of which satisfies $Q(2) = 0$. Moreover, it is unpleasant that Huet's notion of unification does not have "most general unifiers". These difficulties are related; when the right notion of unification is defined, there are "more" unifiers, and there is also a "minimal" or "most general" unifier, of which all the others are "extensions".

This paper contains the definition of this new unification algorithm, and the proof that it always produces a most general unifier.

2 Related Work

This unification algorithm can be used to extend to second-order logic the "backwards Gentzen" theorem-prover described for first-order logic in [Beeson 1991].

As far as I know, the new unification algorithm presented here makes possible the first direct implementation of circumscription, i.e. using second-order logic directly in proof search. Lifschitz has given a method of reducing some cases of circumscription to first-order logic, where it seems certain that existing theorem-provers could handle the transformed problem, although the experiment hasn't actually been carried out([Lifschitz 1985]). A theorem-prover described in [Baker-Ginsberg 1989] and [Ginsberg 1989] is related to circumscription via Lifschitz's result, but does not involve circumscription in its mechanism. Experiments with this algorithm in automatic deduction by circumscription will be described elsewhere.

3 λ-Calculus and definition by cases

We are interested in applications of unification to various logical systems, including second-order logic and various type theories. All of these logics can be expressed as subsystems of λ-calculus, so it is natural to study unification in the general setting of λ-calculus. The system of λ-calculus for which we solve the unification problem is an extension of the usual λ-calculus. We call this system λD. It is formed by adding to the ordinary λ-calculus a new constant **d** for definition by cases, and the following reduction rules. In these rules, equality means "reduces to".

$$\mathbf{d}(x, x, a, b) = a$$

$$\mathbf{d}(x, y, Zy, Zx) = Zx$$

$$\mathbf{d}(x, y, y, x) = x$$

$$\mathbf{d}(x, y, a, a) = a$$

Taking $Z = \lambda x.x$ and $Z = \lambda x.a$, the last two rules follow as equalities from the second, but as reduction rules instead of just equalities they are not superfluous. We also might consider a rule similar to δ-reduction, namely, for distinct normal terms t and s,

$$\mathbf{d}(t, s, a, b) = b$$

but this rule is not needed in our system.[1]

A natural question is whether λD satisfies the Church-Rosser theorem. J. W. Klop showed that it does not (private correspondence). Equality of terms in λD is defined by the transitive closure of the relation "a and b have a common reduct". Klop's counterexample to Church-Rosser leaves open the question whether normal forms are unique in λD, and indeed leaves open the possibility that two terms not containing d, which have no common reduct in $\lambda\eta$-calculus, might still be equal in λD. Indeed, for all we know at this point, **T** and **F** might be equal in λD. The following shows that this is not the case: λD is consistent and indeed is a conservative extension of $\lambda\eta$, for equations between closed terms.

[1] The $\lambda\delta$-calculus involves a constant δ and the reduction rules $\delta\alpha\alpha \to \mathbf{T}$ and $\delta\alpha\beta \to \mathbf{F}$ when α and β are distinct closed normal terms. See [Barendregt 1981], p. 396 for the classical results about $\lambda\delta$-calculus.

Theorem. *Let P and Q be two closed terms not containing* $\mathbf{d}$, *and suppose $P = Q$ in λD-calculus. Then $P = Q$ already in λ-calculus.*

The proof of this theorem is long and technical, and the strict length limit on papers in this volume prohibits its inclusion. To avoid dependency on this result, one can work instead in a typed λ-calculus. To be specific, we consider simply-typed λ-calculus with one ground type, as described in Appendix A of [Barendregt 1981]. To this system we can consistently add a constant $\mathbf{d}$ of each sensible type such that in $\mathbf{d}(a, b, x, y)$, a and b are of ground type and x and y are of the same type (not necessarily ground), and postulate the laws $\mathbf{d}(a, b, x, y) = x$ if $a = b$ and y otherwise. Then the typed versions of the $\mathbf{d}$-laws of λD are valid. The reader with doubts about the theorem just stated can read the rest of the paper as being about this system of typed λ-calculus instead of λD.

We write $t \cong s$ to mean $tx = sx$, where x is a variable not contained in t or s.[2] Note that extensional equality is preserved under reduction: if $a \cong b$ and a reduces to a' and b reduces to b' then $aa' \cong b'$.

The second $\mathbf{d}$-rule above then takes the form

$$\lambda x.\mathbf{d}(x, a, Za, Zx) \cong Z$$

This form of the rule will be used in the most-general-unifier calculation in a subsequent section.

4 Lambda Calculus and Logic

It is well-known that second-order logic can be defined in lambda-calculus (see [Barendregt 1981], p. 570, for basic references, and [Prawitz 1967] for full details). We specify a few notational matters that will be needed to support the proofs in this paper.

It will be convenient to consider λ-calculus as enlarged with a stock of constants, as well as variables. Technically we can just regard half of the usual variables as "constants"; unification will only instantiate the variables, but not the constants. Note that constants can still be used for λ-abstraction, i.e. from within the λ-calculus they are just like variables.

When logic is translated into lambda-calculus, the "object variables" (variables of the logic) become constants in lambda-calculus, as do the constants and function symbols and predicate symbols of the logic. Also the quantifiers and logical symbols such as $\wedge$ and $\rightarrow$ become constants or constant terms (containing no variables) of λ-calculus. The variables of λ-calculus are used as "metavariables" to range over terms of logic. When we look for a proof of a theorem $\exists x A(x)$, we first replace x by a metavariable $\mathbf{X}$ and search for a proof of $A(\mathbf{X})$. We hope that eventually unification will "instantiate" $\mathbf{X}$ to some term. In λ-calculus, the

[2] The η-reduction rule in λ-calculus says that $(\lambda x.t)x$ reduces to t. This is not a rule of the usual λ calculus, but it has been extensively studied. We deliberately use the relation $t \cong s$ rather than equivalence in $\lambda\eta$-calculus, to avoid the technical complications of adding η-reduction and the $\mathbf{d}$-rules to λ-calculus at the same time.

logical metavariable $\mathbf{X}$ becomes a variable, and the object variable x of the logic becomes a constant of λ-calculus.

Second-order logic has two kinds of variables, first-order and second-order. It is customary to use capital letters for second-order variables. This conflicts with the Prolog convention, in which capital letters are used for metavariables. We therefore use boldface for logical metavariables; but in λ-calculus, we use lower-case letters for variables, as usual. Second-order logic permits the formation of λ-terms which are thought of as comprehension terms, i.e. $\lambda x.a(x)$ intuitively represents the set of x such that $a(x)$. Such terms translate directly into λ-terms of λ-calculus, where a λ-term is thought of as a predicate, taking values $\mathbf{T}$ or $\mathbf{F}$.

In logic, predicates and functions can have more than one argument, but in λ-calculus, "currying" is used; that is, $f(x, y)$ is an abbreviation for $(fx)y$. Thus the translation of an atomic formula $P(x, y)$ will be in curried form.

5 A New Unification Algorithm

Huet pointed out in [Huet 1975] that equations involving second-order variables in general have many solutions, even if we require the solutions to be definable by λ-terms. For example, $X(1) = 1$ could be solved by $X = \lambda x.x$ or by $X = \lambda x.1$. Hence, he said, we cannot expect unique "most general unifiers" as in first-order logic, and he went on to give a unification algorithm that produces many unifying substitutions. As pointed out in the introduction, in applications the solutions produced by Huet's algorithm do not suffice, partly because the algorithm does not produce terms for functions defined by cases, but only by pure λ-terms.

One approach to the problem, which I first tried, is to allow terms for functions that are only partially defined. This probably works, but it requires many details, as the necessary formal systems are not in the literature. Here we take a different approach: we allow unification to produce answers containing a new variable. Instead of having some values (which aren't needed to solve the equation at hand) be undefined, we just leave them *unspecified*, by using a new free variable.

To lead up to the definition of unification, we will give an important case of the definition first. Consider the problem of unifying $X(t)$ with s, where X is a variable and t and s are constant terms. To solve this unification problem, we let Y be a new variable and take $X = \lambda x.\mathbf{d}(x, t, s, Yx)$. More precisely, we return the substitution θ whose value on X is the term $\lambda x.\mathbf{d}(x, t, s, Yx)$. Think of this as "if $x = t$ then s else unspecified".

To understand this definition, consider an example: when we unify $X(0)$ with 1, we get $X = \lambda x.\mathbf{d}(x, 0, 1, Yx)$, which takes the value 1 at 0, and elsewhere is "unspecified", that is, has the value Yx where Y is a new variable. By contrast, Huet's unification would give only the constant function $X = \lambda x.1$. Consider a second example: unify $X(1)$ with 1. Huet's unification would give two distinct answers: $X = \lambda x.1$ and $X = \lambda x.x$. Our answer is "more general" than each of these, and more general than infinitely many other special answers obtained by specifying other values of X than $X(1)$. The "most general" answer specifies only $X(1)$, leaving the other values unspecified.

However, in that definition of what it means to unify $X(t)$ and s, it is important that t and s were constant terms. Consider the problem of unifying $X(z)$ and $a(z)$, where a is constant. Do we want $X = \lambda x.\mathbf{d}(x, z, a(z), Yx)$? Well, in some cases we might, but in other cases we want $X = \lambda x.(a(x) \vee Yx)$. For example, if we are trying to prove in second-order logic that $\exists X(a(z) \rightarrow X(z)$, we will first replace X by a metavariable $\mathbf{X}$ and then try to unify $\mathbf{X}(z)$ with $a(z)$. We want to get $X = \lambda z.a(z)$ or some answer more general than that, such as $X = \lambda z.(a(z) \vee Y(z))$.

This raises an issue which could/should have been considered already in ordinary unification. When a theorem-prover replaces a quantifier by a metavariable, there is restriction on the (object) variables that can occur in the answer substitution when it is eventually found. In the above example, when X is replaced by $\mathbf{X}$, the restriction is that the eventual value of $\mathbf{X}$ is not allowed to contain a free occurrence of z. In [Beeson 1991], a simply-programmed inference program allows "wrong" unifications to be temporarily considered and rejected later. An improvement to the program, suggested by N. Shankar, keeps track of these restrictions and rejects unifications that violate them immediately. Considering unification in second-order logic or lambda-calculus brings this "trick" into focus: it should be part of the *definition* of unification. That is, unification takes place relative to a finite set of restrictions on the possible values of variables; these restrictions are part of the input to the unification algorithm. Whether we take $X = \lambda z(a(z) \vee Y(z)$ or $X = \lambda z\mathbf{d}(x, z, a(z), Yx)$ will depend on whether z is forbidden to X or not, i.e. whether X is restricted from taking values depending on z.

We now make these ideas precise.

Definitions. *A* restriction *is a pair consisting of a variable (of lambda-calculus) and a (possibly empty) list of constants.*[3] *An* environment *is a finite list of restrictions.*[4] *If $\langle x, r\rangle$ is a member of the environment E we say that the variable x* occurs *in E or* is mentioned *in E, and that all the members of the list r are* forbidden *to x in E. We say a compound term t is* forbidden *to x in E if it contains a free occurrence of any constant that is forbidden to x in E. A* substitution *is a function from variables to terms. The substitution σ is* legal *for* environment E *provided $\sigma(X)$ is defined for all X that occur in E and that $\sigma(X)$ does not contain free occurrences of any variable or constant forbidden to X in E. The substitution σ* unifies *terms t and s* relative to *E if for some substitution χ whose restriction to E is the identity, we have $t\sigma\chi = s\sigma\chi$.*[5]

The inputs to the unification algorithm are two terms t and s to be unified and an environment E. We say that t and s are to be unified "relative to" the en-

[3] Intuitively, the eventual value of the variable is not allowed to depend on the members of the list.

[4] Intuitively, an environment lists all the variables in use so far, whether or not their eventual values are restricted, together with any restrictions so far imposed.

[5] That is, $t\sigma = s\sigma$ for some values of the the variables not in E.

vironment E. One output of the unification algorithm is a substitution σ which is legal for E, such that $t\sigma = s\sigma$. The usual notion of unification is obtained by taking an environment E with no restrictions on any of the variables occurring in E. But note that the use of restrictions, even in first-order unification, corresponds to the actual use of unification in theorem-proving.

The unification algorithm has a second output, which is a new environment enlarging the input environment E. Here "enlarging" means simply that new variables may have been added.

The algorithm produces an answer substitution only if it succeeds. It can also fail, by terminating but producing a special signal for failure instead of an answer substitution and new environment. And, at least *a priori*, it might fail to terminate.

We now give the precise definition of our unification algorithm. We will suppress mention of the environment, both in input and output, writing $\mathbf{unify}(t, s)$ as usual to denote the output substitution σ. When it is necessary to mention the environment it will be by way of "z is forbidden to x", which means "z is forbidden to x in E, where E is the input environment". When we say, "Y is a new variable", we mean that Y is a variable not occurring in the input environment E, and it is implicit that the output environment will include the new variable Y. It must be remembered that **failure** is a possible result of $\mathbf{unify}(t, s)$.

If a and b are terms containing no variables, and $a = b$ in λD, then $\mathbf{unify}(a, b)$ succeeds, producing the empty substitution. If a and b are terms containing no variables and not containing $\mathbf{d}$, and a and b reduce to distinct normal forms, then $\mathbf{unify}(a, b)$ fails.[6]

If a and b are literally identical terms (which may contain variables) then $\mathbf{unify}(a, b)$ succeeds, producing the identity substitution.

Next, if either input term permits a β-reduction at top level, the following rules are used:

$$\mathbf{unify}(t, (\lambda x.a)s) = \mathbf{unify}(t, a[s/x])$$

$$\mathbf{unify}((\lambda x.a)t, s) = \mathbf{unify}(a[t/x], s)$$

Next, there are two clauses in the definition of **unify** for each of the reduction rules involving $\mathbf{d}$. These rules take the forms

$$\mathbf{unify}(t, s) = \mathbf{unify}(t', s)$$

$$\mathbf{unify}(s, t) = \mathbf{unify}(s, t')$$

where t is the left side of the reduction rule and s is the right side. From now on, we will be dealing with inputs which, while not necessarily normal, permit no reduction at top-level.

[6] The restriction to $\mathbf{d}$-free terms is necessary because of the failure of Church-Rosser in λD; if one uses typed λ-calculus one can dispense with this.

To unify $X(t)$ and s, where t and s are not forbidden to X, we take $X = \lambda x\mathbf{d}(x, t, s, Yx)$. To unify $X(t_1, t_2)$ and s, where t_1, t_2 and s are not forbidden to X, we take

$$X = \lambda x_1 x_2(\mathbf{d}(x_1, t_1, \mathbf{d}(x_2, t_2, s, Y_2 x), Y_1 x)$$

where Y_1 and Y_2 are new variables, and similarly for unifying $X(t_1, \dots, t_n)$ and s.

We have to define when two λ-terms unify. We have

$$\mathbf{unify}(\lambda x.t, \lambda x.s) = \mathbf{unify}(t, s),$$

but we also need to unify two λ-terms with different bound variables after suitable renaming of the bound variables. The simplest way to handle this is to define

$$\mathbf{unify}(\lambda x.a, \lambda y.b) = \mathbf{unify}(a[z/x], b[z/y])$$

where z is a new variable, i.e. not contained in a or b.

In λ-calculus, there are no function and predicate symbols, except the application term formation symbol, Ap. Officially $x(y)$ is the term $Ap(x, y)$. Here our definition corresponds to Robinson unification. To unify ts with pq, we first unify t with p. If this fails, we fail. If it succeeds with substitution σ, we then unify $s\sigma$ and $q\sigma$. Note, however, that the output environment of the first unification becomes the input environment for the second unification. The output is the result of the second unification.

The above clauses define *first-order unification.* This notion of unification will support first-order inference (since it generalizes Robinson unification), but it is not yet sufficient to support second-order inference, in particular the circumscription examples. Consider for example the inference problem in which we have $az \to Xz$. We want to find a value for X more general than $X = \lambda z.az$. If, for example, we also have $bz \to Xz$ we should be able to find by two successive unifications, something more general than $X = \lambda z.(az \vee bz)$. The following additional clause in the definition of unification will permit this:

To unify Xz and az, where z is a constant forbidden to X, we take $X = \lambda z.(az \vee Yz)$, where Y is a new variable. (The convention here is that z can be a single variable or can be $z_1 \dots z_n$.) More generally, if s is a term containing a constant (or constants) z forbidden to X, to unify Xz and s, we take $X = \lambda z.(s \vee Yz)$. The constant z will be forbidden to Y in the output environment, along with any other constants forbidden to X.

Note that the substitution σ which has been defined in this last clause to unify two terms t and s need not satisfy $t\sigma = s\sigma$, since with $t = Xz$ and $s = az$ we have $t\sigma = az \vee Yz$ and $s\sigma = az$. But if we define χ to be the substitution giving Y (which is not part of the input environment) the value $\lambda z.\mathbf{F}$ then we have $t\sigma\chi = s\sigma\chi$, so σ does unify t and s.

There are also mixed cases, where some of the variables are forbidden to X and some are not. For example, to unify Xxz and axz, where z is forbidden to X and x is not, we take $X = \lambda x.\mathbf{d}(x, t, (axz \vee Yz), Ux)$, where Y and U are fresh variables. The general case is only notationally more complex.

Note that unification can fail to terminate on inputs which have no normal form; for example **unify**(Ω, x) fails to terminate where Ω is any term with an infinite β-reduction sequence. On the other hand **unify**(Ω, Ω) does succeed. Usually we will be interested in unifying normal terms, and in that case unification always terminates, as the following theorem shows.

Theorem. *Let t and s be normal terms, and let E be an environment containing all variables in t or s. Then* **unify**(t, s) *terminates, and unifies t and s relative to E.*

Proof: By induction on the complexity (depth) of t and s. Since they are normal terms, all subterms are normal, and the clauses of **unify** that correspond to reduction rules are never used. The other clauses make recursive calls only to **unify** applied to subterms of t and s, which by induction hypothesis do terminate. We still have to prove that **unify**(t, s) unifies t and s. This is obvious for Robinson unification, but now the definition of "unifies" is more general, involving "some values of the new variables", so there is something to be proved, even for the old clauses in the definition. The induction steps corresponding to the new clauses have been given above, in the course of the definition. Consider the proof that **unify**(ts, pq) really unifies ts and pq. Let $\sigma =$ **unify**(t, p). Then by hypothesis $t\sigma\chi = p\sigma\chi$ for some χ which is the identity on the input environment E. Let $\tau =$ **unify**$(s\sigma, q\sigma)$. By induction hypothesis $s\sigma\tau\eta = q\sigma\tau\eta$ for some η which is the identity on E', the environment including E and any new variables introduced by σ. Since E contains all variables in s and q, τ is the identity on any new variables introduced by σ, and since E' is the input environment for the call producing τ, η is the identity on any new variables introduced by τ. Define $\beta = \eta\tau = \tau\eta$, the union of these two disjoint substitutions. Then we will prove $(ts)\sigma\tau\beta = (pq)\sigma\tau\beta$. Note that $\chi\tau = \tau\chi$ since χ is non-identity only on new variables introduced by σ, but τ is the identity on these variables. We calculate as follows:

$$\begin{aligned}
(ts)\sigma\tau\beta &= \\
(t\sigma\tau\beta)(s\sigma\tau\beta) &= \\
(t\sigma\tau\chi)(s\sigma\tau\eta) &= \\
(t\sigma\chi\tau)(s\sigma\tau\eta) &= \\
(p\sigma\chi\tau)(q\sigma\tau\eta) &= \\
(p\sigma\tau\chi)(q\sigma\tau\eta) &= \\
(p\sigma\tau\beta)(s\sigma\tau\beta) &= (pq)\sigma\tau\beta
\end{aligned}$$

Since $\sigma\tau$ is the output substitution **unify**(ts, pq), this completes the proof.

6 The Most General Unifier

Since substitutions are functions whose values are terms, equality between substitutions is defined in terms of equaltiy of terms. In systems allowing λ-terms, equality of terms involves the notion of reduction, rather than syntactic identity, so the notion of equality of substitutions is correspondingly more complicated. In the applications we have in mind, variables will denote functions or predicates, not only individuals. Therefore, the substition which gives a variable the value t should be equal to the substitution which gives the same variable the value s, if s and t are extensionally equal. As explained above, we write $t \cong s$ to mean $tx = sx$, where x is a variable not contained in t or s. We now define two substitutions θ and μ to be a *equal on an environment* E provided $\theta X \cong \mu X$ for all variables X in E.

The notion of one substitution being "more general" than another is defined almost as usual. The usual definition is this: θ is more general than μ if there is a substitution γ such that $\theta\gamma = \mu$. The fact that our unification algorithm introduces "new" variables requires relativizing this definition, as follows:

Definition. *Given an environment E, θ is more general than μ, relative to E, if there is a substitution γ such that $\theta\gamma = \mu$ on E. That is, for all variables X in the finite set E, we have $X\theta\gamma \cong X\mu$.*

It will sometimes happen that γ is defined on variables not in the original environment E. For example, if θ is the substitution produced by unifying terms t and s, and E is the set of variables occurring in t or s, then γ may be defined on some of the "new" variables introduced by the unification algorithm.

Because reduction is allowed in determining equality, the concept of one substitution being more general than another is more complex than in first-order logic, as the following example will illustrate.

Example: Consider the problem $X(1) = 1$. The answer substitution is $X = \lambda x.\mathbf{d}(x, 1, 1, Y(x))$. Both the answers produced by Huet's algorithm can be obtained, up to extensional equivalence, from this answer by instantiating Y to one of the values $Y = \lambda x.x$ or $Y = \lambda x.1$. If we substitute these values for Y and reduce to normal form, we obtain

$$\mathbf{d}(x, 1, 1, (\lambda x.x)(1)) = \mathbf{d}(x, 1, 1, x) = x$$

$$\mathbf{d}(x, 1, 1, (\lambda x.1)(1)) = \mathbf{d}(x, 1, 1, 1) = 1$$

Hence the answer substitution produced by our second-order unification algorithm is more general than each of Huet's two answers. It is also more general than any of the infinitely many variable-free solutions, such as

$$X = \lambda x.\mathbf{d}(x, 1, 1, \mathbf{d}(x, 2, 0, 3))$$

.

Theorem. *(Most general unifier) Let E be an environment. Suppose that p and q are normal terms in λD. Suppose that for some substitution θ legal for E, $p\theta$ and $q\theta$ are identical. Then p and q unify, and the answer substitution is legal for E, and more general than θ.*

Proof: The proof is by induction on the complexity of the term p. If p is a variable, then $\mathbf{unify}(p, q)$ succeeds with answer substitution χ given by $p\chi = q$.

Take $\beta = \theta$. Then to show $\chi\beta = \theta$ it will suffice to show $p\chi\beta = p\theta$, since χ is the identity on variables other than p. But $p\chi\beta = q\theta$ since $p\chi = q$, and $q\theta = p\theta$ by hypothesis, so $p\chi\beta = p\theta$ as desired. The case when q is a variable is treated similarly. We may henceforth suppose that neither p nor q is a variable.

Suppose p is a constant. Then $p\theta$ is p. Therefore $q\theta$ is also p, so q (since it isn't a variable) is p too. Hence **unify**(p, q) succeeds, returning the identity substitution. Similarly if q is constant.

Now we may assume that p and q are both λ-terms or application terms. Since $p\theta$ and $q\theta$ are identical, either both p and q are application terms, or both of them are λ terms.

Let X be a variable and let $\chi = \textbf{unify}(X(s), t)$, where t and s are normal terms not forbidden to X in E. Then by definition of unification,

$$X\chi = \lambda z.\mathbf{d}(x, s, t, Y(x)) \tag{1}$$

where Y is a new variable (outside E). Suppose

$$X(s)\theta = t\theta \tag{2}$$

We want to find a substitution β such that $\theta = \chi\beta$ on E. By (2) we have

$$(X\theta)(s\theta) = t\theta \tag{3}$$

Define β so that $Y\beta = X\theta$, and $Z\beta = Z\theta$ for all variables Z in the environment E (including the case $Z = X$). Calculate:

$$\begin{aligned}
X\chi\beta &= (\lambda x.\mathbf{d}(x, s, t, Y(x))\beta \\
&= \lambda x.\mathbf{d}(x, s\beta, t\beta, (Y\beta)(x)) \\
&= \lambda x.\mathbf{d}(x, s\theta, t\theta, (Y\beta)(x)) \\
&= \lambda x.\mathbf{d}(x, s\theta, t\theta, (X\theta)(x)) \\
&= \lambda x.\mathbf{d}(x, s\theta, (X\theta)(s\theta), (X\theta)(x))
\end{aligned}$$

by (3). In view of the identity $Z \cong \lambda x.\mathbf{d}(x, a, Za, Zx)$, (which is just another expression of the second **d**-rule), applied with $Z = X\theta$, we have

$$X\chi\beta = X\theta$$

Note that this is the step requiring the use of extensional equality. We cannot make this step go through using only β-reduction in the definition of equality of substitutions; and in view of the examples, this seems quite natural. The case in which we have $t_1, \ldots, t_n$ instead of t is handled similarly.

Now consider the case of unifying $X(z)$ and s, where X is a variable and z is a constant forbidden to X in E, and s contains z. By hypothesis, $(X\theta)(z\theta) = (s\theta)$. Since z is constant, $z\theta = z$, so $(X\theta)z = (s\theta)$. Since θ is legal for X in E, by hypothesis, $X\theta$ does not contain z. Let $a = \lambda w.s[w/z]$, so a does not contain z.

Therefore $X\theta \cong a\theta$, i.e. $(X\theta)w = (a\theta)w = s[w/z]$ for any variable or constant w.

Let χ be the result of the unification algorithm, so $X\chi z = az \vee Yz$. Define β to agree with θ on variables occurring in E, and $Y\beta = \lambda z\mathbf{F}$. Then

$$\begin{aligned}
X\chi\beta w &= (\lambda z(az \vee Yz)\beta)w \\
&= (\lambda z((a\beta)z \vee \mathbf{F})\mathbf{w} \\
&= (\lambda z((a\beta)z))w \\
&= (a\beta)w \\
&= (a\theta)w \\
&= (X\theta)w
\end{aligned}$$

Therefore $X\chi\beta \cong X\theta$ as desired. On any variable U other than X, we have $(U\chi\beta)w = U\beta = (U\theta)w$. Hence $\chi\beta = \theta$ as substitutions. This completes the cases corresponding to the new clauses in the definition of unification.

We still have to check the cases corresponding to ordinary unification. Consider the case when $p = f(t_1, t_2)$ and $q = f(s_1, s_2)$. In λ-calculus, there are no function symbols *per se* except the binary symbol **ap** for application. Usually we don't write **ap** explicitly but just write $p(q)$ or pq for $\mathbf{ap}(p, q)$. So officially the only possibility here is $f = \mathbf{ap}$; but this one case is as difficult as the general case in first-order unification, of course.

Let $\chi = \mathbf{unify}(p, q)$. We want to show that for some β we have $\chi\beta = \theta$. We have, by definition of the unification algorithm, $\chi = \chi_1\chi_2$ where

$$\chi_1 = \mathbf{unify}(t_1, s_1) \tag{4}$$

$$\chi_2 = \mathbf{unify}(t_2\chi_1, s_2\chi_1) \tag{5}$$

Assume $p\theta = q\theta$. Then

$$t_1\theta = s_1\theta \tag{6}$$

$$t_2\theta = s_2\theta \tag{7}$$

Then by induction hypothesis, (4), and (6), we have for some β_1,

$$\theta = \chi_1\beta_1 \tag{8}$$

By (7) and (8), we have

$$t_2\chi_1\beta_1 = s_2\chi_1\beta_1 \tag{9}$$

By (5) and the induction hypothesis, we have for some β

$$\chi_2\beta = \chi_1\beta_1 \tag{10}$$

By (9), we have

$$\chi\beta = \chi_1\chi_2\beta = \chi_1\beta_1 = \theta$$

This completes the case in which p and q are binary compound terms with the same function symbol. In case p and q are unary compound terms with the same function symbol, apply the induction hypothesis to the arguments. In case p is a variable not occurring in q, the answer substitution $\chi = \mathbf{unify}(p, q)$ is given by $p\chi = q$, and χ is the identity on other variables. Since by hypothesis $p\theta = q\theta$, we have $p\theta = p\chi\theta$. Since χ is the identity on other variables than p, we have $\theta = \chi\theta$, so θ itself is the desired β. This completes the proof.

7 Unification and Second-Order Logic

We now return to the connection between unification and second-order logic. An innocent victim of the 15-page length limit for this paper was the detailed description of the version of second-order logic we use. The following remarks will have to suffice:

Each of the four quantifier rules has to be taken twice, once for first-order quantifiers and once for second-order quantifiers. In addition there are the λ-rules:

$$\frac{\Gamma \Rightarrow A[t/x]}{\Gamma \Rightarrow (\lambda x.A)t}$$

$$\frac{\Gamma, A[t/x] \Rightarrow \phi}{\Gamma, (\lambda x.A)t \Rightarrow \phi}$$

These rules permit us to reduce λ-terms when trying to construct a proof bottom-up. These rules were implemented at the same time as first-order logic, in the prover GENTZEN [Beeson 1991], but only simple second-order deductions could be performed by GENTZEN, because a powerful second-order unification algorithm was missing. GENTZEN could find the correct instances of mathematical induction for certain proofs, but could not do circumscription proofs, for example.

We define a system **LD** of second order logic with definition by cases. The intuitive idea is

$$\mathbf{d}(x, y, a, b) = a \textbf{ if } \mathbf{x} = \mathbf{y} \textbf{ else } \mathbf{b}$$

In second-order logic, we can take

$$\mathbf{d}(x, y, A, B) = (x = y \rightarrow A) \wedge (x \neq y \rightarrow B)$$

so it is not necessary to add a constant **d** to second-order logic. However, since **d** figures in our unification algorithm and hence in our theorem-prover, the implemented version of second-order logic does contain a constant **d**.

The λ-rules allow us to reduce λ-terms, but in **LD** we also need to reduce **d**-terms. Therefore **LD** includes the following rules, which we call the **d**-rules:

$$\frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow \mathbf{d}(x, x, A, B)}$$

$$\frac{x \neq y, \Gamma \Rightarrow B}{\Gamma \Rightarrow \mathbf{d}(x, y, A, B)}$$

$$\frac{y \neq x, \Gamma \Rightarrow B}{\Gamma \Rightarrow \mathbf{d}(x, y, A, B)}$$

$$\frac{B, \Gamma \Rightarrow C}{x \neq y, \mathbf{d}(x, y, A, B), \Gamma => C}$$

$$\frac{B, \Gamma \Rightarrow C}{y \neq x, \mathbf{d}(x, y, A, B), \Gamma => C}$$

$$\frac{A, \Gamma \Rightarrow C}{\mathbf{d}(x, x, A, B), \Gamma => C}$$

Note that if $\mathbf{d}(x, y, A, B)$ is regarded as an abbreviation, these rules are derived rules of inference. It follows that proofs in **LD** can be translated into proofs in **L**; moreover the same is true of cut-free proofs. It is therefore a matter of convenience only whether we take **d** as defined or primitive.

Extend second-order logic by allowing "metavariables" to stand in place of terms. Call this system **LDM**. Second-order logic **LD** and its extension **LDM** can be translated into λ-calculus by standard techniques. We will be explicit about those "standard techniques." We distinguish some of the variables of λ-calculus and call them *constants*, agreeing not to use them for other purposes. We then specify some constants of λ-calculus to stand for the propositional connectives and for $\forall$ and $\exists$. We denote these constants by the same symbols $\forall$ and $\exists$. Then the translation A' of A is given by

$$(\forall x A)' = \forall(\lambda x.A')$$

$$(\exists x A)' = \exists(\lambda x.A')$$

The translation commutes with the propositional connectives. Identifying formulas of second-order logic with their translations, we can regard unification as defined on formulas of second-order logic. Predicate symbols are simply constants of λ-calculus.

Lemma. *Let A and B be two formulas of second-order logic* **LDM**. *Suppose* **unify**(A, B) *succeeds with answer substitution θ. Then* **LD** *proves $A\theta \Rightarrow B\theta$.*

Proof: By induction on the computation of **unify**(A, B). Note that this is a sensible induction, since when **unify** is called on two formulas of second-order logic, the recursive calls always have arguments which are formulas of second-order logic as well. This is so even when we regard $\forall x A$ as an abbreviation for the λ-term $\forall(\lambda x.A)$, because $\lambda x.A$ is a formula of second-order logic.

The only interesting cases are the steps corresponding to the new clauses in the definition of unification. Consider the case of **unify**(Xt, s), where t and s are not forbidden to X. Then $\theta =$ **unify**(Xt, s) is given by

$$X\theta = \lambda x.\mathbf{d}(x, t, s, Y(x))$$

where Y is a new variable. We have the following derivation in **LD**:

$$\dfrac{\dfrac{s \Rightarrow s}{\mathbf{d}(t,t,s,Y(t)) \Rightarrow s}}{(\lambda x.\mathbf{d}(x,t,s,Y(x)))t \Rightarrow s}$$

Now consider the case of $\mathbf{unify}(Xz, az)$, where z is constant and forbidden to X and does not occur in a. Then we have the following derivation in **LD**:

$$\dfrac{\dfrac{\dfrac{az \Rightarrow az \quad \dfrac{\mathbf{F} \Rightarrow \mathbf{F}}{\mathbf{F} \Rightarrow \mathbf{az}}}{az \vee \mathbf{F} \Rightarrow \mathbf{az}}}{\lambda w(aw \vee \mathbf{F})\mathbf{z} \Rightarrow \mathbf{Az}}}{Xz\theta \Rightarrow az\theta}$$

This completes the proof.

8 Automated Deduction in Second-Order Logic

Robinson's unification algorithm is the key to theorem-proving in first-order logic, whether one combines it with resolution or with "backwards Gentzen" methods, or uses it in an equational theorem-prover. The extensions to the unification algorithm introduced here will have applications in automated deduction, also independent of whether one uses resolution or some other method of proof search.

In [Beeson 1991] one can find the Prolog source code for a theorem-prover for first-order logic, based on bottom-up construction of cut-free proofs in a Gentzen sequent calculus. The prover uses metavariables to stand for as-yet-undetermined terms, and instantiates these metavariables by unification when the proof construction reaches leaf nodes of the proof tree (axioms). We will not assume familiarity with this prover, but only with the general idea of backwards proof-search in a sequent calculus, introducing metavariables when (certain) quantifiers are stripped away, and instantiating the metavariables later by unification. The prover described in [Beeson 1991] can be extended to second-order logic by changing the unification algorithm to the one given in this paper.

We will describe how it proves the theorem

$$a \neq b \Rightarrow \exists X(X(a) \wedge \neg X(b)).$$

First, the quantifier will be "opened up" and the variable X replaced by a metavariable. In [Beeson 1991] we used capital letters for metavariables; this clashes with the convention that capital letters are used in second-order logic for second-order variables, so here we use $\mathbf{X}$ for a metavariable. The goal is now

$$a \neq b \Rightarrow \mathbf{X}(a) \wedge \neg\mathbf{X}(b).$$

This goal is divided into two goals,

$$a \neq b \Rightarrow \mathbf{X}(a) \tag{11}$$

and

$$a \neq b \Rightarrow \neg\mathbf{X}(b) \tag{12}$$

GENTZEN will work on 11 first. It will unify $\mathbf{X}(a)$ with **true**, instantiating the metavariable $\mathbf{X}$ as

$$\mathbf{X} = \lambda x.\mathbf{d}(x, a, \mathbf{true}\ , \mathbf{Y}x) \tag{13}$$

producing the new goal

$$a \neq b \Rightarrow \lambda x.\mathbf{d}(x, a, \mathbf{true}\ , \mathbf{Y}x)a.$$

The clause implementing the $\Rightarrow \lambda$-rule then applies, reducing the goal to

$$a \neq b \Rightarrow \mathbf{d}(a, a, \mathbf{true}\ , \mathbf{Y}x).$$

Then a **d**-rule applies, producing the goal

$$a \neq b \Rightarrow \mathbf{true}$$

which is an axiom. Now GENTZEN begins to work on (12), with the metavariable $\mathbf{X}$instantiated as in 13. Specifically, the goal is

$$a \neq b \Rightarrow \neg\lambda x.\mathbf{d}(x, a, \mathbf{true}\ , \mathbf{Y}x)b$$

The rule $\Rightarrow \neg$ is applied, producing the goal

$$\lambda x.\mathbf{d}(x, a, \mathbf{true}\ , \mathbf{Y}x)b, a \neq b \Rightarrow \mathbf{false}$$

The $\lambda \Rightarrow$ rules is applied, producing the goal

$$\mathbf{d}(b, a, \mathbf{true}\ , \mathbf{Y}b), a \neq b \Rightarrow \mathbf{false}$$

One of the **d**-rules now applies, producing the new goal

$$\mathbf{Y}b, a \neq b \Rightarrow \mathbf{false}$$

The axiom clause now makes a call to $\mathbf{unify}(\mathbf{Y}b, \mathbf{false})$, instantiating $\mathbf{Y}$ as

$$Y = \lambda y.\mathbf{d}(y, b, \mathbf{false}, \mathbf{Z}y)$$

for a new metavariable $\mathbf{Z}$. This makes the attempt to prove $\mathbf{X}a \wedge \neg\mathbf{X}b$ succeed, returning the answer substitution

$$X = \lambda x.\mathbf{d}(x, a, \mathbf{true}\ , \lambda y.\mathbf{d}(y, b, \mathbf{false}, \mathbf{Z}y)x)$$

The proof produced, rewritten in tree form, is as follows, with

$$t = \lambda x.\mathbf{d}(x, a, \mathbf{true}\ , \lambda y.\mathbf{d}(y, b, \mathbf{false}\ , \mathbf{Z}y)x) :$$

$$\dfrac{\dfrac{\dfrac{\dfrac{a \neq b \Rightarrow \mathbf{true}}{a \neq b \Rightarrow \mathbf{d}(a, a, \mathbf{true}\ , \lambda y.\mathbf{d}(y, b, \mathbf{false}\ , \mathbf{Z}y)}}{a \neq b \Rightarrow ta} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{false}\ \Rightarrow \mathbf{false}}{\mathbf{d}(b, b, \mathbf{false}\ , \mathbf{Z}y), a \neq b \Rightarrow \mathbf{false}}}{(\lambda y.\mathbf{d}(y, b, \mathbf{false}\ , \mathbf{Z}y))b, a \neq b \Rightarrow \mathbf{false}}}{\mathbf{d}(b, a, \mathbf{true}\ , (\lambda y.\mathbf{d}(y, b, \mathbf{false}\ , \mathbf{Z}y))b), a \neq b \Rightarrow \mathbf{false}}}{a \neq b, \quad tb \Rightarrow \mathbf{false}}}{a \neq b \Rightarrow \neg tb}}{a \neq b \Rightarrow ta \wedge \neg tb}}{a \neq b \Rightarrow \exists X(Xa \wedge \neg Xb)}$$

This proof illustrates the use of the first new clause in the definition of unification. The circumscription examples use the second new clause as well.

References

[Baker-Ginsberg 1989] A. Baker and M. Ginsberg. A theorem prover for prioritized circumscription. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 463–467, Morgan Kaufmann, Los Altos, Calif. (1989).

[Barendregt 1981] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam (1981).

[Beeson 1991] M. Beeson. Some applications of Gentzen's proof theory in automated deduction. In Schroeder-Heister (ed.), *Extensions of Logic Programming*, Lecture Notes in Artificial Intelligence **475**, Springer-Verlag, Berlin/ Heidelberg/ New York (1991).

[Ginsberg 1989] M. Ginsberg. A circumscriptive theorem prover. *Artificial Intelligence* **39**, No. 2 (1989).

[Huet 1975] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science* **1** (1975) 27–52.

[Lifschitz 1985] V. Lifschitz. Computing circumscription. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 121–127, Morgan Kaufmann, Los Altos, Calif. (1985).

[McCarthy 1986] J. McCarthy. Applications of circumscription to formalizing common-sense knowledge. *Artificial Intelligence* **28** (1986) 89–116.

[Miller and Nadathur] D. Miller and G. Nadathur. An Overview of λ-Prolog In *Proceedings of the Fifth International Symposium on Logic Programming, Seattle, August 1988.*

[Prawitz 1967] D. Prawitz. Completeness and Hauptsatz for second order logic. *Theoria 33*, 246-258.