Practical Higher-Order Pattern Unification With On-the-Fly Raising

Gopalan Nadathur and Natalie Linnell

Department of Computer Science and Engineering, University of Minnesota,
4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455
Email: {gopalan,nlinnell}@cs.umn.edu, Fax: 612-625-0572

Abstract. Higher-order pattern unification problems arise often in computations within systems such as Twelf, λProlog and Isabelle. An important characteristic of such problems is that they are given by equations appearing under a prefix of alternating universal and existential quantifiers. Most existing algorithms for solving these problems assume that such prefixes are simplified to a ∀∃∀ form by an a priori application of a transformation known as raising. There are drawbacks to this approach. Mixed quantifier prefixes typically manifest themselves in the course of computation, thereby requiring a dynamic form of preprocessing that is difficult to support in low-level implementations. Moreover, raising may be redundant in many cases and its effect may have to be undone by a subsequent pruning transformation. We propose a method to overcome these difficulties. In particular, a unification algorithm is described that proceeds by recursively descending through the structures of terms, performing raising and other transformations on-the-fly and only as needed.

1 Introduction

Higher-order unification, or the unification of typed lambda terms modulo the rules of lambda conversion, is a problem that appears to have poor computational properties: most general unifiers may not exist in relevant instances, complete sets of unifiers may be infinite, the search for such unifiers cannot always be nonredundant and unifiability itself is, in general, undecidable. It seems somewhat of an anomaly, therefore, that effective use has been made of this operation in a variety of applications within metalanguages, logical frameworks and proof assistants such as λProlog [9], Twelf [16] and Isabelle [13]. The answer to this puzzle seems to lie in the fact that good programming practice avoids exercising the pathological cases for this form of unification. A discovery of this kind was made by Dale Miller who observed that occurrences of instantiatable (existential) variables in λProlog programs usually satisfy static conditions that lead to unification computations belonging to what is known as the L_λ or higher-order pattern class [6, 12]. For problems in this class, unifiability is decidable and most general unifiers can be provided. Moreover, even though the syntactic restrictions may not be satisfied by all useful programs [5], the unification problems that arise dynamically still usually lie within the L_λ class.

In light of these observations, higher-order pattern unification has adopted a special practical significance. At a coarse level, the unification procedure for simply typed lambda terms that was invented by Huet [3] behaves well on these special problems: it converges on exactly one successful path for each solvable problem and can also be made to terminate in every case. Nevertheless, this procedure has finer-grained characteristics that can be improved: each local step within it still involves examining several competing substitutions and a successful computation may offer only a pre-unifier, conditioned by a solvable but as yet unsolved set of constraints. Both deficiencies can be addressed. Miller has proposed what is ultimately a refinement of Huet's procedure that, for each problem of the L_λ kind, either determines non-unifiability or yields a most general unifier at the end of a non-branching computation [6]. The ideas underlying this procedure has been extended to dependently typed lambda calculi [14, 15] and higher-order rewrite systems [11]. A version of the procedure that has a time and space complexity that is linear in the size of the input terms has been developed [18] and it has also been adapted to use explicit substitutions relative to a special grafting interpretation of instantiatable variables [2].

This work is motivated by a desire to exploit higher-order pattern unification in low-level implementations; in particular, in the *Teyjus* implementation of λProlog [10]. While many variants of the original procedure have been described, none of them seems quite suited to this task. In such a setting, it is important that the processing be driven entirely by a recursive examination of the structures of terms. The original procedure that is given by transformation rules has two rules—pruning and raising—that do not possess this character. Another important property of practical unification problems is that they need to be solved under a mixed prefix of quantifiers [7] that are created in the course of computation. Most procedures other than the original one seem to finesse this issue by assuming that existential quantifiers appear only at the top-level, embedded at most under universal quantifiers corresponding to global constants. [1] Now, it is possible to transform arbitrary prefixes into this specialized form by initially applying a raising transformation to existential quantifiers. However, this preprocessing must be done dynamically and it requires at least some additional bookkeeping at runtime. Moreover, much of this kind of raising may be unnecessary and performing it has the potential of making other necessary steps more expensive than they need to be.

This paper is intended to redress this situation. We describe in it a procedure within which unification takes place relative to a mixed quantifier prefix. However, such prefixes are rendered implicit by tagging existential and universal variables with numbers that count the quantifier alternations prior to the ones binding these variables. The processing is oriented around a recursive traversal of the terms to be unified and consists essentially of simplification and variable binding phases. The latter phase is characterized by an on-the-fly application of

[1] Two exceptions are the approaches in [14] and [2]. The scheme presented in [14] does not generate most general unifiers and, hence, seems not to be complete. We discuss [2] further in Section 7.

the pruning and raising rules in which the numeric tags are used to recognize quantifier orders. The algorithm we describe is meant for use with lambda terms that are polymorphically typed. In this setting it may sometimes be necessary to treat η-convertibility dynamically. Our algorithm supports this capability.

The rest of this paper is organized as follows. The next two sections describe the higher-order pattern unification problem and present a naive procedure for solving it. This procedure is then refined into a more sophisticated form. One refinement, described in Section 4, makes the quantifier prefix implicit. Another refinement, developed in Section 5, factors the algorithm into simplification and variable binding phases. In Section 6 we discuss some aspects relevant to the practical realization of the procedure. We illustrate our procedure in Section 7 and also contrast its behaviour with previously described ones for the same problem. We conclude the paper with a brief discussion of continuing work.

2 Logical Preliminaries

The lambda terms that are of interest to us may contain universal, existential and lambda bound variables. We initially use the symbol u and x, possibly with subscripts, to denote variables of the first two kinds; later the status of such variables will be determined by an explicit quantifier prefix. A variable occurrence that is bound by an abstraction will be represented, following the scheme due to de Bruijn, by a positive number that counts the abstractions up to and including the one binding that particular occurrence. We bunch together a sequence of abstractions and, likewise the arguments in a sequence of applications.

Formally, our terms are given by the syntax rule

$$t ::= x \mid u \mid i \mid \lambda(i, t) \mid t(\bar{t})$$

in which i represents positive integers, n represents natural numbers and $\bar{t}$ represents a sequence of comma separated terms. In an expression of the form $\lambda(i, t)$, i denotes the number of abstractions. In a schematic presentation, we shall allow this number also to be 0, in which case the term is identical to t. Applications are written in a manner reminiscent of first-order syntax rather than in the usual curried form: thus, the term $t_1(t_2, t_3)$ is equivalent to what we would ordinarily write as $((t_1\ t_2)\ t_3)$ in a higher-order language. We actually think of $\bar{t}$ as a vector, writing $|t|$ to denote its length and $\bar{t}[i]$ to refer to its i-th argument. Once again, in a schematic presentation, we shall let $\bar{t}$ be an empty sequence; the expression $t(\bar{t})$ in this case matches with whatever matches with t.

Two terms are considered equal if they can be β-converted to each other. Testing for such equality is based on head normal forms. A term in this form has the structure $\lambda(n, a(\bar{t}))$ where a, called the head of the term, is a universal or existential variable or a de Bruijn index. Although we do not display types explicitly anywhere, we assume that our terms are typed in an ML-like sense. A consequence of this assumption is that every term in fact reduces to a head normal form. A discussion of procedures for producing such a form that exploit explicit substitution notations for the lambda calculus may be found in [4].

Our operative notion of equality also includes η-conversion. This makes it necessary to sometimes consider the η-expansion of terms. We write $t \uparrow j$ to denote the 'lifting' of the term t over j (new) abstractions; this operation, in effect, increments the free variables in t by j. For atomic terms, the computation actually has a simple form: if t is a universal or existential variable then $t\uparrow j = t$ and if it is a de Bruijn index then $t \uparrow j = t + j$. For existential variables, this definition reflects a logical interpretation as opposed to the grafting one in [1]; in particular, these variables cannot be instantiated by terms containing de Bruijn indices bound by external abstractions. The lifting operation is extended to a sequence of terms: if $|\overline{t}| = n$ then $\overline{t}\uparrow j = \overline{t'}$ where $\overline{t'}$ is a sequence of length $n + j$ with $\overline{t'}[i] = \overline{t}[i]\uparrow j$ for $1 \leq i \leq n$ and $\overline{t'}[i] = j - (i - (n+1))$ for $n < i \leq n+j$. This definition applies even when $|\overline{t}| = 0$ and we allow the expressions $t\uparrow j$ and $\overline{t}\uparrow j$ to be used also when j is 0 in which case they are identical to t and $\overline{t}$, respectively. Using these operations, the j-fold η-expansion of the term $\lambda(n, a(\overline{t}))$ in head-normal form is given by the term $\lambda(n + j, (a\uparrow j)(\overline{t}\uparrow j))$, also in head-normal form. Such an η-expansion is sensible only if the type of a allows for it.

A unification problem is defined by two components: a list of equations such that the two terms in each equation have the same type and a quantifier prefix that scopes over the list. In depicting equation lists, we shall use *nil* to denote the empty list and :: to denote an infix, right associative operator that allows a new equation to be added to the front of an existing list. Universal and existential variables may appear in the equations and these are expected to be captured by a quantifier of the corresponding force appearing in the prefix. Intuitively, universal variables correspond to constants whereas existential variables may be instantiated towards solving the equations in a situation where an equation is considered solved when it relates two (closed) terms that are equal modulo β- and η-conversion. A solution to or a unifier for a given problem is a substitution for the existential variables that reduces the list of equations to a solved form. However, the prefix structure restricts what substitutions may legitimately be made for a given existential variable: these must be closed terms, *i.e.*, terms with no existential variables or unbound de Bruijn indices, in which the only constants that are allowed to appear are ones whose quantifier scope includes that of the quantifier governing the existential variable. This constraint determines, for example, that the unification problem given by $\exists x \forall u((x = u) :: nil)$ has no solutions whereas the problem $\forall u \exists x((x = u) :: nil)$ has the solution $\{\langle x, u\rangle\}$.

In the description above, solutions are required to be closed substitutions. We relax this requirement to permit an existential variable to appear in a substitution term provided its quantifier scope includes all the universal quantifiers within the scope of the quantifier governing the variable being substituted for. Thus, the unification problem $\forall u \exists x_1 \exists x_2((x_1 = x_2) :: nil)$ now has $\{\langle x_1, x_2\rangle\}$ as a solution. We further allow a prefix to be extended by the introduction of existential quantifiers over variables not occurring in the equation list and consider solutions to such modified problems to be solutions to the original problem. For instance, the earlier problem may be modified to $\forall u \exists x_1 \exists x_2 \exists x_3((x_1 = x_2) :: nil)$ and then has the solution $\{\langle x_1, x_3\rangle, \langle x_2, x_3\rangle\}$. Substitutions with existential vari-

ables may be thought of as schemas for generating (legitimate) closed solutions. Towards making this idea precise, we first note that substitutions are given by a finite set of variable-term pairs where each pair pertains to a distinct variable and where the term obeys the constraints imposed by a relevant quantifier prefix and that the application of a substitution θ to a term t is denoted by $\theta(t)$. Further, the composition of two substitutions θ_1 and θ_2, written $\theta_1 \circ \theta_2$, is given as follows: $\langle x, t\rangle$ belongs to $\theta_1 \circ \theta_2$ just in case $\langle x, s\rangle \in \theta_2$ and $t = \theta_1(s)$ or there is no pair pertaining to x in θ_2 and $\langle x, t\rangle \in \theta_1$. Now, it is easily seen that if θ is a solution to a unification problem $\mathcal{Q}(E)$ that is legitimate with respect to a prefix $\mathcal{Q}'$ that extends $\mathcal{Q}$ in the permitted fashion and ρ is another substitution that is legitimate with respect to $\mathcal{Q}'$, then $\rho \circ \theta$ is also a solution to $\mathcal{Q}(E)$. The substitution θ then constitutes a schema in the sense that it represents all the closed solutions that can be obtained from it by such a composition. Our interest is eventually only in closed substitutions pertaining to the existential variables appearing in $\mathcal{Q}$, the original prefix. Letting $\theta_1 =_{\mathcal{Q}} \theta_2$ represent the proposition that θ_1 and θ_2 agree on these variables, we say that θ is a most general unifier for $\mathcal{Q}(E)$ just in case it is a solution to this problem that is legitimate with respect an extended prefix $\mathcal{Q}'$ and for every closed solution ρ_1 of $\mathcal{Q}(E)$ there is a substitution ρ_2 that is legitimate with respect to $\mathcal{Q}'$ and such that $\rho_1 =_{\mathcal{Q}} \rho_2 \circ \theta$.

Unification problems may be higher-order in the sense that function variables may by existentially quantified in the prefix. A particular problem illustrating this facet is the following: $\forall u_1 \forall u_2 \exists x((x(u_2) = u_1(u_2)) :: nil)$. This problem has the two incomparable solutions $\{\langle x, \lambda(1, u_1(1))\rangle\}$ and $\{\langle x, \lambda(1, u_1(u_2))\rangle\}$. A *higher-order pattern unification* problem is one where each occurrence of an existential variable in the equation list satisfies the following syntactic constraint: if it appears applied to arguments, then each of these arguments is a distinct de Bruijn index or a distinct universal variable whose quantifier appears within the scope of the quantifier binding the existential variable. The expression $x(u_2)$ in the unification problem just considered does not satisfy this constraint. However, it does obey this requirement in the problem $\forall u_1 \exists x \forall u_2((x(u_2) = u_1(u_2)) :: nil)$ with a modified prefix. The result of this change is that the problem now has only one solution: the substitution $\{\langle x, \lambda(1, u_1(1))\rangle\}$. The existence of most general solutions is a general property of higher-order pattern unification problems [6].

The quantifier prefixes governing the list of equations in a unification problem usually arise from reasoning over predicate formulas in a larger logical system. While our presentation appears to portray these prefixes as fixed entities, it is important to bear in mind that they evolve during computation in practice.

3 Unification via Transformations

We present the first version of our unification procedure in the form of rewrite rules that transform tuples of the form $\langle \mathcal{Q}(E), \theta\rangle$ where $\mathcal{Q}(E)$ is a unification problem and θ is a substitution. In the initial configuration, the first component of the tuple is the problem that we want solved and θ is the empty substitution. The purpose of the rewrite rules is to reduce the differences between the terms in

(1) $\langle \mathcal{Q}((\lambda(n,t) = \lambda(n,s)) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}((t = s) :: E), \theta\rangle$, provided $n > 0$.

(2) $\langle \mathcal{Q}((\lambda(n,t) = \lambda(m,s)) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}((t = \lambda(m-n,s)) :: E), \theta\rangle$,
provided $n > 0$ and $m > n$.

(3) $\langle \mathcal{Q}((a(\overline{t}) = \lambda(m,s)) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}(((a{\uparrow}m)(\overline{t}{\uparrow}m) = s) :: E), \theta\rangle$,
provided a is a de Bruijn index or a universal variable and $m > 0$.

(4) $\langle \mathcal{Q}((f(\overline{t}) = \lambda(n, g(\overline{s}))) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}((f(\overline{t}{\uparrow}n) = g(\overline{s})) :: E), \theta\rangle$,
provided f and g are existential variables and $n > 0$.

(5) $\langle \mathcal{Q}((a(\overline{t}) = a(\overline{s})) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}((\overline{t}[1] = \overline{s}[1]) :: \ldots :: (\overline{t}[n] = \overline{s}[n]) :: E), \theta\rangle$,
where $|\overline{t}| = n$, provided a is a de Bruijn index or a universal variable.

(6) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2((f(\overline{y}) = \lambda(n, a(t_1, \ldots, t_m))) :: E), \theta\rangle \longrightarrow$
$\langle \mathcal{Q}_1 \exists h_1 \ldots \exists h_m \exists f \mathcal{Q}_2((h_1(\overline{y}{\uparrow}n) = t_1) :: \ldots :: (h_m(\overline{y}{\uparrow}n) = t_m) :: \theta'(E)), \theta' \circ \theta\rangle$,
where $\theta' = \{\langle f, \lambda(|\overline{y}| + n, a(h_1(|\overline{y}| + n, \ldots, 1), \ldots, h_m(|\overline{y}| + n, \ldots, 1)))\rangle\}$,
provided a is universally quantified in $\mathcal{Q}_1$ and f does not appear in $\overline{t}$.

(7) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2((f(\overline{y}) = \lambda(n, a(t_1, \ldots, t_m))) :: E), \theta\rangle \longrightarrow$
$\langle \mathcal{Q}_1 \exists h_1 \ldots \exists h_m \exists f \mathcal{Q}_2((h_1(\overline{y}{\uparrow}n) = t_1) :: \ldots :: (h_m(\overline{y}{\uparrow}n) = t_m) :: \theta'(E)), \theta' \circ \theta\rangle$,
where $\theta' = \{\langle f, \lambda(|\overline{y}| + n, a'(h_1(|\overline{y}| + n, \ldots, 1), \ldots h_m(|\overline{y}| + n, \ldots, 1)))\rangle\}$
for $a' = a{\downarrow}(\overline{y}{\uparrow}n)$, provided a appears in $\overline{y}{\uparrow}n$ and f does not appear in $\overline{t}$.

(8) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2((f(\overline{y}) = f(\overline{z})) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}_1 \exists h \exists f \mathcal{Q}_2(\theta'(E)), \theta' \circ \theta\rangle$,
for $\theta' = \{\langle f, \lambda(m, h(\overline{w}))\rangle\}$, where $m = |\overline{y}|$ and $\overline{w} = \overline{\{m - i \mid i \leq m \text{ and } \overline{y}[i] = \overline{z}[i]\}}$.

(9) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2 \exists g \mathcal{Q}_3((f(\overline{y}) = g(\overline{z})) :: E), \theta\rangle \longrightarrow$
$\langle \mathcal{Q}_1 \exists f \exists h \mathcal{Q}_2 \exists g \mathcal{Q}_3((f(\overline{y}) = h(\overline{w} + \overline{z})) :: \theta'(E)), \theta' \circ \theta\rangle$,
where $\overline{w} = \overline{\{u \mid \forall u \text{ appears in } \mathcal{Q}_2\}}$ and $\theta' = \{\langle g, h(\overline{w})\rangle\}$,
provided $\mathcal{Q}_2$ contains at least one universal quantifier.

(10) $\langle \mathcal{Q}_1 \exists f \mathcal{Q}_2 \exists g \mathcal{Q}_3((f(\overline{y}) = g(\overline{z})) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}_1 \exists h \exists f \mathcal{Q}_2 \exists g \mathcal{Q}_3(\theta'(E)), \theta' \circ \theta\rangle$,
for $\theta' = \{\langle f, \lambda(m, h(\overline{u}))\rangle, \langle g, \lambda(n, h(\overline{v}))\rangle\}$,
where $m = |\overline{y}|$, $n = |\overline{z}|$ and $\overline{u} = \overline{w}{\downarrow}\overline{y}$ and $\overline{v} = \overline{w}{\downarrow}\overline{z}$ for $\overline{w} = \overline{y} \cap \overline{z}$,
provided no universal quantifiers appear in $\mathcal{Q}_2$.

Fig. 1. Transformation rules for higher-order pattern unification

the equations. They may postulate substitutions towards this end and these are accumulated in the second component of the tuple. New existential variables may be introduced in the process and the quantifier prefix is, in this case, modified to accommodate them. If a reduction sequence succeeds in transforming the equation list to an empty one in this way, then the substitution component is intended to be a most general solution to the original unification problem.

The specific rules defining the procedure appear in Figure 1. As is evident from their lefthand sides, the action carried out by these rules is based on the form of the first equation in a non-empty list. Prior to attempting a match, the two terms in such an equation must be reduced to their head normal forms. The matching process assumes that the equality symbol is symmetric, *i.e.*, it will attempt a match by also interchanging the left and right sides of the equations

shown in the patterns. An explanation is warranted with respect to some of the notation for terms used in these patterns. The symbols $\overline{t}$ and $\overline{s}$ used in rules (3)-(7) are intended to match with (possibly empty) sequences of arguments in actual terms and the rules may refer to the lengths of these sequences as well as their components subsequent to the match. The symbols $\overline{y}$ and $\overline{z}$ that are used in rules (6)-(10) match with actual argument sequences only if they further satisfy the higher-order pattern restriction: each element of such a sequence must be a distinct de Bruijn index or universal variable that is quantified within the scope of the existential quantifier binding the variable that appears as the head of the term. That this condition is satisfied needs to be ascertained only if the problem is not already known to be of the higher-order pattern unification kind and, in this case, the arguments will also have to be reduced to head-normal form prior to the attempted match. Some explanation pertaining to the action parts of the rules is also relevant. The rules (6)-(10) introduce existential quantifiers over variables shown schematically as h, possibly with subscripts. These variables must be different from ones already appearing in the quantifier prefix. In rule (9), the notation $\overline{S}$ where S is a set is used to denote a sequence created from the elements of S and the expression $\overline{w} + \overline{z}$ is used to represent the sequence obtained from the concatenation of two given ones. In rule (10), we write $\overline{y} \cap \overline{z}$ to represent a sequence of the elements common to $\overline{y}$ and $\overline{z}$. The notation $a \downarrow \overline{z}$ where $\overline{z}$ is a sequence of distinct universal variables and de Bruijn indices and a is an element of this sequence that is used in rule (7) corresponds to the de Bruijn index given by $(|\overline{z}| + 1 - i)$ where $\overline{z}[i] = a$. The expression $\overline{w} \downarrow \overline{z}$ used in rule (10) extends this notation to the situation where $\overline{w}$ is a sequence of elements appearing in $\overline{z}$ as follows: if $|\overline{w}| = n$ then $\overline{w} \downarrow \overline{z} = \overline{w}[1] \downarrow \overline{z}, \ldots, \overline{w}[n] \downarrow \overline{z}$.

The procedure that we have described here is essentially a deterministic adaptation of the one in [6] to a situation where the de Bruijn notation is used and where η-expansion is done on demand. Determinism is obtained by imposing a processing order that is based on a recursive traversal of the structures of the terms to be unified. In this context the rules in Figure 1 may be understood as follows. Rules (1) and (2) implement a descent through abstractions. If the number of abstractions at the top-level in the two terms are mismatched, an η-expansion may be needed. This is done explicitly when the head of the non-abstraction term is *rigid* or unchangeable under substitution (rule (3)) or when both terms have existential variables as their heads, *i.e.*, have heads that are *flexible* (rule (4)). In the only remaining case, η-expansion is folded into the substitution generation process (rules (6) and (7)). Once past abstractions, rule (5) reduces the task of unifying two terms with the same rigid head to that of unifying their (possibly empty) sequence of arguments in a pairwise fashion; typing constraints ensure that the lengths of these sequences are the same. Rule (8) solves an equation between two terms with the same flexible head; if these are applications, typing constraints again ensure that they have the same number of arguments. Rule (10) solves such an equation when the two heads are not identical but are bound by existential quantifiers that scope over the same universal quantifiers. Rule (9) applies the *raising* transformation to prepare the

ground for rule (10) in case the proviso on quantifier scopes is not met initially. The only remaining case relative to higher-order patterns is that where one of the terms is an application with a flexible head and the other term has a rigid head. Rules (6) and (7) attempt to solve this problem at the root using imitation and projection substitutions, respectively, adding new equations to realize the needed recursion over the arguments of the term with the rigid head.

The formal properties of the procedure are given by the following theorem whose proof is to be found in an extended version of this paper:

Theorem 1. *If $\mathcal{Q}(E)$ represents a unification problem, then the sequence of rule applications starting from $\langle \mathcal{Q}(E), \emptyset \rangle$ must terminate. Further, if the last tuple has the form $\langle \mathcal{Q}'(nil), \theta \rangle$, then θ represents a most general solution to $\mathcal{Q}(E)$. Finally, if $\mathcal{Q}(E)$ is a solvable higher-order pattern unification problem, then the tuple at the end must have such a form.*

Theorem 1 shows that the procedure we have outlined in this section is complete for higher-order pattern unification problems. In a more general situation, however, the procedure may terminate because the proviso on the forms of arguments for terms with flexible heads is not satisfied. This can happen even when the problem embodied in the state has a solution. Our procedure is therefore seen not to be complete for general higher-order unification.

4 Eliminating the Quantifier Prefix

The explicit treatment of quantifier prefixes poses practical difficulties: Prefixes grow and shrink as the result of other logical computations and maintaining them therefore requires run-time effort. Using the prefixes also requires that contextual information be examined in the recursive descent through term structure. It is preferable that such a descent be predicated entirely on local information.

Towards understanding how quantifier prefixes may be obviated, we examine the manner in which they are utilized in the unification procedure. These prefixes are relevant to three tasks: (i) determining whether given variable occurrences are of the existential or universal kind, (ii) ascertaining that the arguments of a flexible term satisfy the scoping requirements of higher-order patterns, and (iii) realizing the raising transformation embodied in rule (9). The first of these tasks can also be accomplished by labelling each variable with its associated type. The only additional information the prefix supplies for the second task is the relative order of quantification. However, this information can be maintained more succinctly by associating with each variable a numeric tag that records the number of times an existential quantifier is immediately followed by a universal one in the prefix up to and including the quantifier binding that occurrence. We assume henceforth that this is done and that the tag for a variable y is given by $l(y)$. The test for the satisfaction of the higher-order pattern constraint becomes a local one with these tags: a universal variable u is quantified within the scope of the quantifier for an existential variable x just in case $l(x) < l(u)$.

The information needed for the raising transformation seems more difficult to encode in a local fashion at the outset: rule (9) requires knowledge of the universal quantifiers that intervene between two existential quantifiers in the prefix and this is information that appears not to be available simply from looking at the two (flexible) terms that are to be unified. There is, however, an important observation to be made about the role of rule (9) in the unification procedure. The purpose of this rule is essentially to prepare the stage for an application of rule (10) that solves an equation between two flexible terms with distinct heads. From this perspective, the universal variables over which g, the flexible head of one of the pertinent terms in rule (9), is raised can be factored into two kinds: those that appear as arguments of f, the flexible head of the other term, and those that do not so appear. While rule (9) raises g over the latter kind of variables as well, this raising is redundant since the subsequent application of rule (10) prunes them away. Thus, the collection of variables over which g really needs to be raised can be determined simply by looking at the universal variables that appear as arguments of f and checking if they are quantified outside the scope of the quantifier for g. The last aspect, as we have already noted, can be decided by looking at the tags associated with the two variables.

We introduce notation to represent the operation of raising over a restricted collection of variables: If g is an existential variable and $\overline{y}$ is a sequence of distinct universal variables and de Bruijn indices, then $\overline{y}{\Uparrow}g$ will denote the sequence

$$\overline{\{u \mid u \text{ is a universal variable occurring in } \overline{y} \text{ such that } l(u) \leq l(g)\}}.$$

Now, the modified version of rule (9) and rule (10) both involve traversals over the arguments of the flexible terms that can potentially be carried out simultaneously in an implementation. Towards facilitating this possibility, we combine these rules into one new rule labelled (9'):

(9') $\langle (f(\overline{y}) = g(\overline{z})) :: E, \theta \rangle \longrightarrow \langle \theta'(E), \theta' \circ \theta \rangle$,
for $\theta' = \{\langle f, \lambda(m, h(\overline{q} + \overline{v}))\rangle, \langle g, \lambda(n, h(\overline{p} + \overline{u}))\rangle\}$, where $m = |\overline{y}|$, $n = |\overline{z}|$, h is a new existential variable such that $l(h) = l(f)$, $\overline{p} = \overline{y}{\Uparrow}g$, $\overline{q} = \overline{p}{\downarrow}\overline{y}$, and $\overline{v} = \overline{w}{\downarrow}\overline{y}$ and $\overline{u} = \overline{w}{\downarrow}\overline{z}$ for $\overline{w} = \overline{\{a \mid a \text{ appears in both } \overline{y} \text{ and } \overline{z}\}}$, assuming $f \neq g$ and $l(f) \leq l(g)$.

In this rule, $\overline{p}$ collects the universal variables over which g must eventually be raised by looking at the arguments of f and $\overline{q}$ represents a sequence of projections over the arguments of f that is needed to match $\overline{p}$. The calculation of $\overline{p}$ and $\overline{q}$ captures the cumulative effect of raising as per rule (9) and the application of rule (10) relative to the added arguments. The effect of rule (10) corresponding to the original arguments of g and their counterparts in f is reflected in the calculation of $\overline{v}$ and $\overline{u}$, respectively.

The discussions of this section result in a unification procedure that is obtained by modifying the rules in Figure 1 as follows: First, quantifier prefixes are eliminated from unification problems and numeric and type tags are associated with all universal and existential variables. Second, the new existential variables introduced in rules (6)-(8) are all accorded the same numeric tag as f. Third,

the choice between rules (7) and (8) when a is a universal variable is made by comparing the numeric tags of a and f, picking (7) if $l(a) \leq l(f)$ and attempting to use (8) otherwise. Finally, rules (9) and (10) are replaced by rule (9').

Let $\mathcal{Q}(E)$ be a unification problem that is presented with an explicit quantifier prefix and let E' be a version of the equations in E in which universal and existential variables are distinguished and labelled with numeric tags consistent with the prefix. We shall then say that E' is obtained from $\mathcal{Q}(E)$ by prefix erasure. The important property of the unification procedure described in this section is the content of the following theorem whose proof involves using the intended correspondence between tags and prefixes and the relationship between rule (9') in the present system and the earlier rules (9) and (10).

Theorem 2. *Let E' be obtained from $\mathcal{Q}(E)$ by prefix erasure. Then the sequence of rule applications starting from $\langle E', \emptyset \rangle$ must terminate. Further, if the last tuple has the form $\langle nil, \theta \rangle$, then θ is a most general unifier for $\mathcal{Q}(E)$. Finally, if $\mathcal{Q}(E)$ is a solvable higher-order pattern unification problem, then the tuple at the end must have such a form.*

5 Combining Variable Substitution Steps

The procedure we have at this point solves equations involving two flexible terms immediately. However, it resorts to an incremental process when one term is flexible and the other is rigid and has arguments. This character is manifest in the structure of rules (6) and (7) in Figure 1 that solve the problem if possible at the 'root' and introduce new variables and equations towards solving the rest of the problem in subsequent steps. There is a bookkeeping overhead to this approach that can be avoided by combining the sequence of steps into a mechanism that generates a single composite substitution for solving the entire equation. Such a mechanism would obviously involve a traversal of the structure of the rigid term. Rules (6) and (7) already require such a traversal towards ascertaining that the head of the flexible term does not appear in the arguments of the rigid one. Ideally, these two traversals should be folded into one.

Figure 2 presents a set of rewrite rules that embody a realization of the substitution generation process of the kind desired. These rewrite rules have a pseudo-procedural character in that some of them have as side conditions additional computations using the same set of rules; the symbol $\stackrel{*}{\longrightarrow}$ is to be interpreted in them as a sequence of rewritings. In interpreting these rules, all the notational conventions described in conjunction with Figure 1 are to be utilized. We also assume the following additional conventions: $\overline{s}^R$ represents the reverse of the sequence $\overline{s}$, ε matches with the empty sequence and $\varphi(\overline{t})$ denotes the result of applying the substitution φ to each term in the sequence $\overline{t}$. Further, in attempting to match with the first two rules, *i.e.*, the ones for *mksubst*, we require that the second argument be head normalized and, similarly, before matching with the next four rules, the second argument of *bnd* should be put in head normal form. Finally, the satisfaction of side conditions that involve rewriting requires also that the results of the rewriting have the forms shown for the righthand sides.

$mksubst(f, \lambda(n, f(\overline{z})), \overline{y}, m) \longrightarrow \{\langle f, \lambda(n+m, h(\overline{w}))\rangle\}$,
where $\overline{w} = \overline{\{m+n-i \mid i \leq n+m \text{ and } (\overline{y}{\uparrow}n)[i] = \overline{z}[i]\}}$ and
h is a new existential variable such that $l(h) = l(f)$.

$mksubst(f, t, \overline{y}, m) \longrightarrow \{\langle f, \lambda(m, s)\rangle\} \circ \theta$,
if the head of t is not f and $bnd(f, t, \overline{y}, 0) \xrightarrow{*} \langle\theta, s\rangle$.

$bnd(f, \lambda(n, t), \overline{y}, l) \longrightarrow \langle\theta, \lambda(n, s)\rangle$,
if $n > 0$ and $bnd(f, t, \overline{y}, l+n) \xrightarrow{*} \langle\theta, s\rangle$.

$bnd(f, a(\overline{t}), \overline{y}, l) \longrightarrow \langle\theta, b(\overline{s}^R)\rangle$,
provided $foldbnd(f, \langle\theta, \varepsilon\rangle, \overline{t}, \overline{y}, l) \xrightarrow{*} \langle\theta, \overline{s}\rangle$ and
either a is a universal variable such that $l(a) \leq l(f)$ and $b = a$
or a appears in $\overline{y}{\uparrow}l$ and $b = a{\downarrow}(\overline{y}{\uparrow}l)$.

$bnd(f, g(\overline{z}), \overline{y}, l) \longrightarrow \langle\{\langle g, \lambda(|\overline{z}|, h(\overline{p}+\overline{u}))\rangle\}, h(\overline{q}+\overline{v})\rangle$,
where h is a new existential variable such that $l(h) = l(f)$,
$\overline{p} = (\overline{y}{\uparrow}l){\Uparrow}g$, $\overline{q} = \overline{p}{\downarrow}(\overline{y}{\uparrow}l)$, and $\overline{u} = \overline{w}{\downarrow}\overline{z}$ and $\overline{v} = \overline{w}{\downarrow}(\overline{y}{\uparrow}l)$ for $\overline{w} = (\overline{y}{\uparrow}l){\cap}\overline{z}$,
provided f and g are distinct existential variables such that $l(f) < l(g)$.

$bnd(f, g(\overline{z}), \overline{y}, l) \longrightarrow \langle\{\langle g, \lambda(|\overline{z}|, h(\overline{q}+\overline{v}))\rangle\}, h(\overline{p}+\overline{u})\rangle$,
where h is a new existential variable such that $l(h) = l(g)$,
$\overline{p} = \overline{z}{\Uparrow}f$, $\overline{q} = \overline{p}{\downarrow}\overline{z}$, and $\overline{v} = \overline{w}{\downarrow}\overline{z}$ and $\overline{u} = \overline{w}{\downarrow}(\overline{y}{\uparrow}l)$ for $\overline{w} = (\overline{y}{\uparrow}l){\cap}\overline{z}$,
provided f and g are distinct existential variables such that $l(g) \leq l(f)$.

$foldbnd(f, \langle\theta, \overline{s}\rangle, \varepsilon, \overline{y}, l) \longrightarrow \langle\theta, \overline{s}\rangle$.

$foldbnd(f, \langle\theta, \overline{s}\rangle, (t_1, \overline{t}), \overline{y}, l) \longrightarrow foldbnd(f, \langle\varphi \circ \theta, (s_1, \overline{s})\rangle, \varphi(\overline{t}), \overline{y}, l)$,
provided $bnd(f, t_1, \overline{y}, l) \xrightarrow{*} \langle\varphi, s_1\rangle$.

Fig. 2. Calculating variable bindings

The rules in Figure 2 are intended for solving an equation of the form $f(\overline{y}) = t$ where f is an existential variable, $\overline{y}$ is a sequence of arguments satisfying the higher-order pattern restriction and t is an arbitrary term. Their usage begins with an attempt to rewrite the expression $mksubst(f, t, \overline{y}, |\overline{y}|)$. The outcome of a successful rewriting will be a substitution with a binding for f. The substitution may also bind other variables: t may contain occurrences of existential variables and the solution to the equation may require that substitutions be made for these as well. Of course, not every such equation will be solved: this may happen, as in Section 3, because t violates the higher-order pattern restriction or because $f(\overline{y})$ and t are not unifiable. Such an effect would be manifest in our system by the inability to rewrite $mksubst(f, t, \overline{y}, |\overline{y}|)$ to a substitution. A failure of this kind would arise, in turn, out of the inability to use any rule to rewrite an intermediate expression either because it does not match with the lefthand side of the rule or because of the violation of a side condition.

We comment briefly on the content of the rules in Figure 2, explaining this with reference to the rules in Figure 1. Borrowing imagery from [2], the substitution for f that solves the equation $f(\overline{y}) = t$ may be viewed as an 'inversion'

(1') $\langle(\lambda(n,t) = \lambda(n,s)) :: E, \theta\rangle \longrightarrow \langle(t = s) :: E, \theta\rangle$, provided $n > 0$.

(2') $\langle(\lambda(n,t) = \lambda(m,s)) :: E, \theta\rangle \longrightarrow \langle((t = \lambda(m-n,s)) :: E), \theta\rangle$,
provided $n > 0$ and $m > n$.

(3') $\langle(a(\overline{t}) = \lambda(m,s)) :: E, \theta\rangle \longrightarrow \langle((a{\uparrow}m)(\overline{t}{\uparrow}m) = s) :: E, \theta\rangle$,
provided a is a de Bruijn index or a universal variable and $m > 0$.

(4') $\langle(a(\overline{t}) = a(\overline{s})) :: E, \theta\rangle \longrightarrow \langle(\overline{t}[1] = \overline{s}[1]) :: \ldots :: (\overline{t}[n] = \overline{s}[n]) :: E, \theta\rangle$,
where $|\overline{t}| = n$, provided a is a de Bruijn index or a universal variable.

(5') $\langle(f(\overline{y}) = t) :: E, \theta\rangle \longrightarrow \langle\varphi(E), \varphi \circ \theta\rangle$
provided f is an existential variable and $mksubst(f, t, \overline{y}, |\overline{y}|) \longrightarrow \varphi$.

Fig. 3. Simplified higher-order pattern unification rules

of t relative to $\overline{y}$. That such an inversion has to be performed is an essential difference from first-order unification and its reflection is the third argument to *mksubst*. Now, the only situation in which the equation with f occurring in t has a solution is when this occurrence is at the head. The first rule for *mksubst* treats this case, leaving all others to be handled by *bnd*. In the treatment of the other cases, actual η-expansion of the flexible term is delayed—this is more efficient and also necessary to treat embedded abstractions. The fourth argument of *bnd* provides the information for this expansion when needed. The first rule for *bnd* reflects this treatment of η-expansion. The second rule treats the case when a rigid structure is encountered in the traversal, building rules (6) and (7) into this process. These rules naturally lead to the examination of the arguments of the rigid term. This process is compiled into the definition of *bnd* and is realized through *foldbnd* that 'maps' *bnd* over the arguments. The third and fourth rules for *bnd* use the idea embodied in rule (9') to treat the situation where a flexible term is encountered. The analysis splits into two cases here because, unlike in rule (9'), we cannot guarantee that $l(f) \leq l(g)$. A final aspect to observe is the effect the delaying of η-expansion has on the last three rules for *bnd*.

Our higher-order pattern unification procedure can be simplified based on the substitution computation rules. The changed set of rules are shown in Figure 3. These rules are to be used in the manner already described and the correctness of the resulting procedure follows from Theorem 2 by formalizing the explanation provided of *mksubst* earlier in this section. We omit the details.

6 Some Aspects Relevant to Actual Implementation

The presentation of the procedure of the last section is still at a high-level and some optimizations are possible in an actual implementation. One such aspect concerns the treatment of the lifting transformation on an argument sequence that is needed in the course of generating a substitution term. Although our presentation suggests that this is done at the relevant points in an eager fashion,

a delayed realization that folds lifting into the specific computation that needs it is usually possible. Thus consider the calculation of the sequence

$$\overline{\{m+n-i \mid i \leq n+m \text{ and } (\overline{y}\uparrow n)[i] = \overline{z}[i]\}}$$

that appears in the first rule for *mksubst*. Finding this sequence requires an iteration over $\overline{z}$ and $\overline{y}\uparrow n$. However, rather than adjusting $\overline{y}$ at the outset, the iteration may be driven by $\overline{z}$ and the lifting operation can be performed on demand. A similar observation also applies to the places where this operation is needed in the definition of *bnd*.

Another observation concerns the computation of the raising and pruning sequences of arguments in the last two rules for *bnd*. The presentation of these rules may suggest that the sequences $\overline{p}$, $\overline{q}$, $\overline{u}$, $\overline{v}$ and $\overline{w}$ are to be calculated separately. In reality, however, the calculation of at least $\overline{p}$ and $\overline{q}$ on the one hand and that of $\overline{u}$, $\overline{v}$ and $\overline{w}$ on the other can be coordinated to yield two essential iterations. The last rule for *bnd* presents an even better situation where the entire computation can be carried out in one iteration assuming that the elements of $\overline{q}+\overline{v}$ can be shuffled so long as the elements of $\overline{p}+\overline{u}$ are shuffled in the same way: we sweep through the elements of $\overline{z}$ determining which ones to keep in $\overline{q}+\overline{v}$ either because of raising or because they also occur in $\overline{y}\uparrow l$ and we simultaneously determine the corresponding elements in $\overline{p}+\overline{u}$.

We have treated substitution non-destructively up to this point. This may be changed in an implementation, thereby obviating the explicit application of substitutions to argument sequences and equation lists. In a different direction, our procedure currently fails when it discovers terms that are not higher-order patterns. This failure may be treated by simply deferring the unification problem. This is probably the best decision at the top-level but a different treatment is possible within the substitution computation process. If it is discovered that t is a flexible term that does not have the structure of a higher-order pattern when attempting to rewrite the expression $bnd(f,t,\overline{y},l)$, a substitution term of the form $h(\overline{y}\uparrow l)$ where h is a new existential variable such that $l(h) = l(f)$ may be returned and the equation $h(\overline{y}\uparrow l) = t$ may be deferred.

7 An Example

We illustrate our procedure relative to the unification problem

$$\exists x \forall a \forall b \forall c \exists y \forall d((b(x(a,d)) = b(\lambda(2,(y(1))))) :: nil).$$

Associating the tags 0 with x, 1 with a, b, c and y and 2 with d allows the quantifier prefix to be eliminated, reducing the problem representation to

$$(b(x(a,d)) = b(\lambda(2,(y(1))))) :: nil.$$

Rule (4') of Figure 3 simplifies this to $(x(a,d) = \lambda(2,(y(1)))) :: nil$ and rule (5') then calls for the rewriting of the expression $mksubst(x, \lambda(2,(y(1))),(a,d),2)$ to a substitution that solves the sole equation in this list.

The rule for *mksubst* in Figure 2 that is relevant to this rewriting task is the second one. This rule requires the expression $bnd(x, \lambda(2, (y(1))), (a, d), 0)$ to be transformed into the form $\langle \theta, s \rangle$ where θ composed with the substitution $\lambda(2, s)$ for x is intended to be a solution to the original equation. The first rule for *bnd* applies to this case, leading to the attempt to rewrite the expression $bnd(x, y(1), (a, d), 2)$; observe that $(a, d) \uparrow 2 = (a, d, 2, 1)$ represents the arguments of x after an η-expansion. The evaluation of this last expression actually represents the heart of the entire calculation. The rule relevant to its rewriting is the third one for *bnd*. Conceptually, one component of this rule determines the arguments of x over which y needs to be raised. This part is given by $((a, d)\uparrow 2)\Uparrow y$ that evaluates to (a). The projection over $(a, d)\uparrow 2$ of this sequence, notated as $(a)\downarrow((a, d)\uparrow 2)$, yields (4) that represents the corresponding part of the substitution term being constructed for x. Another component of the rule finds that part of y's arguments that must not be pruned. This is calculated as $((a, d)\uparrow 2)\cap(1)$, yielding the sequence (1). The projection over the arguments of y and x of this sequence have an identical result, both being given by (1). Combining the two parts gives us the substitution $\lambda(1, h(a, 1))$ for y and the term $h(4, 1)$ that represents the corresponding part of the substitution for x; h is a new existential variable with tag 0 here. Embedding the substitution term for x in the right context eventually yields the substitution $\{\langle x, \lambda(4, h(4, 1))\rangle, \langle y, \lambda(1, h(a, 1))\rangle\}$ that is a most general unifier for the given problem.

It is instructive to contrast the calculation described above with the one that results under a "blind" raising that first moves all existential variables to the outermost level. Such a raising transforms the problem into the form

$$\exists x \exists y' \forall a \forall b \forall c \forall d((b(x(a, d)) = b(\lambda(2, (y'(a, b, c, 1))))) :: nil)$$

by substituting $y'(a, b, c)$ for y. The subsequent substitution generation process must then take responsibility for pruning away the unnecessary arguments introduced during the initial raising. The advantages of our approach become evident from this especially when we note that, in the typical setting, the initial raising must be performed dynamically, the quantifier prefixes can get long and, finally, most of the dependencies that are introduced during the indiscriminate raising eventually have to be pruned away.

Most previously described procedures for higher-order pattern unification assume that existential quantifiers have an outermost scope. The comparison of our ideas with these is therefore obvious. The contrast with the approach presented in [2] is more subtle. In the statement of the problem, this work also assumes that all existential variables are quantified at the outermost level. However, by exploiting properties of explicit substitutions and a special interpretation for instantiatable variables, this requirement can be eliminated and, diverging from the underlying theory, the eventual procedure presented in [2] seems to actually allow for mixed quantifier prefixes. Moreover, this procedure does not explicitly utilize quantifier prefixes, basing its behaviour mainly on the manipulation of explicit substitutions. However, even given this, the computation that results can manifest a character that is akin to redundant raising complemented by pruning. In the particular example considered here, the behaviour will, in fact,

be quite similar to that seen under an initial blind raising. A detailed discussion of this matter requires an exposition also of the explicit substitution approach and is, for this reason, beyond the scope of this paper.

While the above discussion indicates both the significance and the novelty of the ideas in this paper, a sceptical reader might also wish to see an experimental comparison of the different approaches. A study of this kind requires an isolation of just the part to be compared within a larger practical system. This is difficult to realize: the question of whether to maintain quantifier prefixes dynamically or to dispense with them is, for instance, quite fundamental and impacts on several aspects of an implementation. Nevertheless, such a study is important and we are currently examining ways to conduct it even if only in a fragmented manner.

8 Conclusion

We have presented in this paper a procedure that allows for the treatment of higher-order pattern unification in the context of a mixed prefix of quantifiers. Computation within this procedure is based on a recursive traversal of the terms to be unified and only uses information that is available in the course of this traversal. A particular characteristic of our approach is an on-the-fly treatment of raising that exploits the interaction between this transformation and pruning. This procedure has actually been implemented in C and incorporated into the *Teyjus* system. We have also realized these ideas in an SML program [8] that has been used in a meta-proof system built by Alwen Tiu and Dale Miller.

This work can be extended in a few different directions. First, there are similarities in the dynamics of our procedure and the one presented in [2] even though we have not utilized explicit substitutions. We would like to understand these connections better since this is likely to shed light on the question of whether the explicit substitutions based approach of [2] is really useful in the higher-order pattern unification setting. In a different direction, we have found it beneficial to employ explicit substitutions implicitly in reduction procedures [4] and we would like to extend this approach also to the unification context. Finally, higher-order pattern unification offers promising possibilities for compilation. Some work has already been done on this topic [17] and we also have recently started a systematic study oriented around a redesign of the λProlog abstract machine that exploits the procedure of this paper.

9 Acknowledgements

This work began while the first author was on a sabbatical visit to the Protheo group at LORIA and INRIA, Nancy and the Comete and Parsifal groups at Ecole Polytechnique and INRIA, Saclay. Support has also been derived from the NSF through a Graduate Fellowship and the grant numbered CCR-0429572 and from the Digital Technology Center at the University of Minnesota.

References

1. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157:183–235, 2000.
2. G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case for higher-order patterns. Technical Report 3591, INRIA, December 1998.
3. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
4. C. Liang, G. Nadathur, and X. Qi. Choices in representation and reduction strategies for lambda terms in intensional contexts. *Journal of Automated Reasoning*, 33:89–132, 2005.
5. S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Conference Record of the Workshop on the λProlog Programming Language*, Philadelphia, July-August 1992.
6. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
7. D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.
8. G. Nadathur and N. Linnell. An SML implementation of higher-order pattern unification, January 2004. Source code distributed over the web via the URL `http://www.cs.umn.edu/~gopalan/code/ho-pattern-unif.tar.gz`.
9. G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, August 1988.
10. G. Nadathur and D.J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in LNAI, pages 287–291. Springer, 1999.
11. T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349. IEEE Press, 1991.
12. T. Nipkow. Functional unification of higher-order patterns. In *Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, June 1993.
13. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
14. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.
15. F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, 1991.
16. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in LNAI, pages 202–206. Springer, July 1999.
17. B. Pientka and F. Pfenning. Optimizing higher-order pattern unification. In F. Baader, editor, *Proceedings of the 19th Conference on Automated Deduction (CADE-19)*, LNCS 2741, pages 473–487. Springer-Verlag, July 2003.
18. Z. Qian. Unification of higher-order patterns in linear time and space. *Journal of Logic and Computation*, 6(3):315–341, 1996.