Linear Unification of Higher-Order Patterns

Zhenyu Qian*
Universität Bremen

Abstract

Higher-order patterns are simply typed λ-terms in η-long form where free variables F only occur in the form $F(x_1, \cdots, x_k)$ with $x_1, \cdots, x_k$ being distinct bound variables. It has been proved in [6] that in the simply typed λ-calculus unification of higher-order patterns modulo α, β and η reductions is decidable and unifiable higher-order patterns have a most general unifier.

In this paper a unification algorithm for higher-order patterns is presented, whose time and space complexities are proved to be linear in the size of input.

1 Introduction

Lambda calculi are suitable frameworks for succinctly representing logical languages with bound variables. This is not only because they can be directly and intuitively used in encoding logical terms and formulae, but also because some of them have been turned into computational realities, e.g. in the logic programming languages λProlog [8] and Elf [14], and in the generic theorem prover Isabelle [12], due to the pioneer work on unification of simply typed λ-terms by Huet [3].

However, unification of simply typed λ-terms is a complex operation since the problem is in general undecidable [2], and unifiable terms may have infinite independent unifiers. Even when one is only interested in the existence of unifiers, where Huet's insight was that terms with free variables at heads are always unifiable and thus need not be further dealt with [3], the unification process could be expensive in time and space because it may be nondeterministic and nonterminating. This has led to search for special classes of λ-terms on which unification is decidable and unifiable terms has a most general unifier.

Miller discovered such a class and gave a unification algorithm to compute the most general unifiers [6]. The terms in the class are those where certain restrictions are placed on occurrences of free variables. Nipkow used the results in the context of higher-order rewriting, reformulated the unification algorithm [9] and presented a functional program for the unification [10]. Pfenning adapted the results for his logic programming language Elf [13] and extended them to the Calculus of Constructions [15]. We follow Nipkow and call these terms *higher-order patterns* (short: *patterns*).

Unification of patterns is a proper compromise of the full unification of simply typed λ-terms [7]: Taking a higher-order logic programming language L_λ based on patterns as proposed in [6], the full unification of simply typed λ-terms can be coded as a L_λ program axiomatizing only the notions of equality and substitution of simply typed λ-terms in a direct and declarative way. The rest of the full unification is addressed by unification of patterns and a backtracking strategy implemented in a L_λ interpreter.

However, unification of patterns may be extremely inefficient in its worst case since so may be unification of first-order terms ([11, 4]) and first-order terms are just special patterns. The purpose of this paper is to tackle unification of patterns from a computational complexity point of view. More precisely, an algorithm for unification of patterns is proposed

*FB Informatik, Universität Bremen, 2800 Bremen 33, Germany. E mail: qian@informatik.uni-bremen.de
Research partially supported by ESPRIT Basic Research WG *COMPASS* 6112.

whose time and space complexities are both linear in the size of the input. The structure of our algorithm has some similarities with Martelli and Montanari's one [4, 5], but ours is not a direct extension of theirs. The features needed in our machine are manipulation of pointers, comparison of labels and random access of array components.

The rest of this paper is organized as follows: In Section 2 basic notions of λ-calculus are reviewed and unification of patterns is presented at a fairly high level. In Section 3 some problems with extending the first-order linear unification algorithms in [11, 4, 5] to unification of patterns are discussed. A linear algorithm for unification of patterns is proposed in Section 4, and its linearity is proved in Section 5. We conclude in Section 6.

2 Preliminaries

Given a set $\mathcal{T}_0$ of *base types*, the set $\mathcal{T}$ of (*simple*) *types* is constructed as usual. For every type $\alpha \in \mathcal{T}$ there exist a set $\mathcal{C}_\alpha$ of *constants* and a countably infinite set $\mathcal{V}_\alpha$ of *variables* such that $\mathcal{C}_{\alpha_1} \cap \mathcal{V}_{\alpha_2} = \{\}$ for any $\alpha_1, \alpha_2 \in \mathcal{T}$ and $\mathcal{C}_{\alpha_1} \cap \mathcal{C}_{\alpha_2} = \mathcal{V}_{\alpha_1} \cap \mathcal{V}_{\alpha_2} = \{\}$ if $\alpha_1 \neq \alpha_2$. Let $\mathcal{C} = \bigcup_{\alpha\in\mathcal{T}} \mathcal{C}_\alpha$ and $\mathcal{V} = \bigcup_{\alpha\in\mathcal{T}} \mathcal{V}_\alpha$. Constants and variables are also called *atoms*.

The set $\mathcal{L}_\alpha$ of *terms of type* $\alpha \in \mathcal{T}$ is constructed as usual. Let $\mathcal{L} = \bigcup_{\alpha\in\mathcal{T}} \mathcal{L}_\alpha$. A term of the form $(s\ t)$ is called an *application*, and $\lambda x.s$ an *abstraction*. The topmost part λx in $\lambda x.t$ is called a *λ-binder* of x, and the term t is said to be *covered* by or *in the scope* of λx. *Free* and *bound* variable occurrences are defined as usual. We use the following abbreviations: $\lambda\overline{x_n}.s$ stands for $\lambda x_1 \ldots \lambda x_n.s$; $a(\overline{u_n})$ stands for $(\ldots((a\ u_1)\ u_2)\ldots u_n)$.

The *size of a term* t is defined as the total number of occurrences of atomic subterms and λ-binders in t.

Terms are only compared modulo α-conversion. Thus we assume from now on that in a term no variable is bound more than once and no variable occurs both bound and free. The set of all bound (or free) variables in a syntactic object O is denoted by $\mathcal{BV}(O)$ (or $\mathcal{FV}(O)$).

Reductions on terms are the usual β and η-reductions, denoted by $\longrightarrow_\beta$ and $\longrightarrow_\eta$, resp. Define $\longrightarrow_{\beta\eta}$ as $\longrightarrow_\beta \cup \longrightarrow_\eta$. Let $\mathcal{X} \in \{\beta, \eta, \beta\eta\}$. We use $\longrightarrow_\mathcal{X}^*$ to denote the reflexive and transitive closure, and $=_\mathcal{X}$ the equivalence relation induced by $\longrightarrow_\mathcal{X}$.

Every term s can be $\mathcal{X}$-reduced to a unique $\mathcal{X}$-normal form $s{\downarrow_\mathcal{X}}$. A term t is a β-normal form if and only if t is of the form $\lambda\overline{x_k}.a(\overline{t_n})$ with $a \in \mathcal{C} \cup \mathcal{V}$ and each t_i being a β-normal form. The atom a is called the *head*. Let $t = \lambda\overline{x_k}.a(\overline{t_n})$ be a β-normal form as above. Then t is called *flexible* if $a \in \mathcal{FV}(t)$, *rigid* if not. Furthermore, t is called an *η-long form* if $a(\overline{t_n})$ is of a base type and each t_i is a η-long form. Every term s has a unique η-long form $s{\downarrow_{l\eta}}$ such that $s{\downarrow_{l\eta}} \longrightarrow_\eta^* s{\downarrow_\beta}$. For single variable x, $x{\downarrow_{l\eta}}$ may still be written as x.

A *substitution* θ is defined as usual and denoted by $\{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ or $\{\overline{x_n \mapsto t_n}\}$. The *domain* of θ is $\mathcal{D}om(\theta) = \{x_1, \cdots, x_n\}$, and the *range* $\mathcal{R}an(\theta) = \{t_1, \cdots, t_n\}$. It is always assumed that before applying θ to a term s in $\theta(s)$, all bound variables in s have been α-converted so that $\mathcal{BV}(s) \cap \mathcal{FV}(\mathcal{R}an(\theta)) = \{\}$. It is also assumed that $\mathcal{BV}(s) \cap \mathcal{D}om(\theta) = \{\}$, hence $\theta(\lambda x.s) = \lambda x.\theta(s)$ holds automatically.

The *restriction* of a substitution θ to a set $\mathcal{W}$ of variables is a substitution $\theta_{|\mathcal{W}} = \{x \mapsto \theta(x) \mid x \in \mathcal{W}\}$. Let σ and θ be substitutions. Then the *composition* $\sigma \circ \theta$ is a substitution defined by $(\sigma \circ \theta)(x) = \sigma(\theta(x))$ for every $x \in \mathcal{V}$. Let $\sigma =_{\beta\eta} \theta$ denote $\sigma(x) =_{\beta\eta} \theta(x)$ for each $x \in \mathcal{V}$. Then σ is said to be more general than θ, denoted as $\sigma \leq_{\beta\eta} \theta$, if there is a substitution σ' such that $\sigma' \circ \sigma =_{\beta\eta} \theta$.

A substitution θ is called a *unifier* of terms s and t if $\theta(s) =_{\beta\eta} \theta(t)$. In this case s and t are said to be *unifiable*. Define a *unification pair* $s =^? t$ as an unordered pair of terms s and t of the same type. A *unification problem* P is a finite multiset of unification pairs. Two unification problems are said to be *equivalent* if they have the same unifiers.

For $\theta = \{\overline{x_n \mapsto t_n}\}$, define $[\theta] = \{\overline{x_n =^? t_n}\}$. Obviously θ is a most general unifier of $[\theta]$. If $\mathcal{D}om(\sigma) \cap \mathcal{D}om(\theta) = \{\}$ then $[\sigma \circ \theta]$ and $[\sigma] \cup [\theta]$ are equivalent.

In the sequel, we use s, t, u and v to denote terms, a and b atoms, c, d and f constants, x, y and z bound variables, X, Y, Z, F, G and H free variables, σ and θ substitutions.

2.1 Higher-order patterns and higher-order pattern unification

Higher-order patterns (short: *patterns*) are η-long forms in which free variables F only occur in the form $F(\overline{x_k})$ with $x_1, \cdots, x_k$, $k \geq 0$, being distinct bound variables. For example, the terms $\lambda xyz.F(y,x)$ and $\lambda xy.y(\lambda z.F(z,y), F(x,y), G(y))$ are patterns, provided that they are η-long forms, whereas $\lambda x.F(c,x)$, $\lambda xy.F(x,x)$ and $\lambda x.F(G(x))$ are not patterns.

Lemma 2.1 *([6]) It is decidable whether two arbitrary patterns are unifiable. Furthermore, a substitution whose range contains only patterns can always be computed as the most general unifier of two arbitrary unifiable patterns.*

From now on we only consider patterns and substitutions whose ranges contain only patterns. Note that for such a substitution σ if t is a pattern then so is $\sigma(t)\downarrow_\beta$.

We represent here an algorithm by Nipkow [9]. The algorithm is given by the following five transformation rules on pairs of substitutions and unification problems, where unification problems are viewed as lists instead of multisets of unification pairs and @ denotes the concatenation operation of lists. The algorithm starts with the pair $\langle\{\}, P\rangle$ for any unification problem P and terminates with $\langle\sigma, \{\}\rangle$ if P is unifiable, in which case σ is the most general unifier of P. It is assumed that a unification pair is always automatically α-converted so that both sides always have the same sequence of outermost λ-binders.

Rule (Rep) propagates solutions.

$$\langle\sigma, \{\lambda\overline{x_k}.F(\overline{x_k}) \stackrel{?}{=} t\}@P\rangle \implies \langle\sigma' \circ \sigma, \sigma'(P)\downarrow_\beta\rangle \qquad \text{(Rep)}$$

if $F \notin \mathcal{FV}(t)$, where $\sigma' = \{F \mapsto t\}$.

Rule (Dec) breaks a unification pair into simpler ones.

$$\langle\sigma, \{\lambda\overline{x_k}.a(\overline{s_n}) \stackrel{?}{=} \lambda\overline{x_k}.a(\overline{t_n})\}@P\rangle \implies \langle\sigma, \{\overline{\lambda\overline{x_k}.s_n \stackrel{?}{=} \lambda\overline{x_k}.t_n}\}@P\rangle \qquad \text{(Dec)}$$

if $a \in C \cup \{\overline{x_k}\}$.

Rule (Bin) finds a partial binding for a head variable.

$$\langle\sigma, \{\lambda\overline{x_k}.F(\overline{y_n}) \stackrel{?}{=} \lambda\overline{x_k}.a(\overline{t_m})\}@P\rangle \implies \langle\sigma' \circ \sigma, \{\overline{\lambda\overline{x_k}.H_m(\overline{y_n}) \stackrel{?}{=} \lambda\overline{x_k}.t_m}\}@\sigma'(P)\downarrow_\beta\rangle \qquad \text{(Bin)}$$

if $F \notin \mathcal{FV}(\overline{t_m})$ and $a \in C \cup \{\overline{y_n}\}$, where $H_1, \cdots, H_m$ are new variables and $\sigma' = \{F \mapsto \lambda\overline{y_n}.a(\overline{H_m(\overline{y_n})})\}$.

Rule (FF-1) finds a unifier of a unification pair of flexible terms with distinct heads.

$$\langle\sigma, \{\lambda\overline{x_k}.F(\overline{y_n}) \stackrel{?}{=} \lambda\overline{x_k}.G(\overline{z_m})\}@P\rangle \implies \langle\sigma' \circ \sigma, \sigma'(P)\downarrow_\beta\rangle \qquad \text{(FF-1)}$$

if F and G are distinct free variables, where $\sigma' = \{F \mapsto \lambda\overline{y_n}.H(\overline{v_p}), G \mapsto \lambda\overline{z_m}.H(\overline{v_p})\}$, $\{v_1, \cdots, v_p\} = \{\overline{y_n}\} \cap \{\overline{z_m}\}$ and H is a new free variable.

Rule (FF-2) finds a unifier of a unification pair of flexible terms with the same head.

$$\langle\sigma, \{\lambda\overline{x_k}.F(\overline{y_n}) \stackrel{?}{=} \lambda\overline{x_k}.F(\overline{z_n})\}@P\rangle \implies \langle\sigma' \circ \sigma, \sigma'(P)\downarrow_\beta\rangle \qquad \text{(FF-2)}$$

where $\sigma' = \{F \mapsto \lambda\overline{y_n}.H(\overline{v_p})\}$, $\{\overline{v_p}\} = \{y_i \mid y_i = z_i, 1 \leq i \leq n\}$ and H is a new free variable.

Inversing the preconditions to the above rules yields the following failure cases. The first case may be described as $\lambda\overline{x_k}.a(\overline{s_n}) =^? \lambda\overline{x_k}.b(\overline{t_m})$, where $a, b \in C \cup \{\overline{x_k}\}$ with $a \neq b$, and is called *clash*. The second and third cases may be described as $\lambda\overline{x_k}.F(\overline{y_n}) =^? \lambda\overline{x_k}.a(\overline{s_m})$, where $F \in \mathcal{FV}(\overline{s_m})$ or $a \in \{\overline{x_k}\} - \{\overline{y_n}\}$. The case $F \in \mathcal{FV}(\overline{s_m})$ is called *cycle* and the case $a \in \{\overline{x_k}\} - \{\overline{y_n}\}$ *bound variable capture*.

Theorem 2.2 *([9])There are no infinite sequences of transformations by the above rules. A unification problem P is unifiable if and only if every sequence of transformations starting with $\langle\{\}, P\rangle$ terminates with $\langle\sigma, \{\}\rangle$, in which case $\sigma_{|\mathcal{FV}(P)}$ is a most general unifier of P.*

In the first-order case, rules (Rep), (Bin) and (FF-1) degenerate into rule (Rep'), and rules (Dec) and (FF-2) into (Dec') as follows:

$$\langle \sigma, \{F \stackrel{?}{=} t\} \cup P \rangle \Longrightarrow \langle \sigma' \circ \sigma, \sigma'(P) \rangle \qquad \text{(Rep')}$$

if $F \notin \mathcal{FV}(t)$, where $\sigma' = \{F \mapsto t\}$.

$$\langle \sigma, \{a(\overline{s_n}) \stackrel{?}{=} a(\overline{t_n})\} \cup P \rangle \Longrightarrow \langle \sigma, \{\overline{s_n \stackrel{?}{=} t_n}\} \cup P \rangle \qquad \text{(Dec')}$$

if $a \in \mathcal{A}$.

Most first-order unification algorithms can be derived from rules (Rep') and (Dec'). In particular, the linear unification algorithms by Paterson and Wegman ([11]) and by Martelli and Montanari ([4, 5]) can be derived from these two rules, where terms are represented by *directed acyclic graphs* (short: *DAGs*), unification problems may be re-organized by the following merging rule

$$\{s \stackrel{?}{=} t, s \stackrel{?}{=} u\} \cup P \Longrightarrow \{s \stackrel{?}{=} t, t \stackrel{?}{=} u\} \cup P$$

and the unification pair, to which rule (Rep') is applied at each stage, is always so selected that no substitutions have to be applied to the rest unification problem.

3 Problems with Extending the First-Order Linear Unification Algorithms

In this section we discuss some problems with extending the linear unification algorithms in [11, 4, 5] to higher-order patterns.

The first problem is due to the propagation of outermost λ-binders in rule (Dec). For example, a unification pair $\lambda\overline{x_k}.f(\overline{a_k}) =^? \lambda\overline{x_k}.f(\overline{b_k})$ of length $O(k)$ may be transformed into $\{\lambda\overline{x_k}.a_1 =^? \lambda\overline{x_k}.b_1, \cdots, \lambda\overline{x_k}.a_k =^? \lambda\overline{x_k}.b_k\}$ of length $O(k^2)$. Making a copy of the λ-binders $\lambda\overline{x_k}$ for each subproblem $s_i =^? t_i$ would lead to at least quadratic space complexity.

Our solution to the problem is to require that whenever rule (Dec) is applied, it should be applied repeatedly to each newly yielded unification pair of two rigid terms. Not all old outermost λ-binders, but only those that do bind some occurrences of bound variables in a flexible term need to be carried over to a final resulting unification pair. All occurrences of bound variables that do not occur in a flexible term have to be eliminated in the further unification process anyway; otherwise the original problem is not unifiable.

For example, unifying $\lambda\overline{x_k}.g(f(F(x_1)), c_1, \cdots, c_k) =^? \lambda\overline{y_k}.g(f(y_2), c_1, \cdots, c_k)$ should directly yield $\lambda x_1.F(x_1) =^? \lambda y_1, y_2$. The unnecessary intermediate unification problem

$$\{\lambda\overline{x_k}.f(F(x_1)) \stackrel{?}{=} \lambda\overline{y_k}.f(y_2), \lambda\overline{x_k}.c_1 \stackrel{?}{=} \lambda\overline{y_k}.c_1, \cdots, \lambda\overline{x_k}.c_k \stackrel{?}{=} \lambda\overline{y_k}.c_k\}$$

will not be really created at all. Note that now the total size of all unification pairs yielded is always linear in the size of the original unification pair. For, each newly created sequence of λ-binders corresponds uniquely to a flexible subterm in the original unification pair and is shorter than the flexible subterm. Continue the unification process for $\lambda x_1.F(x_1) =^? \lambda y_1.y_2$ in the above. Since the occurrence of y_2 is not covered by a corresponding λ-binder, a failure will arise. Intuitively, the failure corresponds to a bound variable capture in unifying $\lambda\overline{x_k}.F(x_1) =^? \lambda\overline{y_k}.y_2$.

The second problem is due to the time required for renaming subterms of bound variables possibly needed before a merging step. For example, in order to merge the unification pairs $\lambda\overline{x_k}.F(\overline{x'_k}) =^? \lambda\overline{x_k}.s$ and $\lambda\overline{x_k}.F(\overline{y'_k}) =^? \lambda\overline{x_k}.t$, the second unification pair may first have to be converted into an equivalent one $\lambda\overline{x_k}.F(\overline{x'_k}) =^? \lambda\overline{x_k}.\phi(t)$ with $\phi = \{\overline{y'_k \mapsto x'_k}\}$. Now a merging step may be performed and result in $\lambda\overline{x_k}.F(\overline{x'_k}) =^? \lambda\overline{x_k}.s$ and $\lambda\overline{x_k}.s =^? \lambda\overline{x_k}.\phi(t)$. The renaming operation in $\phi(t)$ is a possible source of nonlinear behaviors, since an occurrence of a bound variable in t may be involved in many merging steps in the entire unification process and thus need to be renamed many times. Consider the unification problem

$$P_1 = \{\lambda xy.y(F(x,y), F(y,x)) \stackrel{?}{=} \lambda xy.s_1\},$$

where $s_i = y(x(G_i(x,y), G_i(y,x)), s_{i+1}), i = 1, \ldots, n-1$, and $s_n = y(x(G_n(x,y), G_n(y,x)), y(c,c))$. Obviously, P_1 has the size $O(n)$. By rule (Dec), P_1 may be transformed into

$$\{\lambda xy.F(x,y) \stackrel{?}{=} \lambda xy.x(G_1(x,y), G_1(y,x)), \lambda xy.F(y,x) \stackrel{?}{=} \lambda xy.s_2\}.$$

Now $\lambda xy.F(y,x) =^? \lambda xy.s_2$ may be converted into $\lambda xy.F(x,y) =^? \lambda xy.t_2$, where t_2 denotes $\{x \mapsto y, y \mapsto x\}(s_2)$, and the resulting unification pairs may be merged into

$$\{\lambda xy.F(x,y) \stackrel{?}{=} \lambda xy.x(G_1(x,y), G_1(y,x)), \lambda xy.x(G_1(x,y), G_1(y,x)) \stackrel{?}{=} \lambda xy.t_2\}.$$

The first unification pair may be solved by rule (Rep), where no substitution need to be applied to the rest unification problem

$$P_2 = \{\lambda xy.x(G_1(x,y), G_1(y,x)) \stackrel{?}{=} \lambda xy.t_2\}.$$

Continue the unification process with P_2 in the same way as above, it is easy to see that the total number of renaming x and y in the entire unification process is $O(n^2)$.

The merging step in the above example is also necessary when terms are denoted by De Bruijn's representations. So a naive extension of the first-order linear unification algorithms with De Bruijn's representations of patterns has at least quadratic time complexity.

Our solution to the problem is to avoid renaming subterms of bound variables whenever possible. Indeed, such renaming operations would be unnecessary if no attempts are made to keep both sides of a unification pair have the same sequence of outermost λ-binders. In merging $\lambda\overline{x_k}.F(\overline{x'_k}) =^? \lambda\overline{x_k}.s$ and $\lambda\overline{x_k}.F(\overline{y'_k}) =^? \lambda\overline{x_k}.t$, since $\lambda\overline{x_k}.\phi(t) = \lambda\overline{\phi^{-1}(x_k)}.t$ by α-conversion, where $\phi = \{\overline{y'_k \mapsto x'_k}\}$, the second unification pair may also be converted into $\lambda\overline{x_k}.F(\overline{x'_k}) =^? \lambda\phi^{-1}(\overline{x_k}).t$. Due to our solution to the first problem, the size of a sequence of outermost λ-binders, i.e. $\lambda\overline{x_k}$ in this case, is linear in the size of a flexible term, i.e. $F(\overline{x'_k})$ in this case. Thus the time for renaming the sequence, i.e. $\phi^{-1}(\overline{x_k})$ in this case, may be linear in the size of the flexible term. Since a merging step is only applied when the free variable in the flexible term will be solved by rule (Rep) directly afterwards, the flexible term cannot be charged for time expenses in renaming other sequences of outermost λ-binders. Since the flexible term is a subterm in the original unification problem, our solution may be linear.

Now a unification pair may have terms with different sequences of outermost λ-binders. So subterms of bound variables can only be compared modulo α-conversion. This means that for example, in unifying $\lambda\overline{x_k}.x(\cdots) =^? \lambda\overline{y_k}.y(\cdots)$, we need to check whether the λ-binders λx and λy occur at the same position in $\lambda\overline{x_k}$ and $\lambda\overline{y_k}$, resp. Thus each sequence of outermost λ-binders should be implemented by a data structure, where the position of a given λ-binder can be computed in a constant time. However, maintaining such a data structure for each sequence of outermost λ-binders in the entire unification process is nonlinear, since λ-binders may be discarded, and since the same (named) λ-binders may occur in different sequences.

Our solution is to delay the checking operations: Instead of performing a checking operation when and where required, all conditions to be checked are first collected. Afterwards, all bound variables in the collected conditions are replaced by their positions in the corresponding sequences of outermost λ-binders. If some bound variables that should be α-equivalent are replaced by different positions, a failure arises. In fact, this kind of failures correspond to clashes in the original unification problem.

4 A Linear Unification Algorithm for Higher-Order Patterns

4.1 Representations of patterns, multiequations and systems of multiequations

The linearity of first-order unification heavily depends on the DAG representation of terms where only one data structure is dynamically created for a variable and all occurrences of the variable are implemented as pointers to the data structure. We do the similar things for

free variables in patterns. The difference is that now a free variable may have arguments of bound variables. Formally, a pattern can be representated as a DAG such that

1. each occurrence of an atom corresponds to a node labeled with the name of the atom and having no out-arcs,
2. each application $a(\overline{t_n})$ corresponds to a node labeled with the special symbol @ and having $n+1$ ordered out-arcs going to the nodes corresponding to $a, t_1, \cdots, t_n$, resp.,
3. each abstraction $\lambda x.t$ corresponds to a node labeled with λ and having 2 ordered out-arcs leading to the nodes corresponding to x and t,
4. different occurrences of subterms correspond to different nodes, except that occurrences of the same free variable correspond to the same node.

Obviously the root node has no father nodes, and any other node that does not correspond to a free variable has exactly one father node.

We extend the notions of *multiequation* and of *system of multiequations* in [4] as follows: A *multiequation* is a pair $U =^? M$ of a nonempty set U of flexible patterns and a set M of patterns. Usually, patterns in M are rigid. A multiequation $\{s\} =^? M$ may be written as $s =^? M$, and $U =^? \{t\}$ as $U =^? t$. A multiequation $U =^? M$ denotes a unification problem $\{s =^? t \mid t \in U \cup M\}$ where s is some term in U. A *system of multiequations* is defined as (T, S) with T being a sequence of multiequations, called the *solved* part, and S a set of multiequations, called the *unsolved* part, such that

1. the right-hand side of each multiequation in T and S contains at most one term,
2. the left-hand side of each multiequation in T consists of one flexible term, and the right-hand side, if nonempty, consists of a term of the same type,
3. the free variable in the left-hand side of a multiequation in T may only occur in the right-hand sides of preceding multiequations in T and nowhere else.

A DAG representation of a system (T, S) can be obtained by creating a node for each multiequation with out-arcs to all its terms, a node for T (and a node for S) with out-arcs to all its multiequations, and a node for (T, S) with two out-arcs to T and S, resp., where all nodes of the same free variable are identified.

Intuitively, a system of multiequations corresponds to a pair of a substitution and a unification problem in Subsection 2.1. For any given set S_0 of multiequations, our algorithm starts with a system $(\langle\rangle, S_0)$, attempts to transfer multiequations from the unsolved to solved part while preserving the unifiers of the whole system and terminates with a system $(T_f, \{\})$ if S_0 is unifiable.

In the first-order case [4], a multiequation $U =^? M$ in S may be moved to T if no free variables in U occur in $S - \{U =^? M\}$. The same thing cannot be directly done in higher-order case, since some information about free variables in M may be lost. Consider

$$S = \{\lambda xy.F(x) \stackrel{?}{=} \lambda xy.a(G(x,y))\} \cup S'$$

with $F \notin \mathcal{FV}(S')$ as an example. Intuitively the multiequation should not be directly moved to T since it has a left-hand side not equal to F and thus is not directly a substitution of F.

The nature of the problem is that a unifier θ of $\lambda xy.F(x) =^? \lambda xy.a(G(x,y))$ maust satisfy that y does not occur in $\theta(G)(x,y)\downarrow_\beta$. In fact, we may add an additional multiequation $\lambda xy.G(x,y) =^? \lambda xy.H(x)$ to the above S, where H is a new free variable. By rules (Bin) and (FF-1) in Subsection 2.1, the resulting S is equivalent to the original one when restricted to

the original free variables. Now the multiequation $\lambda xy.F(x) =^? \lambda xy.a(G(x,y))$ can be moved from S to T since the information that would have been lost in the above is kept by the additional multiequation. Note that the name of H is immaterial. Therefore we may write $\lambda xy.G(x,y) =^? \lambda xy.H(x)$ as $\lambda y[\lambda x.G(x,y) =^? \{\}]$, where the λy outside the square bracket $[\cdots]$ means that no substitutions of $\lambda x.G(x,y)$ in the unification process may contain y.

In general, the form $\lambda\overline{x_k}.[U =^? M]$ denotes two multiequations

$$\{\lambda\overline{x_k}.s \mid s \in U\} \stackrel{?}{=} \{\lambda\overline{x_k}.t \mid t \in M\} \quad \text{and} \quad \lambda\overline{y_n}.F(\overline{y_n}) \stackrel{?}{=} \lambda\overline{y_n}.H(\overline{z_m})$$

where $F(\overline{y_n})$ is a subterm in U and $\{\overline{z_m}\} = \{\overline{y_n}\} - \{\overline{x_k}\}$. Note that the ordering of λ-binders in $\lambda\overline{x_k}$ may be arbitrary. The form $\lambda\overline{x_k}.[U =^? M]$ can be simplified into $U =^? M$ if all subterms of each x_i in U and M can be recognized as occurrences of bound variable on its own. In this paper bound variables are denoted by lower-case letters x, y, z.

In the sequel, we allow the above extended multiequations in systems of multiequations. The outermost λ-binders in U or M are sometimes called *remaining λ-binders*.

In the above example, the multiequation $\lambda xy.F(x) =^? \lambda xy.a(G(x,y))$ is first rewritten into $\lambda x.F(x) =^? \lambda x.a(G(x,y))$ and then moved to the solved part, while an additional $\lambda x.G(x,y) =^? \{\}$ is created in the unsolved part.

Assume $T_f = (\overline{U_m =^? M_m})$ in a system $(T_f, \{\})$. Let θ_i, $i = 1, \cdots, m$, be defined as

$$\begin{array}{ll} \theta_m = \{F \mapsto t\} & \text{if } U_m = \{\lambda\overline{x_k}.F(\overline{x_k})\} \text{ and } M_m = \{t\} \\ \theta_i = \{\} & \text{if } M_i = \{\} \\ \theta_i = \{F \mapsto \theta_{i+1}(t)\downarrow_\beta\} & \text{if } U_i = \{\lambda\overline{x_k}.F(\overline{x_k})\} \text{ and } M_i = \{t\}. \end{array}$$

Then a most general unifier of T_f can be constructed as $\theta = \theta_1 \cup \cdots \cup \theta_m$. In fact, T_f is a factorized form of θ.

From a given unification pair $\lambda\overline{x_k}.s =^? \lambda\overline{x_k}.t$ a set of multiequations can be constructed as follows: If at least one of $\lambda\overline{x_k}.s$ and $\lambda\overline{x_k}.t$ is flexible then the set consists of one multiequation $U =^? M$, where $U = \{u \in \{\lambda\overline{x_k}.s, \lambda\overline{x_k}.t\} \mid u \text{ is flexible}\}$ and $M = \{\lambda\overline{x_k}.s, \lambda\overline{x_k}.t\} - U$; Otherwise the set consists of two multiequations $U_1 =^? M_1$ and $U_2 =^? M_2$, where $U_1 = U_2 = \{\lambda\overline{x_k}.H(\overline{x_k})\}$ with H being a new free varviable, $M_1 = \{\lambda\overline{x_k}.s\}$ and $M_2 = \{\lambda\overline{x_k}.t\}$. For a unification problem P, the starting set S_0 of multiequations consists of the multiequations constructed from all unification pairs in P. Obviously P and S_0 are equivalent when restricted to $\mathcal{FV}(P)$.

4.2 The algorithm

Our algorithm *Unify* is given below. At each stage, when S is nonempty, the algorithm first selects a set MS of multiequations from S, where no free variables in the left-hand sides of the multiequations in MS occur in $S - MS$ or in the right-sides in MS (line 8). Note that no resulting substitutions of unifying MS need to be applied to the rest of the system. A stack ST is used to keep the sets of multiequations that have been looked at but do not satisfy the condition required in the above. To unify MS the algorithm first merges all multiequations in MS into a multiequation $U =^? M$ (line 9). Then flexible terms in U are unified and the results are put into T (line 10), where B is a set consisting of the original positions of all remaining λ-binders in U after unification. Note that now U contains one term. If M is empty, then nothing has to be done; Otherwise the terms in M are unified w.r.t. the information in B (line 11), and, when no failure arises, a multiequation as a fragment of the final solution with U on the left and the common top layer of all terms in M on the right will be obtained and put into T, and a set of smaller multiequations as a part of the rest unification problem is created and put into S and ST. All sets in the algorithm are implemented as lists, so that sometimes we may talk about "the first (or last) element" in a set. Procedure parameters of complex data structures like lists are implemented as pointers to these data structures.

(1) **Algorithm** *Unify*
(2) **input** A system of multiequations $(\langle\rangle, S)$;
(3) **output** A system of multiequations $(T, \{\})$;
(4) **begin**
(5) Create an empty stack ST;
(6) **while** S is nonempty **do**
(7) **begin**
(8) *Select*(ST, S, MS);
(9) *Merge*$(MS, U =^? M)$;
(10) *UnifyL*(U, T, B);
(11) **if** M is nonempty **then** *UnifyR*(U, M, B, T, S, ST)
(12) **end**
(13) **end**

The procedure *Select* is given in the appendix. It can be seen as an easy extension of its first-order counterpart in [5]. For more details see also [16].

4.3 Merge the selected multiequations

The procedure *Merge*$(MS, U =^? M)$ merges all multiequations in MS into a multiequation $U =^? M$. To begin with, all flexible terms in each left-hand side in MS are α-converted so that they have the same outermost λ-binders (line B-3). Then the procedure takes an arbitrary multiequation in MS as the first $U =^? M$, and starts to merge all other multiequations $U' =^? M'$ in MS one by one into the current $U =^? M$.

First, all left-hand sides are merged. For simplicity, we may factorize the common remaining λ-binders and write $\{\lambda\overline{x_k}.t_1, \cdots, \lambda\overline{x_k}.t_n\}$ as $\lambda\overline{x_k}.\{t_1, \cdots, t_n\}$. Let $U = \lambda\overline{x_k}.V$ and $U' = \lambda\overline{y_m}.V'$ with $F(\overline{x'_n}) \in V$ and $F(\overline{y'_n}) \in V'$. Then a mapping $\phi = \{\overline{y'_n \mapsto x'_n}\}$ may be created (line B-7). Note that $x'_i \notin \{\overline{x_n}\}$ or $y'_i \notin \{\overline{y_m}\}$ means x'_i or y'_i, resp., has been required to be eliminated somewhere. We may compute the common remaining λ-binders for both multiequations modulo α-conversion (line B-8). Now U' may be α-converted and merged into U (line B-9), where it is assumed that all bound variables in V' that are not in $\{\overline{y'_n}\}$ do not occur in V; otherwise we may add an additional renaming operation here.

The resulting left-hand side then consists of terms with a common sequence of remaining λ-binders. The fact that some λ-binders have been discarded in merging the left-hand sides implies that the λ-binders at the same positions in the corresponding right-hand sides should also be discarded (line B-14). The right-hand side is merged in line B-15, where the order of λ-binders are so changed as though the left-hand sides were originally covered by $\lambda\overline{x_k}$.

(B-1) **procedure** *Merge*$(MS, \mathbf{var}\ U =^? M)$
(B-2) **begin**
(B-3) For each $\{\lambda\overline{x_k^1}.t_1, \cdots, \lambda\overline{x_k^n}.t_n\} =^? M$ in MS, replace each $\lambda\overline{x_k^i}.t_i$ by $\lambda\overline{x_k^1}.\{\overline{x_k^i \mapsto x_k^1}\}t_i$;
(B-4) Assume $MS = \{U_1 =^? M_1, \cdots, U_h =^? M_h\}$; $U := U_1$;
(B-5) **for** $i = 2$ to h, assume $U = \lambda\overline{x_k}.V$, $U_i = \lambda\overline{y_m}.V'$, $F(\overline{x'_n}) \in V$ and $F(\overline{y'_n}) \in V'$, **do**
(B-6) **begin**
(B-7) Let ϕ_i be $\{\overline{y'_n \mapsto x'_n}\}$;
(B-8) Let $\{x_{p_1}, \cdots, x_{p_g}\} = \{\overline{x_k}\} \cap \{\overline{\phi_i(y_m)}\}$ be a set with $p_j < p_l$ for $j < l$;
(B-9) $U := \lambda x_{p_1} \cdots \lambda x_{p_g}.(V \cup \phi_i(V'))$
(B-10) **end**;
(B-11) $M := \{\}$; Assume $U = \lambda\overline{x_k}.V$; Let ϕ_1 be the identity map, $\phi_2, \cdots, \phi_h$ as above;
(B-12) **for** $i = 1$ to h, assume $U_i = \lambda\overline{y_m}.V'$ and $M_i = \{\lambda\overline{z_m}.t\}$, **do**
(B-13) **begin**
(B-14) Let $\{p_1, \cdots, p_k\}$ be such that $\phi_i(y_{p_j}) = x_j$; (*$j \le l$ may not imply $p_j \le p_l$.*)
(B-15) $M := M \cup \{\lambda z_{p_1} \cdots \lambda z_{p_k}.t\}$

(B-16) **end**;
(B-17) **end**

As an example, consider

$$MS = \{ \{\lambda x_1x_2x_3.F(x_1,x_2,x_4,x_3), \lambda x_1x_2x_3.G(x_1,x_2,x_3)\} =^? \lambda x_1'x_2'x_3'.s, \{\lambda y_1y_2y_3.F(y_2,y_1,y_3,y_4), \lambda y_1y_2y_3.G(y_2,y_3,y_1)\} =^? \lambda y_1'y_2'y_3'.t \}.$$

Let $i = 2$ in line B-5. Then $\phi_2 = \{y_2 \mapsto x_1, y_1 \mapsto x_2, y_3 \mapsto x_4, y_4 \mapsto x_3\}$ in line B-7, $\{x_{p_1}, \cdots, x_{p_q}\} = \{x_1, x_2\}$ in line B-8 and $U = \lambda x_1x_2.\{F(x_1,x_2,x_4,x_3), G(x_1,x_2,x_3), G(x_1,x_4,x_2)\}$ in line B-9. Furthermore, we have $x_1 = \phi_2(y_2), x_2 = \phi_2(y_1)$ in line B-14. Thus the final M is constructed in line B-15 as $\{\lambda x_1'x_2'.s, \lambda y_2'y_1'.t\}$.

4.4 Unify the left-hand side

Unification of flexible terms corresponds to applications of rules (FF-1) and (FF-2) in Subsection 2.1. Note that U is of the form $\lambda\overline{x_k}.V$. In lines C-6 and C-7, the procedure *UnifyL* computes the bound variables which should be preserved when applying rules (FF-1) and (FF-2). When the **for**-structure from line C-4 to C-8 is finished, a new set X of remaining λ-binders has been found. Now all free variables in U are replaced by a new variable H with arguments from X. In order to adjust the λ-binders in the right-hand side w.r.t. the set X, the procedure also returns a set B of original positions for all λ-binders with bound variables in X.

(C-1) **procedure** *UnifyL*(**var** U, **var** T, **var** B)
(C-2) **begin**
(C-3) Assume $U = \lambda\overline{x_k}.V$ and let $X = \{\overline{x_k}\}$;
(C-4) **for each** $F \in \mathcal{FV}(U)$ with $F(\overline{y_n^1}), \cdots, F(\overline{y_n^m})$ being all subterms in U with F **do**
(C-5) **begin**
(C-6) Let $\{p_1, \cdots, p_q\} = \{i \mid y_i^1 = \cdots = y_i^m\}$ with $p_j < p_l$ for $1 \le j < l \le q$;
(C-7) $X := X \cap \{y_{p_1}^1, \cdots, y_{p_q}^1\}$
(C-8) **end**;
(C-9) Let H be a new variable;
(C-10) **for each** $F \in \mathcal{FV}(U)$, assume $F(\overline{y_n}) \in V$, **do**
(C-11) append $\lambda\overline{y_n}.F(\overline{y_n}) =^? \lambda\overline{y_n}.H(\overline{X})$, marked as "visited", to the end of T;
(C-12) $U := \{\lambda\overline{X}.H(\overline{X})\}$; $B := \{i \mid x_i \in X\}$
(C-13) **end**

As an example, consider the result of the previous example

$$U = \lambda x_1x_2.\{F(x_1,x_2,x_4,x_3), G(x_1,x_2,x_3), G(x_1,x_4,x_2)\}.$$

Considering $G(x_1,x_2,x_3), G(x_1,x_4,x_2)$ we have $X = \{x_1\}$ in line C-7. The multiequations that are put into T in line C-11 are $\lambda\overline{x_4}.F(\overline{x_4}) =^? \lambda\overline{x_4}.H(x_1)$ and $\lambda\overline{x_3}.G(\overline{x_3}) =^? \lambda\overline{x_3}.H(x_1)$. As outputs we have $U = \{\lambda x_1.H(x_1)\}$ and $B = \{1\}$.

4.5 Unify the right-hand side

The procedure *UnifyR* unifies the terms in the right-hand side $M = \{\lambda\overline{x_k^1}.t_1, \cdots, \lambda\overline{x_k^n}.t_n\}$. First, a global array $\Lambda[i]$ is created for noting each sequence of the initial λ-binders $\lambda\overline{x_k^i}$. Let Λ be an array with components $\Lambda[1], \cdots, \Lambda[n]$. Note that the number of arrays in Λ, denoted by *width*(Λ), is fixed as n within a call to *UnifyR*, whereas the number of λ-binders in each array $\Lambda[i]$, denoted by *length*($\Lambda[i]$), may increase, since additional λ-binders may be appended at the end of $\Lambda[i]$.

Two procedures *SimpUnifyR1* and *SimpUnifyR2* are called in lines D-6 and D-7. The procedure *SimpUnifyR1* constructs a scheme C of the common top part of all terms in M, and a set Q of schemes of multiequations, which are the rest unification problems obtained when factorizing C from the terms. We call the outputs "schemes" because they contain

some "scheme variables", which need to be made precise in the procedure *SimpUnifyR2*, where some conditions of clashes and bound variable captures are also checked.

As mentioned in Section 3, not all initial λ-binders need to be left in the rest unification problem. For computing actual remaining λ-binders, the procedure *SimpUnifyR1* appends all λ-binders it encounters to Λ, and notes the requirements of other changes of actual remaining λ-binders. The global variable K created in line D-4 will be used to enumerate the requirements of changes. The procedure *SimpUnifyR1* uses the global variable *BS* created in line D-5 to store the requirements, and also all conditions to be checked. The procedure *SimpUnifyR2* will really compute the actual sequences of remaining λ-binders w.r.t. the requirements stored in *BS*, and check all conditions stored in *BS*.

Assume that *SimpUnifyR2* returns C and Q successfully. Let M be as above and $B = \{p_1, \cdots, p_m\}$ with $p_i < p_j$ for $i < j$. Then the multiequation $U =^? \lambda x^1_{p_1} \cdots \lambda x^1_{p_m}.C$ is a fragment of the final most general unifier (line D-9). The set Q of smaller multiequations is a part of the rest unification problem and put back to S and some of them to ST by the procedure call $AddS(Q, ST, S)$ (line D-10). The procedure *AddS* is given in the appendix.

(D-1) **procedure** $UnifyR(U, \{\overline{\lambda x^1_k.t_1}, \cdots, \overline{\lambda x^n_k.t_n}\}, B, \textbf{var } T, \textbf{var } S, \textbf{var } ST)$
(D-2) **begin**
(D-3) Create a global array $\Lambda[1..n]$ of arrays with initial $\Lambda[i] = \overline{x^i_k}$, $i = 1, \ldots, n$;
(D-4) Let K be a global variable with initial $K = 1$;
(D-5) Create a global empty set *BS*;
(D-6) $SimpUnifyR1(\{(t_1, 1), \cdots, (t_n, n)\}, C, Q)$;
(D-7) $SimpUnifyR2(B, C, Q)$;
(D-8) Assume $B = \{p_1, \cdots, p_m\}$ with $p_i < p_j$ for $i < j$;
(D-9) Append the multiequation $U =^? \lambda x^1_{p_1} \cdots x^1_{p_m}.C$ to the end of T;
(D-10) $AddS(Q, ST, S)$
(D-11) **end**

The procedure *SimpUnifyR1* considers mainly the following cases:

If all terms in N are abstractions (line E-4) then the outermost λ-binder of each term is appended to the corresponding component array of Λ (line E-6). In addition, a requirement of extending the actual remaining λ-binders is noted (line E-7). Then the subterms are considered (line E-8). Before leaving the procedure, a requirement of recovering the old remaining λ-binders is noted (line E-9).

A procedure *NewBinders* to note a requirement of changing the actual remaining λ-binders may be defined as follows:

procedure $NewBinders((D_1, D_2))$
begin $K := K + 1$; Put (D_1, D_2, K) into *BS* **end**

In the case that all terms in N are rigid (line E-11), topmost atoms should be equivalent modulo α-conversion and bound by the actual remaining λ-binders. However, checking these conditions at this place may be expensive in the worst case since the procedure has to find the corresponding λ-binders from Λ for the bound variables. We delay the checking here: The conditions are put into *BS* in line E-13 and to be checked in the procedure *SimpUnifyR2*.

In the case that there is a flexible term in N (line E-17), the flexible term will be first removed from N and the procedure continues to unify the rest of N. Multiequations in [4] may have right-hand sides consisting of arbitrary number of terms. So a corresponding procedure in [4] terminates an long as one of the input terms is flexible, and returns a single multiequation having all flexible input terms on the left and all other input terms on the right. In our case, however, the right-hand side can contain at most one term. Therefore, even when N contains a flexible term, other terms in N have to be further unified (line E-23), until N becomes a singleton (line E-3). The final resulting Q (in line E-25) returned

in our case is a set consisting of all multiequations in Q' returned by the further unification process in line E-23 and a multiequation with the flexible term on the left and the common part of other terms of N on the right.

The strange notation $\Gamma(F, (\overline{y_n}, p), p_1)$ in line E-20 is one of the so-called scheme variables. It will be replaced by a term of the form $F(\overline{y'_n})$ in the procedure *SimpUnifyR2*, where $\overline{y'_n}$ are obtained from $\overline{y_n}$ by renaming all bound variables in $\Lambda[p]$ into corresponding ones in $\Lambda[p_1]$. We delay the renaming here since finding the positions of λ-binders in $\Lambda[p]$ is expensive at the moment. The reason for this renaming is that the procedure *SimpUnifyR1* agrees to return a common part with bound variables corresponding to the λ-binders in the first term of N. Note that p_1 is the index of the first term in N (line E-19). The choices of λz_1 in line E-9, a_1 in line E-15, q in line E-25, and $\lambda x^1_{p_1} \cdots x^1_{p_m}$ in line D-9 of the procedure *UnifyR* are based on this agreement.

The strange notation $\lambda\Gamma((\overline{y_n}, p), K)$ in line E-21 is also a scheme variable. Assume that $\Lambda[p] = \lambda\overline{x_m}$. Let $x_{q_1}, \ldots, x_{q_h}$ be all bound variables in $\{\overline{y_n}\}$ which occur the K-th computed sequence of remaining λ-binders. Then the scheme variable will be replaced by $\lambda\overline{x_{q_h}}$ in *SimpUnifyR2*. The situation is similar for the scheme variables $\lambda\Gamma((\overline{y_n}, p), K)$ and $\lambda\Gamma((\overline{y_n}, q), K)$ in line E-25. Note that the K's in lines E-21 and E-25 may be distinct but the sequences of actual remaining λ-binders are always identical.

When N contains a flexible term, new remaining λ-binders should be computed, as noted in line E-20 by *NewBinders*$((+, (\overline{y_n}, p)))$. Before leaving *SimpUnifyR1* the old remaining λ-binders should be recovered, as noted by *NewBinders*$((-, (\overline{y_n}, p)))$ in line E-27.

If the original input M to *UnifyR* contains only one term then the condition in line E-3 can never be true. In this case, *SimpUnifyR1* simply serves to prepare the conditions which will be used by *SimpUnifyR2* to check failures of bound variable captures in the term.

(E-1) **procedure** *SimpUnifyR1*$(N, \text{var } C, \text{var } Q)$
(E-2) **begin**
(E-3) **if** $N = \{(t, p)\}$ and $width(\Lambda) > 1$ **then** $(C, Q) := (t, \{\})$
(E-4) **else if** $N = \{(\lambda z_1.s_1, p_1), \cdots, (\lambda z_n.s_n, p_n)\}$ **then**
(E-5) **begin**
(E-6) Add z_i to the end of $\Lambda[i]$, $i = p_1, \ldots, p_n$;
(E-7) Let $L = length(\Lambda[p_1])$; *NewBinders*$((+, L))$;
(E-8) *SimpUnifyR1*$(\{(s_1, p_1), \cdots, (s_n, p_n)\}, C', Q')$;
(E-9) *NewBinders*$((-, L))$; $C := \lambda z_1.C'$; $Q := Q'$
(E-10) **end**
(E-11) **else if** $N = \{(a_1(s^1_1, \cdots, s^1_m), p_1), \cdots, (a_n(s^n_1, \cdots, s^n_m), p_n)\}$ **then**
(E-12) **begin**
(E-13) Put $(\{(a_1, p_1), \cdots, (a_n, p_n)\}, K)$ into BS;
(E-14) **for** $i = 1$ **to** m **do** *SimpUnifyR1*$(\{(s^1_i, p_1), \cdots, (s^n_i, p_n)\}, C_i, Q_i)$;
(E-15) $C := a_1(\overline{C_m})$; $Q := \bigcup_{i=1}^{m} Q_i$
(E-16) **end**
(E-17) **else if** there is some $(F(\overline{y_n}), p) \in N$ **then**
(E-18) **begin**
(E-19) Assume (s, p_1) be the first element in N;
(E-20) $C := \Gamma(F, (\overline{y_n}, p), p_1)$; *NewBinders*$((+, (\overline{y_n}, p)))$;
(E-21) **if** $N - \{(F(\overline{y_n}), p)\}$ is empty **then** $Q := \{\lambda\Gamma((\overline{y_n}, p), K).F(\overline{y_n}) =^? \{\}\}$
(E-22) **else begin**
(E-23) *SimpUnifyR1*$(N - \{(F(\overline{y_n}), p)\}, C', Q')$;
(E-24) Assume (t, q) be the first element in $N - \{(F(\overline{y_n}), p)\}$;
(E-25) $Q := Q' \cup \{\lambda\Gamma((\overline{y_n}, p), K).F(\overline{y_n}) =^? \lambda\Gamma((\overline{y_n}, q), K).C'\}$
(E-26) **end**;

(E-27) *NewBinders*$((-,(\overline{y_n},p)))$;
(E-28) **end**
(E-29) **else** failure
(E-30) **end**

As an example, assume that *UnifyR* is called with $M = \{\lambda\overline{x_3}.x_3(\lambda x.t), \lambda\overline{y_3}.y_3(\lambda y.G(y_2)), \lambda\overline{z_3}.F(z_1,z_3)\}$ and $B = \{2,3\}$. An array $\Lambda[1..3]$ is then created with $\Lambda[1] = \overline{x_3}$, $\Lambda[2] = \overline{y_3}$ and $\Lambda[3] = \overline{z_3}$. The procedure *SimpUnifyR1* will be recursively called with $N = \{(x_3(\lambda x.t),1), (y_3(\lambda y.G(y_2)),2),(F(z_1,z_3),3)\}$ in line D-6, with $N = \{(x_3(\lambda x.t),1),(y_3(\lambda y.G(y_2)),2)\}$ in line E-23, with $N = \{(\lambda x.t,1),(\lambda y.G(y_2),2)\}$ in line E-14, with $N = \{(t,1),(G(y_2),2)\}$ in line E-8 and with $N = \{(t,1)\}$ in line E-23. The final results are $\Lambda[1] = \overline{x_3}x$, $\Lambda[i] = \overline{y_3}y$, $\Lambda[3] = \overline{z_3}$, $K = 7$, $BS = \{(+,(z_1z_3,3),2), (\{(x_3,1),(y_3,2)\},2), (+,4,3), (+,(y_2,2),4), (-,(y_2,2),5),(-,4,6),(-,(z_1z_3,3),7)\}$, $C = \Gamma(F,(z_1z_3,3),1)$ and $Q = \{\lambda\Gamma((y_2,2),4).G(y_2) =^? \lambda\Gamma((y_2,1),4).t, \lambda\Gamma((z_1z_3,3),6).F(z_1,z_3) =^? \lambda\Gamma((z_1z_3,1),6).x_3(\lambda x.\Gamma(G,(y_2,2),1))\}$.

The tasks of *SimpUnifyR2* are to check the conditions of clashes (line F-10) and of bound variable captures (line F-18) in *BS* and to replace the scheme variables in C and Q by concrete symbols (lines F-8 and F-20). Note that not only clashes of bound variables at heads but also other kinds of clashes, e.g. when the heads are distinct constants, or partly bound variables and partly constants, are checked here. The procedure finds the positions of λ-binders for occurrenes of bound variables (line F-7), and also computes the positions of all actual remaining λ-binders (lines F-14 – F-17). Note that the symbol $\otimes$ in line F-14 may be either $+$ or $-$, and in F-15 denotes the the usual plus or minus operation of integers.

The positions of actual remaining λ-binders are marked in *TB*. Remaining λ-binders at different places may use different marks. The variable I keeps the mark for the current remaining λ-binders and may be changed in line F-15.

(F-1) **procedure** *SimpUnifyR2*$(B,$ **var** $C,$ **var** $Q)$
(F-2) **begin**
(F-3) Let *max* be the maximum of all *length*$(\Lambda[i])$; Assume $\{1,\ldots,max\} \cap (\mathcal{C} \cup \mathcal{V}) = \{\}$;
(F-4) **for** $p =$ *length*(Λ) **to** 1, assume $\Lambda[p] = \overline{x_m}$, **do**
(F-5) **begin**
(F-6) Let $\phi = \{x_1 \mapsto 1, \cdots, x_m \mapsto m\}$;
(F-7) Replace each $(\overline{y_n},p)$ in *BS*, C or Q by $(\phi(\overline{y_n}),p)$;
(F-8) Replace each $\Gamma(F,(\overline{p_n},p'),p)$ in C or Q by $F(\phi^{-1}(\overline{p_n}))$
(F-9) **end**;
(F-10) **if** there is $(CLA,K) \in BS$ with $(a,i),(b,j) \in CLA$ and $a \neq b$ **then** failure (clash);
(F-11) Create an array $TB[1..max]$ with $TB[i] = 1$ if $i \in B$ and $TB[i] = 0$ if $i \notin B$; $I := 1$;
(F-12) **for** $K' = 2$ **to** K **do**
(F-13) **begin**
(F-14) **if** there is $(\otimes,(\overline{p_n},p),K')$ in *BS* **then**
(F-15) **begin for** $j = 1$ **to** n **do if** $TB[p_j] = I$ **then** $TB[p_j] := I \otimes 1$; $I := I \otimes 1$ **end**;
(F-16) **if** there is $(+,L,K')$ in *BS* **then** $TB[L] := I$;
(F-17) **if** there is $(-,L,K')$ in *BS* **then** $TB[L] := 0$;
(F-18) **if** there is $(CLA,K') \in BS$ with $(q,p) \in CLA$ such that
(F-19) $q \in \{1,\ldots,max\}$ and $TB[q] = I$ do not hold **then** failure (capture)
(F-20) Replace each $\lambda\Gamma((\overline{p_n},p),K')$ in C or Q by $\lambda\overline{x_{q_h}}$ where $\Lambda[p] = \lambda\overline{x_m}$ and $q_1,\ldots,q_h$
(F-21) are all in $\{\overline{p_n}\}$ such that $1 \leq q_j \leq max$, $TB[q_j] = I$ and $q_j < q_l$ for $j < l$
(F-22) **end**
(F-23) **end**

Continuing to compute the previous example with the procedure *SimpUnifyR2*, we will have $BS = \{(+,(13,3),2),(\{(3,1),(3,2)\},2),(+,4,3),(+,(2,2),4),(-,(2,2),5),(-,4,6),(-,(13,3),7)\}$, $C = F(x_1,x_3)$ and $Q = \{\lambda((13,3),6).F(z_1,z_3) =^? \lambda((13,1),6).x_3(\lambda x.G(x_2)),$

$\lambda((2,2),4).G(y_2) =^? \lambda((2,1),4).t\}$ at line F-10. As the results of the procedure we have $C = F(x_1, x_3)$ and $Q = \{\lambda z_3.F(z_1, z_3) =^? \lambda x_3.x_3(\lambda x.G(x_2)), G(y_2) =^? t\}$.

Theorem 4.1 *For any input* $(\{\}, S_0)$*, the algorithm Unify is terminating. Furthermore,* S_0 *is unifiable if and only if Unify terminates with* $(T_f, \{\})$*, where* S_0 *and* T_f *have the same set of unifiers when restricted to the free variables in* S_0*.*

Proof Proof can be found in [16]. □

5 Our Algorithm is Linear

It is assumed that all data structures involved in the algorithm are dynamically created and possibly connected through pointers. Since in principle the space required cannot exceed the running time in such a machine, we will mainly concentrate on the time complexity.

A data structure can always be marked in some way. Changing and checking the mark of a given data structure need only constant time.

As mentioned, systems of multiequations are implemented by DAG's in the usual way, i.e. nodes by dynamically created data structures connected through pointers as required by directed arces. A node of a free variable has a list of father nodes. Each of other nodes has at most one father node.

Sets, multisets or stacks are all implemented as lists Let us recall that eliminating or inserting an element in a list need only constant time when the element to be inserted and the place where the eliminating or inserting operation should be performed are known (see e.g. [1]).

Each bound variable in the starting system may be coded as a unique integer. So an array can be created with indices being (the integers corresponding to) all bound variables in the starting system such that for a given bound variable the corresponding component in the array can always be visited in a constant time. It sufices to allocate a place of a (large enough but) fixed size for each component. Then the size of the array is obviously linear in the size of the starting system.

Using the above array, the time needed in computing the intersection of two sets of bound variables is linear in the total size of the two sets (see e.g. [1]).

First of all, the procedure *Select* is linear since it visits each node in the starting system at most once, and only a fixed number operations may be performed w.r.t. each node. The situation is similar to its first-order counterpart in [5].

The procedure *Merge* is linear since the time required for each call $Merge(MS, U =^? M)$ is linear in the total size of the left-hand sides in MS. Indeed, after this call, all terms on the left will be unified and put into the solved part so that they can never be visited again.

The execution time for the call $UnifyL(U, T, B)$ is linear in the size of U. This can be proved in the similar way as above. Note that the new free variables introduced in line C-9 cannot be more than the original free variables.

Now let us consider the procedure *UnifyR*. The creation of a global array Λ is obviously linear in the size of M. Therefore we need only to prove the linearity of *SimpUnifyR1*, *SimpUnifyR2* and *AddS*.

Intuitively, the procedure *SimpUnifyR1*, like the procedure *Select*, visits each node of the starting system at most once. Remember that in a call $SimpUnifyR1(N, C, Q)$ all nodes in N are from the right-hand sides of some multiequations in the unsolved part. During the procedure call, the nodes in N that have been visited are either simply dropped or moved to the left-hand sides of some multiequations in Q. So these nodes cannot be visited again by *SimpUnifyR1* in the entire unification process. Furthermore, it can be observed that the time required for each operation during the procedure call is always linear in the size of the current nodes being visited. Note that the additional space for the newly created nodes in Λ, BS, C, the remaining outermost λ-binders and the left-hand sides in Q is also linear in

the total size of the visited nodes of N.

To prove the linearity of the procedure *SimpUnifyR2* we assume a list of pointers for each p with $1 \leq p \leq \mathit{width}(\Lambda)$, which connects all pairs of the form $(\overline{y_n}, p)$ and all scheme variables of the form $\Gamma(F, (\overline{p_n}, p'), p)$ in BS, C and Q, as required in lines F-7 and F-8. In fact this list can be easily created during the process of creating BS, C and Q in *SimpUnifyR1*. Furthermore it is also assumed that in the process of creating BS, C and Q in *SimpUnifyR1*, a list of pointers for each K' with $1 \leq K' \leq K$ has also been created, which connects all $(\otimes, (\overline{p_n}, p), K')$, $(\otimes, L, K')$, (CLA, K') and $\lambda\Gamma((\overline{p_n}, p), K')$ as required in lines F-14, F-16, F-17, F-18 and F-20. In a call $\mathit{SimpUnifyR2}(B, C, Q)$, the nodes in Λ, BS, C, in the outermost λ-binders and in the left-hand sides in Q will be visited along the above two lists. The time for the operations associated to a node being visited is always linear in the size of the node.

The time required for the procedure call $\mathit{AddS}(Q, S, ST)$ is linear in the number of multiequations in Q. We need only to partition the time into contant time segments and to assign them to the terms in the left-hand sides in Q. No other executions of *AddS* may be assigned to the same terms since *AddS* only handles multiequations in Q, which are yielded in unifying some right-hand sides.

Theorem 5.1 *The algorithm Unify is linear in the size of its input system.*

6 Conclusion

We have presented a unification algorithm for patterns whose time and space costs are both linear in the size of input. This result may be used as a basis for analyzing computational complexities of higher-order logic programming and higher-order proof systems. The ideas we have used in avoiding potential sources of nonlinear behaviors are also very useful in guiding the design of practical unification algorithms for patterns.

ACKNOWLEDGEMENT

We thank Tobias Nipkow for discussions and the anonymous referees for comments.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[2] W. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

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

[4] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM TOPLAS*, 4(2):258–282, 1982.

[5] A. Martelli and U. Montanari. *Unification in linear time and space: A structured presentation.* Technical Report, Internal Rep. B76-16, Ist. di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, 1976.

[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 of simply typed lambda-terms as logic programming. In P. K. Furukawa, editor, *Proc. 1991 Joint Int. Conf. on Logic Programming*, MIT Press, 1991.

[8] G. Nadathur and D. Miller. An overview of λProlog. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Logic Programming Conference*, MIT Press, 1988.

[9] T. Nipkow. Higher-order critical pairs. In *Proc. 6th LICS*, pages 342–349, 1991.

[10] T. Nipkow. *Practical Unification of Higher-Order Patterns.* Draft, Technische Universität München, 1991.

[11] M. Paterson and M. Wegman. Linear unification. *J. Computer and System Sciences*, 16:158–167, 1978.

[12] L. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385, Academic Press, 1990.

[13] F. Pfenning. Elf: a language for logic definition and verified meta-programming. In *Proc. 4th LICS*, pages 313–322, IEEE Computer Society Press, June 1989.

[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 *Proc. 6th LICS*, pages 74–85, IEEE Computer Society Press, July 1991.

[16] Z. Qian. *Unification of Higher-Order Patterns in Linear Time and Space.* Technical Report, Forschungsbericht 5/92, FB3 Informatik, Universtät Bremen, 1992.

Appendix

(A-1) **procedure** *Select*(**var** ST, **var** S, **var** MS)
(A-2) **begin**
(A-3) **if** ST is nonempty **then** pop ST to MS
(A-4) **else begin** $MS := \{E\}$ for an "unvisited" multiequation E; Mark E as "visited" end;
(A-5) **while** there is an "unvisited" $F(\overline{x_k}) \in \mathcal{U}(MS)$ **do**
(A-6) **begin**
(A-7) **if** there is another "unvisited" occurrence $F(\overline{y_k})$
(A-8) **then begin**
(A-9) Mark $F(\overline{y_k})$ as "visited";
(A-10) if $F(\overline{y_k}) \in \mathcal{U}(E)$ for some multiequation E **then**
(A-11) **if** E is "unvisited" **then**
(A-12) **begin** Mark E as "visited"; $MS := MS \cup \{E\}$ **end**
(A-13) **else begin if** E is in ST then failure (cycle) **end**
(A-14) **else begin**
(A-15) $Node := F(\overline{y_k})$;
(A-16) **repeat** $Node := Father.Node$; Mark N as "visited"
(A-17) **until** $N \in \mathcal{M}(E)$ for some multiequation E;
(A-18) **if** E is "visited" **then** failure (cycle);
(A-19) Push MS in ST; Mark E as "visited"; $MS := \{E\}$
(A-20) **end**
(A-21) **end**
(A-22) **else** mark $F(\overline{x_k})$ as "visited"
(A-23) **end**;
(A-24) $S := S - MS$
(A-25) **end**

(G-1) **procedure** *AddS*(Q, **var** ST, **var** S)
(G-2) **begin**
(G-3) $S := S \cup Q$;
(G-4) **if** some $U' =^? M' \in Q$ with some "visited" $t \in U' \cup M'$ **then**
(G-5) **begin**
(G-6) Mark $U' =^? M'$ as "visited";
(G-7) **if** $t \in U'$ **then**
(G-8) **begin** Pop ST to MS; $MS := MS \cup \{U' =^? M'\}$; Push MS in ST **end**;
(G-9) **if** $t \in M'$ **then** push $\{U' =^? M'\}$ into ST
(G-10) **end**
(G-11) **end**