# Efficient Second-Order Matching*

Régis Curien[1] Zhenyu Qian[2] Hui Shi[2]

[1] CRIN-CNRS & INRIA-Lorraine, 615, rue du jardin botanique, BP 101
54602 Vandoeuvre-les-Nancy Cedex, France. E-mail: curien@loria.fr
[2] Universität Bremen, FB3 Informatik, Bibliothekstr. 4, 28359 Bremen, Germany
E-mail: {qian,shi}@informatik.uni-bremen.de

**Abstract.** The standard second-order matching algorithm by Huet may be expansive in matching a flexible-rigid pair. On one hand, many fresh free variables may need to be introduced; on the other hand, attempts are made to match the heading free variable on the flexible side with every "top layer" on the rigid side and every argument of the heading free variable with every subterm covered by the "top layer". We propose a new second-order matching algorithm, which introduces no fresh free variables and just considers some selected "top layers", arguments of the heading free variable and subterms covered by the corresponding "top layers". A first implementation shows that the new algorithm is more efficient both in time and space than the standard one for a great number of matching problems.

## 1 Introduction

Second-order matching is a useful mechanism. For example, Huet and Lang used second-order matching for program transformations [11]. Boy de la Tour and Caferra [1, 2] used second-order matching to recognize patterns in formulas and to propose lemmas in automatic theorem proving by analogy. Curien followed the similar line to use second-order matching for computing similarities between formulas in automatic analogical proofs [4]. Second-order matching is also used by Kolbe and Walther [12] for reusing proofs. The standard second-order matching algorithm was given by Huet [10] (see also [11]).

Let us use $s \stackrel{\triangleleft}{=} t$ to denote a matching problem, where $s$ is a second-order simply typed $\lambda$-term (short: term) possibly containing free variables and $t$ a term containing no free variables. An essential case that a second-order matching algorithm should consider is the so-called *flexible-rigid* case, where a matching problem $\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t$ with $F$ being a free variable should be solved. A term with a free variable at head is called a *flexible* term. If observing the matching process step by step, the standard algorithm may use an imitation rule for the flexible-rigid case, which yields a partial substitution $\{F \mapsto \lambda y_m.a(H_1(\overline{y_m}), \cdots, H_n(\overline{y_m}))\}$, where $a$ is the head of $t$ and $H_1, \cdots, H_n$ are all fresh free variables. If observing the whole matching process the standard algorithm attempts to match $F$ with every "top layer"[3] of $t$, and every $s_i$ with every subterm covered by a corresponding "top layer". From both aspects as

* Research partially supported by ESPRIT Basic Research WG *COMPASS* 6112 and BMBF Project *UniForM.*

[3] Formally we will define a "top layer" of a term as a context later.

above, the standard algorithm suffers from efficiency problems. The problem with the imitation rule is that it introduces many fresh free variables. In fact, these fresh free variables never need to occur in a final resulting matcher. Thus the efficiency can be increased if they are not introduced at all. The problem with the whole matching process is that many attempts to match $F$ with a "top layer" of $t$, and to match every $s_i$ with every subterm covered by a "top layer" may fail. Thus the efficiency can be increased when some of the failure cases can be foreseen in some way.

In this paper we propose a new algorithm for second-order matching and proves its soundness and completeness. The new algorithm improves the efficiency of the standard algorithm for the flexible-rigid case $\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t$ by directly matching $F$ with a "top layer" of $t$ and solving some matching subproblems consisting of some of $s_1, \cdots, s_m$ and some subterms covered by the "top layer". In addition, the new algorithm does not introduce any fresh free variables and foresees many failure cases in matching subproblems. It should be mentioned that some of the ideas presented in this paper have already been implemented elsewhere, e.g. for inductive proofs in a theorem prover by Boyer and Moore [3]. However, to our knowledge, they have not been studied so systematically and formally as in the current paper.

Theoretically, the new algorithm is by no means worse than the standard one in the sense that for the same original matching problem all matching subproblems encountered during execution of the new algorithm are also encountered during execution of the standard algorithm. However, one may still be wondering whether the increased cost for foreseeing the failure cases in a matching subproblem outweigh the gains in practice. To check this, we implement both Huet's algorithm and the new one. Our measurement shows that the new algorithm has indeed a better performance than Huet's in many cases. In particular, our algorithm is much better for these matching problems $s \stackrel{\triangleleft}{=} t$ where $s$ contains many free variables or $t$ contains many bound variables. Unfortunately, it turns out that our algorithm may have a worse performance than Huet's in the case where many constant symbols (i.e. function symbols with arity 0) occur in $t$.

The paper is organized as follows. Section 2 introduces some basic not(ta)ions. In Section 3, we review the standard second-order matching algorithm formulated via transformations. The new algorithm is presented via transformations in Section 4. The termination, soundness and completeness of the new algorithm are shown in Section 5. In Section 6 we briefly describe an implementation of the new algorithm and compare it with an implementation of the standard algorithm. We conclude in Section 7.

## 2 Preliminaries

Types are constructed in the usual way as in simply typed $\lambda$-calculus. We use $\mathcal{T}_0$ to denote the set of *base types*. We use $\mathcal{T}$ to denote the set of *types*, and write $\alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \beta$ and $\overline{\alpha_n} \rightarrow \beta$ for $(\alpha_1 \rightarrow \cdots (\alpha_n \rightarrow \beta))$. In $\overline{\alpha_n} \rightarrow \beta$, $\beta$ is assumed in $\mathcal{T}_0$. The *order* $O(\alpha)$ of a type $\alpha$ is defined as $O(\alpha) = 1$ for $\alpha \in \mathcal{T}_0$, and $O(\overline{\alpha_n} \rightarrow \beta) = max\{O(\alpha_i) \mid 1 \leq i \leq n\} + 1$ otherwise.

For each $\alpha \in \mathcal{T}$, $\mathcal{C}_\alpha$ and $\mathcal{V}_\alpha$ denote pairwise disjoint denumerable sets of *function symbols* and *variables*, resp. *Constant symbols* are function symbols of base types.

Function symbols and variables are also called *atoms*. We denote $\mathcal{C} = \bigcup_{\alpha \in \mathcal{T}} \mathcal{C}_\alpha$, $\mathcal{V} = \bigcup_{\alpha \in \mathcal{T}} \mathcal{V}_\alpha$ and $\mathcal{A} = \mathcal{C} \cup \mathcal{V}$.

The set $\mathcal{T}_\alpha$ of $\lambda$-*terms* (short: *terms*) of type $\alpha \in \mathcal{T}$ is defined by the usual construction of *application* and *abstraction*. Let $\mathcal{T} = \bigcup_{\alpha \in \mathcal{T}} \mathcal{T}_\alpha$. We use $\tau(t)$ to denote the type $\alpha$ for $t \in \mathcal{T}_\alpha$. In an abstraction $\lambda x.t$ the term $t$ is said to be *covered* by the $\lambda$-*binder* $\lambda x$. The term $(\ldots(a\ s_1)\ldots\ s_n)$ may be written as $a(s_1, \ldots, s_n)$, $a(\overline{s_n})$ or $a(\overline{s})$, and $\lambda x_1.\cdots\lambda x_k.t$ as $\lambda\overline{x_k}.t$ or $\lambda\overline{x}.t$.

For a term $t$ we use $|t|$ to denote the number of atoms in $t$.

Terms are only compared modulo $\alpha$-conversion. Bound and free variables are defined as usual. Furthermore, we always assume that within one term no $\lambda$-binders occur more than once and no variables occur both bound and free, unless otherwise stated. The set of all free variables in a syntactic object $O$ is denoted by $\mathcal{FV}(O)$ and that of all bound variables by $\mathcal{BV}(O)$.

For $\mathcal{X} \in \{\beta, \eta, \beta\eta\}$, we use $\longrightarrow_{\mathcal{X}}$ to denote one step $\mathcal{X}$-reduction and $=_{\mathcal{X}}$ the equivalence relation induced by $\longrightarrow_{\mathcal{X}}$. Let $s$ be a term. Then $s{\downarrow}_{\mathcal{X}}$ denotes the $\mathcal{X}$-normal form such that $s \longrightarrow^*_{\mathcal{X}} s{\downarrow}_{\mathcal{X}}$. If $t$ is a $\beta$-normal form, then $t$ must be of the form $\lambda\overline{x}.a(\overline{s_n})$ with $a \in \mathcal{A}$ and each $s_i$ a $\beta$-normal form. The atom $a$ is called the *head* and denoted by $\mathcal{H}(t)$. The $\beta$-normal form $t$ is called *flexible* if $a \in \mathcal{FV}(t)$ and *rigid* otherwise. A $\beta$-normal form $\lambda\overline{x}.a(\overline{s_n})$ may be said to be *long* if $a(\overline{s_n})$ is of a base type and each $s_i$ is long. Let $s{\uparrow}_\eta$ denote the long $\beta$-normal form (uniquely) obtained by $\eta$-expanding a $\beta$-normal form $s$. For single variable $F$, $F{\uparrow}_\eta$ may still be written as $F$. We only consider long $\beta$-normal forms in the sequel and thus will use $=$ instead of $=_{\beta\eta}$.

A long $\beta$-normal form is said to be *second-order* (or *first-order*) if the types of its free variables are at most of order 2 (or 1, respectively). Huet's second-order matching algorithm works for second-order terms having constants and bound variables of order at most three. In [7], Dowek dropped this restriction.

A *substitution* $\sigma$ is a function denoted by $\{X_1 \mapsto t_1, \cdots, X_n \mapsto t_n\}$ or $\{\overline{X_n \mapsto t_n}\}$, where $X_i \in \mathcal{V}$, $t_i \in \mathcal{T}$ and $\tau(X_i) = \tau(t_i)$ hold for $i = 1, \ldots, n$, the *domain* is denoted by $\mathcal{D}(\sigma) = \{X_1, \cdots, X_n\}$, and the set of *newly introduced variables* by $\mathcal{I}(\sigma) = \bigcup_{i=1,\cdots,n} \mathcal{FV}(t_i)$. The application of a substitution $\sigma$ to a term $s$ is written as $\sigma(s)$. It is always assumed in $\sigma(s)$ that the term $s$ have been $\alpha$-converted such that $\mathcal{BV}(s) \cap \mathcal{I}(\sigma) = \emptyset$, in order to avoid capture of free variables of $\mathcal{I}(\sigma)$ by bound variables of $\mathcal{BV}(s)$. Furthermore, it is always assumed that $\mathcal{BV}(s) \cap \mathcal{D}(\sigma) = \emptyset$, and therefore whenever $s$ is $\lambda x.t$ we have $\sigma(s) = \lambda x.\sigma(t)$ automatically. The *composition* $\sigma\theta$ of two substitutions $\sigma$ and $\theta$ is defined by $(\sigma\theta)(X) = \sigma(\theta(X))$ for every $X \in \mathcal{V}$.

Two substitutions $\sigma$ and $\theta$ are *compatible* if $\sigma(X) = \theta(X)$ holds for all $X \in \mathcal{D}(\sigma) \cap \mathcal{D}(\theta)$. If $\sigma$ and $\theta$ are compatible, then the *union* $\sigma \cup \theta$ are defined as $\sigma \cup \theta(X) = \sigma(X)$ for $X \in \mathcal{D}(\sigma)$, and $\sigma \cup \theta(X) = \theta(X)$ for $X \in \mathcal{D}(\theta)$ and $\sigma \cup \theta(X) = X$ otherwise.

If $\mathcal{W} \subseteq \mathcal{V}$, then the *restriction* $\sigma_{|\mathcal{W}}$ is a substitution defined by $\sigma_{|\mathcal{W}}(X) = \sigma(X)$ for $X \in \mathcal{W}$, $\sigma_{|\mathcal{W}}(X) = X$ otherwise. We use $\sigma_{|-\mathcal{W}}$ to denote $\sigma_{|\mathcal{D}(\sigma)-\mathcal{W}}$.

A substitution $\sigma$ is said to be *normalized* if $\sigma(X)$ is a long $\beta$-normal form for each $X \in \mathcal{D}(\sigma)$. Normalized $\sigma$ satisfies $(\sigma(t){\downarrow}_\beta){\uparrow}_\eta = \sigma(t){\downarrow}_\beta$ for every long $\beta$-normal form $t$. Only normalized substitutions are considered in this paper.

If $\mathcal{W} \subseteq \mathcal{V}$ and $\sigma$ and $\theta$ are two substitutions, then $\sigma = \theta\ [\mathcal{W}]$ means $\sigma(X) = \theta(X)$ for each $X \in \mathcal{W}$. Furthermore, $\sigma \leq \theta\ [\mathcal{W}]$ means that there is a substitution $\rho$ such that $\rho\sigma = \theta\ [\mathcal{W}]$. $[\mathcal{W}]$ may be omitted if $\mathcal{W} = \mathcal{V}$. For any substitution $\sigma$ and finite

$\mathcal{W} \supseteq \mathcal{D}(\sigma)$, there always exists a substitution $\sigma'$ such that $\mathcal{D}(\sigma') \cap \mathcal{I}(\sigma') = \emptyset$, $\mathcal{D}(\sigma) = \mathcal{D}(\sigma')$ and $\sigma \leq \sigma'$ $[\mathcal{W}]$ and $\sigma' \leq \sigma$ $[\mathcal{W}]$. Since $\mathcal{W}$ in question is usually known, we may restrict our attention only to the substitutions like $\sigma'$ without loss of generality. Note that $\mathcal{D}(\sigma') \cap \mathcal{I}(\sigma') = \emptyset$ implies $\sigma'\sigma' = \sigma'$.

*Positions* are strings of integers. Let $p$, $q$ range over positions. The set of all positions of a term $t$, denoted by $\mathcal{P}os(t)$, consists of all positions corresponding to a subterm of $t$ in the following way, where the root position is the empty string, denoted by $\Lambda$:

$$\begin{aligned} t_{|\Lambda} &= t \\ (\lambda x.t)_{|1\cdot p} &= t_{|p} \\ a(t_1, \ldots, t_n)_{|k\cdot p} &= (t_k)_{|p} \quad \text{where } 1 \leq k \leq n \end{aligned}$$

A position $p$ is *above* $q$, denoted by $p \leq q$, when $p$ is a prefix of $q$. We write $p < q$ for $p \leq q$ and $p \neq q$. When $p \not\leq q$ and $q \not\leq p$, $p$ and $q$ are said to be *independent*.

For a term $t$, independent positions $p_1, \ldots, p_k$ of $t$ and terms $s_1, \cdots, s_k$ with $\tau(t_{|p_i}) = \tau(s_i)$ for $i = 1, \ldots, k$, we use $t[s_1, \cdots, s_k]_{p_1,\cdots,p_k}$ to denote the result of replacing each $t_{|p_i}$ in $t$ by $s_i$. Furthermore, we may use the special symbols $\Box_1, \cdots, \Box_k$ to denote holes in a term and call a term $t[\Box_1, \cdots, \Box_k]_{p_1,\cdots,p_k}$ a *context*.

In the sequel, we use $c$, $d$, $e$, $f$, $g$ and $h$ to denote function symbols, $a$ and $b$ atoms, $x$, $y$ and $z$ bound variables, $X$, $Y$, $Z$, $F$, $G$ and $H$ free variables, $s$, $t$, $u$ and $v$ terms, $\sigma$, $\theta$ and $\rho$ substitutions, $p$ and $q$ positions. Furthermore, $\sigma(t)$ always denotes $\sigma(t)\downarrow_\beta$, unless stated otherwise.

## 3 The standard second-order matching

A (second-order) *matching pair* $s \stackrel{\triangleleft}{=} t$ is a pair of second-order terms $s$ and $t$, where $\tau(s) = \tau(t)$ holds, the term $t$ contains no free variables and the term $s$ may contain free variables whose types are of at most second-order. The term $s$ or $t$ can also be referred as the *left-hand side* or *rigid-hand side* of the matching pair. Since only long $\eta$-forms are considered, we may always assume that $s$ and $t$ have the same sequence of outer $\lambda$-binders. A (second-order) *matching problem* is a multiset of matching pairs. We use $S$ to denote a matching problem.

A substitution $\theta$ is called a *matcher* of $s \stackrel{\triangleleft}{=} t$ if $\theta(s) = t$. A substitution is a matcher of $S$ if and only if it is a matcher of each matching pair in $S$. A set of matchers of $S$ is said to be *complete* if whenever $\theta$ is a matcher of $S$, there is a matcher $\sigma$ in the set such that $\sigma \leq \theta$ $[\mathcal{FV}(S)]$ holds.

A matching algorithm can be described via transformations on pairs $\langle S, \theta\rangle$ of matching problems $S$ and substitutions $\theta$. For an input pair $\langle S_0, \{\}\rangle$, the algorithm *succeeds* when there is a sequence of transformations terminating with $\langle \{\}, \sigma\rangle$, in which case $\sigma$ is a matcher of the initial problem $S_0$, fails when there exist no sequences of transformations that terminate with a pair of the above form. In the case of failure, $S_0$ is not matchable.

In this section a second-order matching algorithm is described via the transformations. The transformations are a reformulation of the standard algorithm by Huet [10] (see also [11]). They are in fact a restriction of the corresponding transformations by Snyder and Gallier [19].

The first transformation is the usual simplification rule, which simplifies a matching pair with identical heading symbols on both sides.

$$\langle\{\lambda\overline{x}.a(\overline{s_n}) \trianglelefteq\!\!\!\!= \lambda\overline{x}.a(\overline{t_n})\} \cup S, \sigma\rangle \implies \langle\{\overline{\lambda\overline{x}.s_n \trianglelefteq\!\!\!\!= \lambda\overline{x}.t_n}\} \cup S, \sigma\rangle \qquad \text{(D)}$$

The second rule is the usual imitation rule, which deals with a flexible left-hand side.

$$\langle\{\lambda\overline{x}.F(\overline{s_n}) \trianglelefteq\!\!\!\!= \lambda\overline{x}.a(\overline{t_m})\} \cup S, \sigma\rangle \implies \langle\{\overline{\lambda\overline{x}.H_m(\overline{\sigma'(s_n)}) \trianglelefteq\!\!\!\!= \lambda\overline{x}.t_m}\} \cup \sigma'(S), \sigma'\sigma\rangle \quad \text{(I)}$$

where $\sigma' = \{F \mapsto \lambda\overline{y_n}.a(H_1(\overline{y_n}), \cdots, H_m(\overline{y_n}))\}$, $H_1, \cdots, H_m$ are fresh new free variables of appropriate types.

The third rule is the usual projection rule, which also deals with a flexible left-hand side:

$$\langle\{\lambda\overline{x}.F(\overline{s_n}) \trianglelefteq\!\!\!\!= \lambda\overline{x}.t\} \cup S, \sigma\rangle \implies \langle\{\lambda\overline{x}.\sigma'(s_i) \trianglelefteq\!\!\!\!= \lambda\overline{x}.t\} \cup \sigma'(S), \sigma'\sigma\rangle \qquad \text{(P)}$$

where $\sigma' = \{F \mapsto \lambda\overline{y_n}.y_i\}$ for some $1 \leq i \leq n$.

This algorithm has the failure case when there is a matching pair $\lambda\overline{x}.a(\overline{s}) \trianglelefteq\!\!\!\!= \lambda\overline{x}.b(\overline{t})$ where $a$ is not a free variable and $a \neq b$.

In the sequel we write $\langle S, \sigma\rangle \Longrightarrow_{(R)} \langle S', \sigma'\rangle$ for a step by transformation (R) and $\langle S, \sigma\rangle \Longrightarrow^* \langle S', \sigma'\rangle$ for a finite sequence of transformations.

Let us now see formally why the fresh new free variables $H_1, \cdots, H_m$ introduced in the imitation rule never need to occur in a final resulting matcher.

***Lemma 1.*** *For any matcher $\theta$ of $s \trianglelefteq\!\!\!\!= t$ there is a matcher $\sigma$ of $s \trianglelefteq\!\!\!\!= t$ such that $\sigma \leq \theta\ [\mathcal{FV}(s)]$ and $\mathcal{I}(\sigma) = \emptyset$.*

**Proof** Let us first prove that for a matcher $\theta$ of $s \trianglelefteq\!\!\!\!= t$ if $\mathcal{FV}(\theta(F)) \neq \emptyset$ for some $F \in \mathcal{D}(\theta)$ then $\theta_{|-\{F\}}$ is still a matcher of $s \trianglelefteq\!\!\!\!= t$. Assume that $\theta(F)$ is of the form $\lambda\overline{y_m}.u$, where $m \geq 0$, and $(\theta_{|-\{F\}})(s) = s'$. Then $\theta(s) = \{F \mapsto \lambda\overline{y_m}.u\}(s')$, since $\mathcal{D}(\theta) \cap \mathcal{I}(\theta) = \emptyset$. If we can show that $F \notin \mathcal{FV}(s')$, then the above assertion holds. Since $\mathcal{FV}(t) = \emptyset$, this problem can be reduced to showing that if $F \in \mathcal{FV}(s')$ and $H$ occurs freely in $u$, then $H$ also occurs in $\{F \mapsto \lambda\overline{y_m}.u\}(s')$. We prove this by an induction on the structure of $s'$.

- Let $s'$ be an atom $a$. Then the above assertion holds trivially.
- Let $s'$ be of the form $\lambda\overline{x_n}.F(\overline{u_m})$. Note that the type of $F$ is of at most second-order. If the type of $F$ is of first-order then $m = 0$. If the type of $F$ is of second-order then the type of each $y_i$ must be first-order. In both cases $H$ occurs in $\{F \mapsto \lambda\overline{y_m}.u\}(s')$.
- Let $s'$ be of the form $\lambda\overline{x_n}.a(\overline{u_k})$, where $a$ is an atom other than $F$, and $u_i$ contains $F$ for some $1 \leq i \leq k$. By induction assumption, $H$ occurs $\{F \mapsto \lambda\overline{y_m}.u\}(u_i)$. So, $H$ occurs in $\{F \mapsto \lambda\overline{y_m}.u\}(s')$.

Now assume that $\theta$ is an arbitrary matcher of $s \trianglelefteq\!\!\!\!= t$ and $\mathcal{W} = \{X \in \mathcal{D}(\theta) \mid \mathcal{FV}(\theta(X)) = \emptyset\} \cap \mathcal{FV}(s)$. Let $\sigma$ denote $\theta_{|\mathcal{W}}$. Thus $\sigma \leq \theta\ [\mathcal{FV}(s)]$ and $\mathcal{I}(\sigma) = \emptyset$ hold. By the above assertion, we know that $\sigma$ is a matcher of $s \trianglelefteq\!\!\!\!= t$. □

The following lemma states formally that the process of the standard algorithm attempts to match $F$ with every "top layer" of $t$, and every $s_i$ with every subterm covered by a corresponding "top layer".

***Lemma 2.*** *Let $\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t$ be a matching pair. Assume that $h_1, \cdots, h_k$ are numbers such that $1 \leq h_j \leq m$ for $k \geq 0$ and $j = 1, \ldots, k$, and $q_1, \cdots, q_k$ independent positions of $t$. Then the standard second-order matching algorithm has a transformation sequence*

$$\langle \{\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t\}, \{\}\rangle \Longrightarrow^* \langle \{\lambda\overline{x}.s_{h_j} \stackrel{\triangleleft}{=} \lambda\overline{x}.t_{|q_j} \mid 1 \leq j \leq k\}, \sigma\rangle,$$

*where $\sigma(F) = \lambda\overline{y_m}.t[y_{h_1}, \cdots, y_{h_k}]_{\overline{q_k}}$,*

**Proof** First of all, we prove that there is always a transformation sequence

$$\langle \{\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t\}, \{\}\rangle \Longrightarrow^* \langle \{\lambda\overline{x}.G_j(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t_{|q_j} \mid 1 \leq j \leq k\}, \sigma\rangle,$$

where $G_1, \ldots, G_k$ are fresh free variables, and $\sigma = \{\}$ or $\sigma(F) = \lambda\overline{y_m}.t[G_1(\overline{y_m}), \cdots, G_k(\overline{y_m})]_{\overline{q_k}}$. The proof is easy. In fact, we need only to apply rule (I) repeatedly to $\langle \{\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t\}, \{\}\rangle$ and the resulting pairs until the resulting pair $\langle \{\lambda\overline{x}.G_j(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t_{|q_j} \mid 1 \leq j \leq k\}, \sigma\rangle$ is yielded, where $\sigma(F) = \lambda\overline{y_m}.t[G_1(\overline{y_m}), \cdots, G_k(\overline{y_m})]_{\overline{q_k}}$.

Now by applying rule (P) several times, we may easily obtain

$$\langle \{\lambda\overline{x}.G_j(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t_{|q_j} \mid 1 \leq j \leq k\}, \sigma\rangle \Longrightarrow^* \langle \{\lambda\overline{x}.s_{h_j} \stackrel{\triangleleft}{=} \lambda\overline{x}.t_{|q_j} \mid 1 \leq j \leq k\}, \sigma'\rangle,$$

where $\sigma'(F) = \lambda\overline{y_m}.t[y_{h_1}, \cdots, y_{h_k}]_{\overline{q_k}}$. Hence the assertion of the lemma holds. □

## 4 The new efficient algorithm

The new algorithm consists of

- rule (D) in the previous section for rigid-rigid matching pairs, and
- a new rule (FR) for flexible-rigid matching pairs.

Rule (FR) is based on the fact that if $\lambda\overline{x}.F(\overline{s_m})$ is a second-order term then $s_1, \cdots, s_m$ must be of base types. Therefore, if $\sigma$ is a matcher of $\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t$ then each $\sigma(s_i)$ for $1 \leq i \leq m$ will either occur as a subterm in $t$, or need not be considered at all in proving $\sigma(\lambda\overline{x}.F(\overline{s_m})) = \lambda\overline{x}.t$. Thus we need only to find the occurrences of $\sigma(s_i)$ in $t$. Indeed, computing a matcher may consist in first constructing new matching pairs composed of $s_i$ and some subterms in $t$, then computing the matchers for these new matching pairs and finally using these matchers to build a matcher for the original problem.

Which subterms of $t$ should be used to construct the new matching pairs are determined by two observations. The first observation is that for a term $\lambda\overline{x}.u$ and a substitution $\theta$ if a bound variable $z \in \{\overline{x}\}$ occurs in $\theta(u)$ then it must also occur in $u$, since $\theta$ introduces no new occurrences of $z$. This implies that if a matching pair $\lambda\overline{x}.s \stackrel{\triangleleft}{=} \lambda\overline{x}.t$ is matchable and a bound variable $z \in \{\overline{x}\}$ occurs in $t$ then $z$ must also occur in $s$.

The second observation is that if $\sigma$ is a matcher of $\lambda\overline{x}.F(\overline{s_m}) \stackrel{\trianglelefteq}{=} \lambda\overline{x}.t$ then $\sigma(F) = \lambda\overline{y_m}.t[y_{h_1},\ldots y_{h_k}]_{\overline{q_k}}$, where $\overline{q_k}$ are independent positions of $t$ and $1 \leq h_j \leq m$ holds for $k \geq 0$ and $j = 1,\ldots,k$. Note that

$$\sigma(\lambda\overline{x}.F(\overline{s_m})) \rightarrow_\beta \lambda\overline{x}.t[\sigma(s_{h_1}),\ldots,\sigma(s_{h_k})]_{\overline{q_k}}.$$

Therefore, an occurrence of a bound variable $z \in \{\overline{x}\}$ in $t$ must be in $\sigma(s_{h_j})$, i.e. in $s_{h_j}$, for some $1 \leq j \leq k$. Thus if $p$ is the position of this occurrence of $z$ in $t$ then $q_j \leq p$ holds.

Let us look some simple examples.

The matching pair $\lambda x.F(u) \stackrel{\trianglelefteq}{=} \lambda x.f(x)$ has no matchers if $x$ does not occur in $u$.

Let $\lambda x.F(u,x) \stackrel{\trianglelefteq}{=} \lambda x.g(x,x)$ be a matching pair, where $x$ does not occur in $u$. Then a matcher $\sigma$ always satisfies that $\sigma(F) = \lambda\overline{y_2}.g(x,x)[y_2,y_2]_{1,2}$, i.e. $\sigma(F) = \lambda\overline{y_2}.g(y_2,y_2)$. There is no need to consider $\sigma(F) = \lambda\overline{y_2}.g(x,x)[y_2]_\Lambda$, since $\lambda x.x \stackrel{\trianglelefteq}{=} \lambda x.g(x,x)$ has no matchers.

Let $\lambda x.F(G(x),u) \stackrel{\trianglelefteq}{=} \lambda x.f(c,x)$ be a matching pair, where $u$ contains no occurrences of $x$. Let $\sigma(F) = \lambda\overline{y_2}.s$ be any matcher of the matching pair, then $y_1$ must have an occurrence above 2 in s. So, we only need to consider the following cases:

- $\sigma(F) = \lambda\overline{y_2}.f(c,x)[y_1]_p$ for $p = \Lambda$ or $p = 2$. In fact, if $p = \Lambda$ then $\sigma(F) = \lambda\overline{y_2}.y_1$ and $\sigma$ must match $\{\lambda x.G(x) \stackrel{\trianglelefteq}{=} \lambda x.f(c,x)\}$. If $p = 2$ then $\sigma(F) = \lambda\overline{y_2}.f(c,y_1)$ and $\sigma$ must match $\{\lambda x.G(x) \stackrel{\trianglelefteq}{=} \lambda x.x\}$.
- $\sigma(F) = \lambda\overline{y_2}.f(c,x)[y_1,y_1]_{p,q}$ for $p = 2$ and $q = 1$. Then $\sigma(F) = \lambda\overline{y_2}.f(y_1,y_1)$ and $\sigma$ must match $\{\lambda x.G(x) \stackrel{\trianglelefteq}{=} \lambda x.c, \lambda x.G(x) \stackrel{\trianglelefteq}{=} \lambda x.x\}$. Since $\{\lambda x.G(x) \stackrel{\trianglelefteq}{=} \lambda x.c, \lambda x.G(x) \stackrel{\trianglelefteq}{=} \lambda x.x\}$ is not matchable, there are no matchers in this case.
- $\sigma(F) = \lambda\overline{y_2}.f(c,x)[y_1,y_2]_{p,q}$ for $p = 2$ and $q = 1$. Then $\sigma(F) = \lambda\overline{y_2}.f(y_2,y_1)$ and $\sigma$ must match $\{\lambda x.u \stackrel{\trianglelefteq}{=} \lambda x.c, \lambda x.G(x) \stackrel{\trianglelefteq}{=} \lambda x.x\}$.

The main part of rule (FR) is the function $FFR$, which computes a set of matchers of a flexible-rigid matching pair as follows.

**Function** $FFR(\lambda\overline{x}.F(\overline{s_m}) \stackrel{\trianglelefteq}{=} \lambda\overline{x}.t)$

1. Use the matching algorithm to compute a set $M_q^i$ of matchers of $\lambda\overline{x}.s_i \stackrel{\trianglelefteq}{=} \lambda\overline{x}.t_{|q}$ for each $1 \leq i \leq m$ and each position $q \in \mathcal{P}os(t)$ such that $\tau(s_i) = \tau(t_{|q})$, $t_{|q}$ contains at least one bound variable from $\{\overline{x}\}$ and if a bound variable from $\{\overline{x}\}$ occurs in $t_{|q}$ then it also occurs in $s_i$.
2. Use the matching algorithm to compute a set $M_q^i$ of matchers of $\lambda\overline{x}.s_i \stackrel{\trianglelefteq}{=} \lambda\overline{x}.t_{|q}$ for each $1 \leq i \leq m$ and each position $q \in \mathcal{P}os(t)$ such that $\tau(s_i) = \tau(t_{|q})$ and $q$ is independent of all positions $p$ with $\mathcal{H}(t_{|p}) \in \{\overline{x}\}$.
3. Choose each set $\{\overline{q_k}\}$ of independent positions of $t$ with $k \geq 0$. Assume that the elements $\overline{q_k}$ occur in a fixed order. Choose each sequence of numbers $\overline{h_k}$ such that $t[\Box,\cdots,\Box]_{\overline{q_k}}$ contains no bound variables from $\{\overline{x}\}$ and $M_{q_j}^{h_j}$ is nonempty for $j = 1,\ldots,k$. Choose each substitution $\theta^j \in M_{q_j}^{h_j}$ for each $j = 1,\ldots,k$. Construct a substitution

$$\theta^F = \{F \mapsto \overline{y_m}.t[y_{h_1},\cdots,y_{h_k}]_{\overline{q_k}}\}.$$

If all substitutions $\theta^1, \cdots, \theta^k$, and $\theta^F$ are compatible, then construct a substitution

$$\sigma = \theta^1 \cup \cdots \cup \theta^k \cup \theta^F.$$

4. The function $FFR$ stops and returns the set of all substitutions $\sigma$ constructed in step 3 as the result.

As an example, let us compute $FFR(\lambda\overline{x_2}.F(x_1, x_2, a) \stackrel{\triangleleft}{=} \lambda\overline{x_2}.f(x_1, g(x_1), a))$.

1. The only bound variable in $f(x_1, g(x_1), a))$ on the right is $x_1$. More precisely, $x_1$ occurs in $f(x_1, g(x_1), a))$ at the positions $p_1 = 1$ and $p_2 = 2 \cdot 1$ Note that $x_1$ also occurs in the first argument on the left. The positions $\Lambda$, $p_1$, 2 and $p_2$ are all positions above $p_1$ or $p_2$. Assume that the application of the matching algorithm to the matching pairs yields sets of matchers as follows:

| Matching pairs | Sets of matchers |
|---|---|
| $\lambda\overline{x_2}.x_1 \stackrel{\triangleleft}{=} \lambda\overline{x_2}.f(x_1, g(x_1), a)$ | $M^1_\Lambda = \emptyset$ |
| $\lambda\overline{x_2}.x_1 \stackrel{\triangleleft}{=} \lambda\overline{x_2}.x_1$ | $M^1_{p_1} = \{\{\}\}$ |
| $\lambda\overline{x_2}.x_1 \stackrel{\triangleleft}{=} \lambda\overline{x_2}.g(x_1)$ | $M^1_2 = \emptyset$ |
| $\lambda\overline{x_2}.x_1 \stackrel{\triangleleft}{=} \lambda\overline{x_2}.x_1$ | $M^1_{p_2} = \{\{\}\}$. |

2. The only position independent of $p_1$ and $p_2$ in $f(x_1, g(x_1), a))$ on the right of the original matching pair is $q_1 = 3$. To match each argument on the left with the subterm at $q_1$ on the right, we apply the matching algorithm to $\lambda\overline{x_2}.x_1 \stackrel{\triangleleft}{=} \lambda\overline{x_2}.a$, $\lambda\overline{x_2}.x_2 \stackrel{\triangleleft}{=} \lambda\overline{x_2}.a$ and $\lambda\overline{x_2}.a \stackrel{\triangleleft}{=} \lambda\overline{x_2}.a$. The resulting sets are $M^1_{q_1} = \emptyset$, $M^2_{q_1} = \emptyset$ and $M^3_{q_1} = \{\{\}\}$.
3. The only sets of independent positions of $f(x_1, g(x_1), a)$ that we need to consider are $\{p_1, p_2\}$ and $\{p_1, p_2, q_1\}$.
   - For $\{p_1, p_2\}$, the sets $M^1_{p_1}$ and $M^1_{p_2}$ are the only nonempty sets in consideration. In this case the sequence of numbers corresponding to $p_1, p_2$ is $1, 1$. Thus the substitution $\theta^F$ is $\{F \mapsto \lambda\overline{y_3}.f(y_1, g(y_1), a)\}$. Since $M^1_{p_1}$ and $M^1_{p_2}$ contain only the identity substitution, we may construct $\sigma_1 = \{F \mapsto \lambda\overline{y_3}.f(y_1, g(y_1), a)\}$.
   - For $\{p_1, p_2, q_1\}$, the sets $M^1_{p_1}$, $M^1_{p_2}$ and $M^3_{q_1}$ are the only nonempty sets in consideration. In this case the sequence of numbers corresponding to $p_1, p_2, q_1$ is $1, 1, 3$. Thus the substitution $\theta^F$ is $\{F \mapsto \lambda\overline{y_3}.f(y_1, g(y_1), y_3)\}$. Since $M^1_{p_1}$, $M^1_{p_2}$ and $M^3_{q_1}$ contain only the identity substitution, we may construct a substitution $\sigma_2 = \{F \mapsto \lambda\overline{y_3}.f(y_1, g(y_1), y_3)\}$.
4. The function $FFR$ returns $\{\sigma_1, \sigma_2\}$.

As another example, let us compute $FFR(\lambda x.F(a, F(x, b)) \stackrel{\triangleleft}{=} \lambda x.f(f(b, x), a))$.

1. The only bound variable occurring in $f(f(b, x), a))$ on the right is $x$. More precisely it occurs at position $p_1 = 1 \cdot 2$ in $f(f(b, x), a))$ . Note that it also occurs in the second argument on the left. The positions $\Lambda$, 1 and $p_1$ are all positions above $p_1$. The corresponding matching pairs yield the nonempty sets of matchers as follows:

| Matching pairs | Sets of matchers |
|---|---|
| $\lambda x.F(x,b) \stackrel{\trianglelefteq}{=} \lambda x.f(f(b,x),a))$ | $M^2_\Lambda = \{\theta_1, \theta_2\}$ where $\theta_1 = \{F \mapsto \lambda\overline{y_2}.f(f(b,y_1),a))\}$ $\theta_2 = \{F \mapsto \lambda\overline{y_2}.f(f(y_2,y_1),a))\}$ |
| $\lambda x.F(x,b) \stackrel{\trianglelefteq}{=} \lambda x.f(b,x)$ | $M^2_1 = \{\theta_3, \theta_4\}$ where $\theta_3 = \{F \mapsto \lambda\overline{y_2}.f(y_2,y_1)\}$, $\theta_4 = \{F \mapsto \lambda\overline{y_2}.f(b,y_1)\}$ |
| $\lambda x.F(x,b) \stackrel{\trianglelefteq}{=} \lambda x.x$ | $M^2_{p_1} = \{\theta_5\}$ where $\theta_5 = \{F \mapsto \lambda\overline{y_2}.y_1\}$ |

2. The only position independent of $p_1$ in $f(f(b,x),a))$ on the right of the original matching pair is $q_1 = 2$ and $q_2 = 1 \cdot 1 \cdot 1$. We apply the matching algorithm to the matching pairs $\lambda x.a \stackrel{\trianglelefteq}{=} \lambda x.a$, $\lambda x.F(x,b) \stackrel{\trianglelefteq}{=} \lambda x.a$, $\lambda x.a \stackrel{\trianglelefteq}{=} \lambda x.b$ and $\lambda x.F(x,b) \stackrel{\trianglelefteq}{=} \lambda x.b$. The only nonempty resulting sets of matchers are $M^1_{q_1} = \{\theta_6\}$ where $\theta_6 = \{\}$, $M^2_{q_1} = \{\theta_7\}$ where $\theta_7 = \{F \mapsto \lambda\overline{y_2}.a\}$, $M^1_{q_2} = \emptyset$ and $M^2_{q_2} = \{\theta_8, \theta_9\}$ where $\theta_8 = \{F \mapsto \lambda\overline{y_2}.y_2\}$ and $\theta_9 = \{F \mapsto \lambda\overline{y_2}.b\}$.
3. All sets of independent positions we need to consider are $\{\Lambda\}$, $\{1\}$, $\{1, q_1\}$, $\{p_1\}$, $\{p_1, q_1\}$, $\{p_1, q_2\}$ and $\{p_1, q_1, q_2\}$.
   - For $\{\Lambda\}$, we can choose $\theta_1$ or $\theta_2$ from $M^2_\Lambda$. Thus the corresponding sequence of numbers is 2. Hence the substitution $\theta^F$ is $\{F \mapsto \lambda\overline{y_2}.y_2\}$. Obviously, $\theta_1$ and $\theta^F$ are not compatible, nor $\theta_2$ and $\theta^F$.
   - For $\{1\}$, we can only choose $\theta_3$ or $\theta_4$ from $M^2_1$. Thus the corresponding sequence of numbers is 2. Hence the substitution $\theta^F$ is $\{F \mapsto \lambda\overline{y_2}.f(y_2,a)\}$. It is easy to see that $\theta_3$ and $\theta^F$ are not compatible, nor $\theta_4$ and $\theta^F$.
   - For $\{1, q_1\}$, we can first choose $\theta_3$ from $M^2_1$ and $\theta_6$ from $M^1_{q_1}$. Thus the corresponding sequence of numbers is $2, 1$. Hence the substitution $\theta^F$ is $\{F \mapsto \lambda\overline{y_2}.f(y_2,y_1)\}$. Since $\theta_3$ and $\theta^F$ are identical and $\theta_4$ is the identity substitution, we construct a resulting substitution $\sigma_1 = \theta_3$. As another possibility we can choose $\theta_4$ from $M^2_1$ and $\theta_7$ from $M^2_{q_1}$. Thus the corresponding sequence of numbers is $2, 2$. Hence the substitution $\theta^F$ is $\{F \mapsto \lambda\overline{y_2}.f(y_2,y_2)\}$. Since $\theta_7$ and $\theta^F$ are not compatible, no resulting substitutions can be yielded in this case.
   - For $\{p_1\}$, $\{p_1, q_1\}$, $\{p_1, q_2\}$ and $\{p_1, q_1, q_2\}$, we have to choose $\theta_5$ from $M^2_{p_1}$. Thus the substitution $\theta^F$ is $\{F \mapsto \lambda\overline{y_2}.f(\cdots)\}$. Since $\theta_5$ and $\theta^F$ are not compatible, no resulting substitutions can be yielded in these cases.
4. The function $FFR$ returns $\{\sigma_1\}$.

Now we formulate rule (FR) as follows:

$$\langle \{\lambda\overline{x}.F(\overline{s_n}) \stackrel{\trianglelefteq}{=} \lambda\overline{x}.t\} \cup S, \sigma\rangle \implies \langle \sigma'(S), \sigma'\sigma\rangle \tag{FR}$$

where $\sigma' \in FFR(\lambda\overline{x}.F(\overline{s_n}) \stackrel{\trianglelefteq}{=} \lambda\overline{x}.t)$.

## 5 Termination, soundness and completeness

To prove the termination of the new algorithm we define the complexity measure $CM(S)$ of a matching problem $S$ to be a pair $(\xi_r(S), \xi_l(S))$, where $\xi_r(S)$ is the multiset $\{|t| \mid s \stackrel{\trianglelefteq}{=} t \in S\}$ and $\xi_l(S)$ the multiset $\{|s| \mid s \stackrel{\trianglelefteq}{=} t \in S\}$. The termination proof is based on the standard ordering on multisets of numbers, denoted as $>$, and the lexicographic ordering on pairs, also denoted as $>$.

***Theorem 3.*** *The new algorithm always terminates.*

**Proof** The execution of the new algorithm for an input pair $\langle S,\sigma\rangle$, where $\sigma$ is an arbitrary substitution, can be described as a sequence of applications of rules (D) and (FR) issuing from $\langle S,\sigma\rangle$. Thus the termination of the algorithm follows from that the application of a single rule always terminates and the sequence is always finite. We prove this by an induction on $CM(S)$.

Initially if $CM(S) = (\emptyset,\emptyset)$ then $S=\emptyset$. None of rules (D) and (FR) can be applied. The termination follows trivially.

Assume that $S$ is a nonempty matching problem and rule (D) or (FR) is applicable. First, assume that rule (D) is applicable. Then the sequence is of the form $\underline{\langle S,\sigma\rangle \Longrightarrow_{(D)}} \langle S',\sigma\rangle \Longrightarrow \cdots$, where $S = \{\lambda\overline{x}.a(\overline{s_n}) \stackrel{\triangleleft}{=} \lambda\overline{x}.a(\overline{t_n})\} \cup S_1$ and $S' = \{\lambda\overline{x}.s_n \stackrel{\triangleleft}{=} \lambda\overline{x}.t_n\} \cup S_1$. The application of (D) is obviously terminating. Since $|\lambda\overline{x}.a(\overline{t_n})| > |\lambda\overline{x}.t_i|$ for all $i=1,\ldots,n$, we have $CM(S) > CM(S')$. By induction assumption, the algorithm is terminating for the input pair $\langle S',\sigma\rangle$. Hence the algorithm is terminating for the input pair $\langle S,\sigma\rangle$.

Assume that rule (FR) is applicable. Then the sequence is of the form

$$\langle\{\lambda\overline{x}.F(\overline{s_n}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t\} \cup S_1,\sigma\rangle \Longrightarrow_{(FR)} \langle\sigma'(S_1),\sigma'\sigma\rangle \Longrightarrow \cdots$$

for some substitution $\sigma'$. The termination of rule (FR) is based on that of the function $FFR$. In fact, the only sources of non-termination during execution of the function $FFR$ are the two recursive calls to the algorithm in the steps 1 and 2 of the definition of the function $FFR$. Since both recursive calls have an input matching problem of the form $\lambda\overline{x}.s_i \stackrel{\triangleleft}{=} \lambda\overline{x}.t_{|q}$, where $|t| \geq |t_{|q}|$ and $|F(\overline{s_m})| > |s_i|$ for all $i=1,\ldots,m$, we have $CM(S) > CM(\{\lambda\overline{x}.s_i \stackrel{\triangleleft}{=} \lambda\overline{x}.t_{|q}\})$. By induction assumption the recursive calls are terminating. Hence the application of rule (FR) is terminating. Since $\sigma'(S_1)$ does not change the right-hand sides in $S_1$, we have $\xi_r(S) > \xi_r(\sigma'(S_1))$. By induction assumption, the algorithm for the input $\langle\sigma'(S_1),\sigma'\sigma\rangle$ is terminating. Hence the algorithm with the input $\langle S,\sigma\rangle$ is terminating. □

Now we consider the soundness. First, we show that the function $FFR$ is sound in the following sense.

***Lemma 4.*** *Each $\sigma \in FFR(\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t)$ is a matcher of $\lambda\overline{x_n}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x_n}.t$.*

**Proof** We use the notations in the definition of the algorithm. By step 3, $\sigma$ is of the form $\theta^1\cup\cdots\cup\theta^k\cup\theta^F$, where $\theta^1,\cdots,\theta^k$ and $\theta^F$ are compatible. Note that $\theta^F = \{F \mapsto \lambda y_m.t[y_{h_1},\ldots,y_{h_k}]_{\overline{q_k}}\}$, each $\theta^j$ is a matcher of the problem $\lambda\overline{x}.s_{h_j} \stackrel{\triangleleft}{=} \lambda\overline{x}.t_{|q_j}$ for $j=1,\ldots,k$, and $\overline{q_k}$ are independent positions of $t$. Then

$$\sigma(\lambda\overline{x}.F(\overline{s_m})) = \lambda\overline{x}.t[\overline{\theta^k(s_{h_k})}]_{\overline{q_k}} = \lambda\overline{x}.t[\overline{t_{|q_k}}]_{\overline{q_k}} = \lambda\overline{x}.t.$$

□

***Theorem 5.*** *For a matching problem $S$, if $\langle S,\{\}\rangle \Longrightarrow^* \langle\emptyset,\sigma\rangle$ is a sequence of transformations of the new algorithm, then $\sigma$ matches $S$.*

**Proof** Follows from the above lemma and by an induction on the length of the transformation sequence. □

Now we do some preparations for proving the completeness.

***Lemma 6.*** *Let $\sigma$ be a matcher of $\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t$. Then for some independent positions $\{\overline{q_k}\}$ of $t$ and some numbers $\overline{h_k}$ such that $1 \leq h_j \leq m$ for each $j = 1, \ldots, k$, the following holds:*

1. *$\sigma(F) = \lambda\overline{y_m}.t[y_{h_1}, \cdots, y_{h_k}]_{\overline{q_k}}$,*
2. *$\sigma(\lambda\overline{x}.s_{h_j}) = \lambda\overline{x}.t_{|q_j}$ for each $j = 1, \ldots, k$, and*
3. *if $\mathcal{H}(t_{|p}) \in \{\overline{x}\}$ for some position $p$ then there are $s_{h_j}$ and $q_j$ with some $1 \leq j \leq k$ such that $\mathcal{H}(t_{|p})$ occurs in $s_{h_j}$ and $q_j \leq p$ holds.*

**Proof**

1. Follows directly from the fact that the term $\lambda\overline{x}.F(\overline{s_m})$ is second-order and all subterms $\overline{s_m}$ are of base types.
2. Follows directly from the assertion 1.
3. Assume that $\mathcal{H}(t_{|p}) \in \{\overline{x}\}$ for some $p$. Note that $\sigma(F) = \lambda\overline{y_m}.t[y_{h_1}, \cdots, y_{h_k}]_{\overline{q_k}}$, $\sigma(\lambda\overline{x}.F(\overline{s_m})) = \lambda\overline{x}.t[\overline{\sigma(s_{h_k})}]_{\overline{q_k}}$ and $\sigma(\lambda\overline{x}.F(\overline{s_m})) = \lambda\overline{x}.t$. Since no bound variables from $\{\overline{x}\}$ may occur in $\sigma(X)$ for $X \in \mathcal{D}(\sigma)$, no bound variables from $\{\overline{x}\}$ may occur in $t[y_{h_1}, \cdots, y_{h_k}]_{\overline{q_k}}$. Therefore $\mathcal{H}(t_{|p})$ must occur in some $\sigma(s_{h_j})$ with $1 \leq j \leq k$. Furthermore, $\mathcal{H}(t_{|p})$ must occur in $s_{h_j}$. Since $\lambda\overline{x}.t[\overline{\sigma(s_{h_k})}]_{\overline{q_k}} = \lambda\overline{x}.t$, we have $\sigma(s_{h_j}) = t_{|q_j}$. Since $\mathcal{H}(t_{|p})$ occurs in $\sigma(s_{h_j})$, we have $q_j \leq p$.

□

A call to a matching algorithm for an input matching problem $S$ is said to be *complete* if whenever $\theta$ is a matcher of $S$ the call yields a matcher $\sigma$ of $S$ such that $\mathcal{I}(\sigma) = \emptyset$, $\mathcal{D}(\sigma) \subseteq \mathcal{FV}(S)$ and $\sigma \leq \theta$. A matching algorithm is *complete* if its call for each input matching problem is complete.

***Lemma 7.*** *If every call of the matching algorithm in the step 1 or 2 of the definition of the function $FFR$ is complete, then every call to the function $FFR$ is complete.*

**Proof** Consider the call $FFR(\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t)$. By Lemma 1, we need only to consider those matchers $\theta$ of $\lambda\overline{x}.F(\overline{s_m}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t$ such that $\mathcal{I}(\theta) = \emptyset$.

By Lemma 6-1, $\theta(F) = \lambda\overline{y_m}.t[\overline{y_{h_k}}]_{\overline{q_k}}$ for some independent positions $\overline{q_k}$ of $t$ and some numbers $\overline{h_k}$ such that $1 \leq h_j \leq m$ for each $j = 1, \ldots, k$. By Lemma 6-2, $\theta(s_{h_j}) = \lambda\overline{x}.t_{|q_j}$ holds for $j = 1, \ldots, k$. By Lemma 6-3, if $\mathcal{H}(t_{|p}) \in \{\overline{x}\}$ for some $p$ then there are $s_{h_j}$ and $q_j$ as above such that $\mathcal{H}(t_{|p})$ occurs in $s_{h_j}$ and $q_j \leq p$ holds. This means that the term $\lambda\overline{y_m}.t[\overline{y_{h_k}}]_{\overline{q_k}}$ contains no free occurrences of bound variables from $\{\overline{x}\}$.

Since $\lambda\overline{x}.\theta(s_{h_j}) = \lambda\overline{x}.t_{|q_j}$ for each $1 \leq j \leq k$, a complete call to the matching algorithm in the step 1 or 2 yields a matcher $\theta_j$ of $\lambda\overline{x}.s_{h_j} = \lambda\overline{x}.t_{|q_j}$ such that $\mathcal{I}(\theta_j) = \emptyset$, $\mathcal{D}(\theta_j) \subseteq \mathcal{FV}(\lambda\overline{x}.s_{h_j})$ and $\theta_j \leq \theta$. Since $\mathcal{I}(\theta) = \emptyset$, we have that for each $X \in \mathcal{FV}(s_{h_j})$, either $\theta_j(X) = \theta(X)$ or $X \notin \mathcal{D}(\theta_j)$ holds. Thus the substitutions $\theta^1, \cdots, \theta^k$, and $\theta^F$ are compatible. Hence we may construct a substitution $\sigma$ as in step 3 such that $\mathcal{I}(\sigma) = \emptyset$, $\mathcal{D}(\sigma) \subseteq \mathcal{FV}(\lambda\overline{x}.F(\overline{s_m}))$ and $\sigma \leq \theta$. □

Now we may prove the completeness result.

***Lemma 8.*** *The new matching algorithm is complete.*

**Proof** Let $S$ be a matching problem. By Lemma 1, we need only to consider those matchers $\theta$ of $S$ such that $\mathcal{I}(\theta) = \emptyset$. The proof follows by an induction on $CM(S)$.

Initially, if $CM(S) = (\emptyset, \emptyset)$ then $S = \emptyset$. Thus $\{\}$ is a matcher of $S$ as required.

Assume that $S \neq \emptyset$. Since $\theta$ is a matcher of $S$, either rule (D) or (FR) is applicable. If rule (D) is applicable, then $S = \{\underline{\lambda\overline{x}.a(\overline{s_n}) \stackrel{\triangleleft}{=} \lambda\overline{x}.a(\overline{t_n})}\} \cup S_1$. Therefore we have $\langle S, \{\}\rangle \Longrightarrow_{(D)} \langle S', \{\}\rangle$, where $S' = \{\overline{\lambda\overline{x}.s_n \stackrel{\triangleleft}{=} \lambda\overline{x}.t_n}\} \cup S_1$ and $\theta$ is a matcher of $S'$. Since $CM(S) > CM(S')$, by induction assumption, the new algorithm yields a matcher $\sigma$ of $S'$ as required. The substitution $\sigma$ is also a matcher $\sigma$ of $S$ as required.

If rule (FR) is applicable, then $S = \{\lambda\overline{x}.F(\overline{s_n}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t\} \cup S_1$. Since $CM(S) > CM(\{\lambda\overline{x}.s_j \stackrel{\triangleleft}{=} \lambda\overline{x}.t_{|q}\})$, by induction assumption, every recursive call to the new algorithm in the step 1 or 2 of the definition of the function $FFR$ is complete. By Lemma 7, the call to the function $FFR$ is complete, i.e. it yields a matcher $\sigma$ of $\lambda\overline{x}.F(\overline{s_n}) \stackrel{\triangleleft}{=} \lambda\overline{x}.t$ such that $\mathcal{I}(\sigma) = \emptyset$, $\mathcal{D}(\sigma) \subseteq \mathcal{FV}(\lambda\overline{x}.F(\overline{s_n}))$ and $\sigma \leq \theta$. This implies that $\theta = \sigma \cup \theta_{|-\mathcal{D}(\sigma)}$ and $\theta_{|-\mathcal{D}(\sigma)}$ is a matcher of $\sigma(S_1)$. Note that $\langle S, \{\}\rangle \Longrightarrow_{(FR)} \langle\sigma(S_1), \sigma\rangle$. Since $CM(S) > CM(\sigma(S_1))$, by induction assumption, the new algorithm has a transformation sequence $\langle\sigma(S_1), \{\}\rangle \Longrightarrow^* \langle\emptyset, \sigma_1\rangle$ such that $\sigma_1$ is a matcher of $\sigma(S_1)$, $\mathcal{I}(\sigma_1) = \emptyset$, $\mathcal{D}(\sigma_1) \subseteq \mathcal{FV}(\sigma(S_1))$ and $\sigma_1 \leq \theta_{|-\mathcal{D}(\sigma)}$. Hence the algorithm has a transformation sequence $\langle S, \{\}\rangle \Longrightarrow^* \langle\emptyset, \sigma_1\sigma\rangle$, where $\sigma_1\sigma$ is a matcher of $S$, $\mathcal{I}(\sigma_1\sigma) = \emptyset$, $\mathcal{D}(\sigma_1\sigma) \subseteq \mathcal{FV}(S)$ and $\sigma_1\sigma \leq \theta$. Hence the new algorithm is complete. $\square$

## 6 Implementation and test results

We have implemented the standard second-order matching algorithm [11] and the new algorithm in SML/NJ [8]. We use the data structures for terms in the theorem prover Isabelle [15]. The sizes of the programs are 359 lines and 571 lines of SML source code, respectively[4]. To compare these two algorithms, more than 40 examples have been tested on a SUN SPARC-Station-20 with 64 megabytes of memory.

Terms in a matching pair always have the same type. A term is a *constant*, a *free variable*, a *bound variable*, an *abstraction*, or an *application*. To avoid the renaming of bound variables, de Bruijn's name-free representation is used.

First, we outline the implementation of the standard second-order matching algorithm. Two main functions are used: the function *simplification* corresponds to rule (D); and the function *flex_rigid* contains two local functions corresponding to rules (P) and (I), respectively. These functions are applied to a matching problem and transform it into a set of new ones. For each new matching problem, this process will be continued, until each matching problem is either a substitution or no functions can be applied to it. All substitutions form the complete set of matchers for the given matching problem.

For the new matching algorithm, a given matching problem will first be simplified using the function *simplification*. The function *flex_rigid* in the standard algorithm is replaced by a new function *FFR*, which contains the local functions *right_bound*,

[4] The programs are available under:
`http://www.informatik.uni-bremen.de/~shi/shi_imp.html`

| | Matching pairs | $\mathcal{S}$ | $\mathcal{T}_1$ | $\mathcal{T}_2$ |
|---|---|---|---|---|
| 1 | $\lambda\overline{x_2}.F(x_1, x_2, a_1) \overset{\triangleleft}{=} \lambda\overline{x_2}.f(x_1, g(x_1), a_1)$ | 4 | 0.24 | 0.20 |
| 2 | $\lambda\overline{x_2}.F(x_1, x_2) \overset{\triangleleft}{=} \lambda\overline{x_2}.f(a_1, a_2)$ | 2 | 0.20 | 0.16 |
| 3 | $\lambda\overline{x_2}.F(G_1(x_1), G_2(x_2)) \overset{\triangleleft}{=} \lambda\overline{x_2}.f(g(a_1), h(a_2, a_3))$ | 19 | 0.69 | 0.56 |
| 4 | $\lambda\overline{x_3}.F(G_1(x_1), G_2(x_2), G(x_3)) \overset{\triangleleft}{=} \lambda\overline{x_3}.f(x_1, h(a_2, a_3))$ | 16 | 0.41 | 0.33 |
| 5 | $\lambda\overline{x_3}.F(G_1(x_1), G_2(x_2), G_3(x_3)) \overset{\triangleleft}{=}$ $\lambda\overline{x_3}.f(g(x_1), h_1(x_1, a_1), h_2(a_2, a_3))$ | 125 | 1.97 | 1.38 |
| 6 | $\lambda\overline{x_3}.F(G_1(x_1), G_2(x_2), G_3(x_3)) \overset{\triangleleft}{=}$ $\lambda\overline{x_3}.f(g_1(x_1), g_2(x_2), h_1(x_1, x_2), h_2(a_1, a_2))$ | 265 | 4.38 | 0.78 |
| 7 | $\lambda\overline{x_4}.F(G_1(x_1), G_2(x_2), G_3(x_3), G_4(x_4)) \overset{\triangleleft}{=} \lambda\overline{x_4}.f(g_1(x_1),$ $g_2(x_2), g_3(x_3), h_1(a_1, a_2), h_2(x_1, a_3), h_3(x_2, a_4), h_4(x_1, x_2))$ | $> 10000$ | 577.89 | 158.99 |
| 8 | $\lambda\overline{x_3}.F(G_1(x_1), G_2(x_2), G_3(x_3)) \overset{\triangleleft}{=}$ $\lambda\overline{x_3}.f(g_1(x_1), g_2(x_2), h_1(x_1, x_2), h_2(a_1, a_2), h_3(x_2, a_3))$ | 587 | 11.85 | 4.14 |
| 9 | $\lambda\overline{x_4}.F(G_1(x_1), G_2(x_2), G_3(x_3), G_4(x_4)) \overset{\triangleleft}{=} \lambda\overline{x_4}.f(g_1(x_1),$ $g_2(x_2), g_3(x_3), g_4(x_4), h_1(a_1, a_2), h_2(x_1, a_3))$ | 3462 | 150.69 | 39.75 |
| 10 | $\lambda\overline{x_3}.F(G_1(x_1), G_2(x_2), G(x_3), a_1) \overset{\triangleleft}{=}$ $\lambda\overline{x_3}.f(g_1(x_1), g_2(x_2), h(x_3, a_2), a_1)$ | 113 | 5.78 | 1.00 |
| 11 | $\lambda\overline{x_4}.F(G_1(x_1), G_2(x_2), G_3(x_3), G_4(x_4)) \overset{\triangleleft}{=} \lambda\overline{x_4}.f(g_1(x_1),$ $g_2(x_2), h_1(x_1, x_2), h_2(a_1, a_2), h_3(x_2, a_3), h_4(x_1, a_4))$ | $> 10000$ | 346.26 | 57.67 |
| 12 | $\lambda\overline{x_4}.F(G_1(x_1), G_2(x_2), G_3(x_3)) \overset{\triangleleft}{=}$ $\lambda\overline{x_4}.f(g_1(x_1), g_2(x_2), g_3(x_3), g_4(x_4))$ | 152 | 3.29 | 0.20 |
| 13 | $\lambda\overline{x_4}.F(G_1(x_1), G_2(x_2), G_3(x_3), G_4(x_4)) \overset{\triangleleft}{=}$ $\lambda\overline{x_4}.f(g_1(x_1), g_2(x_2), g_3(x_3), g_4(x_4))$ | 354 | 13.50 | 0.54 |
| 14 | $\lambda\overline{x_4}.F(G_1(x_1), G_2(x_2), G_3(x_3), G_4(x_4)) \overset{\triangleleft}{=}$ $\lambda\overline{x_4}.f(g_1(x_1), g_2(x_2), g_3(x_3), g_4(x_4), h(a_1, a_2))$ | 1359 | 49.86 | 5.27 |
| 15 | $\lambda\overline{x_3}.F(G_1(x_1), G_2(x_2), G_3(x_3)) \overset{\triangleleft}{=} \lambda\overline{x_3}.f(g_1(x_1),$ $g_2(x_1), g_3(x_1), h_1(x_1, a_1), h_2(x_1, a_2), h_3(a_3, a_4))$ | 978 | 26.90 | 44.70 |
| 16 | $\lambda\overline{x_3}.F(G_1(x_1), G_2(x_2), G_3(x_3)) \overset{\triangleleft}{=} \lambda\overline{x_3}.f(g_1(x_1), g_2(x_2),$ $g_3(x_3), h_1(a_1, a_2), h_2(x_1, a_3), h_3(x_2, a_4), h_4(x_1, x_2))$ | 1704 | 54.93 | 81.30 |

Figure 1. Test results.

*right_rest* and *composition*, corresponding to the first three steps described in Section 4, respectively. In the functions *right_bound* and *right_rest* the matching algorithm is called recursively.

In order to show how much time performance gain we can get, the above table contains the CPU-time-costs $\mathcal{T}_1$ and $\mathcal{T}_2$ (in seconds) for both algorithms on several examples. In addition, the table contains the total number of fresh free variables $\mathcal{S}$ for the standard algorithm, which indicate to some extent the additional space cost during execution of the standard algorithm.

We can make the following observations and explanations on the data in the table:

- The first group shows that the new algorithm may have a better performance than the standard one even for very simple matching problems. For the given examples, the new algorithm gets about 20% speedup in execution time.
- The second group consists of some typical applications in our program transformation system. The examples show that the more bound variables the right-hand sides in a matching problem have, the bigger improvement the new algorithm has in execution time than the standard one. The reason is that the mechanism for foreseeing the failure branches in the new algorithm is based on occurrences of bound variables. In average, the new algorithm gets about 80% speedup in execution time for the given examples in this group.
- The third group seems to show that the new algorithm may be less efficient than the standard one for the matching problems, whose right-hand sides have much more constant symbols than bound variables. In general, for subterms independent of occurrences of bound variables, the new algorithm may have to consider all possible combinations of matching subproblems in the flexible-rigid case just like the standard algorithm. Therefore any preparations for foreseeing the failure branches would be useless for such subterms. If such subterms are much more than occurrences of bound variables, then the cost for the preparations may not pay off the cost reduction due to cutting the failure branches.

## 7 Conclusion

We have presented a new second-order matching algorithm. Our implementation shows that the new algorithm can be more efficient than the standard one for many matching problems. The implementation will be used in a prototypical program transformation system [13, 17].

For a certain perspective, the standard second-order matching algorithm is in fact a specialization of an algorithm for unification of arbitrary simply typed $\lambda$-terms [9]. Therefore one may ask whether our algorithm can be extended into a second-order equational matching algorithm, which might be more efficient than higher-order equational unification algorithms [18, 14, 16] used for second-order equational matching [5]. Indeed, Curien and Qian proposed such an algorithm for second-order AC matching [6], but have not implemented it. A motivation for designing a second-order AC matching algorithm is to allow usual binary AC logical operators $\vee$ and $\wedge$ in computing similarities of formulas (cf. [4]).

**Acknowledgement.** We thank the anonymous referees for their corrections and comments.

## References

1. T. Boy de la Tour and R. Caferra. Proof analogy in interactive theorem proving: A method to express and use it via second order pattern matching. In *proceedings of AAAI'87*, 1987.

2. T. Boy de la Tour and R. Caferra. A formale approach to some usually informal techniques used in mathematical reasoning. In *ISSAC'88*, pages 402–406. Lecture Notes in Computer Science, 1988.
3. R. S. Boyer and J. S. Moore. A Theorem Prover for a Computational Logic. In *Proc. of the 10th International Conference on Automated Deduction*, 1990.
4. R. Curien. *Outils pour la preuve par analogie.* PhD thesis, Université Henri Poincaré – Nancy 1, January 1995.
5. R. Curien. Second Order E-matching as a Tool for Automated Theorem Proving. In *Proceedings of EPIA'93 (Portugese Conference on Artificial Intelligence) Porto, Portugal.*, volume 727 of *Lecture Notes in Artificial Intelligence*, 93.
6. R. Curien and Z. Qian. Efficiency for second-order matching: the syntactic and AC-cases. Technical report, 1995. Draft paper.
7. G. Dowek. A Second-order Pattern Matching Algorithm in the Cube of Typed $\lambda$-calculi. In *Mathematical Fundation of Computer Science*, LNCS 520, 1991.
8. R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical report, Dept. of Cmputer Science, University of Edinburg, 1986.
9. G. Huet. A unification algorithm for typed $\lambda$-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
10. G. Huet. *Résolution d'Equations dans les langages d'Ordre 1,2, ...,$\omega$.* Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.
11. G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
12. T. Kolbe and C. Walther. Reusing proofs. In A. Cohn, editor, *ECAI'94, 11th European Conference on Artificial Intelligence*, pages 80 – 84, 1994.
13. B. Krieg-Brüchner, J. Liu, H. Shi, and B. Wolff. Towards correct, effficient and reusable transformational developments. In *"KORSO: Methods, Languages, and Tools for the Construction of Correct Software"*, ages 270–284. LNCS 1009, 1995.
14. T. Nipkow and Z. Qian. Modular higher-order $E$-unification. In R. Book, editor, *Proc. 4th Int. Conf. Rewriting Techniques and Applications*, pages 200-214. LNCS 488, 1991.
15. L. Paulson. *Isabelle - A Generic Theorem prover.* LNCS 828, 1994.
16. Z. Qian and K. Wang. Higher-order $E$-unification for arbitrary theories. In K. Apt, ed., *Proc. 1992 Joint Int. Conf. and Symp. on Logic Programming.* MIT Press, 1992.
17. H. Shi. *Extended Matching with Applications to Program Transformation.* PhD thesis, Universität Bremen, 1994.
18. W. Snyder. Higher-order $E$-unification. In M. Stickel, editor, *Proc. 10th Int. Conf. Automated Deduction*, pages 573–587. Springer-Verlag LNCS 449, 1990.
19. W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *J. Symbolic Computation*, 8(1 & 2):101–140, 1989.