Second-Order Matching via Explicit Substitutions[⋆]

Flávio L. C. de Moura[⋆⋆1] and Fairouz Kamareddine[2] and Mauricio Ayala-Rincón[⋆⋆⋆1]

[1] Departamento de Matemática, Universidade de Brasília, Brasília D.F., Brasil.
flavio@mat.unb.br,ayala@mat.unb.br

[2] School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland. fairouz@macs.hw.ac.uk

Abstract. Matching is a basic operation extensively used in computation. Second-order matching, in particular, provides an adequate environment for expressing program transformations and pattern recognition for automated deduction. The past few years have established the benefit of using explicit substitutions for theorem proving and higher-order unification. In this paper, we will make use of explicit substitutions to facilitate matching: we develop a second-order matching algorithm via the $\lambda\sigma$-style of explicit substitutions. We introduce a convenient notation for matching in the $\lambda\sigma$-calculus. This notation keeps the matching equations separated from the incremental graftings. We characterise an important class of second-order matching problems which is essential to prove termination of the algorithm. In addition, we illustrate how the algorithm works through some examples.

Keywords: Higher-Order Unification, Second-Order Matching, Explicit Substitutions.

1 Introduction

Matching is an important mechanism extensively used in automated deduction and programming languages. For instance, second-order matching has been used in program transformation [HL78,Vis04] and theorem proving [dlTC87,dlTC88].

First-order matching, as well as first-order unification, is decidable and unitary, i.e., when a unifier exists it is unique in the sense that the most general unifier (mgu) exists [Rob65]. Second-order matching is still decidable [Hue76], but the solutions are not necessarily unique and the notion of an mgu no longer exists. In fact, the second-order matching problem[3] $\lambda x.(X\ a) \ll^? \lambda x.(c(b\ a))$, where a, b and c are constants and X is a meta-variable, has two solutions given by $X/\lambda y.(c(b\ a))$ and $X/\lambda y.(c(b\ y))$ and, none of them is an instance of the

⋆ Work supported by funds from CNPq(CT-INFO) 50.6598/04-7 and PRONEX.

⋆⋆ Corresponding author. Supported by Brazilian CAPES Foundation.

⋆⋆⋆ Partially supported by Brazilian CNPq Council.

[3] The type information is omitted to simplify the presentation of the example.

other, and hence there is no mgu. Third and fourth order matching are decidable [Dow94,Pad00], but for higher orders, it remains unknown (for almost thirty years) whether this problem is decidable [Hue76]. In [Loa03], the undecidability of fifth-order β-matching is given, but the proof does not deal with the general case that includes η-conversion.

In [DHK00], Dowek, Hardin and Kirchner gave a general method for higher-order unification for the $\lambda\sigma$-calculus of explicit substitutions. In that paper they prove that the unification problem P has a solution in the simply typed λ-calculus if and only if the translation of this problem in the language of the $\lambda\sigma$-calculus, written P_F has a solution. However, this general unification method, which has been proved adaptable for other explicit substitutions calculi [ARK01], does not decide second-order matching in the $\lambda\sigma$-calculus as we show by a non-terminating counter-example. In addition, [Bur89] shows that matching may behave differently from unification depending on the considered equational theory and, therefore it is of interest to study matching via explicit substitutions.

In this paper we develop a second-order matching algorithm that decides a special subset of $\lambda\sigma$-terms. The contributions of this work are as follows:

1. We characterise an important subset of second-order $\lambda\sigma$-terms which the general method of Dowek, Hardin and Kirchner can decide. This subset contains all the $\lambda\sigma$-terms that can appear in a second-order matching problem derived from another matching problem originated in the simply typed λ-calculus.
2. Since the notation used by Dowek, Hardin and Kirchner is not adequate for matching because it may introduce flexible-flexible equations whose right-hand sides need to be instantiated, we present an adequate notation for dealing with matching in the $\lambda\sigma$-calculus. This notation keeps graftings (first-order substitutions) separated from the matching equations to be unified. This separation will be important during the matching because no variable, which can be instantiated, is included in the right-hand side of a matching equation and, therefore each matching rule will necessarily generate another matching problem.
3. We present a second-order matching algorithm that decides the subset of $\lambda\sigma$-terms characterised in item 1.

Using the $\lambda\sigma$-style of explicit substitutions has the well known advantage of reducing higher-order unification problems into equivalent first-order equational unification problems, and in this way, the variable instantiation mechanism of the λ-calculus is implemented by first-order substitution (grafting). Advantages of this HOU approach include, among others: being closer to implementations which is inherent to explicit substitutions; avoidance of functional encoding of scoping constraints by separating substitutions from reductions and substitutions from unification variables; conceiving HOU as equational unification modulo $\beta\eta$-conversion, which allows for natural mixing of higher order specifications with equational ones as explained in [DHK00]. Nevertheless, since higher-order unification is undecidable [Gol81], it is important to study decidable subproblems over specific λ-terms as well as of its extensions, such as the $\lambda\sigma$-calculus. In this way, this work is worthwhile because the presented algorithm decides the subset of second-order $\lambda\sigma$-terms characterised in item 1.

In the next section we give a brief presentation of the simply typed version of the λ- and $\lambda\sigma$-calculi. In section 3 we start with the characterisation of a subset of $\lambda\sigma$-terms. Afterwards, we define an adequate notation for dealing with matching problems and then, we present a second-order matching algorithm for the $\lambda\sigma$-calculus. Finally, we conclude and give directions for future work.

2 Background

We start this section with a brief presentation of the simply typed λ- and $\lambda\sigma$-calculus and some basic definitions used throughout the paper. The notation used in this presentation uses de Bruijn indexes [dB72] instead of variables with names. This is because de Bruijn's notation is more adequate for implementations of the λ-calculus since α-conversion is no longer needed.

We define types and simply typed λ-terms in de Bruijn notation as usual:

`types` $A ::= K \mid A \rightarrow B$, where K is an atomic type.
`contexts` $\Gamma ::= nil \mid A.\Gamma$
`terms` $a ::= \underline{\mathtt{n}} \mid X \mid (a\ a) \mid \lambda_A.a$, where $n \in \mathbb{N} = \{1, 2, \ldots\}$ and $X \in \mathcal{X}$, the set of meta-variables.

The set of λ-terms built with this grammar is usually denoted by $\Lambda_{dB}(\mathcal{X})$ and the typing rules are as follows:

$$\text{(var)} \quad \frac{}{A.\Gamma \vdash \underline{\mathtt{1}} : A} \qquad\qquad \text{(var n)} \quad \frac{\Gamma \vdash \underline{\mathtt{n}} : B}{A.\Gamma \vdash \underline{\mathtt{n+1}} : B}$$

$$\text{(app)} \quad \frac{\Gamma \vdash a : A \rightarrow B \quad \Gamma \vdash b : A}{\Gamma \vdash\ (a\ b) : B} \qquad\qquad \text{(lambda)} \quad \frac{A.\Gamma \vdash a : B}{\Gamma \vdash \lambda_A.a : A \rightarrow B}$$

The type judgement $\Gamma \vdash a : A$ can also be written as a^{Γ}_{A}.

To each meta-variable X we associate a unique type A and a unique context Γ. We assume that for each type there exists an infinite set of meta-variables with that type. We add the following typing rule for meta-variables:

$$\text{(Metavar)} \quad \Gamma \vdash X : A, \qquad\qquad \text{where } \Gamma \text{ is any context.}$$

β- and η-contraction are defined as usual and $=_{\beta\eta}$ denotes $\beta\eta$-conversion.

Definition 1 (Order of types and terms). *The order of a term is the order of its type and the order of a type A, written as $|A|$, is defined by:*

1. *If A is atomic then $|A| = 1$;*
2. *If $A = B \rightarrow C$ then $|A| = max\{1 + |B|, |C|\}$.*

Unification problems deal with *unification equations* which are defined by:

Definition 2 (Unification equation). *A* unification equation *is an equation of the form $a =^? b$ where a and b are λ-terms of the same type which are well-typed under the same context. The* order *of a unification equation is the highest order of the meta-variables occurring in it. A unification equation is called* *flexible-flexible or rigid-rigid if the left and right-hand sides of the equation are both flexible or rigid terms, respectively. If one term is rigid and the other is flexible (independently of the order) the equation is called flexible-rigid. A unification equation is called* trivial *if it has the form $a =^? a$.*

Example 3. Let $\Gamma = A.A \to A.A \to B.B.nil$ and X and Y meta-variables such that $\Gamma \vdash X : A \to A$ and $\Gamma \vdash Y : (A \to A) \to B$. The unification equation $(\underline{3}(\underline{2}(X\underline{1}))) =^? \underline{4}$ has order 2 (since X has order 2), while $Y\ X =^? \underline{4}$ has order 3.

Definition 4 (Unifier). *A* unifier *for a given unification equation, say $a =^? b$, is a substitution σ such that $a\sigma =_{\beta\eta} b\sigma$.*

Definition 5 (Unification problem). *A* unification problem *is a finite set of unification equations. The* order *of a unification problem is given by the highest order amongst its unification equations. A solution of a unification problem P is a substitution which is a unifier for all equations in P. In other words, a solution for P is a substitution σ such that $P\sigma$ is the trivial unification problem (i.e., formed only by trivial equations).*

Definition 6 (Matching equation[4]). *A* higher-order matching equation *is an equation of the form $a \ll^? b$, where a and b are λ-terms of the same type which are well typed under the same context and, such that the right hand side does not contain meta-variables.*

Definition 7 (Matcher). *A* matcher *for a given matching equation, say $a \ll^? b$, is a substitution σ such that $a\sigma =_{\beta\eta} b$.*

This definition corresponds to the notion of "filtering", which becomes from the assumption that the term to be matched have disjoint variable sets or they can be renamed as usual in rewriting systems and pattern matching. The alternative notion of "semi-unification" ($\exists\sigma, a\sigma =_{\beta\eta} b\sigma =_{\beta\eta} b$) is not treated here [Bur89].

Definition 8 (Matching problem). *A* higher-order matching problem *is a finite set of matching equations. The order of a matching problem is given by the highest order of its meta-variables.*

The $\lambda\sigma$-calculus of explicit substitutions extends the λ-calculus with explicit operators to simulate the substitution (meta-)operation of the λ-calculus.

The syntax of the typed $\lambda\sigma$-calculus is given by

Types	$A ::= K \mid A \to B$	
Contexts	$\Gamma ::= nil \mid A.\Gamma$	
Terms	$a\ ::= \underline{1} \mid X \mid (a\ b) \mid \lambda_A.a \mid a[s]$	where $X \in \mathcal{X}$
Substitutions	$s\ ::= id \mid \uparrow\ \mid a.s \mid s \circ s$	

The set of $\lambda\sigma$-terms is written as $\Lambda_{\lambda\sigma}(\mathcal{X})$.

[4] Adapted from [Dow01]

The $\lambda\sigma$-typing rules are given by:

(var) $\quad A.\Gamma \vdash \underline{1} : A$ (lambda) $\quad \dfrac{A.\Gamma \vdash a : B}{\Gamma \vdash \lambda_A.a : A \to B}$

(app) $\quad \dfrac{\Gamma \vdash a : A \to B \quad \Gamma \vdash b : A}{\Gamma \vdash \ (a\ b) : B}$ (clos) $\quad \dfrac{\Gamma \vdash s \triangleright \Gamma' \quad \Gamma' \vdash a : A}{\Gamma \vdash a[s] : A}$

(id) $\quad \Gamma \vdash id \triangleright \Gamma$ (shift) $\quad A.\Gamma \vdash \uparrow \triangleright \Gamma$

(cons) $\quad \dfrac{\Gamma \vdash a : A \quad \Gamma \vdash s \triangleright \Gamma'}{\Gamma \vdash a.s \triangleright A.\Gamma'}$ (comp) $\quad \dfrac{\Gamma \vdash s'' \triangleright \Gamma'' \quad \Gamma'' \vdash s' \triangleright \Gamma'}{\Gamma \vdash s' \circ s'' \triangleright \Gamma'}$

In addition, to each meta-variable X we associate a unique type T_X and a unique context Γ_X. We assume that for each pair (Γ, A) there is an infinite set of meta-variables X such that $\Gamma_X = \Gamma$ and $T_X = A$. We add the following type rule for meta-variables:

$$\text{(Metavar)} \quad \Gamma_X \vdash X : T_X$$

We use the $\lambda\sigma$-rules and the unification rules (named **Dec-λ**, **Dec-App**, **Dec-Fail**, **Exp-λ**, **Exp-App**, **Normalise** and **Replace**) for the $\lambda\sigma$-calculus as presented in [DHK00].

3 Second-order Matching via Explicit Substitutions

The language of the $\lambda\sigma$-calculus is a non-trivial extension of the language of the λ-calculus, and hence, the decidability of second-order matching arises naturally in the $\lambda\sigma$-calculus. An obvious step to solve second-order matching problems in the $\lambda\sigma$-calculus would be to adapt the higher-order procedure for the $\lambda\sigma$-calculus of [DHK00] to solve second-order matching problems. As we will see in the next section, the procedure given in [DHK00] does not terminate for all second-order matching problems in the $\lambda\sigma$-calculus. Nevertheless, we characterise a sub-set of $\lambda\sigma$-terms for which we can decide second-order matching problems.

3.1 An Important Class of $\lambda\sigma$-terms

In this section we characterise an important class of $\lambda\sigma$-terms, and in the next section, we design a second-order matching algorithm that decides this class. The necessity to define this class is due to the fact that the unification method [DHK00] does not terminate for all second-order matching problems written in the $\lambda\sigma$-style. Hence this method does not decide second-order matching in the $\lambda\sigma$-calculus. The counter-example is the following:

$$X_A^{A \to A.\Gamma}[(\lambda_A.\underline{1}_A^{A.\Gamma})^{\Gamma}_{A \to A}.id^{\Gamma}_{\Gamma}]^{\Gamma}_A =^?_{\lambda\sigma} b^{\Gamma}_A$$

where b is a given closed term, i.e. a term without occurrences of meta-variables, and Γ is a given context.

We can build the following derivation:

$X_A^{A \to A.\Gamma}[(\lambda_A.\underline{1}_A^{A.\Gamma})^{\Gamma}_{A \to A}.id^{\Gamma}_{\Gamma}]^{\Gamma}_A =^?_{\lambda\sigma} b^{\Gamma}_A \to^{\mathbf{Exp-App}}$

$$\begin{array}{l} X_A^{A\to A.\Gamma}[(\lambda_A.\underline{1}_A^{A.\Gamma})_{A\to A}^{\Gamma}.id_\Gamma^\Gamma] =_{\lambda\sigma}^{?} b_A^\Gamma \wedge \\ X_A^{A\to A.\Gamma} =_{\lambda\sigma}^{?} (\underline{1}_{A\to A}^{A\to A.\Gamma} Y_A^{A\to A.\Gamma})_A^{A\to A.\Gamma} \to^{\mathbf{Replace}} \\ (\underline{1}_{A\to A}^{A\to A.\Gamma} Y_A^{A\to A.\Gamma})_A^{A\to A.\Gamma}[(\lambda_A.\underline{1}_A^{A.\Gamma})_{A\to A}^{\Gamma}.id_\Gamma^\Gamma] =_{\lambda\sigma}^{?} b_A^\Gamma \wedge \\ X_A^{A\to A.\Gamma} =_{\lambda\sigma}^{?} (\underline{1}_{A\to A}^{A\to A.\Gamma} Y_A^{A\to A.\Gamma})_A^{A\to A.\Gamma} \to^{\mathbf{Normalise}} \\ Y_A^{A\to A.\Gamma}[(\lambda_A.\underline{1}_A^{A.\Gamma})_{A\to A}^{\Gamma}.id_\Gamma^\Gamma] =_{\lambda\sigma}^{?} b_A^\Gamma \wedge \\ X_A^{A\to A.\Gamma} =_{\lambda\sigma}^{?} (\underline{1}_{A\to A}^{A\to A.\Gamma} Y_A^{A\to A.\Gamma})_A^{A\to A.\Gamma} \end{array}$$

At this point we can repeat the strategy **Exp-App**, **Replace** and **Normalise** since the last problem generated (see the last two lines) is composed by two flexible-rigid equations, the first of which is equivalent to the original problem up to renaming of meta-variables.

The class of $\lambda\sigma$-matching problems that we are going to characterise is strongly based on second-order matching problems that are generated in the simply typed λ-calculus. Let M be a matching problem in the simply typed λ-calculus. In order to solve M in the $\lambda\sigma$-calculus, we need first to rewrite M in the $\lambda\sigma$-language. This translation is given by the following *precooking* function:

Definition 9 (Precooking [DHK00]). *Let $a \in \Lambda_{dB}(\mathcal{X})$ such that $\Gamma \vdash a : A$. To every meta-variable X of type B in the term a, we associate the type B and the context Γ in the $\lambda\sigma$-calculus. The* precooking *of a from $\Lambda_{dB}(\mathcal{X})$ to the set $\Lambda_{\lambda\sigma}(\mathcal{X})$ of $\lambda\sigma$-terms is given by $a_F = F(a, 0)$, where $F(a, n)$ is defined by:*

1. *$F((\lambda_B.a), n) = \lambda_B(F(a, n+1))$.*
2. *$F(\underline{\mathtt{k}}, n) = \underline{1}[\uparrow^{k-1}]$.*
3. *$F((a\ b), n) = (F(a, n)\ F(b, n))$.*
4. *$F(X, n) = X[\uparrow^n]$.*

Notice that $F(1, n)$ and $F(X, 0)$ are resp. $\underline{1}$ and X since $\uparrow^0 = id$. The precooking translation is a function that takes a term from the simply typed λ-calculus and returns an equivalent term in the language of the simply typed $\lambda\sigma$-calculus. This translation is essential to avoid variable capture since the HOU procedure in the $\lambda\sigma$-calculus uses first-order substitution (grafting).

There are two important points that should be emphasised during the precooking translation: first, the unique context associated to each meta-variable in the simply typed λ-calculus in de Bruijn notation is the same unique context associated to the translated meta-variable in the $\lambda\sigma$-calculus, i.e., if $\Gamma \vdash X : A$ then $\Gamma \vdash X_F : A$; second, only meta-variables have their structure changed (in order to avoid variable capture when performing graftings) which means that $\lambda\sigma$-terms without occurrences of meta-variables are always in the image of the precooking translation. This last remark will be particularly important for matching. Although the precooking translation replaces the de Bruijn index $\underline{\mathtt{n}}$ by its codification $\underline{1}[\uparrow^{n-1}]$, here, we avoid using this codification for clarity. To give a better intuition of what happens during the precooking translation, consider a (general) simply typed λ-term a. Suppose that a contains a meta-variable X

which is under the scope of n abstractors:

$$\lambda_{A_1} \dots \lambda_{A_n} . \cdots \; (\; X_B^{A_1 . \cdots . A_n . \Delta} \;) \; \cdots$$

After the precooking translation we get:

$$\lambda_{A_1} \dots \lambda_{A_n} . \cdots \; (\; X_B^{\Delta}[(\uparrow^n)_{\Delta}^{A_1 . \cdots . A_n . \Delta}]_B^{A_1 . \cdots . A_n . \Delta} \;) \; \cdots$$

which is a shorthand for the simultaneous substitution:

$$\lambda_{A_1} \dots \lambda_{A_n} . \cdots \; (\; X_B^{\Delta}[(\underline{\mathtt{n+1}}.\underline{\mathtt{n+2}}. \cdots)_{\Delta}^{A_1 . \cdots . A_n . \Delta}]_B^{A_1 . \cdots . A_n . \Delta} \;) \; \cdots$$

Since in $\lambda\sigma$ one uses grafting, the precooking translation is the correct way to 'protect' the meta-variables and to avoid possible variable capture. The substitution $\uparrow^n$ applied to the meta-variable X, i.e., $X[\uparrow^n]$ means, on one hand, that every free de Bruijn index occurring in the term to be substituted by X must be updated by n and, on the other hand, that the first n terms of any substitution applied to $X[\uparrow^n]$ will be ignored. That is, $X[\uparrow^n][s]$ will be reduced to $X[s_{>n}]$, for any substitution s, where $s_{>n}$ represents the elements in the list s which are in positions greater than n. This means that the redexes related to the abstractors appearing in the initial problem cannot introduce terms in the substitution list applied to meta-variables. Hence, terms to be included in this list should be arguments of β-redexes generated by new abstractors which are created only by the rule **Exp-λ**.

Definition 10 (Unification Path/Matching Path). *Let P be a unification (resp. matching) problem. We say that P' is in the* unification (resp. matching) path *of P if $P \rightarrow^* P'$, where the relation $\rightarrow^*$ means $n \geq 0$ applications of any unification (resp. matching) rules.*

The next proposition characterises second-order problems in the language of the $\lambda\sigma$-calculus that can be decided by the method given by [DHK00].

Proposition 11 (Characterisation of a special subclass of $\lambda\sigma$-terms). *Let P_A^Γ be a second-order unification problem which is in the image of the precooking translation. Then every flexible term occurring in $P_A'^\Gamma$ which is in the unification path of P_A^Γ using the unification rules of [DHK00], and of the form $X[s]$, with X of atomic type and s in σ-normal form, is such that every element in the list s with functional type is a de Bruijn index.*

Proof. The proof is by induction on the size of the derivation that generated the term that contains $X[s]$ as sub-term. Without loss of generality we may assume that P_A^Γ is in $\lambda\sigma$-normal form (otherwise we can apply one step of **Normalise**). We use IH for the induction hypothesis.

If the considered equation belongs to P_A^Γ then by the definition of precooking, the substitution s is of the form $\uparrow^n$, for some $n \geq 0$ and, hence the proposition holds since every term in the substitution $\uparrow^n$ is a de Bruijn index.

Now suppose that the proposition holds for $P_A'^\Gamma$ which by hypothesis is in the unification path of P_A^Γ. Let $P_A''^\Gamma$ be such that $P_A'^\Gamma \rightarrow^r P_A''^\Gamma$ and r is any unification rule as given in [DHK00] except **Dec-Fail** since it does not generate a new unification problem. We have the following cases:

- If r is **Dec-λ** or **Replace** then $X[s]$ was already in P'^{Γ}_{A} since these rules do not change the structure of substitutions. The proposition follows by IH.
- If r is **Dec-App** then either $X[s]$ corresponds to one of the arguments a_i of $\underline{\mathtt{n}}\ a_1 \dots a_p$ or the equation was already in P'^{Γ}_{A}. In both cases, s satisfies the proposition by IH.
- If r is **Exp-λ** then either $X[s]$ is a sub-term of the new equation or it was already in P'^{Γ}_{A}. In the former case, the sole new meta-variable that is introduced has the form Y, that should be seen as $Y[id]$ and then the proposition holds. In the latter case the proposition holds by IH.
- If r is **Exp-App** then either $X[s]$ is a sub-term of one of the terms occurring among the new equations or it was already in P'^{Γ}_{A}. In the former case, all the new meta-variables $H_1, \dots, H_k$ have the form $H_i[id]$ and then the proposition holds. In the latter case the proposition holds by IH.
- If r is **Normalise** then there are two cases that we need to consider:
  1. The application of **Normalise** is preceded by an application of **Exp-λ**: In this case, the newly introduced λ's will generate new β-redexes and the steps are as follows. The selected equation before the application of **Exp-λ** had a sub-term of the form:

     $$X^{\Delta}_{B_1\to\cdots\to B_k\to B}[(\uparrow^n)^{A_1.\cdots.A_n.\Delta}_{\Delta}]^{A_1.\cdots.A_n.\Delta}_{B_1\to\cdots\to B_k\to B}$$

     where $B_1,\dots, B_k$ and B are atomic types since X is second order. After an application of **Exp-λ** followed by **Replace** we have:

     $$(\lambda_{B_1}\cdots\lambda_{B_k}.Y^{B_1.\cdots.B_k.\Delta}_{B})^{\Delta}_{B}[(\uparrow^n)^{A_1.\cdots.A_n.\Delta}_{\Delta}]^{A_1.\cdots.A_n.\Delta}_{B}$$

     The normalisation step consists in pushing the substitution inside the new λ's and then performing β-reductions. After pushing the substitution inside these new abstractors we have a sub-term of the form: $\lambda_{B_1}\cdots\lambda_{B_k}.Y^{B_1.\cdots.B_k.\Delta}_{B}[\underline{1}^{B_1.\cdots.B_k.A_1.\cdots.A_n.\Delta}_{B_1}.\cdots.\underline{\mathtt{k}}^{B_1.\cdots.B_k.A_1.\cdots.A_n.\Delta}_{B_k}.$ $(\uparrow^{k+n})^{B_1.\cdots.B_k.A_1.\cdots.A_n.\Delta}_{\Delta}]^{B_1.\cdots.B_k.A_1.\cdots.A_n.\Delta}_{B}$

     The β-reductions that can be performed now will replace arbitrary elements by the first k de Bruijn indexes in the above substitution list, but since all of these λ's have atomic type the proposition holds. The other terms in the substitution list remain unchanged.
  2. **Normalise** was not preceded by an application of **Exp-λ**: Then, an application of **Normalise** is a consequence of an application of **Exp-App** since the rules **Dec-λ**, **Dec-App**, **Dec-Fail** and **Replace**, do not change the structure of the current terms which, by IH are in normal form. Applications of **Exp-App** do not introduce new abstractions and hence the rule *Beta*[5] does not apply. Application of *Abs* introduces a new de Bruijn index in the substitution list, and hence the proposition still holds. None of the others $\lambda\sigma$-rules introduce new terms in the substitution lists of the current unification problem and the proposition holds by IH. $\square$

[5] See the $\lambda\sigma$-rules in [DHK00] or [ACCL91]

In other words, the substitution s in Proposition 11, has the form $a_1.\cdots.a_p.\uparrow^n$ ($a_p \neq \underline{\mathtt{n}}$), such that all the elements $a_1, \ldots, a_p$ are of atomic type, and the other part of the substitution, i.e., $\uparrow^n$ which is a short hand for $\underline{\mathtt{n+1}}.\underline{\mathtt{n+2}}.\cdots$, is formed by an infinite number of different de Bruijn indexes and is the only part which may have elements of functional type. This result is illustrated as:

$$X[\,\underbrace{a_1.\cdots.a_p.}_{\substack{\text{atomic}\\\text{types}}}\;\underbrace{\underline{\mathtt{n+1}}.\underline{\mathtt{n+2}}.\cdots}_{\substack{\text{at most}\\2^{nd}\text{-order types}}}\,]$$

In section 3.3 we present a second-order matching algorithm for $\lambda\sigma$-problems whose terms belong to the class characterised by Proposition 11. Although this class forms a proper subset of all $\lambda\sigma$-terms, this restriction is not important since this class includes all $\lambda\sigma$-terms that occur in a second-order matching problem which is in the matching path of another matching problem that is in the image of the precooking translation. Thus, this class includes all the terms that can be generated by the unification procedure from a second-order matching problem originated in the simply typed λ-calculus (after the precooking translation).

3.2 The Unification by Transformation Notation

Matching problems are characterised by the fact that terms in the right-hand side of equations cannot be instantiated. Therefore, the first difficulty to use the general rules of [DHK00] is related to applications of the rule **Exp-λ** because it introduces a flexible-flexible equation whose right-hand side needs to be instantiated. As an example, let $a \ll^?_{\lambda\sigma} b$ be a second-order matching problem such that the term a has an occurrence of the meta-variable X of type $A \to A$. An application of a rule like **Exp-λ** would generate a new problem of the form $a <^?_{\lambda\sigma} b \wedge X \ll^?_{\lambda\sigma} \lambda_A.Y$, and of course the meta-variable Y needs to be instantiated. To solve this problem we use a notation based on the so called "unification by transformation" approach [Nip93]. According to this approach, a matching problem will be represented by a pair of the form $\langle\sigma, M\rangle$, where σ is a grafting, and M is a matching problem. The advantage of this notation is that we can define matching rules that do not introduce terms that need to be instantiated in the right-hand side of matching equations because graftings and matching equations are kept in different places. For the above example, an application of a rule with the same behaviour of **Exp-λ** should generate from the matching problem $\langle\{\}, a \ll^?_{\lambda\sigma} b\rangle$ the equivalent matching problem $\langle\{X \mapsto \lambda_A.Y\}, \{a \ll^?_{\lambda\sigma} b\}\rangle$.

This notation is independent of the matching rules and, hence we can characterise solved forms without knowing explicitly the matching rules.

Definition 12 (Solved form). *A* solved form *is a pair of the form $\langle\theta, M\rangle$, where the first element of the pair is a grafting and the second element is either the empty set or a finite set of trivial matching equations, i.e., equations of the form $a \ll^?_{\lambda\sigma} a$.*

Now we are ready to define the matching rules.

3.3 The Second-Order Matching Algorithm

The second-order matching rules are given in Table 1. The rules **Dec-**$_m$**-**λ, **Dec-**$_m$**-App**, **Dec-**$_m$**-Fail** and **Normalise**$_m$ correspond respectively to **Dec-**λ, **Dec-App**, **Dec-Fail** and **Normalise** of [DHK00] written in the unification by transformation notation. The rule **Exp**$_m$**-**λ is the matching version of **Exp-**λ. The difference between them, in addition to the notation, is that **Exp**$_m$**-**λ always replaces a meta-variable of functional type by an abstraction whose body is a fresh meta-variable of atomic type and also applies the generated grafting to the current matching problem. This sole step corresponds to several applications of **Exp-**λ and **Replace**. Note that, if no replacement is done, the rule **Exp-**λ can be applied *ad infinitum*. To avoid such infinite reductions, [DHK00] defined fair strategies. The definition of **Exp**$_m$**-**λ avoids the necessity of defining any strategy because the rules in Table 1 cannot be applied to a given second-order matching problem forever. In fact, for a given equation each rule can be applied only once. The rules **Imit** and **Proj** generate grafting for flexible-rigid equations when the head of the flexible term is a meta-variable of atomic type. The main difference between **Imit** and **Proj** is that the latter does not introduce fresh meta-variables. In addition, while **Proj** may generate several different graftings, for **Imit** we have at most one grafting. Moreover, in the rule **Imit**, the head of the term which replaces X is a de Bruijn index of at most third order. This is because the newly introduced meta-variables have at most second-order.

To prove that the rules of Table 1 always terminate for second-order matching problems whose terms belong to the class characterised by Proposition 11, we need to define an adequate measure. We start by giving the length of a $\lambda\sigma$-term:

Definition 13 (Length of a $\lambda\sigma$-term). *Let $a \in \Lambda_{\lambda\sigma}(\mathcal{X})$. We inductively define $|a|$, the length of a, by:*

- *if $a = X$ or $a = \underline{1}$ then $|a| = 1$*
- *if $a = (b\ c)$ then $|a| = |b| + |c|$*
- *if $a = \lambda.b$ then $|a| = 1 + |b|$*
- *if $a = b[s]$ then $|a| = |b| + ||s||$, where the size of a substitution s, written as $||s||$, is inductively defined as:*
  - *if $s = \uparrow$ or $s = id$ then $||s|| = 0$*
  - *if $s = c.d$ then $||s|| = |c| + ||d||$*
  - *if $s = u \circ v$ then $||s|| = ||u|| + ||v||$*

Definition 14. *Let $M = \{a_1 \ll^?_{\lambda\sigma} b_1, \ldots, a_n \ll^?_{\lambda\sigma} b_n\}$ be a matching problem. Define $\mu(M) = (\xi, \xi', \xi'')$ in the following way:*

- *$\xi = \Sigma_{i=1}^n |b_i|$*
- *ξ' = the number of meta-variables occurring in M*
- *ξ'' = the sum of the order of the type of all meta-variables occurring in M.*

Now denote by $<$ the usual lexicographic order over triples.

Dec$_m$-λ	$\dfrac{\langle \sigma, P \cup \{\lambda_A.a \ll^?_{\lambda\sigma} \lambda_A.b\}\rangle}{\langle \sigma, P \cup \{a \ll^?_{\lambda\sigma} b\}\rangle}$
Dec$_m$**-App**	$\dfrac{\langle \sigma, P \cup \{(\underline{\mathtt{n}}\, a_1 \ldots a_p) \ll^?_{\lambda\sigma} (\underline{\mathtt{n}}\, b_1 \ldots b_p)\}\rangle}{\langle \sigma, P \cup \{a_1 \ll^?_{\lambda\sigma} b_1, \ldots, a_p \ll^?_{\lambda\sigma} b_p\}\rangle}$
Dec$_m$**-Fail**	$\dfrac{\langle \sigma, P \cup \{(\underline{\mathtt{n}}\, a_1 \ldots a_p) \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \ldots b_q)\}\rangle}{Fail}$, if $\mathtt{m} \neq \mathtt{n}$.
Exp$_m$-λ	$\dfrac{\langle \sigma, P\rangle}{\exists Y : (A_1.\cdots.A_k.\Gamma \vdash Y : B), \langle \sigma', P\{X \mapsto \lambda_{A_1} \ldots \lambda_{A_k}.Y\}\rangle}$ if $(\Gamma \vdash X : A_1 \to \cdots \to A_k \to B) \in \mathcal{TV}ar(P)$, $Y \notin \mathcal{TV}ar(P)$, and X is not a solved variable. where $\sigma' = \sigma\{X \mapsto \lambda_{A_1} \ldots \lambda_{A_k}.Y\}$
Imit	$\dfrac{\langle \sigma, P \cup \{X[a_1.\cdots.a_p.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \ldots b_q)\}\rangle}{\langle \sigma', P\sigma' \cup \{(\underline{\mathtt{m-n+p}}\, H_1 \ldots H_q)[a_1\sigma'.\cdots.a_p\sigma'.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \ldots b_q)\}\rangle}$ if X has atomic type and $m > n$. where $\sigma' = \sigma\{X \mapsto (\underline{\mathtt{m-n+p}}\, H_1 \ldots H_q)\}$, $H_1, \ldots, H_q$ are meta variables with appropriate type and with contexts $\Gamma_{H_i} = \Gamma_X (\forall 1 \leq i \leq q)$, and $\underline{\mathtt{m-n+p}}$ is at most third order.
Proj	$\dfrac{\langle \sigma, P \cup \{X[a_1.\cdots.a_p.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \ldots b_q)\}\rangle}{\langle \sigma\{X \mapsto \underline{\mathtt{j}}\}, \{P\{X \mapsto \underline{\mathtt{j}}\} \cup \{a_j\{X \mapsto \underline{\mathtt{j}}\} \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \ldots b_q)\}\rangle}$ if X has atomic type, and the j-th element $(1 \leq j \leq p)$ of the list $a_1.\cdots.a_p$ has the same type of X.
Normalise$_m$	$\dfrac{\langle \sigma, P \cup \{a \ll^?_{\lambda\sigma} b\}\rangle}{\langle \sigma', P \cup \{a' \ll^?_{\lambda\sigma} b'\}\rangle}$ if a or b is not in **Eta**-long form. where a' (resp. b') is the **Eta**-long form of a (resp. b), and σ' is obtained from σ by normalising all its terms. if a (resp. b) is not a solved variable and a (resp. b) otherwise.

Table 1. Second-Order Matching Rules

Proposition 15. *Applications of the rules of Table 1 to second-order matching problems whose terms belong to the class characterised by Proposition 11 always terminate.*

Proof. It is enough to show that $\mu(M)$ decreases after the application of any of the rules in Table 1. We write $M \to^r M'$ to denote one step reduction by one application of rule r. Application of **Dec**$_m$**-**λ decreases the size of both sides of the selected equation (see definition 13), therefore $\mu(M') < \mu(M)$. Application of **Dec**$_m$**-App** replaces one equation by a finite number of new equations formed by sub-terms of the previous problem, therefore ξ decreases and we have that $\mu(M') < \mu(M)$. Application of **Dec**$_m$**-Fail** always stops. Application of **Exp**$_m$**-**λ replaces a meta-variable of functional type by a metavariable of atomic type, therefore ξ'' decreases and the first two components of the current triple remain unchanged, therefore $\mu(M') < \mu(M)$. Application of **Imit** introduces $q \geq 0$ fresh meta-variables to the new matching problem, where q is the number of arguments of the head $\underline{\mathtt{m}}$ of the rigid term in the current equation. If $q = 0$

then no new meta-variable is introduced and, hence ξ' decreases. Otherwise, the new equation $(\underline{\mathtt{m-n+p}}\, H_1 \ldots H_q)[a_1\sigma'.\cdots.a_p\sigma'.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \ldots b_q)$, which is rigid-rigid must be followed by an application of **Dec**$_m$**-App** which decreases $\mu(M)$. Application of **Proj** decreases ξ' since it does not introduce new meta-variables. Application of **Normalise**$_m$ cannot be applied successively because the $\lambda\sigma$-calculus is weakly terminating. In this case, even if it is not the case that $\mu(M) < \mu(M')$ and M' is not trivial, one of the other rules must apply. Therefore the reduction terminates. □

Since we are dealing with matching problems, we have that the image of the graftings corresponding to solved forms are $\lambda\sigma$-terms that are always in the image of the precooking translation. In fact, note that the grafting of a solvable matching problem is always of the form $\{X_1 \mapsto a_1, \ldots, X_k \mapsto a_k\}$, where $a_1, \ldots, a_k$ are closed $\lambda\sigma$-terms, i.e., terms without any occurrences of meta-variables. This fact is formalised by the following proposition:

Proposition 16. *Every solved form of a second-order matching problem, obtained by application of the rules in Table 1, is in the image of the precooking translation.*

Proof. Every closed term is in the image of the precooking translation since we only need to rewrite the $\lambda\sigma$-codification of de Bruijn indexes, say $\underline{\mathtt{1}}[\uparrow^n]$ $(n \geq 0)$, into the usual form $\underline{\mathtt{n}}$. Recall that, for clarity, in all the examples and even in the rules, we write $\underline{\mathtt{n}}$ instead of $\underline{\mathtt{1}}[\uparrow^n]$, although this is not the notation used internally by the $\lambda\sigma$-calculus. □

According to Proposition 16, the solved forms are translated back to the simply typed λ-calculus by rewriting the codification of de Bruijn indexes used by the $\lambda\sigma$-calculus by the corresponding de Bruijn index in the λ-calculus. The whole matching process can be represented by the following scheme:

$$\begin{tikzcd} M \arrow[r, "\textbf{Precooking}"] & M_F \arrow[r, "\text{Matching Algorithm}"] & M'_F \arrow[r, "\textbf{Precooking}^{-1}"] & M' \end{tikzcd}$$

Example 17. Let M be the second-order matching problem given by the equation $\Gamma \vdash \lambda_A.(X\ \underline{3}) \ll^? \lambda_A.(\underline{2}(\underline{4}\underline{3})) : A \to B$, whose context is given by $\Gamma = A \to B.A.A \to A.nil$, where A and B are atomic types and $\Gamma \vdash X : A \to B$. After the precooking translation, we have $\lambda_A.(X[\uparrow]\ \underline{3}) \ll^?_{\lambda\sigma} \lambda_A.(\underline{2}(\underline{4}\underline{3}))$. The algorithm generates the following reduction:

$$\begin{tikzcd}
 & \langle \{\}, \{\lambda_A.(X[\uparrow]\ \underline{3}) \ll^{?}_{\lambda\sigma} \lambda_A.(\underline{2}(\underline{4}\ \underline{3}))\}\rangle \arrow[d, "\mathbf{Dec}_m\textbf{-}\lambda"] & \\
 & \langle \{\}, \{(X[\uparrow]\ \underline{3}) \ll^{?}_{\lambda\sigma} (\underline{2}(\underline{4}\ \underline{3}))\}\rangle \arrow[d, "\mathbf{Exp}_m\textbf{-}\lambda"] & \\
 & \langle \{X \mapsto \lambda_A.Y\}, \{((\lambda_A.Y)[\uparrow]\ \underline{3}) \ll^{?}_{\lambda\sigma} (\underline{2}(\underline{4}\ \underline{3}))\}\rangle \arrow[d, "\mathbf{Normalise}_m"] & \\
 & \langle \{X \mapsto \lambda_A.Y\}, \{Y[\underline{3}.\uparrow] \ll^{?}_{\lambda\sigma} (\underline{2}(\underline{4}\ \underline{3}))\}\rangle \arrow[d, "\mathbf{Imit}"] & \\
 & \langle \{Y \mapsto (\underline{2}H_1), X \mapsto (\lambda_A.(\underline{2}H_1))\}, \{(\underline{2}H_1)[\underline{3}.\uparrow] \ll^{?}_{\lambda\sigma} (\underline{2}(\underline{4}\ \underline{3}))\}\rangle \arrow[d, "\mathbf{Normalise}_m"] & \\
 & \langle \{Y \mapsto (\underline{2}H_1), X \mapsto (\lambda_A.(\underline{2}H_1))\}, \{H_1[\underline{3}.\uparrow] =^{?}_{\lambda\sigma} (\underline{4}\ \underline{3})\}\rangle \arrow[dl, "\mathbf{Imit}"'] \arrow[dr, "\mathbf{Proj}"] & \\
T & & T'
\end{tikzcd}$$

where T' is given by:

$$\begin{tikzcd}
\langle \{H_1 \mapsto \underline{1}, Y \mapsto (\underline{2}\ \underline{1}), X \mapsto \lambda_A.(\underline{2}\ \underline{1})\}, \{\underline{3} \ll^{?}_{\lambda\sigma} (\underline{4}\ \underline{3})\}\rangle \arrow[d, "\mathbf{Dec}_m\mathbf{-Fail}"] \\
Fail
\end{tikzcd}$$

and T is given by:

$$\begin{tikzcd}
 & \langle \{H_1 \mapsto (\underline{4}H_2), Y \mapsto (\underline{2}(\underline{4}H_2)), X \mapsto \lambda_A.(\underline{2}(\underline{4}H_2))\}, \{(\underline{4}H_2)[\underline{3}.\uparrow] \ll^{?}_{\lambda\sigma} (\underline{4}\ \underline{3})\}\rangle \arrow[d, "\mathbf{Normalise}_m"] & \\
 & \langle \{H_1 \mapsto (\underline{4}H_2), Y \mapsto (\underline{2}(\underline{4}H_2)), X \mapsto \lambda_A.(\underline{2}(\underline{4}H_2))\}, \{(\underline{4}H_2[\underline{3}.\uparrow]) \ll^{?}_{\lambda\sigma} (\underline{4}\ \underline{3})\}\rangle \arrow[d, "\mathbf{Dec}_m\textbf{-App}"] & \\
 & \langle \{H_1 \mapsto (\underline{4}H_2), Y \mapsto (\underline{2}(\underline{4}H_2)), X \mapsto \lambda_A.(\underline{2}(\underline{4}H_2))\}, \{H_2[\underline{3}.\uparrow] \ll^{?}_{\lambda\sigma} \underline{3}\}\rangle \arrow[dl, "\mathbf{Imit}"'] \arrow[dr, "\mathbf{Proj}"] & \\
T'' & & T'''
\end{tikzcd}$$

where T'' and T''' are, respectively, given by:

$$\langle\{H_2 \mapsto \underline{3}, H_1 \mapsto (\underline{4}\ \underline{3}), Y \mapsto (\underline{2}(\underline{4}\ \underline{3})), X \mapsto \lambda_A.(\underline{2}(\underline{4}\ \underline{3}))\}, \{\underline{3} \ll^?_{\lambda\sigma} \underline{3}\}\rangle$$

and

$$\langle\{H_2 \mapsto \underline{1}, H_1 \mapsto (\underline{4}\ \underline{1}), Y \mapsto (\underline{2}(\underline{4}\ \underline{1})), X \mapsto \lambda_A.(\underline{2}(\underline{4}\ \underline{1}))\}, \{\underline{3} \ll^?_{\lambda\sigma} \underline{3}\}\rangle$$

To prove completeness and correctness of the matching rules of Table 1, we need to consider only the rules **Exp**$_m$**-**λ, **Imit** and **Proj** because for all the other rules the proof is the same as in [DHK00]. As usual, let us call $\mathcal{U}_{\lambda\sigma}(M)$ the set of all $\lambda\sigma$-unifiers (or matchers) of M.

Proposition 18 (Correctness). *Let $S = \{$**Exp**$_m$**-**λ, **Imit**, **Proj**$\}$. The rules in S are correct, i.e., if $M \to^r M'$ then $\mathcal{U}_{\lambda\sigma}(M') \subseteq \mathcal{U}_{\lambda\sigma}(M)$, where $r \in S$.*

Proof. 1. **Exp**$_m$**-**λ: Suppose that γ is a matcher of $P\{X \mapsto \lambda_{A_1} \dots \lambda_{A_k}.Y\}$. This means that the $\lambda\sigma$-normal form of $P\{X \mapsto \lambda_{A_1} \dots \lambda_{A_k}.Y\}\gamma$ is the trivial problem, i.e, $\gamma \in \mathcal{U}_{\lambda\sigma}(P\{X \mapsto \lambda_{A_1} \dots \lambda_{A_k}.Y\})$ and $\{X \mapsto \lambda_{A_1} \dots \lambda_{A_k}.Y\}\gamma \in \mathcal{U}_{\lambda\sigma}(P)$ which shows that $\mathcal{U}_{\lambda\sigma}(P\{X \mapsto \lambda_{A_1} \dots \lambda_{A_k}.Y\}) \subseteq \mathcal{U}_{\lambda\sigma}(P)$.

2. **Imit**: Let γ be a matcher of $P\sigma' \cup \{(\underline{\mathtt{m}-\mathtt{n}+\mathtt{p}}\ H_1 \dots H_q)[a_1\sigma'.\cdots.a_p\sigma'.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\ b_1 \dots b_q)\}$, where $\sigma' = \sigma\{X \mapsto \underline{\mathtt{m}-\mathtt{n}+\mathtt{p}}\ H_1 \dots H_q\}$. This means that the $\lambda\sigma$-normal form of

$$P\sigma'\gamma \cup \{(\underline{\mathtt{m}-\mathtt{n}+\mathtt{p}}\, H_1 \dots H_q)[a_1\sigma'.\cdots.a_p\sigma'.\uparrow^n]\gamma \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\}$$

is the trivial problem. Therefore, $\sigma'\gamma \in \mathcal{U}_{\lambda\sigma}(P \cup \{X[a_1.\cdots.a_p.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\})$, and hence $\mathcal{U}_{\lambda\sigma}(P\sigma' \cup \{(\underline{\mathtt{m}-\mathtt{n}+\mathtt{p}}\, H_1 \dots H_q)[a_1\sigma'.\cdots.a_p\sigma'.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\}) \subseteq \mathcal{U}_{\lambda\sigma}(P \cup \{X[a_1.\cdots.a_p.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\})$.

3. **Proj**: Let γ be a matcher of $P\{X \mapsto \underline{\mathtt{j}}\} \cup \{a_j\{X \mapsto \underline{\mathtt{j}}\} \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\}$, i.e., the $\lambda\sigma$-normal form of $P\{X \mapsto \underline{\mathtt{j}}\}\gamma \cup \{a_j\{X \mapsto \underline{\mathtt{j}}\}\gamma \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\}$ is the trivial problem. Hence, $\{X \mapsto \underline{\mathtt{j}}\}\gamma$ is a matcher of $P \cup \{X[a_1.\cdots.a_p.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\ b_1 \dots b_q)\}$, i.e., $\{X \mapsto \underline{\mathtt{j}}\}\gamma \in \mathcal{U}_{\lambda\sigma}(P \cup \{X[a_1.\cdots.a_p.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\})$, and since $\{X \mapsto \underline{\mathtt{j}}\}\gamma \in \mathcal{U}_{\lambda\sigma}(P\{X \mapsto \underline{\mathtt{j}}\} \cup \{a_j \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\})$, we have that $\mathcal{U}_{\lambda\sigma}(P\{X \mapsto \underline{\mathtt{j}}\} \cup \{a_j \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\}) \subseteq$
$\mathcal{U}_{\lambda\sigma}(P \cup \{X[a_1.\cdots.a_p.\uparrow^n] \ll^?_{\lambda\sigma} (\underline{\mathtt{m}}\, b_1 \dots b_q)\})$. □

Proposition 19 (Completeness). *Let $S = \{$**Exp**$_m$**-**λ, **Imit**, **Proj**$\}$. The rules in S are complete, i.e., if $M \to^r M'$ then $\mathcal{U}_{\lambda\sigma}(M) \subseteq \mathcal{U}_{\lambda\sigma}(M')$, where $r \in S$.*

Proof. 1. **Exp**$_m$**-**λ: Let θ be a $\lambda\sigma$-unifier of $\langle\sigma, P\rangle$ and $X \in \mathcal{T}var(P)$ such that $\Gamma \vdash X : A_1 \to \dots \to A_k \to B$. Thus $X\theta = a : A_1 \to \dots \to A_k \to B$ and we can assume that a is of the form $\lambda_{A_1} \dots \lambda_{A_k}.b$ with $b : B$. Define θ' such that for all $Z \in Dom(\theta)$, $\theta'(Z) = \theta(Z)$ and $Y\theta = b$ for a new variable $Y \notin Dom(\theta)$ of type B. Then θ' is a $\lambda\sigma$-unifier of $\langle\{X \mapsto \lambda_{A_1} \dots \lambda_{A_k}.Y\}, P\rangle$. Consequently θ is a $\lambda\sigma$-unifier of $\exists(Y : A_1.\cdots.A_k.\Gamma \vdash B), \langle\{X \mapsto \lambda_{A_1} \dots \lambda_{A_k}.Y\}, P\rangle$.

2. **Imit** and **Proj**: Let γ be a matcher of $P \cup \{X[a_1.\cdots.a_p.\uparrow^n] \ll^?_{\lambda\sigma} \underline{\mathtt{m}}\, b_1 \dots b_q\}$, where X has atomic type and $m > n$. Let $X \mapsto \underline{\mathtt{k}}\ c_1 \dots c_r \in \gamma$. Then, we have $P\{X \mapsto \underline{\mathtt{k}}\ c_1 \dots c_r\} \cup \{(\underline{\mathtt{k}}\ c_1 \dots c_r)[a_1.\cdots.a_p.\uparrow^n] \ll^?_{\lambda\sigma} \underline{\mathtt{m}}\, b_1 \dots b_q\} \to^*_{\lambda\sigma}$

$P\{X \mapsto \underline{\mathtt{k}}\, c_1 \dots c_r\} \cup \{\underline{\mathtt{k}}[a_1.\cdots.a_p.\uparrow^n] c_1[a_1.\cdots.a_p.\uparrow^n] \dots c_r[a_1.\cdots.a_p.\uparrow^n]$ $\ll^?_{\lambda\sigma} \underline{\mathtt{m}}\, b_1 \dots b_q\}$.
Now we have two options: $k \leq p$ or $k > p$. In the first case, the previous problem reduces to $P\{X \mapsto \underline{\mathtt{k}}\, c_1 \dots c_r\} \cup \{a_k\, c_1[a_1.\cdots.a_p.\uparrow^n] \dots c_r[a_1.\cdots.a_p.\uparrow^n]$ $\ll^?_{\lambda\sigma} \underline{\mathtt{m}}\, b_1 \dots b_q\}$ and γ is certainly a unifier of it. If $k > p$ then the problem reduces to $P\{X \mapsto \underline{\mathtt{k}}\, c_1 \dots c_r\} \cup \{\underline{\mathtt{k-p+n}}\, c_1[a_1.\cdots.a_p.\uparrow^n] \dots c_r[a_1.\cdots.a_p.\uparrow^n]$ $\ll^?_{\lambda\sigma} \underline{\mathtt{m}}\, b_1 \dots b_q\}$ and it has a solution if and only if $k - p + n = m$ and thus $k = m - n + p$ at the condition that $k > p \Leftrightarrow m - n + p > p \Leftrightarrow m > n$, which gives the condition asserted in the rule **Imit**. $\square$

4 Conclusions and Future Work

We presented a second-order matching algorithm that decides an important subset of $\lambda\sigma$-terms. This subset is important because it contains all the second-order $\lambda\sigma$-terms that can occur in a second-order matching problem which is originated from a matching problem in the simply typed λ-calculus. The algorithm uses an adequate notation for dealing with matching problems since it keeps graftings and matching equations as different entities. This separation is important to avoid the possible introduction of flexible terms that need to be instantiated in the right-hand side of a matching equation.

The study of the possible adaptation of this method to other calculi of explicit substitutions, such as the λs_e-calculus (for which HOU was already adapted [ARK01]) and the suspension calculus, can be helpful to identify advantages and disadvantages of these calculi in practical applications [AMK05]. Moreover, this work can be extended for matching via explicit substitutions using a richer type theory, such as dependent types [Ree03,Muñ01].

There exist different definitions of matching in the literature such as "filtering" and "semi-unification", and in certain cases, matching cannot be seen as a sub-case of higher-order unification[Bur89,Dow01]. As future work, we intend to study how these definitions are related in a higher-order framework and, how they interfere with explicit substitutions environments. In addition, another interesting problem concerns to the existence of a second-order matching algorithm that decides the whole $\lambda\sigma$-calculus and not a sub-class of it, as well as possible extensions of the current algorithm to matching problems of higher orders.
Acknowledgments. We would like to thank Claude Kirchner for the comments that motivated this work and the referees for their suggestions and constructive criticisms.

References

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *J. of Func. Programming*, 1(4):375–416, 1991.

[ARK01] M. Ayala-Rincón and F. Kamareddine. Unification via the λs_e-Style of Explicit Substitution. *The Logical J. of the IGPL*, 9(4):489–523, 2001.

[AMK05] M. Ayala-Rincón, F.L.C. de Moura and F. Kamareddine. Comparing and Implementing Calculi of Explicit Substitutions with Eta-Reduction. R. de Queiroz, B. Poizat and S. Artemov Eds. To appear in Special Issue of *Annals of Pure and Applied Logic* - WoLLIC 2002 selected papers, 2005.

[Bur89] H.J. Burckert. Matching - A Special Case of Unification? *Journal of Symbolic Computation*, 8:523–536, 1989.

[dB72] N.G. de Bruijn. Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indag. Mat.*, 34(5):381–392, 1972.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157:183–235, 2000.

[dlTC87] T. B. 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*, pages 95–99. Morgan Kaufmann, 1987.

[dlTC88] T. B. de la Tour and R. Caferra. A formal approach to some usually informal techniques used in mathematical reasoning. In P. Gianni, editor, *Proc. of the Int. Symposium on Symbolic and Algebraic Computation*, LNCS 358, pages 402–406. Springer Verlag, 1988.

[Dow94] G. Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.

[Dow01] G. Dowek. Higher-Order Unification and Matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 16, pages 1009–1062. MIT press and Elsevier, 2001.

[Gol81] W. Goldfarb. The Undecidability of the Second-Order Unification Problem. *TCS*, 13(2):225–230, 1981.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.

[Hue76] G. Huet. *Résolution d'équations dans les langages d'ordre 1,2,...,ω*. PhD thesis, University Paris-7, 1976.

[Loa03] R. Loader. Higher order β matching is undecidable. *Logic Journal of the IGPL*, 11(1):51–68, 2003.

[Muñ01] C. Muñoz. Proof-term synthesis on dependent-type systems via explicit substitutions. *Theoretical Computer Science*, 266:407–440, 2001.

[Nip93] T. Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.

[Pad00] V. Padovani. Decidability of fourth-order matching. *Mathematical Structures in Computer Science*, 10(3):361–372, 2000.

[Ree03] J. Reed. Extending higher-order unification to support proof irrelevance. In *TPHOLs*, pages 238–252. Springer Verlag, 2003.

[Rob65] J. A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Vis04] E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 2004. Accepted for publication.