Unification via λs_e-Style of Explicit Substitution *

Mauricio Ayala-Rincón†
Departamento de Matemática, Universidade de Brasília
70910-900 Brasília D.F., Brasil
ayala@mat.unb.br

Fairouz Kamareddine
Department of Computing and Electrical Engineering, Heriot-Watt University
Riccarton, Edinburgh EH14 4AS, Scotland
fairouz@cee.hw.ac.uk

ABSTRACT

A unification method based on the λs_e-style of explicit substitution is proposed. This method together with appropriate translations, provide a Higher Order Unification (HOU) procedure for the pure λ-calculus. Our method is influenced by the treatment introduced by Dowek, Hardin and Kirchner using the $\lambda\sigma$-style of explicit substitution. Correctness and completeness properties of the proposed λs_e-unification method are shown and its advantages, inherited from the qualities of the λs_e-calculus, are pointed out. Our method needs only one sort of objects: terms. And in contrast to the HOU approach based on the $\lambda\sigma$-calculus, it avoids the use of substitution objects. This makes our method closer to the syntax of the λ-calculus. Furthermore, detection of redices depends on the search for solutions of simple arithmetic constraints which makes our method more operational than the one based on the $\lambda\sigma$-style of explicit substitution.

Keywords

Higher order unification, lambda-calculus, explicit substitution.

1. INTRODUCTION

After Robinson's successful introduction of his well-known first order *Resolution Principle* based on substitution, unification and resolution [26], much work has been done in order to formalize these basic notions in other settings. Such extensions are essential for amongst other things, automated deduction in higher order logics. The first person to successfully formulate a unification method for the case of higher order logics, specifically for the typed λ-calculus, was Huet [13]. Since then several Higher Order Unification (HOU) approaches have been developed and used in practical languages and theorem provers such as λprolog and Isabelle ([20; 23]). In most of these approaches, the notion of substitution plays an important role. The importance of the notion of substitution led to an explosion of work on making substitutions explicit in recent years. Also, a number of work has been devoted to establish the usefulness of explicit substitution to automated deduction, theorem proving and proof synthesis [18; 19], to programming languages [6] and to HOU [9]. The latter paper shows that in the HOU framework, if substitution was made explicit, many benefits can be obtained in computation. In particular, [9] presented a HOU method based on the $\lambda\sigma$-style of explicit substitution which was proved useful for deduction in the typed λ-calculus and subsequently generalized for treating higher order equational unification problems [17] and restricted for the case of higher order patterns [10]. Here we develop a unification method based on the λs_e-style of explicit substitution which jointly with adequate *pre-cooking* and *back* translations between the languages of the λ-calculus and the λs_e-calculus, as presented in [3] give a HOU procedure, which takes advantage of the qualities of the λs_e calculus. In particular, our approach avoids the use of two different sorts of objects as in the $\lambda\sigma$-calculus. Moreover, decidability of the application of our unification rules (i.e., detection of redices) depends on the search for natural solutions of simple arithmetic constraints. This makes λs_e-HOU more operational than the $\lambda\sigma$-HOU.

1.1 Higher order unification

Higher order objects arise naturally in many fields of computer science. For example, in the context of implementation of functional languages it is necessary to develop mechanisms for the treatment of higher order functions. For instance, the following rewriting system specifies the well-known MAP function, which applies a function to all the elements of a list: $\text{MAP}(f, \text{NIL}) \to \text{NIL}$; $\text{MAP}(f, \text{CONS}(x, l)) \to \text{CONS}(f(x), \text{MAP}(f, l))$, where NIL and CONS are the usual LISP *empty list* and *constructor list function*. Observe that f appears both as a variable and as a functional symbol. From the point of view of first oder rewriting it is not possible to manipulate this kind of objects. In fact, for simple rewriting based deduction processes, such as one-step reduction or critical pair deduction, first order matching and unification do not apply. The solution of these problems, at least in the rewriting context, is the λ-calculus. Rewriting

*Partially supported by EPSRC grant numbers GR/L36963 and GR/L15685.

†Partly supported by FEMAT and CAPES (BEX0384/99-2) Brazilian Foundations. Work carried out during one year visit of this author at the ULTRA Group (*Useful Logics, Type Theory, Rewriting Systems and Their Applications*), Heriot-Watt University.

could be performed modulo the rules of the λ-calculus or combining specifications with the rules of the λ-calculus.

The function MAP is a typical example of a second-order function, but functions of third-order or above have practical interest too. In [22] useful third- until sixth-order functions were presented in the context of combinator parsing.

A simple example of a HOU problem is the search for solutions for the equality $(F(f))(a) = f(F(a))$. A solution is the function identity $\{F/\lambda_x.x\}$, but $\{F(x)/\lambda_x.f^n(x) \mid n \in \mathbb{N}\}$ are solutions too.

HOU is essential in higher order automated reasoning, where it has formed the basis for generalizations of the Resolution Principle in second-order logic.

Huet's work [13] was relevant because he realized that to generalize Robinson's first order Resolution Principle [26] to higher order theories, it is useful to verify the existence of unifiers without computing them explicitly. Huet's algorithm is a semi-decision one that may never stop when the input unification problem has no unifiers, but when the problem has a solution it always presents an explicit unifier. Unification for second-order logic was proved undecidable in general by Goldfarb [12]. Goldfarb's proof is based on a reduction from Hilbert's Tenth Problem. This result shows that there are arbitrary higher order theories where unification is undecidable, but there exist particular higher order languages of practical interest that have a decidable unification problem. In particular, for the second-order case, unification is decidable, when the language is restricted to monadic functions [11]. Another problem of HOU is that the notion of most general unifier does not apply and that a more complex notion than that of complete set of unifiers is necessary. Huet has shown that equations of the form $(\lambda_x.F\ \ a) =^? (\lambda_x.G\ \ b)$ (called *flex-flex*) of third-order may not have minimal complete sets of unifiers and that there may exist an infinite chain of unifiers, one more general than the other, without having a most general one (for references see section 4.1 in [24]).

For a very simple presentation of HOU see [27] and for a detailed introduction in the context of declarative programming see [24].

1.2 Contribution of this work

The $\lambda\sigma$-calculus [1] introduces two different sets of entities, one for terms and one for substitutions. The λs_e-style [15] calculus insists on remaining closer to the λ-calculus and uses a philosophy started by de Bruijn in his system *AUTOMATH* [21] and elaborated extensively through the new item notation [14]. The philosophy states that terms of the λ-calculus are either application terms such as a function applied to an argument; abstraction terms such as a function; or substitution terms or updating terms. Hence, substitution and updating are made explicit in item notation. The advantages of this philosophy are listed in [14] and include remaining as close as possible to the familiar λ-calculus. Therefore, we propose to study HOU in the λs_e-style of explicit substitution, which makes our approach closer to the syntax of the λ-calculus than that of the $\lambda\sigma$-approach in that we avoid the use of two different sorts of objects. We establish the following three properties of λs_e-unification:

1. Correctness: If P and P' are unification problems such that P reduces to P' then every unifier of P' is a unifier of P.
2. Completeness: If P and P' are unification problems such that P reduces to P' then every unifier of P is a unifier of P'.
3. The search for unification redices and the detection of *flex-flex* (i.e. implicitly solvable) equations is simpler in our approach than in the $\lambda\sigma$-approach.

After introducing the (typed) $\lambda\sigma$- and λs_e-calculi (section 2), and the unification approach in the $\lambda\sigma$-calculus (section 3), we present our λs_e-style based unification method in section 4 and some arithmetic properties of the λs_e-unification rules in section 5. We then conclude illustrating how to apply λs_e-unification for solving HOU problems in the pure λ-calculus and discussing future work in section 6. Omitted proofs and references can be found in [2].

2. BACKGROUND

We assume familiarity with the λ-calculus as presented in [5] and with the notion of term algebra $\mathcal{T}(\mathcal{F},\mathcal{X})$ built on a (countable) set of variables $\mathcal{X}$ and a set of operators $\mathcal{F}$. Variables in $\mathcal{X}$ are denoted by upper case last letters of the Roman alphabet $X, Y, ...$ and for a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $var(t)$ denotes the set of variables occurring in t.

Additionally, we assume familiarity with the basic notions of rewriting theory such as *(local) confluence* or (weakly) Church Rosser property (for short, **(W)CR**), normal forms and *strong* and *weak normalization* (for short, **SN** and **WN**, respectively) as presented in [4]. For a *reduction relation* R over a set A, $(A, \to_R)$, we denote with $\to_R^*$ the *reflexive and transitive closure* of $\to_R$. The subscript R is usually omitted. When $a \to^* b$ we say that there exists a *derivation* from a to b. Syntactical identity is denoted by $a = b$.

A **valuation** is a mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F},\mathcal{X})$. The homeomorphic extension of a valuation, θ, from its domain $\mathcal{X}$ to the domain $\mathcal{T}(\mathcal{F},\mathcal{X})$ is called the **grafting** of θ. As usual, valuations and their corresponding grafting valuations are denoted by the same Greek letter. Application of a valuation θ or its corresponding grafting to a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ will be written in postfix notation $t\theta$. The **domain** of a grafting θ, denoted $dom(\theta)$ is defined by $dom(\theta) = \{X \mid X\theta \neq X, X \in \mathcal{X}\}$. Its **range**, denoted $ran(\theta)$, is defined by $ran(\theta) = \cup_{X \in dom(\theta)} var(X\theta)$. The set $var(\theta) = dom(\theta) \cup ran(\theta)$ is the set of variables involved in θ. A valuation and its corresponding grafting θ are explicitly denoted by $\theta = \{X/X\theta \mid X \in dom(\theta)\}$.

The notion of grafting, usually called first order substitution, corresponds to simple substitution without renaming.

2.1 The λ-calculus with names and explicit substitution

Let $\mathcal{V}$ be a (countable) set of variables (different from the ones in $\mathcal{X}$) denoted by lowercase last letters of the Roman alphabet $x, y, ...$

Terms $\Lambda(\mathcal{V})$, of the **λ-calculus with names** are inductively defined by $a ::= x \mid (a\ \ a) \mid \lambda_x.a$. Terms of the forms $\lambda_x.a$ and $(a\ \ b)$ are called *abstractions* and *applications*, respectively. The notions of valuation and grafting from $\mathcal{V}$ and $\Lambda(\mathcal{V})$ to $\Lambda(\mathcal{V})$ are adapted in the obvious way. First order substitution or grafting leads to problems in the λ-calculus. For example, applying the (first order) substitution $\{u/x\}$ to $\lambda_x.(u\ \ x)$ results in $\lambda_x.(x\ \ x)$ which is wrong. Therefore, the λ-calculus with names makes extensive use of *variable renaming* via α-conversion. E.g., renaming x (say as y) in $(\lambda_x.(u\ \ x))\{u/x\}$ results in the correct term $\lambda_y.(x\ \ y)$.

We denote by $\alpha^V(a)$ the α-conversion of a resulting by renaming the variables in $V \subseteq \mathcal{V}$ occurring at $a \in \Lambda(\mathcal{V})$ with *fresh* variables.

DEFINITION 2.1. *Let $\theta = \{x_1/a_1, \ldots, x_n/a_n\}$ be a valuation from $\mathcal{V}$ to $\Lambda(\mathcal{V})$. θ^{ext}, the* **substitution** *that extends θ to $\Lambda(\mathcal{V})$, is defined by structural induction as follows:*

1. $\theta^{ext}(x) = x\theta$;
2. $\theta^{ext}((a\ \ b)) = (\theta^{ext}(a)\ \ \theta^{ext}(b))$;
3. $\theta^{ext}(\lambda_x.a) = \lambda_z.\theta^{ext}((\alpha^{var(\theta)\cup\{x\}}(a))\{x/z\})$, *where z is a fresh variable; i.e., $z \notin var(\theta)$ and z does not occur in $\lambda_x.a$.*

When no confusion arises, both a valuation θ and its corresponding substitution will be denoted by θ. Only in this section we will use prefixed and postfixed notation to remark the difference between substitutions and valuations.

Since free variables are selected randomly, the result of applying a substitution can be conceived as a class of equivalence terms rather than as a unique term.

Now we can define **β-reduction** respectively **η-reduction** as the rewriting relation of the rewrite rule: $\boldsymbol{(\lambda_x.a\ \ b) \to \{x/b\}^{ext}(a)}$ respectively $\boldsymbol{\lambda_x.(a\ \ x) \to a,}$ if $\boldsymbol{x \notin \mathcal{F}var(a)}$, where $\mathcal{F}var(a)$ denotes the set of free variables in a.

The notion of unification in $\Lambda(\mathcal{V})$ differs from the first order one, because bound variables in $\Lambda(\mathcal{V})$ are not affected by unification substitutions. Unification variables in the λ-calculus are free variables. Thus the free variables occurring in terms of a unification problem can be partitioned into true **unification variables** and **constants**, that cannot be bound by the unifiers. Observe that constants, as free variables, cannot be changed by the β-reduction process. However, from the point of view of unification, both constants and bound variables can be considered of the same syntactical category.

2.2 The λ-calculus in de Bruijn notation

To differentiate between unification and constant variables, one can consider unification variables as **meta-variables** in a set $\mathcal{X}$. Thus, λ-calculus is defined as the term algebra, $\Lambda(\mathcal{V}, \mathcal{X})$, over the set of operators $\{\lambda_x._ \mid x \in \mathcal{V}\} \cup \{(\_\ \_)\} \cup \mathcal{V}$ and the set of variables $\mathcal{X}$. In this setting, the previous notion of substitution can be adapted for meta-variables preserving the semantics of both β- and η-reduction. But the most appropriate setting to treat unification meta-variables is the well-known λ-calculus with de Bruijn indices [21], were natural indices are used to denote both bounded variables and constants. Bounded variables are related to their corresponding abstractors by their relative *height*, which is the number of abstractors between them.

For instance, the λ-term $\lambda_x.(\lambda_z.(x\ \ z)\ \ (x\ \ z))$ is translated into $\lambda.(\lambda.(\mathtt{2}\ \ \mathtt{1})\ \ (\mathtt{1}\ \ \mathtt{4}))$. Indices for free variable are appropriately selected to avoid relating them with abstractors.

The set $\Lambda_{dB}(\mathcal{X})$ of **λ-terms in de Bruijn notation** is defined inductively as $a ::= \mathtt{n} \mid X \mid (a\ \ a) \mid \lambda.a$ where $X \in \mathcal{X}$ and $\mathtt{n} \in \mathbb{N} \setminus \{0\}$.

DEFINITION 2.2. *Let $a \in \Lambda_{dB}(\mathcal{X})$, $i \in \mathbb{N}$. The* **i-lift** *of a, denoted by a^{+i}, is defined as:*

a) $X^{+i} = X$ *, for* $X \in \mathcal{X}$ *b)* $(a_1\ \ a_2)^{+i} = (a_1^{+i}\ \ a_2^{+i})$

c) $(\lambda.a_1)^{+i} = \lambda.a_1^{+(i+1)}$ *d)* $\mathtt{n}^{+i} = \begin{cases} \mathtt{n+1}, & \text{if } n > i \\ \mathtt{n}, & \text{if } n \le i \end{cases}$ *for* $n \in \mathbb{N}$.

The **lift** *of a term a is its 0-lift, and is denoted by a^+.*

DEFINITION 2.3. *The* **application of the substitution** *with b of $n \in \mathbb{N}\setminus\{0\}$ on a term a in $\Lambda_{dB}(\mathcal{X})$, denoted $\{n/b\}a$, is defined inductively by:*

1. $\{n/b\}X = X$, *for* $X \in \mathcal{X}$;
2. $\{n/b\}(a_1\ \ a_2) = (\{n/b\}a_1\ \ \{n/b\}a_2)$;
3. $\{n/b\}\lambda.a_1 = \lambda.\{n+1/b^+\}a_1$;
4. $\{n/b\}\mathtt{m} = \begin{cases} \mathtt{m-1}, & \text{if } m > n \\ b, & \text{if } m = n \\ \mathtt{m}, & \text{if } m < n \end{cases}$ *if* $m \in \mathbb{N}$.

DEFINITION 2.4. *Let $\theta = \{X_1/a_1, \ldots, X_n/a_n\}$ be a valuation from the set of meta-variables $\mathcal{X}$ to $\Lambda_{dB}(\mathcal{X})$. The corresponding* **substitution**, *also denoted by θ, is defined inductively by:*

a) $\theta(\mathtt{m}) = \mathtt{m}$ *for* $m \in \mathbb{N}$ *b)* $\theta(X) = X\theta$, *for* $X \in \mathcal{X}$

c) $\theta(a_1\ \ a_2) = (\theta(a_1)\ \ \theta(a_2))$ *d)* $\theta\lambda.a_1 = \lambda.\theta^+(a_1)$

where θ^+ denotes both the valuation $\{X_1/a_1^+, \ldots, x_n/a_n^+\}$ and its associated substitution.

Consider the η-reduction rule in $\Lambda(\mathcal{X})$: $\lambda_x.(a\ \ x) \to a$, if $x \notin \mathcal{F}var(a)$. In $\Lambda_{dB}(\mathcal{X})$, the left side of this rule is written as $\lambda.(a'\ \ 1)$, where a' stands for the corresponding translation of a into the language of $\Lambda_{dB}(\mathcal{X})$. "$x \notin \mathcal{F}var(a)$" means, in $\Lambda_{dB}(\mathcal{X})$, that there are neither occurrences in a' of the index 1 at height zero nor of the index 2 at height one etc. This means, in general, that there exists a term b such that $b^+ = a$. Thus the β-reduction is defined as $(\lambda.a\ \ b) \to \{1/b\}a$ and the η-reduction as $\lambda.(a\ \ 1) \to b$ if $\exists b\ b^+ = a$.

Table 1: $\lambda\sigma$ Rewriting System of the $\lambda\sigma$-calculus

(Beta)	$(\lambda.a\ \ b)$	$\longrightarrow$	$a[b \cdot id]$
(Id)	$a[id]$	$\longrightarrow$	a
(VarCons)	$\mathtt{1}[a \cdot s]$	$\longrightarrow$	a
(App)	$(a\ \ b)[s]$	$\longrightarrow$	$(a[s])\ (b[s])$
(Abs)	$(\lambda.a)[s]$	$\longrightarrow$	$\lambda.a[\mathtt{1} \cdot (s \circ \uparrow)]$
(Clos)	$(a[s])[t]$	$\longrightarrow$	$a[s \circ t]$
(IdL)	$id \circ s$	$\longrightarrow$	s
(IdR)	$s \circ id$	$\longrightarrow$	s
(ShiftCons)	$\uparrow \circ (a \cdot s)$	$\longrightarrow$	s
(Map)	$(a \cdot s) \circ t$	$\longrightarrow$	$a[t] \cdot (s \circ t)$
(Ass)	$(s \circ t) \circ u$	$\longrightarrow$	$s \circ (t \circ u)$
(VarShift)	$\mathtt{1} \cdot \uparrow$	$\longrightarrow$	id
(SCons)	$\mathtt{1}[s] \cdot (\uparrow \circ s)$	$\longrightarrow$	s
(Eta)	$\lambda.(a\ \ \mathtt{1})$	$\longrightarrow$	b if $a =_\sigma b[\uparrow]$

2.3 The $\lambda\sigma$-calculus

DEFINITION 2.5. *The $\lambda\sigma$-calculus is defined as the calculus of the rewriting system $\lambda\sigma$ of Table 1 where* TERMS $a ::= \mathtt{1} \mid X \mid (a\ \ a) \mid \lambda a \mid a[s]$ *and* SUBS $s ::= id \mid \uparrow \mid a.s \mid s \circ s$.

The equational theory associated to the rewriting system $\lambda\sigma$ defines a congruence that we denote by $=_{\lambda\sigma}$. The corresponding congruence obtained by dropping the *Beta* and *Eta* rules is denoted with $=_\sigma$.

The rewriting system $\lambda\sigma$ is locally confluent [1], confluent on substitution-closed terms (i.e., terms without substitution variables) [25] and not confluent on open terms (i.e., terms with term and substitution variables) [8].

PROPOSITION 2.6. *([25]) The $\lambda\sigma$-normal form of any $\lambda\sigma$-term is of one of the following forms: a) λa; b) $(a\ b_1 \ldots b_n)$, where a is either $\mathtt{1}$, $\mathtt{1}[\uparrow^n]$, X or $X[s]$ for s a substitution term different from id in normal form; or c) $a_1 \ldots a_p.\ \uparrow^n$, where $a_1, \ldots, a_p$ are normal terms and $a_p \neq \mathtt{n}$.*

In $\Lambda(\mathcal{X})$ and $\Lambda_{dB}(\mathcal{X})$, the rule $X\{y/t\} = X$, where y is an element of $\mathcal{V}$ or a de Bruijn index, respectively, is necessary because there is no way to suspend the substitution $\{y/t\}$ until X is instantiated. In the $\lambda\sigma$-calculi the application of this substitution can be delayed, since the term $X[s]$ does not reduce to X. Observe that the condition $a =_\sigma b[\uparrow]$ of the *Eta* rule is stronger than the condition $a = b^+$ as $X = X^+$, but there exists no term b such that $X =_\sigma b[\uparrow]$. The fact that the application of a substitution to a meta-variable can be suspended until the meta-variable is instantiated will be used to code the substitution of variables in $\mathcal{X}$ by $\mathcal{X}$-grafting and explicit lifting. Consequently a notion of $\mathcal{X}$-substitution in $\lambda\sigma$-calculi is unnecessary.

2.4 The λs_e-calculus

The λs_e-calculus avoids introducing two different sets of entities and insists on remaining close to the syntax of the λ-calculus. Next to λ and application, the λs_e-calculus introduces substitution (σ) and updating (φ) operators. In the λs_e-calculus, we let a, b, c, etc., range over the set of terms. A term containing neither substitution nor updating operators is called a *pure term*.

DEFINITION 2.7 (λs_e-CALCULUS). *The terms of the λs_e-calculus are given by: $a ::= \mathcal{X} \mid \mathbb{N} \mid (a\ \ a) \mid \lambda.a \mid a\,\sigma^j a \mid \varphi^i_k a$ where $j, i \geq 1$, $k \geq 0$. The set of rules λs_e is given in Table 2. The* **λs_e-calculus** *is the rewriting system generated by the set of rules λs_e. The* **calculus of substitutions associated with the λs_e-calculus** *is the rewriting system generated by the set of rules $s_e = \lambda s_e - \{\sigma\text{-generation}, Eta\}$ and we call it the* **s_e-calculus**.

The equational theory associated with λs_e defines a congruence denoted by $=_{\lambda s_e}$. The congruence obtained by dropping the *σ-generation* and *Eta* rules is denoted by $=_{s_e}$. When we restrict the reduction to these rules, we will use expressions such as s_e-reduction, s_e-normal form, etc, with the obvious meaning.

Intuitively, the substitution operator initiates (*σ-generation*) one-step of β-reduction, from $(\lambda.a\ \ b)$, propagating the associated substitution innermost (*σ-λ-* and *σ-app-transition*). Once this propagation is finished, when necessary, the updating operator is introduced to make the appropriate lift over b (*σ-destruction*). Otherwise the free de Bruijn indices are decremented by one.

Correspondence between the *Eta* rules of the λs_e-calculus and the $\lambda\sigma$-calculus was proved in [2].

Similarly to the $\lambda\sigma$-calculus we can describe operators of the λs_e-calculus over the signature of a first order sorted term algebra $\mathcal{T}_{\lambda s_e}(\mathcal{X})$ built on $\mathcal{X}$, the set of variables of sort TERM and its subsort NAT$\subset$TERM. The set of variables of sort TERM in a term $a \in \mathcal{T}_{\lambda s_e}(\mathcal{X})$ is denoted by $\mathcal{T}var(a)$.

THEOREM 2.8 ([15]).

a) The s_e-calculus is weakly normalizing and confluent.

b) The λs_e-calculus simulates β-reduction.

c) The λs_e-calculus is confluent on open terms.

As a corollary of the characterization of the s_e-normal forms in [15] (Theorem 8) we obtain a characterization of λs_e-normal forms.

COROLLARY 2.9 (λs_e-NORMAL FORMS). *Let a be a λs_e-term. a is in λs_e-normal form iff:*

1. *$a \in \mathcal{X} \cup \mathbb{N}$;*
2. *$a = (b\ \ c)$, where b, c are λs_e-normal forms and b is not an abstraction of the form $\lambda.d$;*

Table 2: Rewriting System of the λs_e-calculus with η-rule

(σ-generation)	$(\lambda.a\ \ b)$	$\longrightarrow$	$a\,\sigma^1\,b$
(σ-λ-transition)	$(\lambda.a)\,\sigma^i b$	$\longrightarrow$	$\lambda.(a\,\sigma^{i+1}\,b)$
(σ-app-transition)	$(a_1\ \ a_2)\,\sigma^i b$	$\longrightarrow$	$((a_1\,\sigma^i b)\ \ (a_2\,\sigma^i b))$
(σ-destruction)	$\mathtt{n}\,\sigma^i b$	$\longrightarrow$	$\begin{cases} \mathtt{n-1} & \text{if } n > i \\ \varphi_0^i\,b & \text{if } n = i \\ \mathtt{n} & \text{if } n < i \end{cases}$
(φ-λ-transition)	$\varphi_k^i(\lambda.a)$	$\longrightarrow$	$\lambda.(\varphi_{k+1}^i\,a)$
(φ-app-transition)	$\varphi_k^i(a_1\ \ a_2)$	$\longrightarrow$	$((\varphi_k^i\,a_1)\ \ (\varphi_k^i\,a_2))$
(φ-destruction)	$\varphi_k^i\,\mathtt{n}$	$\longrightarrow$	$\begin{cases} \mathtt{n+i-1} & \text{if } n > k \\ \mathtt{n} & \text{if } n \le k \end{cases}$
(Eta)	$\lambda.(a\ \ \mathtt{1})$	$\longrightarrow$	$b \quad \text{if} \quad a =_{s_e} \varphi_0^2 b$
(σ-σ-transition)	$(a\,\sigma^i b)\,\sigma^j\,c$	$\longrightarrow$	$(a\,\sigma^{j+1}\,c)\ \sigma^i\,(b\,\sigma^{j-i+1}\,c)$ if $\quad i \le j$
(σ-φ-transition 1)	$(\varphi_k^i\,a)\,\sigma^j\,b$	$\longrightarrow$	$\varphi_k^{i-1}\,a \quad \text{if} \quad k < j < k+i$
(σ-φ-transition 2)	$(\varphi_k^i\,a)\,\sigma^j\,b$	$\longrightarrow$	$\varphi_k^i(a\,\sigma^{j-i+1}\,b) \quad \text{if} \quad k+i \le j$
(φ-σ-transition)	$\varphi_k^i(a\,\sigma^j\,b)$	$\longrightarrow$	$(\varphi_{k+1}^i\,a)\,\sigma^j\,(\varphi_{k+1-j}^i\,b) \quad \text{if} \quad j \le k+1$
(φ-φ-transition 1)	$\varphi_k^i\,(\varphi_l^j\,a)$	$\longrightarrow$	$\varphi_l^j\,(\varphi_{k+1-j}^i\,a) \quad \text{if} \quad l+j \le k$
(φ-φ-transition 2)	$\varphi_k^i\,(\varphi_l^j\,a)$	$\longrightarrow$	$\varphi_l^{j+i-1}\,a \quad \text{if} \quad l \le k < l+j$

3. *$a = \lambda.b$, where b is a λs_e-normal form excluding applications of the form $(c\ \ \mathtt{1})$ such that there exists d with $\varphi_0^2 d =_{s_e} c$;*

4. *$a = b\sigma^j c$, where c is a λs_e-normal form and b is an λs_e-normal form of one of the following forms:*
   a) X, b) $d\sigma^i e$, with $j < i$ or c) $\varphi_k^i d$, with $j \le k$;

5. *$a = \varphi_k^i b$, where b is a λs_e-normal form of one of the following forms:*
   a) X, b) $c\sigma^j d$, with $j > k+1$ or c) $\varphi_l^j c$, with $k < l$.

2.5 Typed λ-calculi

For the sake of clarity we include only the essential notation of the typed $\lambda\sigma$- and λs_e-calculi. Typing rules for the two calculi and additional properties can be found in [2].

We recall that an environment, Γ, in de Bruijn setting is simply a list of types and, in the case of the $\lambda\sigma$-calculus, substitutions receive environments as types. For all the systems we will consider, we take: TYPES $A ::= A \mid A \to B$ and ENVIRS $\Gamma ::= nil \mid A.\Gamma$. The rewrite rules of the typed $\lambda\sigma$- and λs_e-calculi are those of Tables 1 and 2 except that rules involving abstractions are now typed. Thus, for the typed $\lambda\sigma$-calculus we have the typed rules:

(Beta)	$(\lambda_A.a\ \ b)$	$\longrightarrow a\,[b \cdot id]$
(Abs)	$(\lambda_A.a)[s]$	$\longrightarrow \lambda_A.a\,[\mathtt{1} \cdot (s \circ \uparrow)]$
(Eta)	$\lambda_A.(a\ \ \mathtt{1})$	$\longrightarrow b$ if $a =_\sigma b[\uparrow]$

and for the typed λs_e-calculus:

(σ-generation)	$(\lambda_A.a\ \ b)$	$\longrightarrow a\,\sigma^1\,b$
(σ-λ-transition)	$(\lambda_A.a)\,\sigma^i b$	$\longrightarrow \lambda_A.(a\,\sigma^{i+1}\,b)$
(φ-λ-transition)	$\varphi_k^i(\lambda_A.a)$	$\longrightarrow \lambda_A.(\varphi_{k+1}^i\,a)$
(Eta)	$\lambda_A.(a\ \ \mathtt{1})$	$\longrightarrow b$ if $a =_s \varphi_0^2 b$

Characterization of η-long normal forms in the typed $\lambda\sigma$- and λs_e-calculi is necessary to simplify the set of rules of the unification algorithms. Essentially, the use of η-long normal forms guarantees that meta-variables of functional type $A \to B$ are instantiated with typed terms of the form $\lambda_A.a$.

DEFINITION 2.10 (η-LONG NORMAL FORM IN $\lambda\sigma$). *Let a be a $\lambda\sigma$-term of type $A_1 \to \ldots \to A_n \to B$ in the environment Γ and in $\lambda\sigma$-normal form. The* **η-long normal form** *of a, written a', is defined by:*

$$a' = \begin{cases} \lambda_C.b' & \text{if } a = \lambda_C.b \\ \lambda_{A_1}\ldots\lambda_{A_n}\,(\mathtt{k+n}\,c_1\ldots c_p\,\mathtt{n'}\ldots\mathtt{1'}) & \text{if } a = (\mathtt{k}\ b_1\ldots b_p) \\ \lambda_{A_1}\ldots\lambda_{A_n}\,(X[s']\,c_1\ldots c_p\,\mathtt{n'}\ldots\mathtt{1'}) & \text{if } a = (X[s]\,b_1\ldots b_p) \end{cases}$$

where in the second clause, c_i is the η-long normal form of the normal form of $b_i[\uparrow^n]$; and in the third clause, c_i is the η-long normal form of $b_i[\uparrow^n]$ and if $s = d_1\ldots d_q.\uparrow^k$ then $s' = e_1\ldots e_q.\uparrow^{k+n}$ where e_i is the η-long normal form of $d_i[\uparrow^n]$.

DEFINITION 2.11 (η-LONG NORMAL FORM IN λs_e). *Let a be a λs_e-term of type $A_1 \to \ldots \to A_n \to B$ in the environment Γ and in λs_e-normal form. The* **η-long normal form** *of a, written a', is defined by:*

$$a' = \begin{cases} \lambda_C.b' & \text{if } a = \lambda_C.b \\ \lambda_{A_1}\ldots\lambda_{A_n}\,(c_1\ldots c_p\,\mathtt{n'}\ldots\mathtt{1'}) & \text{if } a = (b_1\ldots b_p) \\ \lambda_{A_1}\ldots\lambda_{A_n}\,(d'\sigma^{i+n}e'\,\mathtt{n'}\ldots\mathtt{1'}) & \text{if } a = b\sigma^i c \\ \lambda_{A_1}\ldots\lambda_{A_n}\,(\varphi_k^i c'\,\mathtt{n'}\ldots\mathtt{1'}) & \text{if } a = \varphi_k^i b \end{cases}$$

where in the second clause, c_i is the η-long normal form of the normal form of $\varphi_0^{n+1} b_i$; in the third clause, d', e' are the η-long normal forms of the normal forms of $\varphi_0^{n+1} b$ and $\varphi_0^{n+1} c$, respectively; and in the fourth clause, c' is the η-long normal form of the normal form of $\varphi_0^{n+1} b$.

The set of unification rules of the two unification methods are constructed by combining the different types of η-long normal forms enumerated in the previous two Definitions obtaining different types of equational problems. For the unification setting based on the λs_e-style an additional characterization of λs_e-normal terms whose main operators are either σ or φ will be useful in order to combine directly η-long normal forms of type 2 (See subsection 4.1) with the ones of type 3. and 4. This simplifies the comparison of both unification approaches.

DEFINITION 2.12 (LONG NORMAL FORM). *For both $\lambda\sigma$- and λs_e-terms,* **long normal forms** *are defined as the η-long normal form of the corresponding $\beta\eta$-normal forms.*

In both typed $\lambda\sigma$- and λs_e-calculi we have that two terms are $\beta\eta$-equivalent iff they have the same long normal form.

3. UNIFICATION IN THE $\lambda\sigma$-CALCULUS

In this section we present briefly notions and results on higher order unification in the $\lambda\sigma$-style of explicit substitution given in [9]. Equational problems are restricted to substitution-closed terms (for which $\lambda\sigma$ is confluent), because $\lambda\sigma$ is not confluent on substitution-open terms. Since the main goal is to provide a mechanism to solve unification problems in the λ-calculus this restriction is harmless.

Let $\mathcal{T}(\mathcal{F}, \mathcal{X})$ be a term algebra over a set of function symbols $\mathcal{F}$ and a countable set of variables $\mathcal{X}$ and let $\mathcal{A}$ be an $\mathcal{F}$-algebra. A **unification problem** over $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is a first order formula without universal quantifier or negation, whose atoms are of the form $\mathbb{F}$, $\mathbb{T}$ or $s =^?_{\mathcal{A}} t$. Unification problems are written as disjunctions of existentially quantified conjunctions of atomic equational unification problems: $D = \bigvee_{j\in J} \exists \vec{w_j} \bigwedge_{i\in I_j} s_i =^?_{\mathcal{A}} t_i$. When $|J| = 1$, the unification problem is called a **unification system**. Variables in the set $\vec{w}$ of a unification system $\exists \vec{w} \bigwedge_{i\in I} s_i =^?_{\mathcal{A}} t_i$ are bound and all other variables are free. $\mathbb{T}$ and $\mathbb{F}$ stand for the empty conjunction and disjunction, respectively. Of course, the empty disjunction corresponds to an unsatisfiable problem.

A **unifier** of a unification system $\exists \vec{w} \bigwedge_{i\in I} s_i =^?_{\mathcal{A}} t_i$ is a grafting σ such that $\mathcal{A} \models \exists \vec{w} \bigwedge_{i\in I} s_i\sigma_{|\vec{w}} = t_i\sigma_{|\vec{w}}$ where $\sigma_{|\vec{w}}$ denotes the restriction of the grafting σ to the domain $\mathcal{X}\backslash\vec{w}$. A unifier of $\bigvee_{j\in J} \exists \vec{w_j} \bigwedge_{i\in I_j} s_i =^?_{\mathcal{A}} t_i$ is a grafting σ that unifies at least one of the unification systems. The set of unifiers of a unification problem, D, or system, P, is denoted by $\mathcal{U}_{\mathcal{A}}(D)$ or $\mathcal{U}_{\mathcal{A}}(P)$, respectively.

DEFINITION 3.1. *A* **$\lambda\sigma$-unification problem** *P is a unification problem in the algebra $\mathcal{T}_{\lambda\sigma}(\mathcal{X})$ modulo the equational theory presented by $\lambda\sigma$. An* **equation** *of such a problem is denoted $a =^?_{\lambda\sigma} b$, where a and b are substitution-closed $\lambda\sigma$-terms of the same sort. An equation is called trivial when it is of the form $a =^?_{\lambda\sigma} a$. The set of variables of sort* TERM *in P is denoted by $\mathcal{T}var(P)$.*

We present a set of rewrite rule schemata used to simplify unification problems. The objective of applying the rules is to obtain a description of the set of unifiers. Basic decomposition rules for unification should be applied modulo the usual boolean simplification rules as presented in [9].

DEFINITION 3.2. *The set of* **$\lambda\sigma$-unification rules** *for the typed $\lambda\sigma$-unification problems is defined as the set of rules {Dec-λ, Dec-App, App-Fail, Exp-λ, Replace, Normalize} in Table 4, replacing the equality $=^?_{\lambda s_e}$ by $=^?_{\lambda\sigma}$, including the rule Exp-App defined in Table 3.*

Since $\lambda\sigma$ satisfies CR and WN the search can be restricted to η-long normal solutions that are graftings of the form $\{X/(\mathtt{n}\, a_1 \dots a_p)\}$ and $\{X/(Z[s]\, a_1 \dots a_p)\}$ or $\{X/\lambda.a\}$, when the type of X is atomic or functional, respectively.

Equations of the form $(\mathtt{n}\ a_1 \dots a_p) =^?_{\lambda\sigma} (\mathtt{m}\ b_1 \dots b_q)$ are transformed by *Dec-App* and *App-Fail* into the empty disjunction when $\mathtt{n} \neq \mathtt{m}$, as it has no solution, or into the conjunction $\bigwedge_{i=1..p} a_i =^?_{\lambda\sigma} b_i$, when $\mathtt{n} = \mathtt{m}$. The rules *Normalize* and *Dec-λ* normalize equations of the form $\lambda a =^?_{\lambda\sigma} \lambda b$ into equations of the form $a' =^?_{\lambda\sigma} b'$. The rule *Exp-$\lambda$* generates the grafting $\{X/\lambda.Y\}$ for a variable X of type $A \to B$, where Y is a new variable of type B.

For an equation $X[a_1 \dots a_p.\ \uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\ b_1 \dots b_q)$, with X atomic, solutions can only be grafting valuations of the form $\{X/(\mathtt{r}\ c_1 \dots c_k)\}$, where $\mathtt{r} \in \{1, \dots, p\} \cup \{m - n + p\}$. The rule *Exp-App* advances in direction towards the solution.

The rule *Replace* simply propagates to the current unification problem the grafting $\{X/a\}$, which corresponds to equations $X =^?_{\lambda\sigma} a$ previously added.

DEFINITION 3.3. *A unification system P is a* **$\lambda\sigma$-solved form** *if it is a conjunction of non trivial equations of the following forms:*

- *(Solved)* $X =^?_{\lambda\sigma} a$, *where the variable X does not occur anywhere else in P and a is in long normal form. Such an equation and variable are said to be* **solved** *in P.*
- *(Flex-Flex)* *non solved equations of the form $X[a_1 \dots a_p.\ \uparrow^n] =^?_{\lambda\sigma} Y[a'_1 \dots a'_{p'}.\ \uparrow^{n'}]$, where $X[a_1 \dots a_p.\ \uparrow^n]$ and $Y[a'_1 \dots a'_{p'}.\ \uparrow^{n'}]$ are long normal terms with X and Y of atomic type.*

In [9] it is shown that: any $\lambda\sigma$-solved form has $\lambda\sigma$-unifiers; deduction by the $\lambda\sigma$-unification rules of a well typed equation gives rise only to well typed equations, $\mathbb{T}$ and $\mathbb{F}$; solved problems are normalized for the $\lambda\sigma$-unification rules; and,

Table 3: *Exp-App* $\lambda\sigma$**-unification rule**

(Exp-App)	$P \wedge X[a_1 \dots a_p.\uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \dots b_q) \quad \rightarrow \quad P \wedge X[a_1 \dots a_p.\uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \dots b_q) \ \wedge \bigvee_{r \in R_p \cup R_i} \exists H_1 \dots H_k, X =^?_{\lambda\sigma} (\mathtt{r}\ H_1 \dots H_k)$	
	if X has an atomic type and is not solved where $H_1, \dots, H_k$ are variables of appropriate types, not occurring in P, with the environments $\Gamma_{H_i} = \Gamma_X$, R_p is the subset of $\{1, \dots, p\}$ such that $(\mathtt{r}\ H_1 \dots H_k)$ has the right type, $R_i =$ if $m \geq n+1$ then $\{m-n+p\}$ else $\emptyset$	

if a system is a conjunction of equations that cannot be reduced by the $\lambda\sigma$-unification rules then it is solved. These facts enabled [9] to prove completeness and correctness of the $\lambda\sigma$-unification rules.

4. λs_e-UNIFICATION

Normal form characterization of λs_e-terms jointly with WN and CR properties are the essential requirements to develop a unification method for the λs_e-calculus.

4.1 λs_e-normal forms

We present a characterization of λs_e-normal forms whose main operators are either σ or φ (i.e. of type 3. and 4. in Corollary 2.9). This will help simplify our presentation of the unification rules and of the *flex-flex* equations.

Observe that left arguments of the σ operator or arguments of the φ operator at λs_e-normal forms are neither applications, nor abstractions, nor de Bruijn indices. For instance, $\varphi^j_i(a\ \ b) \rightarrow (\varphi^i_k a\ \ \varphi^i_k b)$, $(a\ \ b)\sigma^i c \rightarrow (a\sigma^i c\ \ b\sigma^i c)$. Hence, the sole possibility is to have a meta-variable as a left argument. Thus one has to consider terms with alternating sequences of operators φ and σ whose left innermost argument is a meta-variable; for instance, $((\varphi^{j_3}_{i_3}((\varphi^{j_1}_{i_1} X)\sigma^{i_2} a))\sigma^{i_4} b)\sigma^{i_5} c$.

DEFINITION 4.1. *Let t be a λs_e-normal term whose root operator is either σ or φ and let X be its left innermost meta-variable. Denote by $\psi^{j_k}_{i_k}$ the operator at the k^{th} position following the sequence of operators φ and σ, considering only left arguments of the σ operators, in the innermost outermost ordering. Additionally, if $\psi^{j_k}_{i_k}$ corresponds to an operator φ then j_k and i_k denote its super and subscripts, respectively and if $\psi^{j_k}_{i_k}$ corresponds to an operator σ then $j_k = 0$ and i_k denotes its superscript. Let a_k denote the corresponding right argument of the k^{th} operator if $\psi^{j_k}_{i_k} = \sigma^{i_k}$ and the empty argument if $\psi^{j_k}_{i_k} = \varphi^{j_k}_{i_k}$. The* **skeleton** *of t written $sk(t)$ is $\psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p)$.*

EXAMPLE 4.2. *Let t be a λs_e-normal term of the form $((\varphi^{j_3}_{i_3}((\varphi^{j_1}_{i_1} X)\sigma^{i_2} a))\sigma^{i_4} b)\sigma^{i_5} c$. Then its skeleton is $sk(t) = \psi^0_{i_5}\psi^0_{i_4}\psi^{j_3}_{i_3}\psi^0_{i_2}\psi^{j_1}_{i_1}(X, a, b, c)$.* •

LEMMA 4.3. *Let t be a λs_e-normal term whose root operator is either σ or φ and let $sk(t) = \psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p)$. Successive subscripts i_k and i_{k+1} satisfy the following:*

1. *$i_k > i_{k+1}$ if ψ_k and ψ_{k+1} are both σ operators or both φ operators;*
2. *$i_k \geq i_{k+1}$ if ψ_k and ψ_{k+1} are φ and σ operators, respectively;*
3. *$i_k > i_{k+1} + 1$ if ψ_k and ψ_{k+1} are σ and φ operators, respectively.*

PROOF. By simple analysis of the arithmetic constraints at the λs_e rewrite rules. □

4.2 Unification in the λs_e-calculus

Unification notions (such as an *equation* $a =^?_{\lambda s_e} b$ or a *set of unifiers* $\mathcal{U}_{\lambda s_e}(P)$) are defined analogously to the $\lambda\sigma$ unification setting of the previous section.

DEFINITION 4.4. *The set of* λs_e**-unification rules** *for the typed λs_e-unification problems is defined as the set of rules in Table 4.*

Since λs_e is CR and WN, the search can be restricted to η-long normal solutions that are graftings that bind functional variables to η-long normal terms of the form $\lambda.a$ and atomic variables into η-long normal terms of the form $(\mathtt{k}\ b_1 \dots b_p)$ or $a\sigma^i b$ or $\varphi^i_k a$, where in the first case $\mathtt{k}$ could be omitted and p could be zero. The use of the η rule is important to reduce the number of cases (or unification rules) to be considered when defining the unification algorithm, but as for the $\lambda\sigma$-calculus, the η-rule can be dropped [9]. As for the $\lambda\sigma$-unification, *Normalize* and *Dec-*λ use the fact that λs_e is CR and WN to normalize equations of the form $\lambda.a =^?_{\lambda s_e} \lambda.b$ into $a' =^?_{\lambda s_e} b'$ and the rule *Replace* propagates the grafting $\{X/a\}$ corresponding to equations $X =^?_{\lambda s_e} a$. *Exp-*λ generates the grafting $\{X/\lambda.Y\}$ for a variable X of type $A \rightarrow B$, where Y is a new variable of type B.

Equations of the form $(\mathtt{n}\ a_1 \dots a_p) =^?_{\lambda s_e} (\mathtt{m}\ b_1 \dots b_q)$ are transformed by the rules *Dec-App* and *App-Fail* into the empty disjunction when $\mathtt{n} \neq \mathtt{m}$, as it has no solution, or into the conjunction $\bigwedge_{i=1..p} a_i =^?_{\lambda s_e} b_i$, when $\mathtt{n} = \mathtt{m}$. Remember that by terms of the form $(\mathtt{n}\ a_1 \dots a_p)$ we also mean those where $\mathtt{n}$ is omitted or $p = 0$. Analogously, the rules *Dec-*σ and *Dec-*φ decompose equations with leading operators σ and φ. But, the corresponding rules *σ-Fail* and *φ-Fail* should omit *flex-flex* equations as the following example shows.

EXAMPLE 4.5. *Let $(\lambda.(\lambda.(X\ \ \mathtt{2})\ \ \mathtt{1})\ \ Y) =^?_{\lambda s_e} (\lambda.(Z\ \ \mathtt{1})\ \ U)$ be a unification problem, where X, Y, Z and U are meta-variables of types $A \rightarrow A, A, A \rightarrow A$ and A, respectively.*

Then $(\lambda.(\lambda.(X\ \ \mathtt{2})\ \ \mathtt{1})\ \ Y) \rightarrow^* ((X\sigma^2 Y)\sigma^1(\varphi^1_0 Y)\ \ \varphi^1_0 Y)$ and $(\lambda.(Z\ \ \mathtt{1})\ \ U) \rightarrow^* (Z\sigma^1 U\ \ \varphi^1_0 U)$. Thus by applying the rule Normalize to the original equation we obtain the equation

Table 4: λs_e-unification rules

(Dec-λ)	$P \wedge \lambda_A.a =^?_{\lambda s_e} \lambda_A.b \quad \rightarrow \quad P \wedge a =^?_{\lambda s_e} b$
(Dec-App)	$P \wedge (\mathtt{n}\ a_1 \dots a_p) =^?_{\lambda s_e} (\mathtt{n}\ b_1 \dots b_p) \quad \rightarrow \quad P \bigwedge_{i=1..p} a_i =^?_{\lambda s_e} b_i$
(App-Fail)	$P \wedge (\mathtt{n}\ a_1 \dots a_p) =^?_{\lambda s_e} (\mathtt{m}\ b_1 \dots b_q) \quad \rightarrow \quad \mathbb{F}$ if $\mathtt{n} \neq \mathtt{m}$
(Dec-σ)	$P \wedge a\sigma^i b =^?_{\lambda s_e} c\sigma^i d \quad \rightarrow \quad P \wedge a =^?_{\lambda s_e} c \wedge b =^?_{\lambda s_e} d$
(σ-Fail)	$P \wedge a\sigma^i b =^?_{\lambda s_e} c\sigma^j d \quad \rightarrow \quad \mathbb{F}$ if $i \neq j$ and $a\sigma^i b =^?_{\lambda s_e} c\sigma^j d$ is not *flex-flex*
(Dec-φ)	$P \wedge \varphi^i_k a =^?_{\lambda s_e} \varphi^i_k b \quad \rightarrow \quad P \wedge a =^?_{\lambda s_e} b$
(φ-Fail)	$P \wedge \varphi^i_k a =^?_{\lambda s_e} \varphi^j_l b \quad \rightarrow \quad \mathbb{F}$ if $i \neq j$ or $k \neq l$ and $\varphi^i_k a =^?_{\lambda s_e} \varphi^j_l b$ is not *flex-flex*
(Exp-λ)	$P \quad \rightarrow \quad \exists(Y : A.\Gamma \vdash B), P \wedge X =^?_{\lambda s_e} \lambda_A.Y$ if $(X : \Gamma \vdash A \rightarrow B) \in \mathcal{T}var(P), Y \notin \mathcal{T}var(P)$, and X is a unsolved variable
(Exp-App)	$P \wedge \psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p) =^?_{\lambda s_e} (\mathtt{m}\ b_1 \dots b_q) \quad \rightarrow \quad P \wedge \psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p) =^?_{\lambda s_e} (\mathtt{m}\ b_1 \dots b_q) \quad \wedge \quad \bigvee_{r \in R_p \cup R_i} \exists H_1, \dots, H_k, X =^?_{\lambda s_e} (\mathtt{r}\ H_1 \dots H_k)$ if $\psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p)$ is the skeleton of a λs_e-normal term and X has an atomic type and is not solved where $H_1, \dots, H_k$ are variables of appropriate types, not occurring in P, with the environments $\Gamma_{H_i} = \Gamma_X$, R_p is the subset of $\{i_1, \dots, i_p\}$ of superscripts of the σ operator such that $(\mathtt{r}\ H_1 \dots H_k)$ has the right type, $R_i = \bigcup_{k=0}^{p}$ if $i_k \geq m+p-k-\sum_{l=k+1}^{p} j_l > i_{k+1}$ then $\{m+p-k-\sum_{l=k+1}^{p} j_l\}$ else $\emptyset$, where $i_0 = \infty, i_{p+1} = 0$
(Replace)	$P \wedge X =^?_{\lambda s_e} a \quad \rightarrow \quad \{X/a\}P \wedge X =^?_{\lambda s_e} a$ if $X \in \mathcal{T}var(P), X \notin \mathcal{T}var(a)$ and $a \in \mathcal{X} \Rightarrow a \in \mathcal{T}var(P)$
(Normalize)	$P \wedge a =^?_{\lambda s_e} b \quad \rightarrow \quad P \wedge a' =^?_{\lambda s_e} b'$ if a or b is not in long normal form where a' is the long normal form of a if a is not a solved variable and a otherwise. b' is defined from b identically

$((X\sigma^2 Y)\sigma^1(\varphi^1_0 Y)\ \ \varphi^1_0 Y) =^?_{\lambda s_e} (Z\sigma^1 U\ \ \varphi^1_0 U)$ *which can be decomposed into* $(X\sigma^2 Y)\sigma^1(\varphi^1_0 Y) =^?_{\lambda s_e} Z\sigma^1 U \ \wedge\ \varphi^1_0 Y =^?_{\lambda s_e} \varphi^1_0 U$ *and subsequently into* $(X\sigma^2 Y) =^?_{\lambda s_e} Z \ \wedge\ \varphi^1_0 Y =^?_{\lambda s_e} U \ \wedge\ Y =^?_{\lambda s_e} U$.

Since $\forall n \in \mathbb{N}$, $\varphi^1_0 \mathtt{n} \rightarrow \mathtt{n}$, *the equation* $\varphi^1_0 Y =^?_{\lambda s_e} U$ *always has solutions and solutions of the last two equations are graftings of the form* $\{Y/V, U/V\}$. *Additionally, observe that the first equation has also a variety of solutions: take* $\{X/\mathtt{n}\}$; *thus if* $n > 2$, $\{Z/\mathtt{n}-1\}$ *else if* $n = 2$, $\{Z/\varphi^2_0 Y\}$ *else* $\{Z/\mathtt{1}\}$.

Analogously, by normalization and decomposition with the $\lambda\sigma$*-unification rules we have*

$$(\lambda.(\lambda.(X\ \ \mathtt{2})\ \ \mathtt{1})\ \ Y) =^?_{\lambda\sigma} (\lambda.(Z\ \ \mathtt{1})\ \ U) \rightarrow_{Normalize}$$

$$(X[Y.Y.id]\ \ Y) =^?_{\lambda\sigma} (Z[U.id]\ \ U)$$

which can be decomposed into $X[Y.Y.id] =^?_{\lambda\sigma} Z[U.id] \ \wedge\ Y =^?_{\lambda\sigma} U$. *A further step of replacement gives the corresponding flex-flex equation in the language of the* $\lambda\sigma$*-calculus* $X[Y.Y.id] =^?_{\lambda\sigma} Z[Y.id]$. •

In the $\lambda\sigma$-calculus, *Exp-App* advances in direction towards solutions for equations of the form

$$X[a_1 \dots a_p.\uparrow^n] =^?_{\lambda s_e} (\mathtt{m}\ b_1 \dots b_q)$$

where X is an unsolved variable of an atomic type. The λs_e-unification rule *Exp-App* has the analogous role for λs_e-unification problems. Use of λs_e-normal forms in *Exp-App* is not essential. This is done with the sole objective of simplifying the case analysis presented in the definition of the rule and its completeness proof. In fact, this can be dropped and be subsequently incorporated as an efficient unification strategy, where before applying *Exp-App* λs_e-unification problems are normalized.

EXAMPLE 4.6. *From the unification problem*

$$\lambda.(\lambda.(Y\ \ \mathtt{1})\ \ \lambda.(X\ \ \mathtt{1})) =^? \lambda.(\lambda.V\ \ \lambda.W)$$

we reach the two equations:

- $(Y[\lambda.(X\ \ \mathtt{1}).id]\ \ \lambda.(X\ \ \mathtt{1})) =^?_{\lambda\sigma} V[\lambda.W.id]$
- $(Y\sigma^1\lambda.(X\ \ \mathtt{1})\ \ \lambda.(\varphi^1_1\ \ \mathtt{1})) =^?_{\lambda s_e} V\sigma^1\lambda.W$

After applying the corresponding Exp-App rules, with $V =^?_{\lambda\sigma} (V_1\ \ V_2)$ *and* $V =^?_{\lambda s_e} (V_1\ \ V_2)$, *additional equations appear:* $\lambda.(X\ \ \mathtt{1}) =^?_{\lambda\sigma} V_2[\lambda.(X\ \ \mathtt{1}).id]$ *and* $\lambda.(\varphi^1_1 X\ \ \mathtt{1}) =^?_{\lambda s_e} V_2\sigma^1\lambda.(X\ \ \mathtt{1})$. *Solutions result by selecting the case* $V_2 =^?_{\lambda\sigma} \mathtt{1}$ *or correspondingly* $V_2 =^?_{\lambda s_e} \mathtt{1}$. •

DEFINITION 4.7. *A unification system* P *is a* λs_e**-solved form** *if it is a conjunction of non trivial equations of the following forms:*

(Solved) $X =^?_{\lambda s_e} a$, *where the variable* X *does not occur anywhere else in* P *and* a *is in long normal form. Such an equation and variable are said to be* **solved** *in* P.

(Flex-Flex) *non solved equations between long normal forms whose root operator is* σ *or* φ *which can be represented as equations between their skeleton:* $\psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p) =^?_{\lambda s_e} \psi^{l_q}_{k_q} \dots \psi^{l_1}_{k_1}(Y, b_1, \dots, b_q)$.

REMARK 4.8. *Consider a* λs_e*-normal form* t *whose root operator is either* σ *or* φ *and with skeleton of the form* $sk(t) = \psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p)$. *Then by binding* X *with* $\mathtt{n}$, $n > i_1$, *one obtains the normal form* $t \to^* \mathtt{n} + \sum_{k=1}^{p} \mathtt{j}_k - \mathtt{p}$. *This is a direct consequence of lemma 4.3.* •

LEMMA 4.9. *Any* λs_e*-solved form has* λs_e*-unifiers.*

PROOF. Since solved forms appearing in a system P straightforwardly define bindings between variables that do not appear anywhere else in P and in terms in long normal form, it is enough to prove that *flex-flex* equations have unifiers.

Let P be a system in λs_e-solved form including a *flex-flex* equation of the form

$$\psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p) =^?_{\lambda s_e} \psi^{l_q}_{k_q} \dots \psi^{l_1}_{k_1}(Y, b_1, \dots, b_q)$$

This equation has always solutions. Select for instance bindings $\{X/\mathtt{n}, Y/\mathtt{m}\}$ such that $n > i_1, m > l_1$ and $n + \sum_{r=1}^{p} j_r - p = m + \sum_{r=1}^{q} k_r - q$ (see Remark 4.8). □

Now we show some properties of the λs_e-unification rules.

LEMMA 4.10 (WELL-TYPEDNESS). *Deduction by applying the* λs_e*-unification rules of a well typed equation gives rise only to well typed equations,* $\mathbb{T}$ *and* $\mathbb{F}$.

PROOF. It is proved by analyzing, rule by rule, the type of the resulting transformed equation. □

LEMMA 4.11. *Solved problems are normalized for the* λs_e*-unification rules and, conversely, if a system is a conjunction of equations that cannot be reduced by applying the* λs_e*-unification rules then it is solved.*

PROOF. *Solved* and *flex-flex* equations are normalized. Conversely, suppose P is a non solved system. Then P contains an equation $a =^?_{\lambda s_e} b$ that is neither *solved* nor *flex-flex*. Supposing that neither *Normalize* nor *Replace* apply and according to the characterization of λs_e-normal forms at Corollary 2.9, we have:

Firstly, if a if of the form $\lambda.a'$ then, since b is long normal, the sole possibility of having a well typed equation implies b is of the form $\lambda.b'$ and rule *Dec-*λ applies.

Secondly, suppose that a is of the form $(\mathtt{k}\ a_1 \dots a_p)$. Then if b is of the form $(\mathtt{l}\ b_1 \dots b_q)$, then either *Dec-App* or *Fail-App* apply (remember here that both $\mathtt{k}$ and $\mathtt{l}$ could be omitted and p and q could be zero). If b has root operator σ or φ then rule *Exp-App* applies.

Cases of equations between terms with main operators σ and φ are either *flex-flex* or can be reduced by the rules *Dec-*σ, *Dec-*φ, σ*-Fail* or φ*-Fail*. □

DEFINITION 4.12. *Let* P *and* P' *be* λs_e*-unification problems, let "rule" denote the name of a* λs_e*-unification rule and "*$\to^{rule}$*" its corresponding deduction relation over unification problems. We define the following properties of rule:* **correctness***: 1.* $P \to^{rule} P'$ *implies* $\mathcal{U}_{\lambda s_e}(P') \subseteq \mathcal{U}_{\lambda s_e}(P)$. **completeness***:* $P \to^{rule} P'$ *implies* $\mathcal{U}_{\lambda s_e}(P) \subseteq \mathcal{U}_{\lambda s_e}(P')$.

THEOREM 4.13 (CORRECTNESS AND COMPLETENESS). *The* λs_e*-unification rules are correct and complete.*

PROOF. We present a sketch of the interesting verification of completeness of the *Exp-App* rule.

Consider $P \wedge \psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p) =^?_{\lambda s_e} (\mathtt{m}\ b_1 \dots b_q)$ and a λs_e-unifier θ of this unification problem. Then $\theta(X) = (\mathtt{r}\ c_1 \dots c_s)$ and since $\psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(X, a_1, \dots, a_p)$ is the skeleton of a λs_e-normal form, we have two cases to consider: either r differs from all i_k such that $\psi^{j_k}_{i_k}$ corresponds to a σ operator or $r = i_k$ for some k such that $\psi^{j_k}_{i_k} = \sigma^{i_k}$.

In the first case, let $i_0 = \infty$ and $i_{p+1} = 0$ and suppose that $i_{k+1} < r \leq i_k$ for some $0 \geq k \geq p$ such that either $k = p$ or $\psi^{j_k}_{i_k}$ corresponds to a σ operator. Then $\psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(\mathtt{r}, a'_1, \dots, a'_p) \to^* \mathtt{r} + \sum_{l=k+1}^{p} \mathtt{j}_l - (\mathtt{p} - \mathtt{k})$.

If $r = i_k$ for some $1 \leq j \leq p$ corresponding to a σ operator, then we have the following derivation:

$$\psi^{j_p}_{i_p} \dots \psi^{j_1}_{i_1}(\mathtt{r}, a'_1, \dots, a'_p) \to^* \psi^{j_p}_{i_p} \dots \psi^{j_k}_{i_k}(\mathtt{r}, a'_k, \dots, a'_p) \to$$

$$\psi^{j_p}_{i_p} \dots \psi^{j_{k+1}}_{i_{k+1}}(\varphi_0^{i_k} a'_k, \dots, a'_p) \to^* \varphi_0^{i_k - p + k + \sum_{l=k+1}^{p} j_l} a'_k$$

In both cases θ is clearly a solution of

$$\exists H_1, \dots, H_k, X =^?_{\lambda s_e} (\mathtt{r}\ H_1\ \dots\ H_k)$$

by selecting $H_1, \dots, H_k$ appropriately and, consequently, it is solution of the original problem and

$$\bigvee_{r \in R_p \cup R_i} \exists H_1, \dots, H_k, X =^?_{\lambda s_e} (\mathtt{r}\ H_1 \dots H_k)$$

□

5. ARITHMETIC PROPERTIES OF THE λs_e-UNIFICATION RULES

The arithmetic constraint that naturally has emerged when developing the *Exp-App* λs_e-unification rule is more expressive than the one of the unification setting based on the $\lambda\sigma$-style. This, jointly with an efficient arithmetic deductive

method, speed up the verification of possible splittings and the search for solutions in the corresponding case analysis.

For the case of the $\lambda\sigma$-calculus, equations of the form

$$X[a_1 \dots a_p.\ \uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \dots b_q)$$

may have solutions of the form $(\mathtt{r}\, H_1 \dots H_k)$, where $r - p + n = m$. In fact, $\mathtt{1}[\uparrow^{r-1}][a_1 \dots a_p.\ \uparrow^n] \to^* \mathtt{1}[\uparrow^{r-1-p+n}]$.

In [9] the $\lambda\sigma$-calculus is presented using only the de Bruijn index 1. Thus the detection of the previous kind of solutions is very inefficient. In fact, observe that since $\uparrow^n$ abbreviates $(n-1)$-compositions of $\uparrow$, finding the first component $\mathtt{1}[\uparrow^{r-1}]$ of these possible solutions can be done only after realizing a process of enumeration of the p a_i components and the $(n-1)$ $\uparrow$ of $\mathtt{n} \equiv \mathtt{1}[\uparrow^{n-1}]$. Since λs_e-terms are written using all the natural indices, one can state that searching for redices of the unification rules and determining solved and *flex-flex* equations in our unification setting are more efficient than in the language of the $\lambda\sigma$-calculus.

We show that the first numeric components of bindings for a meta-variable X of solutions of equations of the form

$$\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(X, a_1, \dots, a_p) =^?_{\lambda s_e} (\mathtt{m}\, b_1 \dots b_q)$$

are determined in a unique way.

LEMMA 5.1. *Let $\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(X, a_1, \dots, a_p)$ be a skeleton of a λs_e-normal term and suppose that $n > k_1$ and $m \le k_p$. Then $\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(n, a_1, \dots, a_p) \to^* n - p + \sum_{r=1}^{p} j_r > m$.*

PROOF. Firstly, observe that since $k_1, \dots, k_p$ is a decreasing sequence, we have $n > k_1 \ge \dots \ge k_p \ge m$ and thus $k_1 - k_p < n - m$ which implies $m \le n - (k_1 - k_p + 1)$.

Secondly, observe that $\sum_{r=1}^{p} j_r \ge 0$. Thus the sole possibility to have $n - p + \sum_{r=1}^{p} j_r \le n - (k_1 - k_p + 1)$ is $p - 1 \ge k_1 - k_p$. We consider two cases:

If $p - 1 = k_1 - k_p$ then $\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(n, a_1, \dots, a_p) \to^* n - p + \sum_{r=1}^{p} j_r \ge n - p = n - (k_1 - k_p + 1) \ge m$. Moreover, observe that if there exists some operator φ, say $\psi^{j_i}_{k_i}$ in the sequence of the skeleton, then $\sum_{r=1}^{p} j_r \ge j_i > 0$ which implies $n - p + \sum_{r=1}^{p} j_r > m$. If the sequence consists only of σ operators, then $m < k_p$ and also $n - p + \sum_{r=1}^{p} j_r > m$.

If $p - 1 > k_1 - k_p$ then there exists at least one $1 \le i < p$ such that $\psi^{j_i}_{k_i} = \varphi^{j_i}_{k_i}$ and $\psi^{j_{i+1}}_{k_{i+1}} = \sigma^{k_{i+1}}$ being $k_i = k_{i+1}$. Thus $\psi^{j_{i+1}}_{k_{i+1}} \psi^{j_i}_{k_i}(n, a_i, a_{i+1}) \to \psi^{j_{i+1}}_{k_{i+1}}(n + j_i - 1, a_{i+1}) \to n + j_i - 2 \ge n - 1$. For each of these subsequences we have the analogous situation, obtaining for the whole sequence $n - p + \sum_{r=1}^{p} j_r > n - (k_1 - k_p + 1) \ge m$. □

LEMMA 5.2 (UNICITY). *Consider the equation*
$\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(X, a_1, \dots, a_p) =^?_{\lambda s_e} (\mathtt{m}\, b_1 \dots b_q)$
where $\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(X, a_1, \dots, a_p)$ is the skeleton of a λs_e-normal form. The first numerical component of bindings for the meta-variable X of solutions of this equations is unique.

PROOF. Observe firstly the three possible cases for bindings $\{X/(\mathtt{n} \dots)\}$:

1. $n \le k_p$: $\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(n, a_1, \dots, a_p) \to^* \psi^{j_p}_{k_p}(n, a_p)$. Since case $n = k_p$ thus $\psi^{j_p}_{k_p} = \varphi^{j_p}_{k_p}$, we have $\psi^{j_p}_{k_p}(n, a_p) \to n$.

2. $k_{i+1} < n \le k_i$: we have $\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(n, a_1, \dots, a_p) \to^* \psi^{j_p}_{k_p} \dots \psi^{j_i}_{k_i}(n, a_i, \dots, a_p)$. Since case $n = k_i$ we have $\psi^{j_i}_{k_i} = \varphi^{j_i}_{k_i}$, then in the two cases: $n = k_i$ and $n < k_i$, we have $\psi^{j_p}_{k_p} \dots \psi^{j_i}_{k_i}(n, a_i, \dots, a_p) \to \psi^{j_p}_{k_p} \dots \psi^{j_{i+1}}_{k_{i+1}}(n, a_{i+1}, \dots, a_p) \to^* n - (p - i) + \sum_{r=i+1}^{p} j_r$.

3. $k_1 < n$: $\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(n, a_1, \dots, a_p) \to^* n - p + \sum_{r=1}^{p} j_r$.

We analyze the more general case of natural numbers between subscripts k. Select $k_{i+1} < n_1 \le k_i$ and $k_{l+1} < n_2 \le k_l$, for $i > l$. Then $\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(n_1, a_1, \dots, a_p) \to^* n_1 - (p - i) + \sum_{r=i+1}^{p} j_r$ and $\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(n_2, a_1, \dots, a_p) \to^* n_2 - (p - l) + \sum_{r=l+1}^{p} j_r$.

Since $k_1, \dots, k_p$ is a decreasing sequence we have $n_1 < n_2$. By the previous Lemma we obtain:

$\psi^{j_i}_{k_i} \dots \psi^{j_{l+1}}_{k_{l+1}} \dots \psi^{j_1}_{k_1}(n_2, a_1, \dots, a_i) \to^*$
$\psi^{j_i}_{k_i} \dots \psi^{j_{l+1}}_{k_{l+1}}(n_2, a_{l+1}, \dots, a_i) \to^* n_2 - (i - l) + \sum_{r=l+1}^{i} j_r > n_1$.

Then $n_2 - (p - l) + \sum_{r=l+1}^{p} j_r > n_1 - (p - i) + \sum_{r=i+1}^{p} j_r$, which concludes the proof. □

Observe that when searching for solutions of

$$\psi^{j_p}_{k_p} \dots \psi^{j_1}_{k_1}(X, a_1, \dots, a_p) =^?_{\lambda s_e} (\mathtt{m}\, b_1 \dots b_q)$$

a binding for X to an application should be selected, whose first component is a natural number n such that for some i, $k_{i+1} < n \le k_i$ and $n - (p - i) + \sum_{r=i+1}^{p} j_r = m$. This corresponds to searching for solutions of an integer linear problem.

6. CONCLUSIONS

Advantages of the here proposed unification method, with respect to the one formulated by Dowek, Hardin and Kirchner in [9], are mainly consequences of the inherent differences between both styles of explicit substitution of the two calculi in question: the λs_e- and $\lambda\sigma$-calculi.

1. In our unification setting we remain close to the λ-calculus because we don't need to use more than one kind of objects: the objects of the λ-calculus. We don't use substitution objects as is done in the $\lambda\sigma$-unification approach. From this point of view, we think that our approach is more clear semantically; mainly, because the principal intention and obvious application of any unification via explicit substitution in some version of the λ-calculus is, of course, to solve unification problems in the pure λ-calculus.

2. Because of the fact that for both methods, the *Normalize* unification rule depends on the subjacent properties

of the λs_e and $\lambda\sigma$ rewrite rules, correspondingly, and that the underlying reduction processes based on the λs_e- and $\lambda\sigma$-calculi are incomparable (see for instance [16]), one cannot say that λs_e-unification is more (or less) efficient than the unification setting proposed in [9]. But at least one can state that searching for redices of the unification rules (and determining solved and *flex-flex* equations) is more efficient, since λs_e terms are written using natural indices. Of course, in the praxis, this problem can be easily solved in the $\lambda\sigma$ setting by overloading the notation $\mathtt{n}$ to represent the corresponding $\lambda\sigma$-term $(\mathtt{1}[\uparrow^{n-1}])$ incorporating to the unification mechanism the necessary built-in linear arithmetic deductive method.

Additionally, we think that the arithmetic constraint that naturally results when defining the *Exp-App* unification rule in the λs_e setting is more expressive than the one of the $\lambda\sigma$. This, jointly with an efficient arithmetic deductive method, speed up the verification of possible splittings and the search for solutions in the corresponding case analysis.

As pointed out in [9], the use of explicit substitution enables us to translate HOU problems into first order ones. This results in a simpler development and analysis of HOU methods. The proposed unification method and its further developments are not only relevant because of the obvious necessity of analyzing, developing and implementing HOU procedures to improve performance (and expressiveness) of the current higher order deductive systems and languages. We think that our work is also important because of the necessity of comparing the advantages, disadvantages and appropriateness of both the λs_e- and $\lambda\sigma$-style of explicit substitution in a practical and relevant setting incrementing in this way the theoretic knowledge about the properties of the involved calculi.

In order to obtain a HOU procedure useful in practice, an efficient and complete unification strategy was developed in [3]. In [9] the rules for unification of $\lambda\sigma$-terms are related to HOU on the pure λ-calculus by the *pre-cooking* and *back* translations. This was also done for the λs_e-calculus in [3].

In the sequel we present in an informal way one example on how to apply our unification method to HOU problems in the λ-calculus. For a formal presentation see [3].

Observe that unifying two terms a and b in the λ-calculus consists in finding a *substitution* θ such that $\theta(a) =_{\beta\eta} \theta(b)$. But in the λ-calculus (and in the $\lambda\sigma$-calculus) as well as in the λs_e-calculus the notion of substitution is different from the first order one or grafting, as was shown in Section 2. Thus using the notation of substitution in Definitions 2.3 and 2.4 a unifier in the λ-calculus of the problem $\lambda.X =^?_{\beta\eta} \lambda.\mathtt{2}$ is not a term $t = \theta X$ such that $\lambda.t =^?_{\beta\eta} \lambda.\mathtt{2}$ but a term $t = \theta X$ such that $\theta(\lambda.X) = \lambda.\theta^+(X) = \lambda.\mathtt{2}$ as $\{X/t\}\lambda.X = \lambda.\{X/t^+\}X = \lambda.t^+$ and not $\lambda.t$. This observation can be extended to any unifier and by translating appropriately λ-terms $a, b \in \Lambda_{dB}(\mathcal{X})$, the HOU problem $a =^?_{\beta\eta} b$ can be reduced to equational unification. In [9] a translation called *pre-cooking* from $\Lambda_{dB}(\mathcal{X})$ terms into the language of the $\lambda\sigma$-calculus is given such that searching for solutions of the corresponding $\lambda\sigma$-unification problem corresponds to searching for solutions of the higher order problem $a =^?_{\beta\eta} b$. In the following example, we illustrate informally the analogous situation in the λs_e-calculus.

EXAMPLE 6.1. *Consider the higher order unification problem* $\lambda.(X\ \mathtt{2}) =^?_{\beta\eta} \lambda.\mathtt{2}$, *where* $\mathtt{2}$ *and* X *are of type* A *and* $A \to A$, *respectively. Observe that applying a substitution solution* θ *to the* $\Lambda_{dB}(\mathcal{X})$*-term* $\lambda.(X\ \mathtt{2})$ *gives* $\theta(\lambda.(X\ \mathtt{2})) = \lambda.(\theta^+(X)\ \mathtt{2})$. *Then in the* λs_e*-calculus we are searching for a grafting* θ' *such that* $\theta'(\lambda.(\varphi_0^2(X)\ \mathtt{2})) =_{\lambda s_e} \lambda.\mathtt{2}$. *Correspondingly, in the* $\lambda\sigma$*-calculus the term* $\lambda.(X\ \mathtt{2})$ *is translated or pre-cooked into* $\lambda.(X[\uparrow]\ \mathtt{2})$. *Then we should search for unifiers for the problem* $\lambda.(\varphi_0^2(X)\ \mathtt{2}) =^?_{\lambda s_e} \lambda.\mathtt{2}$.

Now we apply λs_e*-unification rules to the former problem. By applying Dec-*λ *and Exp-*λ *we get* $(\varphi_0^2(X)\ \mathtt{2}) =^?_{\lambda s_e} \mathtt{2}$ *and subsequently* $\exists Y(\varphi_0^2(X)\ \mathtt{2}) =^?_{\lambda s_e} \mathtt{2} \wedge X =^?_{\lambda s_e} \lambda.Y$. *Then by applying Replace and Normalize we obtain* $\exists Y(\varphi_0^2(\lambda.Y)\ \mathtt{2}) =^?_{\lambda s_e} \mathtt{2} \wedge X =^?_{\lambda s_e} \lambda.Y$ *and* $\exists Y(\varphi_1^2 Y)\sigma^1\mathtt{2} =^?_{\lambda s_e} \mathtt{2} \wedge X =^?_{\lambda s_e} \lambda.Y$. *Now, by applying rule Exp-app we obtain*

$$(\exists Y(\varphi_1^2 Y)\sigma^1\mathtt{2} =^?_{\lambda s_e} \mathtt{2} \wedge X =^?_{\lambda s_e} \lambda.Y) \quad \wedge$$

$$(Y =^?_{\lambda s_e} \mathtt{1} \vee Y =^?_{\lambda s_e} \mathtt{2})$$

which by Replace gives

$$((\varphi_1^2\mathtt{1})\sigma^1\mathtt{2} =^?_{\lambda s_e} \mathtt{2} \wedge X =^?_{\lambda s_e} \lambda.\mathtt{1}) \quad \vee$$

$$((\varphi_1^2\mathtt{2})\sigma^1\mathtt{2} =^?_{\lambda s_e} \mathtt{2} \wedge X =^?_{\lambda s_e} \lambda.\mathtt{2})$$

and, finally, by Normalize

$$(\mathtt{2} =^?_{\lambda s_e} \mathtt{2} \wedge X =^?_{\lambda s_e} \lambda.\mathtt{1}) \vee (\mathtt{2} =^?_{\lambda s_e} \mathtt{2} \wedge X =^?_{\lambda s_e} \lambda.\mathtt{2})$$

In this way substitution solutions $\{X/\lambda.\mathtt{1}\}$ *and* $\{X/\lambda.\mathtt{2}\}$ *are found.*

To complete the analysis observe that by definition of substitution (Definitions 2.3, 2.4) and beta reduction in $\Lambda_{dB}(\mathcal{X})$ *we have* $\{X/\lambda.\mathtt{1}\}(\lambda.(X\ \mathtt{2})) = \lambda.(\{X/(\lambda.\mathtt{1})^+\}(X)\ \mathtt{2}) = \lambda.(\lambda.\mathtt{1}^{+1}\ \mathtt{2}) = \lambda.(\lambda.\mathtt{1}\ \mathtt{2}) =_\beta \lambda.\mathtt{2}$ *and* $\{X/\lambda.\mathtt{2}\}(\lambda.(X\ \mathtt{2})) = \lambda.(\{X/(\lambda.\mathtt{2})^+\}(X)\ \mathtt{2}) = \lambda.(\lambda.\mathtt{2}^{+1}\ \mathtt{2}) = \lambda.(\lambda.\mathtt{3}\ \mathtt{2}) =_\beta \lambda.\mathtt{2}$. *Observe that the last application of beta reduction is as follows:* $(\lambda.\mathtt{3}\ \mathtt{2}) =_\beta \{\mathtt{1/2}\}(\mathtt{3}) = \mathtt{2}$. •

In general, before the unification process, a λ-term a should be translated into the λs_e-term a' resulting by simultaneously replacing each occurrence of a meta-variable X at position i in a with $\varphi_0^{k+1}X$, where k is the number of abstractors between the root position of a, ε, and position i. If $k = 0$ then the occurrence of X remains unchanged.

In [7] it was shown that for an efficient implementation of the $\lambda\sigma$-HOU approach, the use of terms decorated with their corresponding types and environments is useful. For instance, observe that for applying unification rules such as *Exp-App* and *Exp-*λ, it is necessary to know the types and the environments of subterms of the current unification problem. In relation with that implementation, where repeated execution of a type-checking algorithm is avoided by decorating terms, our HOU approach has the clear advantage of having less expensive decorations than the ones of the $\lambda\sigma$-HOU. This is a consequence of the fact that decorations of substitution objects are more expensive than those of term objects.

7. REFERENCES

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

[2] M. Ayala-Rincón and F. Kamareddine. Higher Order Unification via λs-Style of Explicit Substitution. Technical report, Computer and Electrical Engineering, Heriot-Watt University, Edinburgh, December 1999. Available at `http://www.cee.hw.ac.uk/ultra`.

[3] M. Ayala-Rincón and F. Kamareddine. Strategies for Simply-Typed Higher Order Unification via λs_e-Style of Explicit Substitution. In R. Kennaway, editor, *Third International Workshop on Explicit Substitutions Theory and Applications to Programs and Proofs (WESTAPP 2000)*, Norwich, England, July 2000.

[4] F. Baader and T. Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[5] H. Barendregt. *The Lambda Calculus : Its Syntax and Semantics (revised edition)*. North Holland, 1984.

[6] Z.-e.-A. Benaissa, P. Lescanne, and K. H. Rose. Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution. In *Programming Languages: Implementations, Logics and Programs PLILP'96*, volume 1140 of *LNCS*, pages 393–407. Springer, 1996.

[7] P. Borovanský. Implementation of Higher-Order Unification Based on Calculus of Explicit Substitutions. In M. Bartošek, J. Staudek, and J. Wiedermann, editors, *Proceedings of the SOFSEM'95: Theory and Practice of Informatics*, volume 1012 of *Lecture Notes on Computer Science*, pages 363–368. Springer Verlag, 1995.

[8] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence Properties of Weak and Strong Calculi of Explicit Substitutions. *Journal of the ACM*, 43(2):362–397, 1996. Also as *Rapport de Recherche* INRIA 1617, 1992.

[9] G. Dowek, T. Hardin, and C. Kirchner. Higher-order Unification via Explicit Substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

[10] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via Explicit Substitutions: The Case of Higher-Order Patterns. In M. Maher, editor, *Proc. of the 1996 Joint International Conference and Symposium on Logic Programming*, Logic Programming, pages 259–273, Bonn, Germany, Sept. 1996. MIT press.

[11] W. Farmer. A Unification Algorithm for Second-Order Monadic Terms. *Annals of Pure and Applied Logic*, 39:131–174, 1988.

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

[13] G. P. Huet. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[14] F. Kamareddine and R. P. Nederpelt. A useful λ-notation. *Theoretical Computer Science*, 155:85–109, 1996.

[15] F. Kamareddine and A. Ríos. Extending a λ-calculus with Explicit Substitution which Preserves Strong Normalisation into a Confluent Calculus on Open Terms. *Journal of Functional Programming*, 7:395–420, 1997.

[16] F. Kamareddine and A. Ríos. Relating the $\lambda\sigma$- and λs-Styles of Explicit Substitutions. *Journal of Logic and Computation*, 10(3):399–431, 2000.

[17] C. Kirchner and C. Ringeissen. Higher-order Equational Unification via Explicit Substitutions. In *Proc. Algebraic and Logic Programming*, volume 1298 of *LNCS*, pages 61–75. Springer, 1997.

[18] L. Magnusson. *The implementation of ALF - a proof editor based on Martin Löf's Type Theory with explicit substitutions*. PhD thesis, Chalmers, 1995.

[19] C. Muñoz. Proof-Term Synthesis on Dependent-Type Systems via Explicit Substitution. Technical report, ICASE, Institute for Computer Applications in Science and Engineering, MS 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA, October 1999.

[20] G. Nadathur and D. S. Wilson. A representation of lambda terms suitable for operations on their intentions. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 341–348, 1990.

[21] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected papers on Automath*. North-Holland, Amsterdam, 1994.

[22] C. Okasaki. FUNCTIONAL PEARL Even Higher-Order Functions for Parsing or Why Would Anyone Ever Want to Use a Sixth-Order Function? *Journal of Functional Programming*, 8(2):195–199, March 1999.

[23] L. Paulson. Isabelle: The next 700 Theorem Provers. *Logic and Computer Science*, pages 361–386, 1990.

[24] C. Prehofer. Progress in Theoretical Computer Science. In R. V. Book, editor, *Solving Higher-Order Equations: From Logic to Programming*. Birkhäuser, 1997.

[25] A. Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites*. PhD thesis, Université de Paris 7, 1993.

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

[27] W. Snyder and J. Gallier. Higher-Order Unification Revisited: Complete Sets of Transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.