

# Higher-order unification via explicit substitutions

## Extended Abstract

Gilles Dowek

Thérèse Hardin

Claude Kirchner

INRIA-Rocquencourt  
B.P. 105  
78153 Le Chesnay Cedex  
France  
Gilles.Dowek@inria.fr

LITP & INRIA-Rocquencourt  
4 Place Jussieu  
75252 Paris Cedex 05  
France  
Therese.Hardin@litp.ibp.fr

INRIA Lorraine & CRIN  
BP 101  
54602 Villers-lès-Nancy Cedex  
France  
Claude.Kirchner@loria.fr

### Abstract

*Higher-order unification is equational unification for  $\beta\eta$ -conversion. But it is not first-order equational unification, as substitution has to avoid capture. In this paper higher-order unification is reduced to first-order equational unification in a suitable theory: the  $\lambda\sigma$ -calculus of explicit substitutions.*

### Introduction

Unification is the kernel of deduction processes used in theorem provers and programming languages. Syntactic unification initiated by Herbrand and developed by Robinson has been extended in two ways: equational unification and higher-order unification. The same idea is at the root of both extensions. It consists in incorporating in the unification process some knowledge about the underlying theory, in the higher-order case,  $\beta\eta$ -conversion [2] and in the case of a first-order theory, some equational axioms, such as associativity [12]. Nevertheless although the goal is the same, the unification processes designed so far are fairly different. In this paper, we show that higher-order unification can be reduced to first-order equational unification in a suitable theory.

Higher-order unification is equational unification for  $\beta\eta$ -conversion. But it is not first-order equational unification. This is due to some particularities of the substitution in  $\lambda$ -calculus. Let us first develop this point, in order to present the framework of the paper.

The well-known  $\lambda$ -calculus is defined as follows. Let  $\mathcal{V}$  be a set of variables, written  $x, y$ , etc. The terms of  $\Lambda(\mathcal{V})$ , the  $\lambda$ -calculus with names, are inductively defined by:

$$a ::= x \mid (a \ a) \mid \lambda x.a$$

Substitution cannot be defined in the following first-order way:

1.  $\{x \mapsto a\}y = a$  if  $x = y$  and  $y$  otherwise,
2.  $\{x \mapsto a\}(b \ c) = (\{x \mapsto a\}b \ \{x \mapsto a\}c)$ ,
3.  $\{x \mapsto a\}(\lambda y.b) = \lambda y.\{x \mapsto a\}b$ .

As well-known, this definition has two severe drawbacks. First  $\{x \mapsto a\}(\lambda x.x) = \lambda x.a$  although  $\{x \mapsto a\}(\lambda y.y) = \lambda y.y$ , so the computation is incorrect. Second  $\{x \mapsto y\}(\lambda y.x) = \lambda y.y$  although  $\{x \mapsto y\}(\lambda z.x) = \lambda z.y$ , so there is a capture.

Thus, during the replacement, bound variables renaming (also called  $\alpha$ -conversion) is needed to ensure the correctness of computations. The correct definition of substitution [4] is given by:

1.  $\{x/a\}y = a$  if  $x = y$  and  $y$  otherwise,
2.  $\{x/a\}(b \ c) = (\{x/a\}b \ \{x/a\}c)$ ,
3.  $\{x/a\}(\lambda y.b) = \lambda z.(\{x/a\}\{y \mapsto z\}a)$  where  $z$  is a fresh variable.

In the following, we keep the name *substitution* for the substitution of  $\lambda$ -calculus (denoted by  $\{x/a\}b$ ) and we call *grafting* the first-order substitution (denoted by  $\{x \mapsto a\}b$ ).

Because of these particularities of substitution in  $\lambda$ -calculus, the methods for equational unification (such as narrowing) built upon grafting, cannot be used for higher-order unification, which needs specific algorithms, extensively studied in [10, 13]. Both sorts of algorithms compute solutions incrementally. But in the first-order framework, to explore the graftings of a variable  $X$  being instantiated by a term whose head symbol is  $f$ , we perform the elementary grafting step:

$$X \mapsto (f \ Y_1 \ \dots \ Y_p),$$

where  $Y_1, \dots, Y_p$  are new variables.

In contrast, this elementary substitution step cannot be done in a higher-order framework by the substitution:

$$X / \lambda x_1 \dots \lambda x_n . (f Y_1 \dots Y_p).$$

Indeed, the arguments of  $f$  may depend on the variables  $x_j$  and thus  $Y_i$  should be substituted by a term containing some  $x_j$ . But the mechanism of substitution of  $\lambda$ -calculus forbids such captures. Therefore the information that the variables  $x_j$  can indeed occur in the arguments of  $f$  needs to be functionally handled leading to the classical elementary substitution:

$$X / \lambda x_1 \dots \lambda x_n . (f (Y_1 x_1 \dots x_n) \dots (Y_p x_1 \dots x_n)).$$

These functional handlings of scope would be avoided if grafting were used instead of substitution. In this way higher-order unification would be reduced to equational unification. But, replacing substitution by grafting raises two major problems.

First, the unification problems formulated in terms of grafting and substitution are not the same. For instance the problem  $\lambda x . Y =_{\beta\eta}^? \lambda x . x$ , has a solution  $Y \mapsto x$  with grafting but no solution with substitution. So, while substitution is too constrained and leads to over-complicated elementary substitutions as seen above, grafting is too liberal as it takes care of no constraint. Thus even if we succeed in formulating an algorithm for grafting-unification in  $\lambda$ -calculus, we could not deduce directly an algorithm for substitution-unification.

There is a second point. Equational unification algorithms such as narrowing use the fact that grafting and reduction commute. But grafting and  $\beta\eta$ -reduction do not commute in  $\lambda$ -calculus. For instance  $((\lambda x . Y) a)$  reduces to  $Y$ , although the term  $((\lambda x . x) a)$  obtained by grafting  $x$  to  $Y$  reduces to  $a$  and not to  $x$ . Thus reducing an equation would change the set of its solutions. This difficulty comes from the interaction between two different calculi (the  $\beta\eta$ -conversion and the instantiation by the unification process) during which the variables play a double game. We may consider that there are two kinds of variables: those which are only concerned by  $\beta\eta$ -conversion and those which are only concerned by the unification process.

Our goal in this paper is to reduce higher-order unification to first-order equational unification. To achieve this goal we need to set up a calculus where reduction and grafting commute. In such a calculus, if  $x$  is a reduction variable and  $Y$  a unification one, the

application of the substitution  $x/a$  to the term  $Y$ , during the reduction of  $((\lambda x . Y) a)$  for instance, must be delayed until  $Y$  is instantiated. This will make grafting and reduction commute. In other words, we need to describe at the object level how the application of a substitution initiated by reduction works. Such an internalization of substitutions was already required for describing implementations of  $\lambda$ -calculi and has motivated the development of  $\lambda\sigma$ -calculus [1, 5] which is a first-order equational theory.

In this framework, the variables used by  $\beta\eta$ -conversion are coded by de Bruijn indices. Unification variables can also be coded by de Bruijn indices, but here, we have chosen to code these unification variables by variables of the free first-order  $\lambda\sigma$ -algebra. This choice is required to reduce higher-order unification to first-order equational unification.

In the following we give a cursory overview of  $\lambda\sigma$ -calculus. Unification in this calculus can be performed by already known algorithms such as narrowing for weakly terminating and confluent rewrite systems [14]. But we present a specialized algorithm for a greater efficiency. At last we show how to relate unification in  $\lambda$ -calculus and in  $\lambda\sigma$ -calculus.

Thus we come up with a new higher-order unification algorithm, which respects the structure of  $\lambda$ -terms but eliminates some burdens of the previous algorithms, in particular the functional handling of scopes. Huet's classical algorithm [10, 13] can be seen as a strategy of our algorithm, each of its steps is decomposed in elementary ones, giving a more atomic description of the unification process. Moreover, solved forms of [10, 13] can easily be computed from our solved forms.

Such an attempt to reduce higher-order unification to equational one has been already done, for example in [7] with the combinators S, K, I, but, as usual with combinatory logic, it needs an explicit handling of extensionality and destroys the structure of the  $\lambda$ -terms.

This extended abstract presents results detailed in [9] to which the reader is referred for complete proofs and extensions. For the standard notions on  $\lambda$ -calculus, unification and equational logic, we refer the reader to the full paper and to [4, 11].

## 1 Explicit substitutions

The  $\lambda\sigma$ -calculus is a first-order rewriting system, introduced to provide an explicit treatment of substitutions initiated by  $\beta$ -reductions. Here, we shall use the  $\lambda\sigma$ -calculus described in [1], in its typed version [1, 6], but similar free calculi with explicit substitutions can be used in the same way provided they are confluent and weakly terminating on the free algebra

<b>Types</b>	$A ::= K \mid A \rightarrow B$
<b>Contexts</b>	$\Gamma ::= \text{nil} \mid A.\Gamma$
<b>Terms</b>	$a ::= 1 \mid X \mid (a \ b) \mid \lambda_A.a \mid a[s]$
<b>Substitutions</b>	$s ::= \text{id} \mid \uparrow \mid a : A \cdot s \mid s \circ t$

Figure 1: The syntax of  $\lambda\sigma$ -terms.

generated by term variables. As shown in the following definitions, this calculus contains the  $\lambda$ -calculus, written in de Bruijn notation, as a proper subsystem, the  $\beta$  and  $\eta$  reductions being simply particular strategies of application of its rules.

In this calculus, the term  $a[s]$  represents the term  $a$  on which the substitution  $s$  has to be applied. The simplest substitutions are list of terms build with the constructors “.” (cons) and “*id*” (nil). Applying the substitution  $(a_1 \cdots a_p \cdot \text{id})$  to a term  $a$  replaces the de Bruijn indices  $1, \dots, p$  in  $a$  by the terms  $a_1, \dots, a_p$  and decrements accordingly by  $p$  the remaining de Bruijn indices. The substitution “ $\uparrow$ ” increments de Bruijn indices when needed. At last the composition operation is needed in order to ensure confluence by transforming a sequence of substitutions application to a single simultaneous one. For example, the term  $(\lambda X \ Y)[s]$  reduces to  $X[Y.\text{id}][s]$  but also to  $X[1.(s \circ \uparrow)][Y[s].\text{id}]$ . With composition these two terms reduce to  $X[Y[s].s]$  allowing to recover confluence.

Using a set of atomic types that are denoted  $K$  and a set of variables denoted  $X$ , Figure 1 gives the syntax of  $\lambda\sigma$ -terms. Notice that we do not have substitution variable in the calculus we consider here.

Typing rules associate to each term  $a$  a context  $\Gamma$  and a type  $T$ . We call  $\Gamma \vdash T$  the sort of  $a$ , which is written  $a : \Gamma \vdash T$  or simply  $\Gamma \vdash a : T$ . To each variable  $X$  is associated a unique sort, i.e. a unique context  $\Gamma_X$  and a unique type  $T_X$ . The typing rules are described in Figure 2.

The reduction rules defining the semantics of this typed calculus are given in Figure 3. The full set of rules is called  $\lambda\sigma$  as the whole set but the rule **Beta** is called  $\sigma$ .

**Proposition 1.1**  $\lambda\sigma$ -calculus is confluent and weakly terminating.

## 2 A Unification Algorithm for $\lambda\sigma$

The  $\lambda\sigma$ -calculus is a very ordinary first-order theory, so we can use standard techniques, such as conditional narrowing for unification. However we shall

<i>(var)</i>	$A.\Gamma \vdash 1 : A$
<i>(lambda)</i>	$\frac{A.\Gamma \vdash b : B}{\Gamma \vdash \lambda_A b : A \rightarrow B}$
<i>(app)</i>	$\frac{\Gamma \vdash a : A \rightarrow B \quad \Gamma \vdash b : A}{\Gamma \vdash (a \ b) : B}$
<i>(clos)</i>	$\frac{\Gamma \vdash s \triangleright \Gamma' \quad \Gamma' \vdash a : A}{\Gamma \vdash a[s] : A}$
<i>(id)</i>	$\Gamma \vdash \text{id} \triangleright \Gamma$
<i>(shift)</i>	$A.\Gamma \vdash \uparrow \triangleright \Gamma$
<i>(cons)</i>	$\frac{\Gamma \vdash a : A \quad \Gamma \vdash s \triangleright \Gamma'}{\Gamma \vdash a : A \cdot s \triangleright A.\Gamma'}$
<i>(comp)</i>	$\frac{\Gamma \vdash s'' \triangleright \Gamma'' \quad \Gamma'' \vdash s' \triangleright \Gamma'}{\Gamma \vdash s' \circ s'' \triangleright \Gamma'}$
<i>(Metavar)</i>	$\Gamma_X \vdash X : T_X$

Figure 2: Typing rules for  $\lambda\sigma$ -terms.

<b>Beta</b>	$(\lambda_A.a)b \rightarrow a[b.\text{id}]$
<b>App</b>	$(a \ b)[s] \rightarrow (a[s] \ b[s])$
<b>VarCons</b>	$1[(a : A).s] \rightarrow a$
<b>Id</b>	$a[\text{id}] \rightarrow a$
<b>Abs</b>	$(\lambda_A.a)[s] \rightarrow \lambda_A.(a[1 : A.(s \circ \uparrow)])$
<b>Clos</b>	$(a[s])[t] \rightarrow a[s \circ t]$
<b>IdL</b>	$\text{id} \circ s \rightarrow s$
<b>ShiftCons</b>	$\uparrow \circ ((a : A).s) \rightarrow s$
<b>AssEnv</b>	$(s_1 \circ s_2) \circ s_3 \rightarrow s_1 \circ (s_2 \circ s_3)$
<b>MapEnv</b>	$((a : A).s) \circ t \rightarrow (a[t] : A).(s \circ t)$
<b>IdR</b>	$s \circ \text{id} \rightarrow s$
<b>VarShift</b>	$1.\uparrow \rightarrow \text{id}$
<b>Scons</b>	$1[s].(\uparrow \circ s) \rightarrow s$
<b>Eta</b>	$\lambda_A.(a \ 1) \rightarrow b$ if $a =_\sigma b[\uparrow]$

Figure 3: Reduction rules for  $\lambda\sigma$ .

design a much more efficient algorithm taking advantages of the rewriting system  $\lambda\sigma$ . Before giving a formal description of the algorithm we illustrate its principal features. We are using the following notations:  $a =_{\lambda\sigma}^? b$  denotes an equation to be solved in  $\lambda\sigma$ . The terms  $a$  and  $b$  are assumed to have the same sort. A conjunction of such equations is called a system. The set of term variables of a system  $P$  or of a term  $a$  is denoted  $\text{Var}(P)$  (resp.  $\text{Var}(a)$ ).

Since  $\lambda\sigma$  is a confluent and weakly terminating system, equations can be normalized. This is done by the rule **Normalize** presented in Figure 4. Then as a term  $\lambda a$  reduces only to terms of the form  $\lambda a'$  where  $a$  reduces to  $a'$ , an equation of the form  $\lambda a =_{\lambda\sigma}^? \lambda b$  can be simplified to  $a =_{\lambda\sigma}^? b$ . This is done by using the rule **Dec- $\lambda$** . In the same way, an equation  $(\mathbf{n} a_1 \dots a_p) =_{\lambda\sigma}^? (\mathbf{n} b_1 \dots b_p)$  can be simplified to  $a_1 =_{\lambda\sigma}^? b_1, \dots, a_p =_{\lambda\sigma}^? b_p$  by the rule **Dec-app1** and an equation of the form  $(\mathbf{n} a_1 \dots a_p) =_{\lambda\sigma}^? (\mathbf{m} b_1 \dots b_q)$  with  $n \neq m$  can be simplified to the unsatisfiable problem  $\mathbb{F}$  by the rule **Dec-app2**, as it has no solution.

Again, as  $\lambda\sigma$  is a confluent and weakly terminating system, we can restrict the search to normal  $\eta$ -long solutions that are grafting of the form  $\{X \mapsto \lambda a\}$ , when the type of  $X$  is functional and  $\{X \mapsto (\mathbf{r} a_1 \dots a_p)\}$  and  $\{X \mapsto (Z[s] a_1 \dots a_p)\}$  when the type of  $X$  is atomic.

When the type of a variable  $X$  is  $A \rightarrow B$ , a step towards the solution  $\{X \mapsto \lambda a\}$  is done by performing, using the rule **Exp- $\lambda$** , the grafting  $\{X \mapsto \lambda Y\}$  where  $Y$  is a new variable of type  $B$ . For instance, the problem  $(X \ 1) =_{\lambda\sigma}^? 1$  where  $X$  has type  $A \rightarrow A$  is transformed by the grafting  $\{X \mapsto \lambda Y\}$  into the problem  $((\lambda Y) \ 1) =_{\lambda\sigma}^? 1$  that reduces, with **Normalize**, to  $Y[1.id] =_{\lambda\sigma}^? 1$  where  $Y$  is a variable of type  $A$ . Note that the reduction substitution cannot be applied to  $Y$ .

Then, since  $Y$  has an atomic type, a normal solution of this last equation can only be a grafting of the form  $\{Y \mapsto (Z[s] a_1 \dots a_k)\}$  or  $\{Y \mapsto (\mathbf{r} a_1 \dots a_k)\}$ . A grafting of the form  $\{Y \mapsto (Z[s] a_1 \dots a_k)\}$  is obviously not a solution, as the normal form of the term  $(Z[s] a_1 \dots a_k)[1.id]$  cannot be  $1$ . Thus all the solutions are of the form  $\{Y \mapsto (\mathbf{r} a_1 \dots a_k)\}$ . A step towards such a solution is done by performing the grafting  $\{Y \mapsto (\mathbf{r} H_1 \dots H_k)\}$  where  $H_1, \dots, H_k$  are new variables. In this example  $\mathbf{r}$  can only be  $1$  or  $2$ , as otherwise the head variable of the normal form of  $(\mathbf{r} H_1 \dots H_k)[1.id]$  would be  $\mathbf{r} - 1$  and thus different from  $1$ .

More generally when we have an equation of the form  $X[a_1 \dots a_p \cdot \uparrow^n] =_{\lambda\sigma}^? (\mathbf{m} b_1 \dots b_q)$  where  $X$  has

an atomic type, the solutions can only be of the form  $\{X \mapsto (\mathbf{r} c_1 \dots c_k)\}$  where  $\mathbf{r} \in \{1, \dots, p\} \cup \{m - n + p\}$ . A step towards this solution is done by the rule **Exp-app**, instanciating  $X$  by  $(\mathbf{r} H_1 \dots H_k)$ .

As usual, when describing unification algorithms by transformation rules, performing a grafting  $\{X \mapsto a\}$  on a system  $P$  is implemented by first adding the equation  $X =_{\lambda\sigma}^? a$  to  $P$  and then using the rule **Replace** to propagate this constraint on the variable  $X$ . This permits to describe the solutions of unification problems as problems in solved forms and to let the unification rules be transformation rules preserving the solutions.

The only equations that are not treated by the rules above are of the form  $(X[a_1 \dots a_p \cdot \uparrow^n] =_{\lambda\sigma}^? Y[a'_1 \dots a'_p \cdot \uparrow^{n'}])$ . As in  $\lambda$ -calculus, such equations, called flexible-flexible, always have solutions.

**Definition 2.1** A system  $P$  is a  $\lambda\sigma$ -solved form if it is a conjunction of non trivial equations of the following shapes:

**Solved:**  $(X =_{\lambda\sigma}^? a)$  where the variable  $X$  does not appear anywhere else in  $P$  and  $a$  is in long normal form (which corresponds to the long normal form of  $\lambda$ -calculus). Such an equation is said *solved in  $P$* .

**Flex-flex:**  $(X[a_1 \dots a_p \cdot \uparrow^n] =_{\lambda\sigma}^? Y[a'_1 \dots a'_p \cdot \uparrow^{n'}])$ , where the two members of the equation are long normal forms, the type of the variables  $X$  and  $Y$  is atomic and the equation is not solved.

**Proposition 2.1** Any  $\lambda\sigma$ -solved form has  $\lambda\sigma$ -unifiers.

Now, the first-order unification process in  $\lambda\sigma$  can be expressed using the unification rules given in the Figure 4 in the style of [11].

**Theorem 2.1** Any fair strategy applying the rules in **Unif**, where any application of a rule **Exp-\*** is immediately followed by an application of **Replace** defines a correct and complete unification procedure for the  $\lambda\sigma$ -calculus. In other words, given a unification problem  $P$ :

- if the application of a finite number of the **Unif** rules leads to a disjunction of systems such that one of them is in solved form then the problem  $P$  is unifiable and a solution is given by a solution of the solved form,
- if  $P$  is a unifiable problem then **Unif** leads in a finite number of steps to a disjunction of systems such that one of them is in solved form,

<b>Dec-<math>\lambda</math></b>	$P \wedge \lambda_A a =_{\lambda\sigma}^? \lambda_A b$ $\rightarrow$ $P \wedge a =_{\lambda\sigma}^? b$
<b>Dec-app1</b>	$P \wedge (\mathbf{n} a_1 \dots a_p) =_{\lambda\sigma}^? (\mathbf{n} b_1 \dots b_p)$ $\rightarrow$ $P \wedge (\bigwedge_{i=1..p} a_i =_{\lambda\sigma}^? b_i)$
<b>Dec-app2</b>	$P \wedge (\mathbf{n} a_1 \dots a_p) =_{\lambda\sigma}^? (\mathbf{m} b_1 \dots b_q)$ $\rightarrow$ $\mathbb{F}$ $\text{if } n \neq m$
<b>Exp-<math>\lambda</math></b>	$P$ $\rightarrow$ $\exists Y : (A.\Gamma \vdash B), P \wedge X =_{\lambda\sigma}^? \lambda_A Y$ $\text{if } (X : \Gamma \vdash A \rightarrow B) \in \mathcal{V}ar(P), Y \notin \mathcal{V}ar(P),$ $\text{and } X \text{ is not a solved variable}$
<b>Exp-app</b>	$P \wedge X[a_1 \dots a_p \cdot \uparrow^n] =_{\lambda\sigma}^? (\mathbf{m} b_1 \dots b_q)$ $\rightarrow$ $P \wedge X[a_1 \dots a_p \cdot \uparrow^n] =_{\lambda\sigma}^? (\mathbf{m} b_1 \dots b_q)$ $\wedge \bigvee_{r \in R_p \cup R_i} \exists H_1, \dots, H_k, X =_{\lambda\sigma}^? (\mathbf{x} H_1 \dots H_k)$ $\text{if } X \text{ has an atomic type and is not solved}$ $\text{where } H_1, \dots, H_k \text{ are variables of appropriate types, not occurring in } P,$ $\text{with the contexts } \Gamma_{H_i} = \Gamma_X, R_p \text{ is the subset of } \{1, \dots, p\} \text{ such}$ $\text{that } (\mathbf{x} H_1 \dots H_k) \text{ has the right type,}$ $R_i = \text{if } m \geq n + 1 \text{ then } \{m - n + p\} \text{ else } \emptyset$
<b>Replace</b>	$P \wedge X =_{\lambda\sigma}^? a$ $\rightarrow$ $\{X \mapsto a\}(P) \wedge X =_{\lambda\sigma}^? a$ $\text{if } X \in \mathcal{V}ar(P), X \notin \mathcal{V}ar(a) \text{ and } (a \text{ is a variable } \Rightarrow a \in \mathcal{V}ar(P))$
<b>Normalize</b>	$P \wedge a =_{\lambda\sigma}^? b$ $\rightarrow$ $P \wedge a' =_{\lambda\sigma}^? b'$ $\text{if } a \text{ or } b \text{ is not in long normal form}$ $\text{where } a' \text{ (resp. } b') \text{ is the long normal form of } a \text{ (resp. } b) \text{ if } a \text{ (resp. } b) \text{ is}$ $\text{not a solved variable and } a \text{ (resp. } b) \text{ otherwise}$

Figure 4: **Unif**: Rules for  $\lambda\sigma$  unification.

- if **Unif** terminates, the problem obtained is a disjunction of systems in solved form, this disjunction is a description of a complete set of unifiers of  $P$ .

The main step of the completeness proof (presented in the full paper [9]) is to show that relatively to a given solution, the **Unif** rules are making an appropriate noetherian complexity measure decreasing.

We also show in the full paper that similar rules permit to perform unification for the  $\beta$ -rule only.

### 3 Unification in $\lambda$ -calculus

Having an algorithm for unification in  $\lambda\sigma$  does not provide directly an algorithm for unification in  $\lambda$ -calculus. First the former concerns grafting while the latter concerns substitution. Then as  $\lambda\sigma$  is a strict extension of  $\lambda$ -calculus in de Bruijn notation, a problem could have a solution in  $\lambda\sigma$  corresponding to no solution in  $\lambda$ -calculus. We show in this section that unification in  $\lambda$ -calculus can be reduced to unification in  $\lambda\sigma$ .

As already said, substituting  $t$  for  $X$  in some equation  $a =_{\beta\eta}^? b$  needs some  $\alpha$ -conversion. With de Bruijn notation  $\alpha$ -conversion is done by an adjustment of indices called *lifting*. Remark that substitution can be performed by first lifting  $t$ , then grafting it for  $X$  and that the structure of  $a$  and  $b$  determines solely how this lifting has to be done. As lifting operators are explicit in  $\lambda\sigma$ -calculus, we can *pre-cook* the terms  $a$  and  $b$  by stuffing them with the relevant lifting operators. It remains then only to apply graftings.

This pre-cooking of a  $\lambda$ -term  $a$ , written  $a_F$  is defined as  $a_F = F(a, 0)$  where:

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

**Proposition 3.1** *Pre-cooking is an homomorphism from  $\lambda$ -calculus to  $\lambda\sigma$ -calculus:*

1.  $a =_{\beta\eta} b$  if and only if  $a_F =_{\lambda\sigma} b_F$ ,
2.  $(\{X_1/b_1, \dots, X_p/b_p\}a)_F = \{X_1 \mapsto b_{1F}, \dots, X_p \mapsto b_{pF}\}a_F$ .

**Theorem 3.1** *Let  $a =_{\beta\eta}^? b$  be a unification problem in  $\Lambda_{DB}(\mathcal{X})$ . The equational problem  $a_F =_{\lambda\sigma}^? b_F$  has a solution if and only if the higher-order problem  $a =_{\beta\eta}^? b$  has a solution.*

The proof of the above direct implication is straightforward using Proposition 3.1.

Proving the converse is more subtle and needs in particular to show that from a solved form obtained from  $a_F =_{\lambda\sigma}^? b_F$  by the **Unif** system, one can compute a solution of this solved form which is in the image of  $F$ .

So we may find a solution to a problem  $a =_{\beta\eta}^? b$  in  $\lambda$ -calculus by pre-cooking it and finding a solution in  $\lambda\sigma$  to  $a_F =_{\lambda\sigma}^? b_F$ .

What remains to be described is how sets of solutions of the two problems correspond.

Suppose that  $a_F =_{\lambda\sigma}^? b_F$  is transformed using **Unif** into a problem  $P$  which is a disjunction of solved forms. These forms are not in general in the codomain of  $F$ : they are not pre-cooked terms. To describe the solutions of  $a =_{\beta\eta}^? b$ , we compute a problem  $Q$  equivalent to  $P$  such that  $Q$  is in the codomain of  $F$ , then we pre-cook back  $Q$  into a disjunction of solved forms in  $\lambda$ -calculus.

We need two rules to compute  $Q$  which are dual of some rules of **Unif**: they are described in Figure 5.

**Proposition 3.2** *The system **Unif**, augmented by the anti-rules described in Figure 5, remains sound and complete.*

**Theorem 3.2** *(Description of the solutions) Let  $a =_{\beta\eta}^? b$  be a unification problem in a context  $\Gamma$  in  $\lambda$ -calculus such that  $a_F =_{\lambda\sigma}^? b_F$  has a normal form  $P$  by the system **Unif**. Let  $Q$  be the system obtained by applying the **Anti-\*** and **Replace** rules to  $P$  to equations and variables whose contexts are strict extensions of  $\Gamma$ . Then,  $P$  and  $Q$  have the same solutions. Furthermore,  $Q$  is in the image of  $F$  and denoting  $R = F^{-1}(Q)$ , the system  $R$  is in solved form in the sense of  $\lambda$ -calculus [13] and its solutions are also solutions of  $a =_{\beta\eta}^? b$ . If we apply the trivial solution of [13] to solve the flexible-flexible equations of  $R$  we get the same solution as in Theorem 3.1. Moreover, any solution of  $a =_{\beta\eta}^? b$  can be expressed as the  $F^{-1}$  image of a solution of  $a_F =_{\lambda\sigma}^? b_F$ .*

In fact the rules **Anti-\*** can be applied at any stage of the transformations of **Unif**. This permits to simulate step by step Huet's algorithm [10, 13] which can be seen as a particular strategy for the system **Unif** extended by the two rules above. Indeed, by Proposition 3.1 applying an elementary substitution:

$$X/\lambda^n(\mathbf{r} (K_1 n \dots 1) \dots (K_p n \dots 1))$$

<p><b>Anti-Exp-<math>\lambda</math></b> <math>P</math> <math>\rightarrow</math> <math>\exists Y (P \wedge X =_{\lambda\sigma}^? (Y[\uparrow] 1))</math>  if <math>X \in \mathcal{V}ar(P)</math> such that <math>\Gamma_X = A.\Gamma'_X</math>  where <math>Y \in \mathcal{X}</math>, and <math>T_Y = A \rightarrow T_X, \Gamma_Y = \Gamma'_X</math></p> <p><b>Anti-Dec-<math>\lambda</math></b> <math>P \wedge a =_{\lambda\sigma}^? b</math> <math>\rightarrow</math> <math>P \wedge \lambda_A a =_{\lambda\sigma}^? \lambda_A b</math>  if <math>a =_{\lambda\sigma}^? b</math> is well-typed in a context <math>\Delta = A.\Delta'</math></p>
--

Figure 5: Anti Rules

in an equation  $a =_{\beta\eta}^? b$  consists in grafting the pre-cooking of this term in  $a_F =_{\lambda\sigma}^? b_F$ :

$$X \mapsto \lambda^n(\mathbf{r} (K_1[\uparrow^n] n \dots 1) \dots (K_p[\uparrow^n] n \dots 1))$$

and this grafting is obtained by applying **Exp- $\lambda$**   $n$  times:

$$X \mapsto \lambda^n Y$$

then **Exp-app** once:

$$X \mapsto \lambda^n(\mathbf{r} H_1 \dots H_p)$$

at last, applying **Anti-Exp- $\lambda$**   $n$  times to each new variable  $H_i$ :

$$X \mapsto \lambda^n(\mathbf{r} (K_1[\uparrow^n] n \dots 1) \dots (K_p[\uparrow^n] n \dots 1)).$$

In the same way, a simplification step of an equation consists in applying **Dec- $\lambda$**   $n$  times, **Dec-app** once and **Anti-Dec- $\lambda$**   $n$  times to each equation.

## 4 Examples

In order to illustrate the approach of higher order unification presented in this paper, let us solve the problem

$$\lambda y.(X a) =_{\beta\eta}^? \lambda y.a$$

with  $a : A, X : A \rightarrow A$ . This equation is encoded in de Bruijn terms, using the context  $\Gamma = A.nil$  into  $\lambda(X 2) =_{\beta\eta}^? \lambda 2$  and then pre-cooked into

$$\lambda(X[\uparrow] 2) =_{\lambda\sigma}^? \lambda 2$$

. With the rule **Dec- $\lambda$**  we get:

$$(X[\uparrow] 2) =_{\lambda\sigma}^? 2$$

applying the rule **Exp- $\lambda$**  yields:

$$\exists Y (((X[\uparrow] 2) =_{\lambda\sigma}^? 2) \wedge (X =_{\lambda\sigma}^? \lambda Y))$$

where  $\Gamma_Y = A.\Gamma$  and  $T_Y = A$ . With the rule **Replace** we get:

$$\exists Y (((\lambda Y)[\uparrow] 2) =_{\lambda\sigma}^? 2) \wedge (X =_{\lambda\sigma}^? \lambda Y)$$

which can be normalized by **Normalize**:

$$\exists Y ((Y[2. \uparrow] =_{\lambda\sigma}^? 2) \wedge (X =_{\lambda\sigma}^? \lambda Y))$$

the rule **Exp-app** can then be applied:

$$\begin{aligned} & \exists Y (((Y[2. \uparrow] =_{\lambda\sigma}^? 2) \wedge (X =_{\lambda\sigma}^? \lambda Y) \wedge (Y = 1))) \\ & \vee \\ & \exists Y (((Y[2. \uparrow] =_{\lambda\sigma}^? 2) \wedge (X =_{\lambda\sigma}^? \lambda Y) \wedge (Y = 2))) \end{aligned}$$

and the rule **Replace** yields:

$$\begin{aligned} & ((1[2. \uparrow] =_{\lambda\sigma}^? 2) \wedge (X =_{\lambda\sigma}^? \lambda 1)) \\ & \vee \\ & ((2[2. \uparrow] =_{\lambda\sigma}^? 2) \wedge (X =_{\lambda\sigma}^? \lambda 2)) \end{aligned}$$

which finally reduces using the rule **Normalize** into:

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

This problem is a disjunction of solved forms, the first gives the solution  $\lambda x.x$  and the second the solution  $\lambda x.a$ .

## A more elaborated example

let us solve now the ‘‘classical’’ equation:

$$(f (X a)) =_{\beta\eta}^? (X (f a))$$

with  $f : A \rightarrow A, X : A \rightarrow A, a : A$ . This equation is encoded in de Bruijn terms, using the context  $\Gamma = A \cdot A \rightarrow A \cdot nil$ , into  $(2 (X 1)) =_{\beta\eta}^? (X (2 1))$ . Then pre-cooking using the variable  $X$  with context  $\Gamma$  and type  $A \rightarrow A$ , yields  $(2 (X 1)) =_{\lambda\sigma}^? (X (2 1))$ .

Applying the rules in **Unif**, we get the following derivation:

we continue with the second system:

$$\begin{aligned}
& \{ (2 (X 1)) =_{\lambda\sigma}^? (X (2 1)) \\
\longrightarrow \mathbf{Exp-\lambda} & \\
& \left\{ \begin{array}{l} (2 (X 1)) =_{\lambda\sigma}^? (X (2 1)) \\ X =_{\lambda\sigma}^? \lambda Y \quad (\Gamma_Y = A \cdot \Gamma, T_Y = A) \end{array} \right. \\
\longrightarrow \mathbf{Replace} & \\
& \left\{ \begin{array}{l} (2 (\lambda Y 1)) =_{\lambda\sigma}^? (\lambda Y (2 1)) \\ X =_{\lambda\sigma}^? \lambda Y \end{array} \right. \\
\longrightarrow \mathbf{Normalize} & \\
& \left\{ \begin{array}{l} (2 Y[1.id]) =_{\lambda\sigma}^? Y[(2 1).id] \\ X =_{\lambda\sigma}^? \lambda Y \end{array} \right. \\
\longrightarrow \mathbf{Exp-app} & \\
& \left\{ \begin{array}{l} (2 Y[1.id]) =_{\lambda\sigma}^? Y[(2 1).id] \\ Y =_{\lambda\sigma}^? 1 \\ X =_{\lambda\sigma}^? \lambda Y \end{array} \right. \\
\vee \left\{ \begin{array}{l} (2 Y[1.id]) =_{\lambda\sigma}^? Y[(2 1).id] \\ Y =_{\lambda\sigma}^? (3 H_1) \quad (\Gamma_{H_1} = \Gamma_Y, T_{H_1} = A) \\ X =_{\lambda\sigma}^? \lambda Y \end{array} \right. & \\
\longrightarrow \mathbf{Replace} & \\
& \left\{ \begin{array}{l} (2 1[1.id]) =_{\lambda\sigma}^? 1[(2 1).id] \\ Y =_{\lambda\sigma}^? 1 \\ X =_{\lambda\sigma}^? \lambda 1 \end{array} \right. \\
\vee \left\{ \begin{array}{l} (2 (3 H_1)[1.id]) =_{\lambda\sigma}^? (3 H_1)[(2 1).id] \\ Y =_{\lambda\sigma}^? (3 H_1) \\ X =_{\lambda\sigma}^? \lambda Y \end{array} \right. & \\
\longrightarrow \mathbf{Normalize} & \\
& \left\{ \begin{array}{l} (2 1) =_{\lambda\sigma}^? (2 1) \\ Y =_{\lambda\sigma}^? 1 \\ X =_{\lambda\sigma}^? \lambda 1 \end{array} \right. \\
\vee \left\{ \begin{array}{l} (2 (2 H_1[1.id])) =_{\lambda\sigma}^? (2 H_1)[(2 1).id] \\ Y =_{\lambda\sigma}^? (3 H_1) \\ X =_{\lambda\sigma}^? \lambda (3 H_1) \end{array} \right. &
\end{aligned}$$

We get a first solved form:  $Y =_{\lambda\sigma}^? 1 \wedge X =_{\lambda\sigma}^? \lambda 1$  and

$$\begin{aligned}
& \longrightarrow \mathbf{Dec-app1} \\
& \left\{ \begin{array}{l} (2 H_1[1.id]) =_{\lambda\sigma}^? H_1[(2 1).id] \\ Y =_{\lambda\sigma}^? (3 H_1) \\ X =_{\lambda\sigma}^? \lambda (3 H_1) \end{array} \right. \\
\longrightarrow \mathbf{Exp-app} & \\
& \left\{ \begin{array}{l} (2 H_1[1.id]) =_{\lambda\sigma}^? H_1[(2 1).id] \\ H_1 =_{\lambda\sigma}^? 1 \\ Y =_{\lambda\sigma}^? (3 H_1) \\ X =_{\lambda\sigma}^? \lambda (3 H_1) \end{array} \right. \\
\vee \left\{ \begin{array}{l} (2 H_1[1.id]) =_{\lambda\sigma}^? H_1[(2 1).id] \\ H_1 =_{\lambda\sigma}^? (3 H_2) \quad (\Gamma_{H_2} = \Gamma_Y, T_{H_2} = A) \\ Y =_{\lambda\sigma}^? (3 H_1) \\ X =_{\lambda\sigma}^? \lambda (3 H_1) \end{array} \right. & \\
\longrightarrow \mathbf{Replace} & \\
& \left\{ \begin{array}{l} (2 1[1.id]) =_{\lambda\sigma}^? 1[(2 1).id] \\ H_1 =_{\lambda\sigma}^? 1 \\ Y =_{\lambda\sigma}^? (3 1) \\ X =_{\lambda\sigma}^? \lambda (3 1) \end{array} \right. \\
\vee \left\{ \begin{array}{l} (2 (3 H_2)[1.id]) =_{\lambda\sigma}^? (3 H_2)[(2 1).id] \\ H_1 =_{\lambda\sigma}^? (3 H_2) \\ Y =_{\lambda\sigma}^? (3 (3 H_2)) \\ X =_{\lambda\sigma}^? \lambda (3 (3 H_2)) \end{array} \right. & \\
\longrightarrow \mathbf{Normalize} & \\
& \left\{ \begin{array}{l} (2 1) =_{\lambda\sigma}^? (2 1) \\ H_1 =_{\lambda\sigma}^? 1 \\ Y =_{\lambda\sigma}^? (3 1) \\ X =_{\lambda\sigma}^? \lambda (3 1) \end{array} \right. \\
\vee \left\{ \begin{array}{l} (2 (2 H_2[1.id])) =_{\lambda\sigma}^? (1 H_2)[(2 1).id] \\ H_1 =_{\lambda\sigma}^? (3 H_2) \\ Y =_{\lambda\sigma}^? (3 (3 H_2)) \\ X =_{\lambda\sigma}^? \lambda (3 (3 H_2)) \end{array} \right. &
\end{aligned}$$

We get another solved form:  $H_1 =_{\lambda\sigma}^? 1 \wedge Y =_{\lambda\sigma}^? (3 1) \wedge X =_{\lambda\sigma}^? \lambda (3 1)$ , and a system that obviously will get rewritten by **Unif** forever, generating all the (infinitely many) solved forms of this system.

Now if we consider the two previous solved forms, they are both in the image of  $F$ , and pre-cooking them back to  $\lambda$ -terms we get for the first  $X =_{\beta\eta}^? \lambda x.x$  and for the second  $X =_{\beta\eta}^? \lambda x.(f x)$  that correspond clearly to two solutions of the initial problem.

## Conclusion

In this paper we have developed an algorithm for higher-order unification based on an algorithm for unification in the equational theory  $\lambda\sigma$ . Thus we have shown that higher-order unification problems could be

very simply expressed as first-order ones. The main point in this paper, is that the use of a language of explicit substitutions with meta-variables is really a benefit. It allows to separate substitutions initiated by reduction and substitutions of unification variables and to clarify their respective roles in higher-order unification. This separation permits to avoid the encoding of scoping constraints by  $\beta\eta$ -conversion, which was one of the burdens of previous algorithms. Moreover by using a language which is an extension of  $\lambda$ -calculus, our algorithm remains close to Huet's one, which can be seen as a particular strategy for ours.

We hope that this new framework, that allowed us to understand higher-order unification as first-order equational one, will be useful for some other purposes. In particular, mixing higher-order specifications with equational ones may be done just by extending  $\lambda\sigma$  with new symbols and new equations, this may be a way to reduce higher-order equational unification to first-order. Also, this framework might be a good one to study decidable subproblems of unification.

This new approach of higher-order unification has now to be implemented and tested in real size systems like higher-order prologs or interactive proof checkers. More generally it remains to be understood what explicit substitutions can bring to such systems. This work has to be carried with a precise analysis of the algorithm, in order to define strategies as lazy as possible. For example, there is no need to compute  $\lambda\sigma$ -normal forms at every step.

A major continuation of this work is its extension to unification in richer  $\lambda$ -calculi, such as the calculi of Barendregt's cube [3]. In this case, the functional expression of scoping constraints leads to technical difficulties [8] that may be simplified using explicit substitutions. At last, this work suggests that higher-order logic itself should be expressed with explicit substitutions. Then, higher-order resolution would be equational resolution in this theory.

### Acknowledgements

This work has been supported partly by the French Inter-PRC operation "Mécanisation du raisonnement". We acknowledge careful readings from Martin Abadi, Peter Borovansky, Daniel Briaud, César Muñoz and anonymous referees.

### 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] P. Andrews. Resolution in type theory. *Journal of Symbolic Logic*, 36:414–432, 1971.
- [3] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*. Clarendon Press, 1992.
- [4] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.
- [5] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 95. To appear. Also as 1992 INRIA report 1617.
- [6] P.-L. Curien and A. Rios. Un résultat de complétude pour les substitutions explicites. *Comptes rendus de l'Académie des Sciences de Paris*, 312(I):471–476, 1991.
- [7] D. J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114:273–298, 1993.
- [8] G. Dowek. A complete proof synthesis method for the cube of type systems. *Journal of Logic and Computation*, 3(3):287–315, 1993.
- [9] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. Research Report 94R243, CRIN, December 1994.
- [10] G. Huet. A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.
- [11] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.
- [12] G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.
- [13] W. Snyder and J. Gallier. Higher order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1 & 2):101–140, 1989. Special issue on unification. Part two.
- [14] A. Werner. Normalizing narrowing for weakly terminating and confluent systems. Technical report, Karlsruhe University, October 1994.