Information and Computation **157**, 183–235 (2000)
doi:10.1006/inco.1999.2837, available online at http://www.idealibrary.com on IDEAL®

Higher Order Unification via Explicit Substitutions

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

Thérèse Hardin

LIP6 and INRIA-Rocquencourt, Université Paris 6, 4 Place Jussieu, 75252 Paris Cedex 05, France
E-mail: Therese.Hardin@inria.fr

and

Claude Kirchner

LORIA and INRIA, 615, rue du Jardin Botanique, 54600 Villers-lès-Nancy Cedex, France
E-mail: Claude.Kirchner@loria.fr

Higher order unification is equational unification for $\beta\eta$-conversion. But it is not first order equational unification, as substitution has to avoid capture. Thus, the methods for equational unification (such as narrowing) built upon grafting (i.e., substitution without renaming) cannot be used for higher order unification, which needs specific algorithms. Our goal in this paper is to reduce higher order unification to first order equational unification in a suitable theory. This is achieved by replacing substitution by grafting, but this replacement is not straightforward as it raises two major problems. First, some unification problems have solutions with grafting but no solution with substitution. Then equational unification algorithms rest upon the fact that grafting and reduction commute. But grafting and $\beta\eta$-reduction do not commute in λ-calculus and reducing an equation may change the set of its solutions. This difficulty comes from the interaction between the substitutions initiated by $\beta\eta$-reduction and the ones initiated by the unification process. Two kinds of variables are involved: those of $\beta\eta$-conversion and those of unification. So, we need to set up a calculus which distinguishes between these two kinds of variables and such that reduction and grafting commute. For this purpose, the application of a substitution of a reduction variable to a unification one must be delayed until this variable is instantiated. Such a separation and delay are provided by a calculus of explicit substitutions. Unification in such a calculus can be performed by well-known algorithms such as narrowing, but we present a specialised algorithm for greater efficiency. At last we show how to relate unification in λ-calculus and in a calculus with explicit substitutions. Thus, we come up with a new higher order unification algorithm which eliminates some burdens of the previous algorithms, in particular the functional handling of scopes. Huet's algorithm can be seen as a specific strategy for our algorithm, since each of its steps can be decomposed into elementary ones, leading to a more atomic description of the unification process. Also, solved forms in λ-calculus can easily be computed from solved forms in $\lambda\sigma$-calculus.

Key Words: explicit substitutions; higher order unification; equational unification.

INTRODUCTION

In the original presentation of higher order logic, β and η-conversion were defined as axioms of the theory [9, 3]. The use of these axioms in a proof searching method leads to the replacement of a given proposition by any of its $\beta\eta$-equivalent forms, which is quite inefficient. This leads to a more suitable presentation of higher order logic, where $\beta\eta$-equivalent propositions are identified. Blind enumeration of equivalent propositions is then avoided, but instead of looking for syntactically equal instances of two terms, unification has to search for convertible instances of those terms. In other words, the conversion steps of the early proof searching methods are integrated within the unification process [2]: this is the so-called higher order unification or unification in simply typed λ-calculus [26].

A similar approach has been applied successfully to many theories of interest by removing equational axioms like associativity and commutativity from the theory and integrating them within the unification process, thus leading to the use of equational unification [43] in the deduction process.

Thus, higher order unification is merely equational unification for $\beta\eta$. But it is not first order equational unification: as λ is a binding operator, λ-calculus is not a first order algebra and substitution is not first order. Therefore, the classical first order methods for equational unification, like narrowing for example, cannot be used and higher order unification still needs specific algorithms. In this paper we show how higher order unification can be reduced to first order equational unification in a suitable equational theory.

The main difference between substitution of λ-calculus and first order substitution, here called *grafting*, is that the former performs renaming to avoid captures. This complicates higher order unification algorithms a lot. Indeed, consider a problem where a variable X occurs and θ a solution to this problem. The normal form of the term θX is

$$\theta X = \lambda x_1 \cdots \lambda x_n.(f\, a_1 \cdots a_p).$$

The symbol f is called the head symbol of this term and the terms $a_1, ..., a_p$ are its arguments. Of course, the bound variables $x_1, ..., x_n$ may occur in these arguments.

As for equational unification, we would like in a first step towards this substitution to choose the head symbol f and to generate new variables for its arguments leading to the elementary substitution

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

But this does not work in λ-calculus, as the substitution mechanism would forbid substituting, for example, x_1 for Y_1 while x_1 may occur in a_1. Thus, elementary substitutions in higher order unification have the form

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

The information that the variables x_j can indeed occur in the arguments of f needs to be functionally handled explicitly. Using grafting instead of substitution for unification will allow us to avoid this functional handling of scopes.

However using grafting instead of substitution for unification in λ-calculus raises two major problems. First, some scoping constraints are essential and they would need to be handled explicitly. For instance, in the unification problem $\lambda x.Y =^?_{\beta\eta} \lambda x.x$, a term substituted for Y cannot contain x. When the replacement is the substitution of λ-calculus, the bound variable x is renamed if the term substituted for Y contains the variable x and thus the constraint is automatically satisfied. If the replacement is grafting, then this constraint must explicitly be recorded, leading to a side calculus. Second, and more seriously, grafting and $\beta\eta$-reduction do not commute, while substitution and reduction do. 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 application of substitution by the unification algorithm. The difference is at the variable level rather than at the calculus level: the variables that can be meaningfully instantiated by unification are never the variables that can be instantiated by $\beta\eta$-reduction, i.e., the bound variables. Thus, we need to distinguish the two kinds of variables and to set up a calculus where reduction and unification grafting do not interfere. In such a calculus, the application of the substitution x/a to the term Y, during the reduction of $((\lambda x.Y)\ a)$, must be delayed until the unification variable Y is instantiated.

In other words, we need to describe at the object level how the application of a substitution initiated by reduction works. Such an internalisation of the substitution calculus was already required for describing implementations of λ-calculi and has motivated the development of $\lambda\sigma$-calculus [1, 11] which is a first order rewriting system. This calculus has been designed to describe $\beta\eta$-reduction step by step, the λ-terms being written in de Bruijn notation.

As we want to distinguish two kinds of variables in λ-calculus, we cannot directly use the embedding of the λ-calculus in de Bruijn notation into the $\lambda\sigma$-calculus. To extend it, we need to express how we translate unification variables. They can either be coded as de Bruijn indices together with the variables or as meta-indices, i.e., in a calculus with two kinds of de Bruijn indices or kept as named identifiers. As we want these unification variables to be the variables of a first order algebra, we rather keep names.

In $\lambda\sigma$-calculus grafting and reduction commute, so we can use grafting for unification, but we still need to handle the scoping constraints initially given in the problem. In fact, using the expressive power of $\lambda\sigma$-calculus, these scoping constraints can be internalised in the expression of the problem. We define a translation (that we call precooking) of λ-calculus into $\lambda\sigma$-calculus such that a problem $a =^?_{\beta n} b$ has a solution (substitution) in λ-calculus if and only if its translation $a_F =^?_{\lambda\sigma} b_F$ has a solution (grafting) in $\lambda\sigma$. The main idea of the translation is the following: if a variable X occurs in the problem $a =^?_{\beta\eta} b$, under some λ's, then substituting t for X in $a =^?_{\beta\eta} b$ needs some processing (called lifting) of t to avoid capture. Thus, substitution in λ-calculus contains two steps: first lifting then grafting. As lifting operators are explicit in $\lambda\sigma$-calculus, lifting will be incorporated in the unification problem $a_F =^?_{\lambda\sigma} b_F$ and thus searching for a substitution solution of the problem $a =^?_{\beta\eta} b$ is reduced to searching for a grafting solution of the problem $a_F =^?_{\lambda\sigma} b_F$.

As λ-calculus is a strict subset of $\lambda\sigma$-calculus, we need to show that a problem has solutions in λ-calculus if and only if its translation has solutions in $\lambda\sigma$. In this way we reduce higher order unification to first order equational unification in $\lambda\sigma$. This solution is completely different from a reformulation of Huet's algorithm [26] in $\lambda\sigma$, as such an approach would not reap the benefit of the first order framework, in particular the use of grafting.

To solve equational unification problems in $\lambda\sigma$, we can use a classical algorithm such as narrowing. In fact we will design a more efficient algorithm, for this particular theory. This algorithm can be understood as a kind of optimised narrowing.

At last we show that in contrast with the reduction of higher order unification to equational unification in a combinatory language [16], the unification algorithm for $\lambda\sigma$ can simulate Huet's algorithm, since every step of this algorithm can be simulated by a sequence of steps of our algorithm. We also show that this is not the most efficient way to use the unification algorithm of $\lambda\sigma$ as this simulation includes a lot of steps that functionally code and decode scoping constraints, which are not needed.

The structure of the paper is as follows: First we introduce the λ and $\lambda\sigma$ calculi and show in detail how the notions of substitution and grafting interact with $\beta\eta$-reduction. We then introduce, in Section 2, type information in the calculi and we make explicit a first order presentation of the typed $\lambda\sigma$-calculus. In Section 3 we define a translation called precooking from λ-calculus to $\lambda\sigma$-calculus that reduces substitution in λ-calculus to grafting in $\lambda\sigma$-calculus. After introducing the standard notions of equational unification and the appropriate notion of solved forms, Section 4 is devoted to the description of a rule-based unification procedure for the typed $\lambda\sigma$-calculus and to its proof of correctness and completeness. Section 5 then shows how the previous unification procedure can be applied to solve the problem of higher order unification thanks to precooking. The paper ends with detailed examples.

This paper is the full version, including full motivations and proofs, of the extended abstract which appeared in Ref. [19]. We have chosen to make this version quite detailed since it is important for the reader to have an explicit description of all the fundamental choices we have made in the design of the framework.

1. FROM λ-CALCULUS TO λσ-CALCULUS

In this section we motivate and introduce the calculi used in the paper: λ-calculus with names, λ-calculus with de Bruijn indices, and then $\lambda\sigma$-calculus. The presentation given here is slightly different from the classical ones, as we emphasise the role of unification variables and substitution.

In these calculi, the key notion is always the notion of substitution. However, in these formalisms the word substitution is heavily overloaded. First, we can substitute a bound variable of the calculus, as in substitutions initiated by β-reduction. Then we may substitute a unification variable of a term schema. Also, some substitutions avoid bound variable capture by renaming or lifting and others such as first order substitutions do not. So we are in particular insisting in this section on the differences between substitution and grafting.

1.1. First Order Algebras

First, in order to fix notations, we recall the classical definition of first order substitution which will be called *grafting* in this paper. Let $\mathcal{T}(\mathcal{F}, \mathcal{X})$ be the free algebra built on a set $\mathcal{X}$ of variables (denoted X, Y, ...) and a set $\mathcal{F}$ of operators. The set of variables of a term a is denoted $Var(a)$.

DEFINITION 1.1. A *valuation* is a function from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$. The $\mathcal{X}$-*grafting* (usually called *first order substitution*), extending a valuation θ and written $\bar{\theta}$, is the homomorphism defined by:

1. if X is a variable of $\mathcal{X}$ then $\bar{\theta}(X) = \theta(X)$,
2. if $a_1, ..., a_k \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $f \in \mathcal{F}$, then $\bar{\theta}(f(a_1, ..., a_k)) = f(\bar{\theta}(a_1), ..., \bar{\theta}(a_k))$.

The *domain* of a grafting θ is the set of variables that are not trivially mapped to themselves,

$$Dom(\theta) = \{X \mid X \in \mathcal{X} \text{ and } \theta(X) \neq X\},$$

and the set of variables introduced by θ is called its *range*, defined by:

$$Ran(\theta) = \bigcup_{X \in Dom(\theta)} Var(\theta(X)).$$

The set of all variables involved in θ is $Var(\theta) = Dom(\theta) \cup Ran(\theta)$.

Notation. Let θ be a valuation binding the variables $X_1, ..., X_n$ to the terms $a_1, ..., a_n$; we write the grafting associated to θ: $\bar{\theta} = \{X_1 \mapsto a_1, ..., X_n \mapsto a_n\}$. As usual, $\bar{\theta}$ is also written θ and $\mathcal{X}$-grafting is simply called *grafting* when there is no ambiguity.

EXAMPLE 1.1. Applying $\theta = \{X \mapsto f(X, Z)\}$ on the term $a = g(b, g(X, X))$ results in the term $\theta(a) = g(b, g(f(X, Z), f(X, Z)))$.

1.2. λ-Calculus with Names

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

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

First, we may try to consider the λ-calculus as a first order language, with a binary infix operator with no name that we denote when needed $\_$ for the application (formally $(a\, b)$ would be the infix notation for the term in prefix notation $\_(a, b)$) and an infinite number of unary operators called abstractors $\lambda x.\_$ indexed on $\mathscr{V}$.

A term $\lambda x.a$ is intended to represent a function of x, whose body is a. Its application to an actual parameter b is expected to return the value of a, where the formal parameter x is replaced by b. So, we need to define this replacement operation. In a first order setting, the natural notion of replacement is the one of *grafting* on the term algebra $\mathscr{T}\{\lambda x.\_,, \_\}, \mathscr{V})$. Using this notion, the replacement of formal arguments by actuals (the β-reduction) would be defined by:

$$(\lambda x.a)\, b \to \{x \mapsto b\}\, a.$$

But this definition of grafting has two severe drawbacks with respect to the intended semantics of the calculus. First if $\theta = \{x \mapsto a\}$ then $\theta(\lambda x.x) = \lambda x.a$ although $\theta(\lambda y.y) = \lambda y.y$, so the computation can be incorrect. Second, if $\theta = \{x \mapsto y\}$ then $\theta(\lambda y.x) = \lambda y.y$ although $\theta(\lambda z.x) = \lambda z.y$, so there can be a capture.

Thus, during the replacement, bound variables renaming (also called α-conversion) is needed to ensure the correctness of computations and we must forget about first order substitutions. So let us recall the classical definition of substitution in λ-calculus [4].

Definition 1.2. Let $a \in \Lambda(\mathscr{V})$. The set of *free variables* of a is written $FV(a)$. For a valuation θ binding the variables $x_1, ..., x_n$ to the terms $a_1, ..., a_n$, the *substitution* extending θ and written $\bar{\theta} = \{x_1/a_1, ..., x_n/a_n\}$ is defined by:

1. $\bar{\theta}x = \theta x$,
2. $\bar{\theta}(a\, b) = (\bar{\theta}a\; \bar{\theta}b)$,
3. $\bar{\theta}(\lambda y.a) = \lambda z.(\bar{\theta}\{y/z\}\, a)$ where z is a fresh variable, i.e., a variable such that $\theta z = z$, z does not occur in a and for every $x \in FV(a)$, $z \notin FV(\theta x)$.

Renaming bound variables (α-conversion) has to be done before performing the substitution under a λ. This is combined with the substitution of a fresh variable z for y in order to avoid capture. As the choice of this variable is not unique, this substitution is actually defined on classes of α-equivalent terms (see, for instance, [33]).

Definition 1.3. The β-reduction is the rewriting relation defined by the rule:

$$(\lambda x.a)\, b \to \{x/b\}\, a.$$

The η-reduction is the rewriting relation defined by the rule:

$$\lambda x.(a\, x) \to a \qquad \text{if} \quad x \notin FV(a).$$

1.3. Unification Variables

At first glance, a unification problem is given by two terms a and b, and a solution to such a problem is a substitution making a and b equal. In a given λ-term, we can distinguish several kinds of variables. First, bound variables are not concerned by unification substitutions, then free variables are separated in two classes: *constants* which cannot be substituted during unification and true *unification variables* which define the unification problem. We may keep the same syntactical category for bound variables and constants: both of them cannot be instantiated by unification. Their only difference is that the latter happen to be not bound by a λ and thus remain unchanged during the β-reduction process.

There are two possible choices for the status of unification variables: they may either be mere elements of $\mathscr{V}$ or be considered as metavariables, i.e., external to the β-process. If we take the first choice (see [26]), Definition 1.2 applies directly, but we need to keep at a metalevel a distinction between constants and true unification variables during the unification process.

Here we favour the second choice, i.e., we have two syntactical categories, one for bound variables and constants represented by $\mathscr{V}$ and another for unification variables, called metavariables and represented by $\mathscr{X}$. Although this choice is not the more economical, it eases the understanding of higher order unification as equational unification.

Thus λ-calculus is defined as an algebra defined on a set $\mathscr{X}$ of variables (our metavariables, written X, Y, ...) and a set of operators containing a set of constants $\mathscr{V}$ (the variables of the calculus of Section 1.2, written x, y, ...), a set of unary abstractors indexed on $\mathscr{V}$, and the application.

Definition 1.4. $\Lambda(\mathscr{V}, \mathscr{X})$, the set of open λ-terms, is inductively defined as

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

where $x \in \mathscr{V}$ and $X \in \mathscr{X}$.

We have now two notions of substitution. The first works on $\mathscr{V}$ and is needed for β-reduction. The second works on $\mathscr{X}$; it is used for unification.

The substitution of elements of $\mathscr{V}$ (or $\mathscr{V}$-substitution) is defined as above with the extra clause

$$\theta(X) = X \qquad \text{if} \quad X \in \mathscr{X}.$$

Let us turn now to the substitution of metavariables. Is it possible to reduce it to a $\mathscr{X}$-grafting? With the separation of name spaces, the first drawback of $\mathscr{X}$-grafting disappears: variables bound by a λ are elements of $\mathscr{V}$; thus they remain unchanged by an $\mathscr{X}$-substitution. But, the second is still present. If we substitute a variable by a term containing constants, some of them may be captured by abstractors as, for instance, in $\{X \mapsto x\}(\lambda x.X) = \lambda x.x$. So we need to introduce the notion of substitution with bound variables renaming.

DEFINITION 1.5. Let θ be a valuation (i.e., a function from $\mathscr{X}$ to $\Lambda(\mathscr{V}, \mathscr{X})$), the *substitution* written $\bar{\theta}$ is the extension of the valuation such that:

1. $\bar{\theta}(X) = \theta(X)$,
2. $\bar{\theta}(x) = x \quad$ if $\quad x \in \mathscr{V}$,
3. $\bar{\theta}(a\, b) = (\bar{\theta}(a)\; \bar{\theta}(b))$,
4. $\bar{\theta}(\lambda y.a) = \lambda z.\bar{\theta}(\{y/z\}\; a)$ where z is a fresh variable.

As usual $\bar{\theta}$ is also written θ.

Remark. Grafting and reduction do not commute. For instance, consider $a = ((\lambda x.X)\, y)$ and $\theta = \{X \mapsto x\}$. Then a β-reduces to X but $\theta(a) = ((\lambda x.x)\, y)$ does not reduce to $\theta(X) = x$. In the same way, the term $\lambda x.(X\, x)$ η-reduces to X but $\theta(\lambda x.(X\, x)) = \lambda x.(x\, x)$ does not reduce to $\theta(X) = x$.

PROPOSITION 1.6. *Substitution and reduction commute.*

Proof. For β-reduction we have $((\lambda x.a)\, b) \rightarrow^{\beta} \{x/b\}\; a$ and $\theta((\lambda x.a)\, b) = (\theta(\lambda x.a)\; \theta b) = (\lambda z.\theta(\{x/z\}\; a)\; \theta b) \rightarrow^{\beta} \{z/\theta(b)\}(\theta(\{x/z\}\; a)$. So we have to prove the equality $\theta(\{x/b\}\; a) = \{z/\theta(b)\}(\theta(\{x/z\}\; a)$, but this is an application of the *substitution lemma* [4] on commutation of substitutions.

For η-reduction we have $\lambda x.(a\, x) \rightarrow a$ and $\theta(\lambda x.(a\, x)) = \lambda z.\theta(\{x/z\}(a\, x)) = \lambda z.\theta(\{x/z\}\; a\, z)$ that reduces to θa as z has no occurrence in $\theta\{x \mapsto z\}\; a$. ∎

1.4. λ-Calculus in de Bruijn Notation

λ-calculus deals with actual names for bound variables. It is well known that these names are irrelevant both for computation and reasoning. Moreover, we need to rename these bound variables in order to ensure the correctness of the substitution. Thus substitution is not really defined on terms but on α-equivalence classes. This name management may be avoided with another formulation of λ-calculus: the de Bruijn notation [14]. We give here a presentation slightly different from the original one, as we add metavariables.

The intuitive idea of the de Bruijn notation is simple. To perform correctly the β-reduction, it suffices to find the occurrences of the formal parameter which has to be replaced by an actual one. In λ-calculus with names, this is indicated by the identity between the name of the variable at a given occurrence and the name of the λ's variable of the β-redex. It may also be indicated by the *binding height* of an occurrence, that is, by the number of λ's one has to cross between this occurrence and its binder.

DEFINITION 1.7. The set $\Lambda_{DB}(\mathscr{X})$ of λ-terms in de Bruijn's notation is defined inductively as

$$a ::= \mathtt{n} \mid X \mid \lambda\, a \mid (a\, a),$$

where $\mathtt{n}$ is an integer greater than or equal to $\mathtt{1}$ and $X \in \mathscr{X}$ (some authors use 0 as the first de Bruijn number, see [10] for example).

Note that we replace bound variables and constants by indices, but we keep the names for metavariables. This distinction is not done in the original de Bruijn calculus.

In the original notation of de Bruijn, free variables of a term a are ordered into a list $(x_0 \cdots x_n)$, called a *referential* and written $\mathscr{R}$. The cons operation (i.e., addition of a name in front of a list) is written as usual by ".". Then, a is coded as a subterm of $\lambda x_0 \cdots \lambda x_n.a$. Here, only $\mathscr{V}$-variables (i.e., bound variables and constants, but not unification variables) are recorded in the referential $\mathscr{R}$.

Definition 1.8. Let $\mathscr{R}$ be a referential. Let $a \in \Lambda(\mathscr{V}, \mathscr{X})$ such that all the free $\mathscr{V}$-variables of a are declared in $\mathscr{R}$. The de Bruijn translation of a, written $tr(a, \mathscr{R})$, is defined by:

1. $tr(x, \mathscr{R}) = \mathtt{j}$, where $\mathtt{j}$ is the place number of the first occurrence of x in the list $\mathscr{R}$,
2. $tr(X, \mathscr{R}) = X$,
3. $tr((a\,b), \mathscr{R}) = (tr(a, \mathscr{R})\; tr(b, \mathscr{R}))$,
4. $tr(\lambda\, x.a, \mathscr{R}) = \lambda\,(tr(a, x.\mathscr{R}))$.

So, during the de Bruijn translation, the referential is incremented when crossing a λ. For example, a closed λ-term (i.e., without free $\mathscr{V}$-variables) may be translated in de Bruijn notation, according to an empty referential. Note that this translation does not correspond to a bijection between $\mathscr{V}$ and $\mathbf{N}$: a bound variable may be represented by different numbers. For example, $\lambda x\, \lambda y.(x(\lambda z.(z\, x))\, y)$ is written $\lambda(\lambda(\mathtt{2}(\lambda\, \mathtt{1}\ \mathtt{3})\ \mathtt{1}))$.

Definition 1.9. The *λ-height* of an occurrence u in a term a is the number of λ's at prefix occurrences of u. It is written $|u|$.

Let u be an occurrence of a de Bruijn number $\mathtt{p}$ in a given term $tr(a, \mathscr{R})$. If $\mathtt{p} \leqslant |u|$, then $\mathtt{p}$ is said to be *bound* in $tr(a, \mathscr{R})$; otherwise it is a *free* index of a representing a constant encoded as a $\mathscr{V}$-variable at position $\mathtt{p} - |u|$ in the referential.

In this context, it is natural to define the β-reduction by

$$((\lambda\, a)\; b) \rightarrow \{\mathtt{1}/b\}\; a,$$

where $\{\mathtt{1}/b\}$ is the substitution of the index $\mathtt{1}$ by the term b. So, we need now to define this substitution which corresponds to the substitution of elements of $\mathscr{V}$.

Suppose that $((\lambda\, a)\, b)$ is expressed in a referential $\mathscr{R}$, then b is also expressed in $\mathscr{R}$ but a is expressed in $x.\mathscr{R}$, where x is a name for the variable bound by the λ. The term $a\{\mathtt{1}/b\}$ is expressed in $\mathscr{R}$ and is obtained from a by replacing the indices referring to x in $x.\mathscr{R}$ by b. An index $\mathtt{n}$ at an occurrence u in a refers to this x if and only if $\mathtt{n} = |u| + 1$. When we substitute such an index by b, this term b is placed under $|u|$ more λ's and must be correctly updated: indices in b referring to a variable of the referential $\mathscr{R}$ have still to refer to the same variable. Therefore free indices in b must be incremented by $|u|$. Also, as a is expressed in $x.\mathscr{R}$ and $a\{\mathtt{1}/b\}$ is expressed in $\mathscr{R}$, all the free indices of a which do not refer to the first variable of $x.\mathscr{R}$ must be decremented by 1. We first define the lifting operation that increments by 1 all the free indices of a term.

Definition 1.10. Let $a \in \Lambda_{DB}(\mathscr{X})$. The term a^+, called *lift* of a, is defined by $a^+ = \ell ft(a, 0)$ where $\ell ft(a, i)$ is inductively defined by:

1. $\ell ft((a_1\ a_2), i) = (\ell ft(a_1, i)\ \ell ft(a_2, i))$,
2. $\ell ft(\lambda\, a, i) = \lambda\, (\ell ft(a, i+1))$,
3. $\ell ft(X, i) = X$,
4. $\ell ft(\mathtt{m}, i) = \begin{cases} \mathtt{m+1} & \text{if } m > i, \\ \mathtt{m} & \text{if } m \leqslant i. \end{cases}$

Lemma 1.11. *Let a be a term of $\Lambda(\mathscr{V}, \mathscr{X})$ expressed in a referential $\mathscr{R}$. Let v be any variable of $\mathscr{V}$, not belonging to $\mathscr{R}$. Then, $tr(a, \mathscr{R})^+ = tr(a, v.\mathscr{R})$.*

Proof. It suffices to prove the following equality:

$$\ell ft(tr(a, x_1 \cdots x_i.\mathscr{R}), i) = tr(a, x_1 \cdots x_i.v \cdot \mathscr{R}).$$

This is done by structural induction on a, for all i and all $\mathscr{R}$. ∎

We now define the analogous of the $\mathscr{V}$-substitution of $\Lambda(\mathscr{V}, \mathscr{X})$. Here this substitution replaces indices.

Definition 1.12. The substitution by b at the λ-height $(n-1)$ in a, written $\{\mathtt{n}/b\}\ a$, is defined by induction as follows:

$$\begin{aligned} \{\mathtt{n}/b\}(a_1\ a_2) &= (\{\mathtt{n}/b\}\ a_1\ \{\mathtt{n}/b\}\ a_2) \\ \{\mathtt{n}/b\}\ X &= X \\ \{\mathtt{n}/b\}\ \lambda\, a &= \lambda\, (\{\mathtt{n+1}/b^+\}\ a) \end{aligned}$$

$$\{\mathtt{n}/b\}\ \mathtt{m} = \begin{cases} \mathtt{m-1} & \text{if } m > n & (\mathtt{m} \in FV(a)) \\ b & \text{if } m = n & (\mathtt{m} \text{ bound by the } \lambda \text{ of the Beta-redex}) \\ \mathtt{m} & \text{if } m < n & (\mathtt{m} \in BV(a)). \end{cases}$$

Definition 1.13. The β-reduction is defined by:

$$((\lambda\, a)\, b) \rightarrow \{\mathtt{1}/b\}\ a.$$

Example 1.2. The term $\lambda\, x.((\lambda\, y.(x\, y))\, x)$ is written $\lambda((\lambda(\mathtt{2}\ \mathtt{1}))\ \mathtt{1})$ in de Bruijn notation. It β-reduces to $\lambda\, x.(x\, x)$ when using explicit variable names or to $\lambda(\{\mathtt{1}/\mathtt{1}\}(\mathtt{2}\ \mathtt{1})) = \lambda(\mathtt{1}\ \mathtt{1})$ in de Bruijn notation.

Remark. To substitute a term b for an index $\mathtt{n}$ in a term of the form λa, we substitute the index $\mathtt{n+1}$ in a and we lift the term b. Usually, the lift of the actual parameter b is done, not after each traversal of an abstractor, but globally, when the formal parameter is reached. This choice is only a matter of taste.

Remark. The free indices are decremented by 1 by the substitution. Thus, the same operation takes into account the replacement of $\mathtt{1}$ by b and the decrement of all the free indices of a. Indeed $\{\mathtt{n}/b\}$ replaces $\mathtt{n}$ by b and decrements by 1 all the free indices that refer to an element of the referential greater than $\mathtt{n}$. So we do *not* define a general notion of simultaneous substitution $\{\mathtt{n}/b, \mathtt{m}/c\}$ as the decrementing effect of such a substitution would be more technical (see [40] and the full papers [39, 41] for a complete development of this idea). The simultaneous substitution which can be easily defined is the substitution of an initial segment of the natural numbers $\{\mathtt{1}/a_1, \mathtt{2}/a_2, ..., \mathtt{n}/a_n\}$. In this case all the other indices in the term have to be decremented by $\mathtt{n}$. Note that such a substitution is better represented as a list $a_1, ..., a_n$ than as a set of pairs.

We now turn to η-reduction. This relation is defined, in the classical setting, by the rule $\lambda x.(a\,x) \rightarrow^{\eta} a$, if x is not a free variable of a. Let $\mathscr{R}$ be a referential containing the constants of a. As a is coded by $tr(a, \mathscr{R})$ and $\lambda x.(a\,x)$ by $tr(\lambda x.(a\,x), \mathscr{R}) = \lambda(tr(a, x \cdot \mathscr{R})\;\mathtt{1}) = \lambda(tr(a, \mathscr{R})^{+}\;\mathtt{1})$, we get the following definition for η:

Definition 1.14. The η-reduction in $\Lambda_{DB}(\mathscr{X})$ is defined by the rule:

$$\lambda(a\;\mathtt{1}) \rightarrow b \qquad \text{if} \quad b \in \Lambda_{DB}(\mathscr{X}) \text{ is such that } a = b^{+}.$$

Proposition 1.15. *For a term a in $\Lambda_{DB}(\mathscr{X})$, there exists a term b such that $a = b^{+}$ if and only if, for any occurrence u of an index $\mathtt{p}$ in a, $\mathtt{p} \neq |u| + 1$.*

Proof. The if part comes from the definition of b^{+}: $\ell ft(\mathtt{m}, i)$ is never equal to $\mathtt{i+1}$. For the only if part, it suffices to build, from a, the term b by replacing any free number $\mathtt{n}$ of a by $\mathtt{n-1}$. ∎

At last we define the $\mathscr{X}$-substitution. As usual the notations θ and $\bar{\theta}$ are identified and the substitution θ^{+} is defined by $\theta^{+} = \{X_1/a_1^{+}, ..., X_n/a_n^{+}\}$ when $\theta = \{X_1/a_1, ..., X_n/a_n\}$.

Definition 1.16. Let θ be a valuation from $\mathscr{X}$ to $\Lambda_{DB}(\mathscr{X})$; the associated *substitution* $\bar{\theta}$ is defined by the rules:

1. $\bar{\theta}(X) = \theta(X)$,
2. $\bar{\theta}(\mathtt{n}) = \mathtt{n}$,
3. $\bar{\theta}(a_1\;a_2) = (\bar{\theta}(a_1)\;\bar{\theta}(a_2))$,
4. $\bar{\theta}(\lambda a) = \lambda(\bar{\theta}^{+}(a))$.

The relation between the substitution in λ-calculus with names and the substitution in λ-calculus with de Bruijn indices is expressed by the following proposition.

Proposition 1.17. *Let $a \in \Lambda(\mathscr{V}, \mathscr{X})$ and $\theta = \{X_1/a_1, ..., X_n/a_n\}$ be a term and a substitution of the λ-calculus with names. Let $\theta' = \{X_1/tr(a_1, \mathscr{R}), ..., X_n/tr(a_n, \mathscr{R})\}$, then $tr(\theta(a), \mathscr{R}) = \theta'(tr(a, \mathscr{R}))$.*

Proof. By induction on the structure of a. We only give the nontrivial case $a = \lambda v.b$, assuming that θ is $\{X/c\}$.

$$\begin{aligned} tr(\{X/c\}(\lambda\, v.b), \mathscr{R}) &= tr(\lambda\, w.\{X/c\}\{v/w\}\ b, \mathscr{R}) \\ &= \lambda\,(tr(\{X/c\}\{v/w\}b, w.\mathscr{R})) \\ \{X/tr(c, \mathscr{R})\}\ tr(\lambda\, v.b, \mathscr{R}) &= \{X/tr(c, \mathscr{R})\}(\lambda\,(tr(b, v.\mathscr{R}))) \\ &= \lambda\,(\{X/tr(c, \mathscr{R})^{+}\}(tr(b, v.\mathscr{R}))) \end{aligned}$$

We have $tr(b, v.\mathscr{R}) = tr(\{v/w\}\ b, w.\mathscr{R})$ and, as w is a fresh variable, $tr(c, \mathscr{R})^{+} = tr(c, w.\mathscr{R})$. So, we can apply the induction hypothesis, which allows the following conclusion:

$$\{X/tr(c, w.\mathscr{R})\}(tr(\{v/w\}b, w.\mathscr{R})) = tr(\{X/c\}(\{v/w\}b), w.R). \quad \blacksquare$$

1.5. λσ-Calculus

The $\lambda\sigma$-calculi [1, 11] are first order rewriting systems introduced to provide an explicit treatment of substitutions initiated by β-reductions. They contain the λ-calculus, written in de Bruijn notation, as a proper subsystem, the β and η reductions being simply the results of a particular strategy application of their rules. They differ by their treatment of substitution, which leads to slightly different confluence properties. Here, we shall use the $\lambda\sigma$-calculus described in [1] and we try to give an intuitive presentation of it in the following.

*Internalising operations as operator*s. Let us consider an algebra $\mathscr{A}$ whose domain is denoted A and a computable operation F from A to A. For instance, let us consider the set of natural numbers expressed with the symbols 0, S and the function F that associates to each natural number its double: $F(0) = 0$, $F(S(0)) = S(S(0))$, $F(S(S(0))) = S(S(S(S(0))))$, etc. A smooth way to define this operation is to extend the term language by adding an operator f internalising the operation F and rules rewriting expressions containing the symbol f into other expressions that eventually do not contain the symbol f. For instance,

$$\begin{gathered} f(0) \to 0 \\ f(S(X))) \to S(S(f(X))). \end{gathered}$$

Indeed any computable function can be expressed this way, even using a linear regular lonesome rule [13].

λ-calculus is such an internalisation of the application operation on functional expressions. Indeed, consider, for instance, the functional expressions $e_1 = \lambda x.x$ and $e_2 = 0$. The application operation is defined in such a way that e_1 applied to e_2 gives 0. Instead of defining the application operation directly one extends the expression language such that $((\lambda x.x)\ 0)$ is also an expression and set rewrite rules (β-reduction) such that:

$$(\lambda x.x)\ 0 \to 0.$$

But, λ-calculus internalises operations only half-way. Indeed application is internalised, but substitution and variable renaming (or lifting) are still external operations. $\lambda\sigma$-calculus goes one step further by internalising also substitutions and lifting.

A first try. When the operation $\{\mathtt{n}/b\}\ a$ is internalised as a term $a[\mathtt{n}/b]$ and the lifting operation a^+ as $(a\uparrow)$, one meets several difficulties. In order to express the rewrite rules for computing $(a\uparrow)$ one would need to introduce another operator internalising the operation ℓft. Then, the introduction of metavariables makes this system nonconfluent. Indeed,

$$((\lambda((\lambda X)\ Y))\ Z) \to ((\lambda X)\ Y)[\mathtt{1}/Z] \to X[\mathtt{1}/Y][\mathtt{1}/Z],$$

and

$$\begin{aligned}((\lambda((\lambda X)\ Y))\ Z) &\to ((\lambda X)\ Y)[\mathtt{1}/Z] \to (\lambda X[\mathtt{2}/Z\uparrow]\ Y[\mathtt{1}/Z])\\ &\to X[\mathtt{2}/Z\uparrow][\mathtt{1}/Y[\mathtt{1}/Z]].\end{aligned}$$

The term $\lambda((\lambda X)\ Y)\ Z$ reduces to $X[\mathtt{1}/Y][\mathtt{1}/Z]$ and to $X[\mathtt{2}/Z\uparrow][\mathtt{1}/Y[\mathtt{1}/Z]]$, which are both normal.

Simultaneous substitutions. In order to find a common normal form to these two terms, we introduce a notion of simultaneous substitution. We shall see that these two terms reduce to a common normal form $X[\mathtt{1}/Y[\mathtt{1}/Z], \mathtt{2}/Z]$. A simultaneous substitution is nicely represented as a list of terms. Lists are represented as usual with an operator *cons* (written ".") and an operator for the empty list (written *id* as it represents the identity substitution). The substitution $a_1 . a_2 \cdots a_n . id$ replaces $\mathtt{1}$ by a_1, ..., $\mathtt{n}$ by a_n *and decrements by n all the other (free) indices in the term.*

Then the operator $\uparrow$, which internalises the lifting operator a^+, can be seen as the infinite simultaneous substitution $\mathtt{2}.\mathtt{3}.\mathtt{4}.\ldots$. We also introduce a composition operator $\circ$. Finally, the index $\mathtt{n+1}$ is often written $\mathtt{1}[\uparrow \circ \cdots \circ \uparrow]$ or $\mathtt{1}[\uparrow^n]$. Note that $\uparrow^0$ is conventionally equal to *id* and thus $\mathtt{1}[\uparrow^0] = \mathtt{1}[id] = \mathtt{1}$. For more details see [1] and [11].

$\lambda\sigma$-calculus. Terms in this calculus are built as follows:

Definition 1.18. Let $\mathscr{X}$ be a set of term metavariables and $\mathscr{Y}$ be a set of substitution metavariables. The set $\mathscr{T}_{\lambda\sigma}(\mathscr{X}, \mathscr{Y})$ of *terms* and of *explicit substitutions* is inductively defined as

$$a ::= \mathtt{1} \mid X \mid (a\ a) \mid \lambda a \mid a[s]$$

$$s ::= Y \mid id \mid \uparrow \mid a.s \mid s \circ s$$

with $X \in \mathscr{X}$ and $Y \in \mathscr{Y}$.

One can also see $\mathscr{T}_{\lambda\sigma}(\mathscr{X}, \mathscr{Y})$ as a first order sorted algebra built on $\mathscr{X}$, the set of variables of sort `term`, $\mathscr{Y}$, the set of variables of sort `substitution`, and the operators described by:

$$\begin{array}{rll}
\mathtt{1}: & & \rightharpoonup \mathtt{term} \\
(\_\_): & \mathtt{term}\ \mathtt{term} & \rightharpoonup \mathtt{term} \\
(\lambda\,\_): & \mathtt{term} & \rightharpoonup \mathtt{term} \\
\_[\_]: & \mathtt{term}\ \mathtt{substitution} & \rightharpoonup \mathtt{term} \\
id: & & \rightharpoonup \mathtt{substitution} \\
\uparrow: & & \rightharpoonup \mathtt{substitution} \\
\_.\_: & \mathtt{term}\ \mathtt{substitution} & \rightharpoonup \mathtt{substitution} \\
\_\circ\_: & \mathtt{substitution}\ \mathtt{substitution} & \rightharpoonup \mathtt{substitution}.
\end{array}$$

Note that since the arrow is a very overloaded symbol, in particular in this paper (where it is used at least for rewriting and as a type constructor), we prefer to represent the rank information using the arrow $\rightharpoonup$. The set of variables of a term a of sort $\mathtt{term}$ is denoted by $\mathcal{TV}ar(a)$.

General Assumption. *Except if explicitly mentioned, we are always assuming that all the terms considered do not contain any substitution variable.*

Notation. The term $((((a\,a_1)\,a_2)\cdots)\,a_n)$ is written as usual $(a\,a_1\cdots a_n)$.

The $\lambda\sigma$-calculus is defined as the term rewriting system defined in Fig. 1. The rewrite rules in $\boldsymbol{\lambda\sigma}$ can be read as equational axioms and they define an equational theory whose congruence is denoted $=_{\lambda\sigma}$. If we drop the rules **Beta** and **Eta**, we get the rewriting system $\boldsymbol{\sigma}$, which performs the application of substitutions. The corresponding equational theory is written $=_\sigma$.

For η-reduction it would be natural to have the axiom

$$a = \lambda(a[\uparrow]\,\mathtt{1}).$$

Unfortunately if we orient this equational axiom as a rewrite rule $\lambda(a[\uparrow]\,\mathtt{1}) \to a$, an infinite set of critical pairs is generated (and provides a nice open problem to the

Beta	$(\lambda a)b$	$\to$	$a[b.id]$
App	$(a\ b)[s]$	$\to$	$(a[s]\ b[s])$
VarCons	$1[a.s]$	$\to$	a
Id	$a[id]$	$\to$	a
Abs	$(\lambda a)[s]$	$\to$	$\lambda(a[1.(s\circ\uparrow)])$
Clos	$(a[s])[t]$	$\to$	$a[s\circ t]$
IdL	$id\circ s$	$\to$	s
ShiftCons	$\uparrow\circ(a.s)$	$\to$	s
AssEnv	$(s_1\circ s_2)\circ s_3$	$\to$	$s_1\circ(s_2\circ s_3)$
MapEnv	$(a.s)\circ t$	$\to$	$a[t].(s\circ t)$
IdR	$s\circ id$	$\to$	s
VarShift	$1.\uparrow$	$\to$	id
Scons	$1[s].(\uparrow\circ s)$	$\to$	s
Eta	$\lambda(a\ 1)$	$\to$	b
			if $a =_\sigma b[\uparrow]$

FIG. 1. $\boldsymbol{\lambda\sigma}$—The $\lambda\sigma$-term rewriting system.

schematisation community [32, 8]), so we rather express it as the conditional rewrite rule **Eta**.

The main properties of **λσ** are:

1. The term rewriting system **λσ** is locally confluent on any $\lambda\sigma$-term, open or closed [1].

2. **λσ** is confluent on substitution-closed terms (i.e., on terms without substitution variable) [44].

3. **λσ** is *not* confluent on open terms (i.e., terms with term and substitution variables) [11].

The patterns of the normal forms of $\lambda\sigma$-terms of sorts `term` and `substitution` are given by the following results:

PROPOSITION 1.19 [44]. *Any $\lambda\sigma$-term in normal form for* **λσ** *is of one of the following forms*:

1. *$\lambda\, a$ where a is in normal form,*

2. *$(a\, b_1 \cdots b_p)$, where a and the b_i are in normal form and a is either* $\mathtt{1}$, $\mathtt{1}[\uparrow^n]$, *X, or $X[s]$ where s is a substitution term in normal form and different from id,*

3. *$a_1 \cdots a_p \cdot \uparrow^n$, where $a_1, ..., a_p$ are in normal form and $a_p \neq \mathtt{n}$.*

In λ-calculus with names (resp. with de Bruijn indices) we have a rule $X\{y/t\} = X$ where y is an element of $\mathscr{V}$ (resp. a de Bruijn index). This rule is needed because we have no way to suspend the substitution $\{y/t\}$ until X is instantiated. In $\lambda\sigma$-calculus we can delay the application of this substitution as the term $X[s]$ does not reduce to X. Note that, in particular, the condition $a =_\sigma b[\uparrow]$ is stronger than $a = b^+$ as $X = X^+$ in λ-calculus but there exists no term b such that $X =_\sigma b[\uparrow]$.

The application of a substitution to a metavariable can be suspended until the metavariable is instantiated. This will be used to code substitution of variables in $\mathscr{X}$ by $\mathscr{X}$-grafting and explicit lifting. Thus a notion of $\mathscr{X}$-substitution in $\lambda\sigma$-calculus is not needed.

By definition of rewriting, the $\lambda\sigma$-reduction relation is compatible with first order substitution, which is, as already said, called grafting here. So we get the following proposition.

PROPOSITION 1.20. *$\mathscr{X}$-grafting and $\lambda\sigma$-reduction commute.*

2. ADDING TYPING INFORMATION

As said in the Introduction, the right framework for performing unification is typed λ-calculus. So we are now introducing types for the calculi presented above.

2.1. Typed λ-Calculus with de Bruijn Indices

Contexts are used to record the types of free variables. In λ-calculus with de Bruijn indices, these contexts are lists of types. For instance, the context $A_1 . A_2 \cdots A_n . nil$ associates the type A_i to the index i.

DEFINITION 2.1. The syntax of simply typed λ-calculus using de Bruijn indices is

$$\begin{array}{ll} \textbf{Types} & A ::= K \mid A \to B \\ \textbf{Contexts} & \Gamma ::= nil \mid A.\Gamma \\ \textbf{Terms} & a ::= \mathtt{n} \mid X \mid (a\,b) \mid \lambda_A.a \end{array}$$

with $X \in \mathcal{X}$, and the typing rules are:

$$(var\ 1) \qquad A.\Gamma \vdash \mathtt{1} : A$$

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

$$(lambda) \qquad \frac{A.\Gamma \vdash b : B}{\Gamma \vdash \lambda_A.b : A \to B}$$

$$(app) \qquad \frac{\Gamma \vdash a : A \to B \qquad \Gamma \vdash b : A}{\Gamma \vdash (a\,b) : B}.$$

Notation. The length of a context Γ is written $|\Gamma|$.

In a calculus with metavariables, to each metavariable X we associate a unique type T_X. We assume that for each type T there is an infinite set of variables X such that $T_X = T$ and we add the typing rule

$$(Metavar) \quad \Gamma \vdash X : T_X,$$

where Γ is any context. This means in particular that the type of variable is imposed to be independent of the context where it appears.

Remark. Typing and grafting are not compatible. Indeed, consider the context,

$$\Gamma = A.(A \to A) \to (B \to A) \to A.nil,$$

and a variable X of type A, then we have:

$$\Gamma \vdash (\mathtt{2}\ \lambda_A.X\ \lambda_B.X) : A.$$

The variable X and $\mathtt{1}$ have the same type A in Γ, but when applying the grafting $\{X \mapsto \mathtt{1}\}$ we get the term $(\mathtt{2}\ \lambda_A.\mathtt{1}\ \lambda_B.\mathtt{1})$ which is not well typed.

PROPOSITION 2.2. *Typing and substitution are compatible; i.e., if X is a variable of type B and $\Gamma \vdash a : A$ and $\Gamma \vdash b$: B then $\Gamma \vdash \{X/b\}\ a : A$.*

Proof. By induction on the structure of a.

1. If $a = X$, then $A = B$.
2. If $a = \mathtt{n}$ then $\{X/b\}\ a = \mathtt{n}$.

3. If $a = (a_1\ a_2)$, we conclude with the induction hypothesis.

4. If $a = \lambda_C.a_1$ then $A = C \to D$ and $C.\Gamma \vdash a_1 : D$. As $\{X/b\}\ \lambda_C.a_1 = \lambda_C.(\{X/b^+\}a_1)$, in order to apply the induction hypothesis we need to know that $C.\Gamma \vdash X : B$, which is true by definition, and that $C.\Gamma \vdash b^+ : B$. In order to prove this last fact, i.e., that if $\Gamma \vdash b : B$, then $C.\Gamma \vdash b^+ : B$, we prove the more general result:

$$\text{If } C_i \cdots C_1.\Gamma \vdash b : B \qquad \text{then} \quad C_i \cdots C_1.C.\Gamma \vdash \ell ft(b, i) : B.$$

This is done by induction on the structure of b.

(a) If $b = X$, then $\ell ft(b, i) = X$.

(b) If $b = \mathtt{n}$. If $n > i$ then the type of $\mathtt{n}$ in $C_i \cdots C_1.\Gamma$ is the one of $\mathtt{n-i}$ in Γ. The type of $\ell ft(\mathtt{n}, i) = \mathtt{n+1}$ is the one of $\mathtt{n+1-(i+1)}$ in Γ. If $n \leqslant i$ then the type of $\mathtt{n}$ is C_{i-n+1} in both cases.

(c) If $b = (b_1\ b_2)$, we conclude with the induction hypothesis.

(d) $b = \lambda_D.c$ then $B = D \to E$ and $D.C_i \cdots C_1.\Gamma \vdash c : E$. By the induction hypothesis $D.C_i \cdots C_1.C.\Gamma \vdash \ell ft(c, i+1)$: E, thus $C_i \cdots C_1.C.\Gamma \vdash \ell ft(\lambda_D.c, i) : C$. ▌

2.2. *Typed λσ-Calculus*

In $\lambda\sigma$-calculus, we need to type not only terms, but also substitutions. As contexts are lists of types, they are used as types for substitutions which are merely lists of terms. We use the notation $s \rhd \Gamma$ to express the fact that the substitution s has type Γ.

Definition 2.3 [12, 1]. The syntax of typed $\lambda\sigma$-calculus is

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

with $X \in \mathscr{X}$. The typing rules are:

$$(var) \qquad A.\Gamma \vdash \mathtt{1} : A$$

$$(lambda) \qquad \frac{A.\Gamma \vdash b : B}{\Gamma \vdash \lambda_A b : A \to B}$$

$$(app) \qquad \frac{\Gamma \vdash a : A \to B \qquad \Gamma \vdash b : A}{\Gamma \vdash (a\ b) : B}$$

$$(clos) \qquad \frac{\Gamma \vdash s \rhd \Gamma' \qquad \Gamma' \vdash a : A}{\Gamma \vdash a[s] : A}$$

$$(id) \qquad \Gamma \vdash id \rhd \Gamma$$

$$(shift) \qquad A.\Gamma \vdash \uparrow \rhd \Gamma$$

$$(cons) \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash s \rhd \Gamma'}{\Gamma \vdash a.s \rhd A.\Gamma'}$$

$$(comp) \qquad \frac{\Gamma \vdash s'' \rhd \Gamma'' \qquad \Gamma'' \vdash s' \rhd \Gamma'}{\Gamma \vdash s' \circ s'' \rhd \Gamma'}.$$

Remark. Note that a and $a[s]$ have the same type, but in general are in different contexts. For instance, we have:

$$nat.nil \vdash \mathtt{1} : nat$$

and

$$nat.nil \vdash id \rhd nat.nil;$$

thus

$$nat.nil \vdash \mathtt{1}.\, id \rhd nat.nat.nil.$$

As we have also

$$nat.nat.nil \vdash \mathtt{2} : nat,$$

we deduce

$$nat.nil \vdash \mathtt{2}[\mathtt{1}.id] : nat.$$

The reduction rules of typed $\lambda\sigma$-calculus are simply defined by adding to the rules in **λσ** the relevant typing information. Below are the rules from **λσ** where type information has been added; the other rules are unchanged:

$$\textbf{Beta} \quad (\lambda_A a)\, b) \to a[b.id]$$

$$\textbf{Abs} \quad (\lambda_A a)[s] \to \lambda_A(a[1.(s \circ \uparrow)]).$$

Note that the typed version of σ has the same properties (termination and confluence) as the untyped one. One can also clearly type the **eta** rule. The resulting term rewriting system is also called **λσ**.

When we consider a calculus with metavariables, we want more than in λ-calculus, as we want that typing and *grafting* to be compatible. Thus, to each metavariable X we associate a *unique* type T_X and a *unique* context Γ_X. We assume that for each pair (Γ, T) there is an infinite set of variables X such that $\Gamma_X = \Gamma$ and $T_X = T$. We add to the previous typing rules the following one, in order to type metavariables:

$$(Metavar) \quad \Gamma_X \vdash X : T_X.$$

PROPOSITION 2.4. *Grafting and typing are compatible; i.e., if X is a variable and b be a term such that $\Gamma_X \vdash b : T_X$ then for every Δ, a, and A such that $\Delta \vdash a : A$ we have $\Delta \vdash \{X \mapsto b\}\ a : A$ and for every Δ, s, and Δ' such that $\Delta \vdash s \rhd \Delta'$ we have $\Delta \vdash \{X \mapsto b\}\ s \rhd \Delta'$.*

Proof. By a simultaneous induction on the structure of typing derivation of $\Delta \vdash a : A$ and $\Delta \vdash s \rhd \Delta'$. ▮

Associating to every metavariable a type and also a context is a strong requirement. For instance, neither the term $a = (Y\, X\, X[\uparrow])$ nor the term $b = (Y\, \lambda X\, X)$ can be typed in any context. Indeed if, for instance, a were typeable in a context Γ then both X and $X[\uparrow]$ should be typed in Γ. Thus X should be typed both in Γ and in $A.\Gamma$ which is impossible by the rule above. We exclude such terms on purpose: in $\lambda\sigma$-calculus we are more interested in grafting than in substitution. Grafting the index $\mathtt{2}$ to X in the first term would give the term $(Y\ \mathtt{2}\ \mathtt{3})$, and thus the two occurrences of X would refer to different variables of the context.

PROPOSITION 2.5. *The typed $\lambda\sigma$ calculus is:*

- *confluent* [44],
- *weakly normalising; see, for example,* [38] *or* [22].

Let us now present the notion of η-long normal form that is used in the rest of this work.

DEFINITION 2.6 (η-long normal form). Let a be a $\lambda\sigma$-term of type $A_1 \to \cdots \to A_n \to B$ in the context Γ and in $\lambda\sigma$-normal form. The η-long normal form of a, written a', is defined by:

1. If $a = \lambda_C b$ then $a' = \lambda_C b'$,
2. If $a = (\mathtt{k}\, b_1 \cdots b_p)$ then $a' = \lambda_{A_1} \cdots \lambda_{A_n}(\mathtt{k+n}\ c_1 \cdots c_p\ \mathtt{n}' \cdots \mathtt{1}')$, where c_i is the η-long normal form of the normal form of $b_i[\uparrow^n]$.
3. If $a = (X[s]\, b_1 \cdots b_p)$ then $a' = \lambda_{A_1} \cdots \lambda_{A_n}(X[s']\ c_1 \cdots c_p\ \mathtt{n}' \cdots \mathtt{1}')$, where c_i is the η-long normal form of the normal form of $b_i[\uparrow^n]$ and if $s = d_1 \cdots d_q . \uparrow^k$ then $s' = e_1 \cdots e_q . \uparrow^{k+n}$ where e_i is the η-long form of $d_i[\uparrow^n]$.

This definition is well founded as we now prove using an induction based on the lexicographic ordering on the triple consisting in the number of occurrences of metavariables the size of the term and the size of its type. The size of a normal term is defined by:

1. $|\lambda\, a| = 1 + |a|$,
2. $|(n\, a_1 \cdots a_p)| = 1 + |a_1| + \cdots + |a_p|$,
3. $|(X[s]\, a_1 \cdots a_n)| = 1 + |a_1| + \cdots + |a_n|$.

The size of a type is defined as usual by $|A \to B| = 1 + |A| + |B|$ where the size of an atomic type is 1.

LEMMA 2.7. *For any $\lambda\sigma$-normal term a we have $|(a[\mathtt{1} \cdots \mathtt{p}.\uparrow^n])| = |a|$.*

Proof. By induction on the structure of a. ▮

LEMMA 2.8. *Definition* 2.6 *is correct and well founded.*

Proof. In case 1, the number of occurrences of metavariables is kept and the size of the term is strictly decreasing. In case 2, when $p \neq 0$ the number of metavariables is decreasing and the size of the term is strictly decreasing, and when $p = 0$, the type is strictly decreasing. In case 3, the number of metavariables is strictly decreasing. ∎

DEFINITION 2.9. The *long normal form* of a term is the η-long form of its $\beta\eta$-normal form.

PROPOSITION 2.10. *Two terms are $\beta\eta$-equivalent if and only if they have the same long normal form.*

Remark. Note that in $\lambda\sigma$-calculus, the reduction of a η-redex may create a σ-redex. For instance, the term $X[\lambda(\mathtt{2}\ \mathtt{1}).\uparrow]$ reduces to $X[\mathtt{1}.\uparrow]$ then to $X[id]$ and then to X. Thus to compute the long normal form we need to reduce all the redexes (including the η ones) before expanding the term.

Note also that substitutions also could be written in extensional forms by using a kind of surjective pairing: *id* could be rewritten in $(\mathtt{1}.\uparrow)$ when type permits. But we do not make these expansions in this work.

2.3. A First Order Specification of Typed λσ-Calculus

In this paper our goal is to use the usual tools of first order equational unification, such as narrowing. Thus we need to show that the above term rewriting system can be expressed in a first order many-sorted fashion. This is the purpose of this section.

Let us first determine the sorts we are using. For a given term a of the calculus, since the type A of this term makes sense only in some context Γ, we consider this context as the first part of the sort of a term, the second part of which is simply the type itself. A sort for a term is thus a couple which is denoted $\Gamma \vdash A$ and a sort for a substitution is similarly a couple of two contexts denoted $\Gamma \vdash \Gamma'$. This brings us to the following formal definitions that are given in an ELAN [30]-like syntax.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textit{module SortLangage}\ [\textit{BaseType}]}\\
\multicolumn{3}{l}{\textit{sort Type Context TermSort SubstitutionSort};}\\
\textit{op} & & \\
- & : \textit{BaseType} & \rightharpoonup \textit{Type};\\
\rightarrow & : \textit{Type Type} & \rightharpoonup \textit{Type};\\
\textit{nil} & : & \rightharpoonup \textit{Context};\\
\_\cdot\_ & : \textit{Type Context} & \rightharpoonup \textit{Context};\\
\vdash & : \textit{Context Type} & \rightharpoonup \textit{TermSort};\\
\vdash & : \textit{Context Context} & \rightharpoonup \textit{SubstitutionSort};\\
\multicolumn{3}{l}{\textit{end of module}}
\end{array}
$$

We use the elements in `TermSort` and `SubstitutionSort` for sorting our $\lambda\sigma$-terms. Note that we get a refinement of the unsorted case in the sense that the two sorts `term` and `substitution` are now expanded in all the elements of `TermSort` and `SubstitutionSort`.

We introduce an infinity of new symbols that are the disambiguations of the operators $\mathtt{1}$, $\_\_$, λ, $\_[\_]$, id, $\uparrow$, $.$, $\circ$ by the appropriate type and context. We assign to each variable a sort $\Gamma \vdash A$. Note that this coding is consistent with the fact that, in $\lambda\sigma$-calculus, each metavariable is associated to a unique context and type. Then we give a rank to these symbols. These ranks are rephrasing of the typing rules of Definition 2.3, but here we simply use the usual rules for typing many-sorted terms with the assigned ranks.

$$\begin{array}{lll}
\mathtt{1}^{\Gamma}_{A} & : & \rightharpoonup A.\Gamma \vdash A \\
(\_\_)^{\Gamma}_{A\to B, A} & : \Gamma \vdash A\to B \quad \Gamma\vdash A & \rightharpoonup \Gamma \vdash B\\
(\lambda\_)^{\Gamma}_{A,B} & A.\Gamma\vdash B & \rightharpoonup \Gamma\vdash A\to B\\
\_[\_]^{\Gamma,\Gamma'}_{A} & : \Gamma'\vdash A \quad \Gamma\vdash\Gamma' & \rightharpoonup \Gamma\vdash A\\
id^{\Gamma} & : & \rightharpoonup \Gamma\vdash\Gamma\\
\uparrow^{\Gamma}_{A} & : & \rightharpoonup A.\Gamma\vdash\Gamma\\
\_\cdot^{\Gamma,\Gamma'}_{A}\_ & : \Gamma\vdash A \quad \Gamma\vdash\Gamma' & \rightharpoonup \Gamma\vdash A.\Gamma'\\
\_\circ^{\Gamma,\Gamma',\Gamma''}\_ & : \Gamma\vdash\Gamma'' \quad \Gamma''\vdash\Gamma' & \rightharpoonup \Gamma\vdash\Gamma'
\end{array}$$

A sort $\Gamma \vdash A$ is naturally interpreted as the set of all the ground terms that have type A in the context Γ. Note that there may be empty sorts; for instance, the sort $A.nil \vdash B$ is empty while the sort $A.nil \vdash A$ is not.

One can also formalise the same first order signature by using a polymorphic order-sorted signature as in [24], in which case the set of operators remains the same and becomes overloaded on the appropriate sorts.

EXAMPLE 2.1. Using these notations, the term $(\mathtt{1}^{\Gamma}_{A\to B} \;\; \mathtt{1}^{\Gamma}_{A})$ is not well formed since the first argument of the application is of sort $A\to B.\Gamma \vdash A \to B$ and the second one is of sort $A.\Gamma \vdash A$ which does not match the sort specification of the application. On the contrary one can check that the term $(\mathtt{1}^{A.\Gamma}_{A\to B} \;\; \mathtt{1}^{\Gamma}_{A}[\uparrow^{A.\Gamma}_{A\to B}])$ is well formed.

From this infinite set of symbols one can deduce the respective (infinite) set of rewrite rules expanding the $\boldsymbol{\lambda\sigma}$ rewrite rules.

PROPOSITION 2.11. *Let a be a well-typed $\lambda\sigma$-term in a context Γ. There is a unique well-sorted term a' that is a disambiguation of a.*

Proof. By induction on the term structure. ▮

PROPOSITION 2.12. *The sorted reduction system is weakly normalising and confluent.*

3. DEALING WITH FREE VARIABLES

Unifying two terms a and b in λ-calculus is finding a substitution θ such that $\theta(a) = \theta(b)$. The main difference between unification in λ-calculus and equational unification is that equational unification is a search for *graftings* and unification in λ-calculus is a search for *substitutions*. Thus, a unifier in λ-calculus of the problem

$$\lambda X =^?_{\beta\eta} \lambda 2$$

is not a term $t = \theta X$ such that $\lambda t = \lambda 2$, which would be a $\lambda\sigma$-unifier, but a term $t = \theta X$ such that

$$\lambda(t^+) = \lambda 2$$

as $(\lambda X)\{X/t\} = \lambda(t^+)$ and not λt. But, if t does not contain any metavariable, then $t^+ =_\sigma t[\uparrow]$. Therefore a closed unifier in λ-calculus is also a term such that $\lambda(t[\uparrow]) = \lambda 2$, i.e., a solution of the equational problem $\lambda(X[\uparrow]) =^?_{\beta\eta} \lambda 2$. The translation of the λ-term λX to the $\lambda\sigma$-term $\lambda(X[\uparrow])$ is called *precooking*. This translation will be used in Section 5 in order to reduce higher order unification to unification in $\lambda\sigma$.

3.1. *Precooking*

Since in $\lambda\sigma$-calculus metavariables have both a context and a type, when translating a λ-term a in $\lambda\sigma$ we need to assign a context to each metavariable X of a. We choose the one of a.

Definition 3.1. Let $a \in \Lambda_{DB}(\mathscr{X})$ such that $\Gamma \vdash a : T$. To every variable X of type U in the term a, we associate the type U and the context Γ in $\lambda\sigma$-calculus. The precooking of a from $\Lambda_{DB}(\mathscr{X})$ to $\mathscr{T}_{\lambda\sigma}(\mathscr{X})$ is defined by $a_F = F(a, 0)$ where $F(a, n)$ is defined by:

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

Proposition 3.2. *If $\Gamma \vdash a : T$ in $\Lambda_{DB}(\mathscr{X})$, then $\Gamma \vdash a_F : T$ in $\lambda\sigma$-calculus.*

Proof. We prove the following more general result.
If $A_1 \cdots A_n.\Gamma \vdash a : T$ in λ-calculus, if every variable of a is associated to the context Γ, then $A_1 \cdots A_n.\Gamma \vdash F(a, n) : T$.
This is done by induction on the structure of a, for all n.

1. $a = \lambda_B.b$; Then $T = B \to C$ and $B.A_1 \cdots A_n.\Gamma \vdash b : C$. So $B.A_1 \cdots A_n.\Gamma \vdash F(b, n+1) : C$ and $A_1 \cdots A_n.\Gamma \vdash F(\lambda\, b, n) = \lambda\,(F(b, n+1)) : B \to C$.

2. $a = X$; By definition, in λ-calculus, $\Gamma \vdash X : T$. By hypothesis, in $\lambda\sigma$-calculus, $\Gamma \vdash X : T$ and $A_1 \cdots A_n.\Gamma \vdash X[\uparrow^n] = F(X, n) : T$.

3. The cases $a = \mathtt{n}$ and $a = (a_1\ a_2)$ are easy.

The following proposition relates $\mathscr{X}$-substitution in $\Lambda_{DB}(\mathscr{X})$ and $\mathscr{X}$-grafting in $\lambda\sigma$-calculus and justifies the precooking.

PROPOSITION 3.3. *Let $a, b_1, ..., b_p$ be λ-terms in de Bruijn notation. Then*:

$$(a\{X_1/b_1, ..., X_p/b_p\})_F = \{X_1 \mapsto b_{1F}, ..., X_p \mapsto b_{pF}\}\ a_F.$$

Proof. We prove by induction on the structure of a the more general statement that for all i:

$$F(a\{X_1/b_1^{+^i}, ..., X_p/b_p^{+^i}\}, i) = \{X_1 \mapsto b_{1F}, ..., X_p \mapsto b_{pF}\}\ F(a, i).$$

The result follows for $i = 0$. The only difficult point is when a is one of the X_j, say X; in this case we need

$$F(b^{+^i}, i) = F(X, i)\{X \mapsto b_F\},$$

i.e.,

$$F(b^{+^i}, i) = b_F[\uparrow^i].$$

To prove this, we show by induction on the structure of b that for all i and k:

$$F(\ell f t^i(b, k), i+k) = F(b, k)[\mathtt{1} \cdots \mathtt{k}. \uparrow^{i+k}].$$

• If $b = \mathtt{m}$ and $m \leqslant k$ then $\ell f t^i(\mathtt{m}, k) = \mathtt{m}$; thus $F(\ell f t^i(b, k), i+k) = \mathtt{m}$ and $F(b, k)[\mathtt{1} \cdots \mathtt{k}. \uparrow^{i+k}] = \mathtt{m}$ also.

• If $b = \mathtt{m}$ and $m > k$ then $\ell f t^i(\mathtt{m}, k) = \mathtt{m+i}$; thus $F(\ell f t^i(b, k), i+k) = \mathtt{m+i}$ and $F(b, k)[\mathtt{1} \cdots \mathtt{k}. \uparrow^{i+k}] = \mathtt{m+i}$ also.

• If $b = X$, we get $X[\uparrow^{i+k}]$ for the two members.

• If $b = \lambda c$, then $\ell f t(b, k) = \lambda\, (\ell f t(c, k+1))$, so $\ell f t^i(b, k) = \lambda\, (\ell f t^i(c, k+1))$. Then, $F(\ell f t^i(b, k), i+k) = \lambda\, (F(\ell f t^i(c, k+1), i+k+1))$ and $F(b, k)[\mathtt{1} \cdots \mathtt{k}. \uparrow^{i+k}] = \lambda\, (F(c, k+1)[\mathtt{1} \cdots \mathtt{k+1}. \uparrow^{i+k+1}])$ and we conclude using the induction hypothesis.

• If $b = (c\ d)$. By induction.

PROPOSITION 3.4. *Let a be a λ-term.*

1. *If $a \xrightarrow{\beta} b$, then $a_F \xrightarrow{\lambda\sigma} b_F$.*
2. *If $a \xrightarrow{\eta} b$, then $a_F \xrightarrow{\eta} b_F$.*
3. *If a is $\beta\eta$-normal then a_F is $\lambda\sigma$-normal.*
4. *$a =_{\beta\eta} b$ if and only if $a_F =_{\lambda\sigma} b_F$.*

Proof. 1. We show that if $a \xrightarrow{\beta} b$, then for every n, $F(a, n) \xrightarrow{\lambda\sigma} F(b, n)$ and the result follows for $n = 0$. This is done by structural induction on a, the only non-trivial case is $a = (\lambda a_1)\, a_2$, where we need to show a substitution lemma:

$$F(a_1, n+1)[F(a_2, n).id] = F(a_1\{\mathtt{1}/a_2\}, n).$$

We show a more general lemma, for all n and p:

$$F(a_1, n+1+p)[\mathtt{1} \cdots \mathtt{p} \cdot (F(a_2, n).id) \circ \uparrow^p] = F(a_1\{\mathtt{p+1}/a_2^{+p}\}, n+p).$$

This is done again by structural induction on a_1, for all n and p. All cases are easy computations but the one $a_1 = \mathtt{p+1}$, which makes a_2 appear. We need to prove:

$$F(a_2, n)[\uparrow^p] = F(a_2^{+p}, n+p).$$

This is done by induction on p. As $F(a_2, n)[\uparrow^{p+1}] =_\sigma (F(a_2, n)[\uparrow^p])[\uparrow]$, we get by induction hypothesis, $F(a_2, n)[\uparrow^{p+1}] = F(a_2^{+p}, n+p)[\uparrow]$. We conclude using the following equality,

$$F(b, n)[\uparrow] = F(b^+, n+1),$$

which is derived from the more general fact:

$$F(b, n+r)[\mathtt{1} \cdots \mathtt{r}.\uparrow^{r+1}] = F(\ell f t(b, r), n+r+1).$$

This follows from an easy induction on b.

2. We have $(\lambda(b^+\ \mathtt{1}))_F = \lambda(F(b^+, 1)\ \mathtt{1})$; thus we need to show that $F(b^+, 1) = b_F[\uparrow]$. But we have proved, in the previous proposition, the following more general result

$$F(\ell f t^i(b, k), i+k) = F(b, k)[\mathtt{1} \cdots \mathtt{k}.\uparrow^{i+k}].$$

3. By induction on the structure of a.

4. If $a =_{\beta\eta} b$ then $a_F =_{\lambda\sigma} b_F$ by induction on the structure of the proof of the derivation $a =_{\beta\eta} b$. Conversely, if $a_F =_{\lambda\sigma} b_F$ then let a' be the normal form of a and b' be the normal form of b. The term a_F reduces to a'_F and b_F reduces to b'_F. Thus $a'_F =_{\lambda\sigma} b'_F$. As these terms are normal, we have $a'_F = b'_F$. As the precooking translation is injective we get $a' = b'$ and thus $a =_{\beta\eta} b$.

PROPOSITION 3.5. *Let a and b be two λ-terms. There exists terms $N_1, ..., N_p$ such that $a\{X_1/N_1, ..., X_p/N_p\} =_{\beta\eta} b\{X_1/N_1, ..., X_p/N_p\}$ if and only if there exists terms $M_1, ..., M_p$ in the image of the precooking translation* such that $\{X_1 \mapsto M_1, ..., X_p \mapsto M_p\}\ a_F =_{\lambda\sigma} \{X_1 \mapsto M_1, ..., X_p \mapsto M_p\}\ b_F$.

Proof. By Propositions 3.3 and 3.4

3.2. On the Use of Substitution Variables

Since in $\lambda\sigma$-calculus substitutions are explicit, another way to formulate unification problems could be to find an explicit substitution σ such that $a[\sigma] =_{\lambda\sigma} b[\sigma]$ which means to solve in the algebra $\mathcal{T}_{\lambda\sigma}(\mathcal{Y})$ the equation $a[Y] =^?_{\lambda\sigma} b[Y]$ modulo the equational theory $\lambda\sigma$. Surprisingly, such an approach leads also to the formalism presented in the previous section. Let us detail why.

The term grafted to Y should necessarily be of the form

$$Y \mapsto a_1 \cdots a_p \cdot \uparrow^n,$$

where p is the length of the type of Y, i.e., the number of free variables in a and b.

Thus we can now see this problem as finding terms $a_1, ..., a_p$ such that

$$a[a_1 \cdots a_p \cdot \uparrow^n] =_{\lambda\sigma} b[a_1 \cdots a_p \cdot \uparrow^n];$$

i.e., we are looking for a grafting θ of term variables, a $\lambda\sigma$-unifier of:

$$a[X_1 \cdots X_p \cdot \uparrow^n] =^?_{\lambda\sigma} b[X_1 \cdots X_p \cdot \uparrow^n].$$

Let us now concentrate on one side of the equation, for example the one concerning a. Let us consider a free number $\mathtt{i+k}$ occurring at λ-height k.

By $\lambda\sigma$-reduction of $a[X_1 \cdots X_p \cdot \uparrow^n]$ we get

$$\mathtt{i+k}[\mathtt{1}.(\mathtt{1}.(\cdots(X_1 \cdots X_p \cdot \uparrow^n) \circ \uparrow) \circ \uparrow)],$$

which $\lambda\sigma$-normalises to $X_i[\uparrow^k]$.

We thus exactly get the result of the precooking defined above. This means that it is equivalent to formalise the problem of unification in λ-calculus in the $\lambda\sigma$-calculus either by solving the precooked equations in $\mathscr{T}_{\lambda\sigma}(\mathscr{X})$ or by solving the equation $a[Y] =^?_{\lambda\sigma} b[Y]$. Since the first formulation is more suitable (in particular it will be anyway the result of the normalisation of the term when using substitution variables), we choose it in this work.

Note that the previous discussion permits the extension of a unification method when we have simultaneously term variables and substitution variables. Of course, this algorithm only makes sense if the typed $\lambda\sigma$-calculus with substitution variables is confluent. Recall that the untyped $\lambda\sigma$-calculus with substitution variables is not, but the problem is still open for the typed calculus.

4. UNIFICATION IN THE λσ-CALCULUS

Now that we have seen how to translate equations on λ-terms to equations on $\lambda\sigma$-terms, we address the problem of solving such equations on $\lambda\sigma$-terms modulo the equational theory **λσ**. The problems we focus on are substitution-ground; i.e., they may contain term variables but no substitution variables.

The unification problems we consider in this section are conditional equational first order unification problems; i.e., we use the fact that typed $\lambda\sigma$-calculus is a conditional rewriting system (conditional because of the η-rule) that is confluent and weakly normalising and a solution to a unification problem is a *grafting* that makes the two terms equal (compare to the usual presentation of unification in λ-calculus where a solution is a *substitution*).

Methods for conditional equational first order unification are well known. For instance we could use the first order many-sorted equational theory presented by the confluent set of rules **λσ** to perform (conditional) narrowing or better basic conditional narrowing [27, 28, 35, 47]. But since the system is only weakly normalising we would need to use techniques from [47] to show completeness. But this will be highly inefficient since almost all rewrite rules initiate a narrowing derivation on any $\lambda\sigma$-term. Thus we choose here to design a specific set of rules to perform $\lambda\sigma$-unification.

4.1. *Equational Unification*

Let us first recall briefly the unification notions we will need in the following. For details see [29].

DEFINITION 4.1. Let $\mathcal{F}$ be a set of function symbols, $\mathcal{X}$ be a set of variables, and $\mathcal{A}$ be an $\mathcal{F}$-algebra. A $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-*unification problem* (*unification problem* for short) is a first order formula without negation or universal quantifier whose atoms are **T**, **F** and $s =^?_{\mathcal{A}} t$, where s and t are terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We call an *equation* on $\mathcal{A}$ any $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification problem $s =^?_{\mathcal{A}} t$.

Equational problems will be written as a disjunction of existentially quantified conjunctions:

$$\bigvee_{j\in J} \exists \vec{w}_j \bigwedge_{i\in I_j} s_i =^?_{\mathcal{A}} t_i.$$

When $|J| = 1$ the problem is called a *system*. Variables $\vec{w}$ in a system written $P = \exists \vec{w} \bigwedge_{i\in I} s_i =^?_{\mathcal{A}} t_i$ are called *bound*, while the other variables are called *free*. Their respective sets are denoted by $\mathcal{BV}ar(P)$ and $\mathcal{V}ar(P)$.

DEFINITION 4.2. An $\mathcal{A}$-*unifier* of an $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification system,

$$P = \exists \vec{w} \bigwedge_{i\in I} s_i =^?_{\mathcal{A}} t_i,$$

is a grafting σ such that:

$$\mathcal{A} \models \exists \vec{w} \bigwedge_{i\in I} \sigma_{|\mathcal{X}-\vec{w}}(s_i) = \sigma_{|\mathcal{X}-\vec{w}}(t_i).$$

A $\mathcal{A}$-unifier of a $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification problem $D = \bigvee_{j\in J} P_j$, where all the P_j are $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification systems, is a grafting σ such that σ unifies at least one of the P_j.

We denote by $\mathcal{U}_{\mathcal{A}}(D)$ the set of unifiers of D. This is abbreviated $\mathcal{U}(D)$ when $\mathcal{A}$ is clear from the context. Similarly when clear from the context $\mathcal{A}$-unifiers are called unifiers and $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification is called unification.

When the $\mathcal{F}$-algebra considered is the quotient algebra of a set of terms by the congruence defined by a set of equational axioms E, i.e., $\mathcal{A} = \mathcal{T}(\mathcal{F}, \mathcal{X})/E$, then we denote $=^?_{\mathcal{A}}$ by $=^?_E$.

Definition 4.3. Given an equational problem P, $CSU_{\mathscr{A}}(P)$ is a *complete set of unifiers* of P for the algebra $\mathscr{A}$ if:

(i) $CSU_{\mathscr{A}}(P) \subseteq \mathscr{U}_{\mathscr{A}}(P)$, (correctness)

(ii) $\forall \theta \in \mathscr{U}_{\mathscr{A}}(P), \exists \sigma \in CSU_{\mathscr{A}}(P)$ such that $\sigma \leqslant_{\mathscr{A}}^{\mathscr{V}ar(P)} \theta$, (completeness)

(iii) $\forall \sigma \in CSU_{\mathscr{A}}(P), \mathscr{R}an(\sigma) \cap \mathscr{D}om(\sigma) = \emptyset$. (idempotency)

$CSU_{\mathscr{A}}(P)$ is called a *complete set of most general unifiers* of P in $\mathscr{A}$, and written $CSMGU_{\mathscr{A}}(P)$, if:

(iv) $\forall \alpha, \beta \in CSMGU_{\mathscr{A}}(P), \alpha \leqslant_{\mathscr{A}}^{\mathscr{V}ar(P)} \beta$ implies $\alpha = \beta$, (minimality)

or in other words: any two graftings of $CSMGU_{\mathscr{A}}(P)$ are not comparable for the quasi ordering $\leqslant_{\mathscr{A}}^{\mathscr{V}ar(P)}$.

In order to find a complete set of unifiers, we intend to simplify a given unification problem into simpler ones until one gets in an almost obvious way a description of the set of unifiers. This simplification is very conveniently described by transformation rules that are simply first order rewrite rule schemata. For example, considering a given signature $\mathscr{F} = \{f, g, a\}$, the transformation rule

$$\textbf{Decompose} \quad P \wedge f(s_1, ..., s_n) =^? f(t_1, ..., t_n)$$
$$\rightarrow \quad P \wedge s_1 =^? t_1 \wedge \cdots \wedge s_n =^? t_n$$

transforms the unification problem

$$x =^? a \wedge f(x, a) =^? f(g(y), z)$$

into

$$x =^? a \wedge x =^? g(y) \wedge a =^? z.$$

These rules are applied, for any unification problems P and Q, any equation e, and any term t modulo the usual Boolean simplification rules described in Fig. 2, where the connectors $\vee$ and $\wedge$ are assumed to be associative and commutative and where **F** denotes a unification problem without solution (the empty disjunction) and **T** a system always true (the empty conjunction). It is well known that this system of transformations of equational problems preserves the set of unifiers.

This view of solving unification problems has been initiated by [23, 34] and is fully developed in [29]. We can summarise it as follows. First choose the intended solved forms that characterise which equational problems actually correspond to most general unifiers or more generally to the desired simplified form of equational problems (think of the so-called flexible–flexible equations). Then determine the transformation rules: for each possible equational problem which is not in solved form, write transformation rules replacing this set with an equivalent one. Finally determine the appropriate control: this will determine which application of the transformation rules is intended in order to reach the solved forms. This approach corresponds in fact to express as a computational system the solving process [30].

Associativity- $\wedge$	$(P_1 \wedge P_2) \wedge P_3$	$=$	$P_1 \wedge (P_2 \wedge P_3)$
Associativity- $\vee$	$(P_1 \vee P_2) \vee P_3$	$=$	$P_1 \vee (P_2 \vee P_3)$
Commutativity- $\wedge$	$P_1 \wedge P_2$	$=$	$P_2 \wedge P_1$
Commutativity- $\vee$	$P_1 \vee P_2$	$=$	$P_2 \vee P_1$

Trivial	$P \wedge (s =^? s)$	$\to$	P
AndIdemp	$P \wedge (e \wedge e)$	$\to$	$P \wedge e$
OrIdemp	$P \vee (e \vee e)$	$\to$	$P \vee e$
SimplifAnd1	$P \wedge \mathbf{T}$	$\to$	P
SimplifAnd2	$P \wedge \mathbf{F}$	$\to$	$\mathbf{F}$
SimplifOr1	$P \vee \mathbf{T}$	$\to$	$\mathbf{T}$
SimplifOr2	$P \vee \mathbf{F}$	$\to$	P
Distrib	$P \wedge (Q \vee R)$	$\to$	$(P \wedge Q) \vee (P \wedge R)$
Propag	$\exists \vec{z} : (P \vee Q)$	$\to$	$(\exists \vec{z} : P) \vee (\exists \vec{z} : Q)$
Elimin0	$\exists z : P$	$\to$	P if $z \notin \mathcal{V}ar(P)$
Elimin1	$\exists z : z =^? t \wedge P$	$\to$	P if $z \notin \mathcal{V}ar(P) \cup \mathcal{V}ar(t)$

FIG. 2. **Simplif**, simplification rules for unification problems.

4.2. Transformation Rules for λσ-Unification

Let us first specialise the definitions of the previous section to $\lambda\sigma$-terms:

DEFINITION 4.4. 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 $\boldsymbol{\lambda\sigma}$. An *equation* of such a problem is denoted $a =^?_{\lambda\sigma} b$ where a and b are two substitution-closed $\lambda\sigma$-terms of *the same sort* in `TermSort`. An equation is called *trivial* when of the form $a =^?_{\lambda\sigma} a$. The set of variables of sort `term` in a unification problem P is denoted $\mathcal{T}\mathcal{V}ar(P)$. The set of all $\lambda\sigma$-unifiers of a problem P is denoted $\mathcal{U}_{\lambda\sigma}(P)$.

Remark. In a sorted language, we usually have a symbol $=$ and thus an equation symbol $=^?$ for every sort. Thus in our case to each equation is associated in a canonical way a context and a type.

Before giving a formal description of the algorithm we illustrate its principal features. Since $\lambda\sigma$ is a confluent and weakly terminating system, equations can be normalised. This is done by the rule **Normalise** presented in the set of transformation rules **Unif** given in Fig. 3. Then as a term λa 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-λ**. In the same way, an equation $(\mathtt{n}\, a_1 \cdots a_p) =^?_{\lambda\sigma} (\mathtt{n}\, b_1 \cdots b_p)$ can be simplified to $a_1 =^?_{\lambda\sigma} b_1, ..., a_p =^?_{\lambda\sigma} b_p$ by the rule **Dec-App** and an equation of the form $(\mathtt{n}\, a_1 \cdots a_p) =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \cdots b_q)$ with $n \neq m$ can be simplified to the unsatisfiable problem **F** by the rule **Dec-Fail**, as it has no solution.

As $\boldsymbol{\lambda\sigma}$ is a weakly terminating system, we can restrict the search to normal η-long solutions that are graftings of the form $\{X \mapsto \lambda a\}$ when the type of X is functional, and $\{X \mapsto (\mathtt{n}\, a_1 \cdots a_p)\}$ and $\{X \mapsto (Z[s]\, a_1 \cdots a_p)\}$ when the type of X is atomic.

Dec-λ $P \wedge \lambda_A a =^?_{\lambda\sigma} \lambda_A b$
$\to$
$P \wedge a =^?_{\lambda\sigma} b$

Dec-App $P \wedge (\mathtt{n}\ a_1\ \ldots\ a_p) =^?_{\lambda\sigma} (\mathtt{n}\ b_1\ \ldots\ b_p)$
$\to$
$P \wedge (\bigwedge_{i=1..p} a_i =^?_{\lambda\sigma} b_i)$

Dec-Fail $P \wedge (\mathtt{n}\ a_1\ \ldots\ a_p) =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q)$
$\to$
F
if $n \neq m$

Exp-λ P
$\to$
$\exists Y : (A.\Gamma \vdash B),\ \ P \wedge X =^?_{\lambda\sigma} \lambda_A Y$
if $(X : \Gamma \vdash A \to B) \in \mathcal{TVar}(P), Y \notin \mathcal{TVar}(P)$,
and X is not a solved variable

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

Replace $P \wedge X =^?_{\lambda\sigma} a$
$\to$
$\{X \mapsto a\}(P) \wedge X =^?_{\lambda\sigma} a$
if $X \in \mathcal{TVar}(P), X \notin \mathcal{TVar}(a)$ and $a \in \mathcal{X} \Rightarrow a \in \mathcal{TVar}(P)$

Normalise $P \wedge a =^?_{\lambda\sigma} b$
$\to$
$P \wedge a' =^?_{\lambda\sigma} b'$
if a or b is not in long normal form
where a' (resp. b') is the long normal form of a (resp. b) if a (resp. b) is not a solved variable and a (resp. b) otherwise

FIG. 3. **Unif**, the basic rules for unification in **λσ**.

When the type of a variable X is $A \to B$, a step towards the solution $\{X \mapsto \lambda a\}$ is done by performing, using the rule **Exp-λ**, the grafting $\{X \mapsto \lambda Y\}$ where Y is a new variable of type B. For instance, the problem $(X\,\mathtt{1}) =^?_{\lambda\sigma} \mathtt{1}$ where X has type $A \to A$ is transformed by the grafting $\{X \mapsto \lambda Y\}$ into the problem $((\lambda Y)\,\mathtt{1}) =^?_{\lambda\sigma} \mathtt{1}$ that reduces to $Y[\mathtt{1}.id] =^?_{\lambda\sigma} \mathtt{1}$ where Y is a variable of type A.

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

More generally when we have an equation of the form $X[a_1 \cdots a_p . \uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\ b_1 \cdots b_q)$ where X has an atomic type, the solutions can only be of the form $\{X \mapsto (\mathtt{r}\ c_1 \cdots c_k)\}$ where $\mathtt{r} \in \{1, ..., p\} \cup \{m-n+p\}$. A step towards this solution is done by the rule **Exp-app**, instantiating X by $(\mathtt{r}\ H_1 \cdots H_k)$.

As usual, when describing unification algorithms by atomic 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 a description of the solutions of unification problems as problems in solved forms and an allowance of the unification rules be transformation rules preserving the solutions.

The only equations that are not treated by the rules above are of the form:

$$X[a_1 \cdots a_p . \uparrow^n] =^?_{\lambda\sigma} Y[a'_1 \cdots a'_{p'} . \uparrow^{n'}].$$

As in λ-calculus, such equations, called *flexible–flexible*, always have solutions.

Let us now formally define the concepts that we have just informally introduced above.

Definition 4.5. A system P is a *$\lambda\sigma$-solved form* if all its metavariables are of atomic type and it is a conjunction of nontrivial equations of the following forms:

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

Flexible–flexible: $X[a_1 \cdots a_p . \uparrow^n] =^?_{\lambda\sigma} Y[a'_1 \cdots a'_{p'} . \uparrow^{n'}]$, where $X[a_1 \cdots a_p . \uparrow^n]$ and $Y[a'_1 \cdots a'_{p'} . \uparrow^{n'}]$ are long normal terms and the equation is not solved.

Note that in the definition above some of the n, n', p, p' may be zero. Examples of flexible–flexible equations are $X =^?_{\lambda\sigma} Y[X. \uparrow]$ (i.e., $X[id] =^?_{\lambda\sigma} Y[X. \uparrow])$ or $X[\mathtt{1}. \uparrow^2] =^?_{\lambda\sigma} Y[\uparrow^3]$ which are well sorted in the appropriate context. These last equations are solved forms but they contain unsolved variables.

Lemma 4.6. *Any $\lambda\sigma$-solved form has $\lambda\sigma$-unifiers.*

Proof. As solved equations define a part of the unifier we only need to solve the flexible-flexible equations. Consider such an equation

$$\dfrac{\dfrac{B_1 \cdots B_n . \Gamma \vdash a_i : A_i \qquad B_1 \cdots B_n . \Gamma \vdash \uparrow^n \rhd \Gamma}{B_1 \cdots B_n . \Gamma \vdash a_1 \cdots a_p . \uparrow^n \rhd A_1 \cdots A_p . \Gamma} \qquad A_1 \cdots A_p . \Gamma \vdash X : T}{B_1 \cdots B_n . \Gamma \vdash X[a_1 \cdots a_p . \uparrow^n] : T}.$$

Thus $\Gamma_X = A_1 \cdots A_p . \Gamma$ and $\Delta = B_1 \cdots B_n . \Gamma$ for some Γ. Thus $|\Gamma_X| = (|\Delta| - n) + p$. In the same way we get $|\Gamma_Y| = (|\Delta| - n') + p'$.

Now, for each atomic type T consider a variable Z_T of sort $nil \vdash T$ that does not occur in P. Let θ be the grafting that binds every nonsolved variable X of type T to the term $Z_T[\uparrow^{|\Gamma_X|}]$.

When we apply this grafting to the flexible–flexible equations we get:

$$Z_T[\uparrow^{|\Delta|}] = Z_T[\uparrow^{|\Delta|}].$$

And thus these equations are satisfied by this grafting; therefore all the equations of θP are solved.

So, let ϕ be the grafting binding every solved variable Y occurring in an equation $Y =^?_{\lambda\sigma} t$ of θP to the term t. The grafting $\phi \circ \theta$ is a $\lambda\sigma$-unifier to P. ▌

As mentioned before, in the previous definition the sort of Z is empty. We will see in Section 5 that one can choose more interesting solutions when the problem comes from the λ-calculus.

The transformation rules needed for unification in the typed $\lambda\sigma$-unification are described in Fig. 3. Additional unification rules given in Fig. 4 could be added to the previous ones in order to improve efficiency and to avoid systematic replacement.

Remember that all these unification rules are applied together with the simplification rules described in Fig. 2 which are applied eagerly. In order to show that the **Unif** rules are correct and complete with respect to $\lambda\sigma$-unification, we shall now, following the lines of [29], prove the following results:

- the **Unif** normal forms of equation systems are solved forms,
- the application of each rule in **Unif** preserves the set of $\lambda\sigma$-unifiers,
- for any $\lambda\sigma$-unifier γ of a system S, there exists a solved form representing γ.

The following lemmas formally present these different steps and their technical requisites.

First, let us show that the **Unif** transformations do not introduce ill-typed terms:

LEMMA 4.7. *The transformation by the rules in* **Unif** *of a well-typed equation gives rise only to well-typed equations* **T** *and* **F**.

Proof. This follows from a rule by rule analysis of the transformation. ▌

LEMMA 4.8. *Any solved problem is normalised with respect to* **Unif**. *Conversely, if a system P is a conjunction of equations irreducible by the rules of* **Unif**, *then it is solved.*

Proof. It is clear that any solved form is in normal form for **Unif**. Conversely, let P be a nonsolved system; let us show that it is reducible by **Unif**. Since P is not a solved form, it contains an equation $a =^?_{\lambda\sigma} a'$ that is neither solved nor flexible–flexible.

The first possibility is to have an equation of the form $X =^?_{\lambda\sigma} a$, where X appears somewhere else in P, in which case **Replace** applies.

If a or a' are not in long normal form, then we apply the rule **Normalise**.

The other cases where a and a' are long normal terms—which allows us to infer that if a is not a λ-abstraction, then it has an atomic type—are summarised in the following table, using the result of Proposition 1.19 to describe the form of the terms involved and assuming that $\vec{b}$ and $\vec{b}'$ both consist in at least one element. Note that the table is symmetric.

$a =^?_{\lambda\sigma} a'$	$\lambda\, b$	$(\mathtt{n}\, \vec{b})$	$(X[s]\vec{b})$ or $(X\, \vec{b})$	$X[s]$	X
$\lambda\, b'$	**Dec-λ**	Ill-typed	Ill-typed	Ill-typed	Ill-typed
$(\mathtt{n}'\, \vec{b}')$		**Dec-App** or **Dec-Fail**	**Exp-λ**	**Exp-App**	**Replace** or **Exp-App**
$(X'[s']\, \vec{b}')$ or $(X'\, \vec{b}')$			**Exp-λ**	**Exp-λ**	**Replace** or **Exp-λ**
$X'[s']$				Flexible–flexible	Flexible–flexible or **Replace**
X'					**Replace**

∎

LEMMA 4.9. *Any rule* **r** *in* **λσ-Unif** *is correct, i.e.:*

$$P \overset{\mathbf{r}}{\mapsto\!\!\!\twoheadrightarrow} P' \Rightarrow \mathcal{U}_{\lambda\sigma}(P') \subseteq \mathcal{U}_{\lambda\sigma}(P).$$

Proof. By easy inspection of the rules. ∎

LEMMA 4.10. *Any rule* **r** *in* **λσ-Unif** *is complete, i.e.:*

$$P \overset{\mathbf{r}}{\mapsto\!\!\!\twoheadrightarrow} P' \Rightarrow \mathcal{U}_{\lambda\sigma}(P) \subseteq \mathcal{U}_{\lambda\sigma}(P').$$

Proof. Let us check it for all the rules:

Normalise: clear.

Dec-λ: Assume that θ is a $\lambda\sigma$-unifier of $(\lambda\, a =^?_{\lambda\sigma} \lambda b)$; then $\lambda\theta(a) =_{\lambda\sigma} \lambda\theta(b)$ and since no rule from **λσ** could be applied on top of these terms, we necessarily have $\theta(a) =_{\lambda\sigma} \theta(b)$.

Dec-App: Let θ be a $\lambda\sigma$-unifier of

$$(\mathtt{n}\, a_1 \cdots a_p) =^?_{\lambda\sigma} (\mathtt{n}\, b_1 \cdots b_p);$$

then since the system is weakly terminating and confluent on substitution-closed $\lambda\sigma$-terms,

$$\begin{aligned} \theta(\mathtt{n}\, a_1 \cdots a_p) =_{\lambda\sigma_\tau} \theta(\mathtt{n}\, b_1 \cdots b_p) \quad &\Leftrightarrow \\ (\mathtt{n}\, \theta(a_1) \cdots \theta(a_p)) =_{\lambda\sigma_\tau} (\mathtt{n}\, \theta(b_1) \cdots \theta(b_p)) \quad &\Leftrightarrow \\ \forall 1 \leqslant i \leqslant p,\ \theta(a_i) =_{\lambda\sigma_\tau} \theta(b_i), \end{aligned}$$

which means that θ is $\lambda\sigma$-unifier of:

$$P \wedge \left(\bigwedge_{i=1 \cdots p} a_i =^?_{\lambda\sigma} b_i \right).$$

Dec-Fail: The last proof schema shows that if $n \neq m$ there is no θ which is a $\lambda\sigma$-unifier of:

$$P \wedge ((\mathtt{n}\, a_1 \cdots a_p) =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \cdots b_q)).$$

Exp-λ: If θ is a $\lambda\sigma$-unifier of P and if $X\colon \Gamma \vdash A \to B \in \mathcal{TV}ar(P)$, then $\theta(X) = a : A \to B$ and we can assume that a is of the form $\lambda_A a'$ with $a' : B$. Let us define θ' such that $\forall X \in \mathcal{D}om(\theta)$, $\theta'(X) = \theta(X)$, and $\theta'(Y) = a'$. Then θ' is a $\lambda\sigma$-unifier of $P \wedge X =^?_{\lambda\sigma} \lambda_A Y$, which shows that θ is a $\lambda\sigma$-unifier of $\exists(Y : A.\Gamma \vdash B)$, $P \wedge X =^?_{\lambda\sigma} \lambda_A Y$.

Exp-App: Since by hypothesis $X[a_1 \cdots a_p . \uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \cdots b_q)$ is unifiable by θ, the unifier should be such that $\theta(X) = (\mathtt{r}\, c_1 \cdots c_s)$. Thus it verifies:

$$\begin{aligned}
&(\mathtt{r}\, c_1 \cdots c_s)[a'_1 \cdots a'_p . \uparrow^n] &&=^?_{\lambda\sigma} (\mathtt{m}\, b'_1 \cdots b'_q) \Leftrightarrow \\
&(\mathtt{r}[a'_1 \cdots a'_p . \uparrow^n] \cdots) &&=^?_{\lambda\sigma} (\mathtt{m}\, b'_1 \cdots b'_q) \Leftrightarrow \\
&(\mathtt{1}[\uparrow^{r-1} \circ (a'_1 \cdots a'_p . \uparrow^n)] \cdots) &&=^?_{\lambda\sigma} (\mathtt{m}\, b'_1 \cdots b'_q).
\end{aligned}$$

So either $r \leqslant p$ in which case the equation becomes

$$\begin{aligned}
&(\mathtt{1}[a'_r \cdots a'_p . \uparrow^n] \cdots) &&=^?_{\lambda\sigma} (\mathtt{m}\, b'_1 \cdots b'_q) \Leftrightarrow \\
&(a'_r\, c_1[a'_1 \cdots a'_p . \uparrow^n] \cdots) &&=^?_{\lambda\sigma} (\mathtt{m}\, b'_1 \cdots b'_q)
\end{aligned}$$

(in this case θ is clearly solution of $\exists H_1, ..., H_k,\ X =^?_{\lambda\sigma} (\mathtt{r}\, H_1 \cdots H_k)$), or $r > p$ in which case the equation becomes

$$(\mathtt{1}[\uparrow^{n+(r-p)}] \cdots) =^?_{\lambda\sigma} (\mathtt{m}\, b'_1 \cdots b'_q),$$

and it has a solution if and only if $n + r - p = m$, thus if $r = m - n + p$, at the condition that $r > p \Leftrightarrow m - n + p > p \Leftrightarrow m \geqslant n + 1$, which gives the condition asserted in rule **Exp-App**.

So in all the cases, the grafting θ is the solution of:

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

$$\wedge \bigvee_{r \in R_p \cup R_i} \exists H_1, ..., H_k,\ X =^?_{\lambda\sigma} (\mathtt{r}\, H_1 \cdots H_k).$$

Note that the condition on m and n in fact allows us to cut the search space by guessing the bindings that will not a priori fail. When $r \leqslant p$ then this corresponds to the projection transformation in the higher order unification algorithm. When $r \geqslant p$ this corresponds to the imitation transformation.

Replace: The proof works like in the first order case. ∎

4.3. A Complete Strategy for $\lambda\sigma$-Unification

It is clear that some strategies in applying the rules of **Unif** are not terminating. A typical case is when the problem has no solution, it may not terminate. Another example is that **Exp-λ** can be applied infinitely many times on a system if no replacement is done. We are thus proving the completeness of a particular class of strategies which are built on any *fair* application of the following rules or group of rules:

Normalise or **Dec-λ** or **Dec-App** or **Dec-Fail** or **Replace** or

Exp-λR = (**Exp-λ**; **Replace**) or

Exp-AppR = (**Exp-App**; **Replace**).

We call this class of strategies **UnifReplace** since it applies replacement eagerly after a **Exp-*** rule. This principle can be modified, in particular using the merging rule and an appropriate handling of the variables, in order to get a Martelli–Montanari-like unification algorithm for $\lambda\sigma$.

These rules are assumed to be applied in a *fair* way on the problem to be solved, which means that in one disjunction of the systems, none of the constitutive systems is left forever without applying transformation rules on it.

In order to present the completeness proof in a more comprehensible way, we divide a problem P in two parts Q and R containing respectively the nonsolved and solved equations of P. The idea of the proof is to show that all the above elementary groups of rules decrease a complexity measure based on the grafting θ that we want to mimic using the unification algorithm. For a solved system R consisting only of solved equations, we denote by $Graft(R)$ the canonical grafting associated to R. For example if $R=(X=^?_{\lambda\sigma} a)$ then $Graft(R)=\{X\mapsto a\}$.

In all this section, we assume θ to be a $\lambda\sigma$-normalised grafting solution of the unification problem P.

DEFINITION 4.11. For a system of equations $P=(Q,R)$ and a $\lambda\sigma$-normalised grafting θ solution of P, we define the transformations $(Q,R,\theta)\rightarrow^{\mathbf{r}}(Q',R',\theta')$ as follows:

1. $(Q,R,\theta)\rightarrow^{\textbf{Normalise}}(Q',R',\theta)$, where Q' and R' are the normalised forms of Q and R as defined in **Unif**.

2. $(Q\wedge\lambda_A a=^?_{\lambda\sigma}\lambda_A b, R, \theta)\rightarrow^{\textbf{Dec-}\boldsymbol{\lambda}}(Q',R',\theta)$, where:

   - $Q'=(Q\wedge a=^?_{\lambda\sigma} b)$ and $R'=R$ when $a=^?_{\lambda\sigma} b$ is not solved (with respect to $Q\wedge R$),
   - $Q'=Q$ and $R'=(R\wedge a=^?_{\lambda\sigma} b)$ when $a=^?_{\lambda\sigma} b$ is solved.

3. $(Q\wedge(\mathrm{n}\,a_1\cdots a_p)=^?_{\lambda\sigma}(\mathrm{n}\,b_1\cdots b_p), R, \theta)\rightarrow^{\textbf{Dec-App}}(Q',R',\theta)$, where Q' consists in Q and the unsolved equations (with respect to $Q\wedge R$) in $\bigwedge_{i=1..p} a_i=^?_{\lambda\sigma} b_i$ and R' consists in R and the solved equations (with respect to $Q\wedge R$) in $\bigwedge_{i=1..p} a_i=^?_{\lambda\sigma} b_i$.

4. $(Q\wedge X=^?_{\lambda\sigma} b, R, \theta)\rightarrow^{\textbf{Replace}}(\{X\mapsto b\}(Q), R\wedge X=^?_{\lambda\sigma} b, \theta)$

5. (Q, R, θ)
$\rightarrow^{\textbf{Exp-}\boldsymbol{\lambda}\textbf{R}}$
$(\{X \mapsto \lambda_A Y\}\, Q, R \wedge X =^?_{\lambda\sigma} \lambda_A Y, \theta - \{X \mapsto \lambda_A a\} + \{Y \mapsto a\})$
when $\theta(X) = \lambda_A a$.

6. $(Q \wedge X[a_1 \cdots a_p . \uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \cdots b_q), R, \theta)$
$\rightarrow^{\textbf{Exp-AppR}}$
$(\{X \mapsto (\mathtt{r}\, H_1 \cdots H_k)\}(Q \wedge X[a_1 \cdots a_p . \uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \cdots b_q),$
$R \wedge X =^?_{\lambda\sigma} (\mathtt{r}\, H_1 \cdots H_k),$
$\theta - \{X \mapsto (\mathtt{r}\, c_1 \cdots c_k)\} + \{H_i \mapsto c_i\})$

when $\theta(X) = (\mathtt{r}\, c_1 \cdots c_k)$.

We call this set of rules **UStrat**.

LEMMA 4.12. *For a system* (Q, R) *having for a solution the* $\lambda\sigma$*-normalised grafting* θ*, there is no infinite derivation issued from* (Q, R, θ) *using the transformations of* **UStrat**.

Proof. We use as size of a grafting the sum of the sizes of the terms in its image: $|\theta| = \sum_{t \in \mathcal{R}an(\theta)} |t|$.

Let us first prove that there is no infinite sequence of rule applications involving **Dec-***, **Replace**, and **Normalise**. We define the complexity of a system $P = (Q, R)$ to be $\tau(P) = (|\mathcal{V}ar(Q)|, \{(\kappa_i, \max(|a_i|, |b_i|)\}_{a_i =^?_{\lambda\sigma} b_i \in Q})$, where $\kappa_i = 0$ when a_i and b_i are both in long normal form and 1 otherwise.

We compare the complexities lexicographically, using the standard ordering on naturals for the first component and the multiset ordering for the second component itself ordered by the lexicographic ordering on naturals (for a definition of these orderings see [15]).

We check now that each application of one of the rules **Dec-***, **Replace**, and **Normalise** decreases the complexity of the system on which it is applied.

Replace decreases always the number of unsolved variables, i.e. $|\mathcal{V}ar(Q)|$.

Dec-* never increases κ_i since normal forms are preserved by decomposition, and it always decreases the size of the equation to which it is applied.

Normalise may decrease the number of unsolved variables but always decreases the size of one of the κ_i.

In order to prove the termination of the application of the whole set of rules, let us add to the previous complexity measure of a system P a first component consisting in the size of the grafting θ: $\rho(P) = (|\theta|, \tau(P))$.

Since any application of **Exp-λR** or **Exp-AppR** makes the size of θ strictly decreasing and all the other transformations do not change it, this proves that the application of the above transformations is terminating. ∎

The previous result can be refined in order to prove completeness when dealing with rules like the merging rule in Fig. 4, by using a more sophisticated complexity measure including the solving level of an equation. Since the rules in **UStrat** are terminating, let us now see how they allow us to build the solutions:

LEMMA 4.13. *If θ is $\lambda\sigma$-solution of the system Q and if $(Q, R, \theta) \to^{\mathbf{r}} (Q', R', \theta')$ then θ' is a $\lambda\sigma$-solution of Q' and*:

$$\theta \circ Graft(R) =_{\lambda\sigma}^{\mathcal{V}ar(Q,R)} \theta' \circ Graft(R').$$

Proof. For all the rules except the **Exp-*** ones, $\theta = \theta'$ and since the transformations preserve the solutions, θ' is a $\lambda\sigma$-solution of Q'. Note that the equality modulo $\lambda\sigma$ is introduced by possible normalisation steps.

For the **Exp-λR** rule, because of the definition of θ and of Q', θ' is $\lambda\sigma$-solution of Q'.

Let Z be a variable in $\mathcal{V}ar(Q, R)$ then:

- if $Z = X$, in this case θ should be such that

$$\theta(X) = (\theta \circ Graft(R))(X) = \theta(X) = \lambda_A a, \text{ and } (\theta' \circ Graft(R'))(X) = \theta'(\lambda_A Y) = \lambda_A a.$$

- if $Z \neq X$ then by definition the two graftings give the same image of Z.

The proof is similar in the case of **Exp-AppR**. ▮

LEMMA 4.14. *Starting from the problem $P_0 = (Q_0, R_0)$ having the $\lambda\sigma$-normalised $\lambda\sigma$-solution θ_0 and applying the rules defined in* **UStrat** *leads to a finite derivation,*

$$(Q_0, R_0, \theta_0) \to (Q_1, R_1, \theta_1) \to \cdots \to (Q_n, R_n, \theta_n),$$

such that $\theta_0 =_{\lambda\sigma}^{\mathcal{V}ar(P_n)} \theta_n \circ Graft(R_n)$ where θ_n is a solution of the solved form Q_n.

Proof. Due to the definition of R_0, we have $\theta_0 = \theta_0 \circ Graft(R_0)$.

Following the previous lemmas, the derivation issued from (Q_0, R_0, θ_0) should be of finite length n and we get

$$\begin{aligned}\theta_0 \circ Graft(R_0) &=_{\lambda\sigma}^{\mathcal{V}ar(P_0)} \theta_1 \circ Graft(R_1) =_{\lambda\sigma}^{\mathcal{V}ar(P_1)} \cdots \\ &=_{\lambda\sigma}^{\mathcal{V}ar(P_n)} \theta_n \circ Graft(R_n),\end{aligned}$$

where $Q_n \wedge R_n$ should be a solved form (otherwise this would not be the end of the derivation by Lemma 4.8) and $\mathcal{V}ar(P_0) \supseteq \mathcal{V}ar(P_1) \supseteq \cdots \supseteq \mathcal{V}ar(P_n)$ since the set of variables of the unification problems (thus excluding the existentially bounded variables) could only decrease, due to the **Normalise** rule. ▮

As mentioned before, we assume that the rules in **Unif** are applied using the strategy **UnifReplace** in a fair way.

THEOREM 4.15. *The rules in* **Unif** *describe a correct and complete $\lambda\sigma$-unification procedure in the sense that, given a $\lambda\sigma$-unification problem P*:

- *if* **Unif** *leads in a finite number of steps to a disjunction of systems such that one of them is solved, then the problem P is $\lambda\sigma$-unifiable and a solution to P is the solution constructed in Lemma* 4.6 *for a solved constitutive system,*

- *if P has a unifier θ then the strategy* **UnifReplace** *leads in a finite number of steps to a disjunction of systems such that one constitutive system is solved and has θ as a unifier.*

Occur-Check $P \wedge X =^?_{\lambda\sigma} a$
$\rightarrow$
$\mathbf{F}$
if $X \in \mathcal{V}arRS(a)$

Merge $P \wedge X =^?_{\lambda\sigma} a \wedge X =^?_{\lambda\sigma} b \rightarrow P \wedge X =^?_{\lambda\sigma} a \wedge a =^?_{\lambda\sigma} b$

FIG. 4. Unif+, more rules for unification.

Proof. By application of the previous results on termination, completeness and correction. ▌

Remarks. 1. If the strategy **UnifReplace** terminates, the problem obtained is a disjunction of systems that are all in solved form. This disjunction is a description of a complete set of $\lambda\sigma$-unifiers of P consisting in the union of the $\lambda\sigma$-unifiers of all the solved forms obtained.

2. We have shown how to solve $\lambda\sigma$-unification problems using the rules in **Unif** with the strategy **UnifReplace**. This can be improved in many ways depending on the use of such a procedure. In particular the use of the **Merging** rule, described in Fig. 4, will give a Martelli and Montanari taste to the resulting algorithm.

3. In order to fail more often (note that the algorithm loops on equations like $X = (\mathtt{1}\ X)$), one can introduce an occur-check rule as described in Fig. 4.

The **Occur-Check** rule needs to define the usual notion of *variables on a strict rigid path*, denoted $\mathcal{V}arRS(a)$ and inductively defined on terms as follows:

- $X \in VarRS(\lambda\ a)$ if $X \in VarR(a)$,
- $X \in VarRS(\mathtt{n}\ a_1 \cdots a_n)$ if $X \in \bigcup_{i=1}^{n} VarR(a_i)$,
- $X \in VarR(a)$ if $X \in VarRS(a)$,
- $X \in VarR(X[s]\ a_1 \cdots a_n)$.

For example, X is on a rigid path in $(\mathtt{1}\ (X[Z]\ Y))$, but Y and Z are not.

4. The use of a calculus of explicit substitution like $\lambda\sigma$, which allows substitution composition, is fundamental in order to allow a simple expression of the transformation rules like **Exp-App**. Note also that the weak termination and confluence on substitution ground terms (thus containing `term` variables) of the calculus is crucial in the normalising rule. This allows us in particular to obtain solved forms that describe the set of open $\lambda\sigma$-unifiers.

4.4. Comparison with Conditional Narrowing

As mentioned above, we could have used conditional narrowing instead of the transformations above. The main difference between the two methods is that we are exploiting our knowledge of the $\boldsymbol{\lambda\sigma}$ term rewriting system to restrict the search space:

- As our system is normalising, we consider only normalised equations (as in normalised narrowing).
- We deeply use the fact that λ is a constructor, i.e., that a term of the form λa reduces only to terms on the form $\lambda a'$ such that a reduces to a', to simplify equations

of the form $\lambda a =^?_{\lambda b}$ to $a =^?_{\lambda\sigma} b$. In the same way, a term of the form $(\mathtt{n}\, a_1 \cdots a_p)$ whose type is atomic reduces only to terms of the form $(\mathtt{n}\, a'_1 \cdots a'_p)$ where a_i reduces on a'_i; we can simplify equations of the form $(\mathtt{n}\, a_1 \cdots a_p) = (\mathtt{n}\, b_1 \cdots b_p)$ to $a_1 = b_1$, ..., $a_p = b_p$ and $(\mathtt{n}\, a_1 \cdots a_p) = (\mathtt{m}\, b_1 \cdots b_q)$ with $n \neq m$ to failure.

- As our system is normalising, we substitute only normal terms to variables. As the long normal form of a closed term of a functional type is always λa and the long normal form of a closed term of an atomic type is always $(\mathtt{r}\, a_1 \cdots a_k)$, where k is the arity of $\mathtt{r}$, we consider only graftings of the form $X \mapsto \lambda Y$ if X has a functional type and $X \mapsto (\mathtt{r}\, H_1 \cdots H_k)$ if X has an atomic type. When we have an equation of the form $X[a_1 \cdots a_p . \uparrow^n] = (\mathtt{m}\, b_1 \cdots b_q)$ where X has an atomic type and when we take the grafting $X \mapsto (\mathtt{r}\, H_1 \cdots H_k)$ we know that if r is different from $1, ..., p$ and $m - n + p$, then we can anticipate the failure of the grafting $X \mapsto (\mathtt{r}\, H_1 \cdots H_k)$.
- Finally the flexible–flexible equations always have solutions.

Note that while the existence of solutions to flexible–flexible equations and the failure anticipation just mentioned are properties of the system we are considering, the use of normalisation and the simplification of rigid contexts are optimisations to narrowing of course applicable to other theories.

4.5. *Dropping the η-Rule*

Higher order unification algorithms come in two flavours according to the considered equational theory: $\beta\eta$-conversion or β-conversion. The algorithms for $\beta\eta$-conversion are simpler. Redundancy due to the synthesis of η-equivalent solutions is avoided and the use of the long normal form permits the determination of the shape of a term in function of its type (a long normal term of type $A \to B$ is always an abstraction). Nevertheless, algorithms can be designed for β-conversion alone. We show here how the algorithm above can be adapted to the theory $\lambda\sigma$, i.e., without using the **eta** axiom.

- When we have an equation of the form

$$(X[a_1 \cdots a_p . \uparrow^n]\; e_1 \cdots e_l) =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \cdots b_q)$$

we lose completeness if we apply only the rule **Exp-λ** to X as a term of a functional type need not be an abstraction anymore. We must also apply the rule **Exp-App**. Thus the rule **Exp-App** must be rephrased, removing the condition that X has an atomic type. Moreover we have to introduce some undeterminism between this rule and **Exp-λ**, leading to a single rule **Exp**.

$$\begin{array}{ll} \textbf{Exp} & P \wedge ((X[a_1 \cdots a_p . \uparrow^n]\; e_1 \cdots e_l) =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \cdots b_q)) \\ & \to \\ & \quad P \wedge (X[a_1 \cdots a_p . \uparrow^n]\; e_1 \cdots e_l) =^?_{\lambda\sigma} (\mathtt{m}\, b_1 \cdots b_q) \\ & \quad\quad \wedge \bigvee_{r \in R_p \cup R_i} \exists H_1, ..., H_k,\, X =^?_{\lambda\sigma} (\mathtt{r}\, H_1 \cdots H_k) \\ & \quad\quad \vee Q \end{array}$$

if X is not solved

where $H_1, ..., H_k$ are variables of the appropriate type not occurring in P with the contexts $\Gamma_{H_i} = \Gamma_X$, R_p is the subset of $\{1, ..., p\}$ such that $(\mathtt{r}\ H_1 \cdots H_k)$ has the right type, $R_i =$ if $m \geqslant n+1$ then $\{m-n+p\}$ else $\varnothing$ and $Q = \exists Y\ X =^?_{\lambda\sigma} \lambda Y$ if X has a functional type and $Q = \mathbf{F}$ otherwise.

• We have a new kind of equations that were forbidden when terms were in long normal form:

$$(\mathtt{n}\ a_1 \cdots a_p) =^?_{\lambda\sigma} \lambda b.$$

These equations obviously have no solutions; thus we add the rule:

$$\textbf{Dec-App-}\boldsymbol{\lambda}\ P \wedge ((\mathtt{n}\ a_1 \cdots a_p) =^?_{\lambda\sigma} \lambda b) \to \mathbf{F}.$$

• Also, a new kind of equations that were forbidden when terms were in long normal form is:

$$(X[a_1 \cdots a_p \,.\, \uparrow^n]\ e_1 \cdots e_l) =^?_{\lambda\sigma} \lambda b.$$

For these equations, we can either instantiate X by a term of the form $(\mathtt{r}\ H_1 \cdots H_k)$, $r \in \{1, ..., p\}$, or λY. Thus we add the rule:

$$\textbf{Exp2}\quad P \wedge ((X[a_1 \cdots a_p \,.\, \uparrow^n]\ e_1 \cdots e_l) =^?_{\lambda\sigma} \lambda b)$$

$$\to$$

$$P \wedge (X[a_1 \cdots a_p \,.\, \uparrow^n]\ e_1 \cdots e_l) =^?_{\lambda\sigma} \lambda b$$

$$\wedge \bigvee_{r \in R_p} \exists H_1, ..., H_k, X =^?_{\lambda\sigma} (\mathtt{r}\ H_1 \cdots H_k)$$

$$\vee\ \exists Y\ X =^?_{\lambda\sigma} \lambda Y$$

if X is not solved

where $H_1, ..., H_k$ are variables of the appropriate type not occurring in P with the contexts $\Gamma_{H_i} = \Gamma_X$, R_p is the subset of $\{1, ..., p\}$ such that $(\mathtt{r}\ H_1 \cdots H_k)$ has the right type.

In a similar way as with the **eta** rule, this extended set of transformation rules can be proved correct and complete.

5. APPLICATION TO UNIFICATION IN λ-CALCULUS

In this section, we prove that the higher order problem $a =^?_{\beta\eta} b$ has a solution if and only if the precooked equational problem $a_F =^?_{\lambda\sigma} b_F$ has a solution. Moreover, solutions of the higher order problem are completely described from the equational solutions.

5.1. *Grafting Unification and Substitution Unification*

PROPOSITION 5.1. *Let $a =^?_{\beta\eta} b$ be a unification problem in $\Lambda_{DB}(\mathscr{X})$. If the problem $a =^?_{\eta} b$ has a solution, then the problem $a_F =^?_{\lambda\sigma} b_F$ also has a solution.*

Proof. By Proposition 3.5, if the problem $a =^?_{\beta\eta} b$ has a solution $\{X_1/t_1, ..., X_n/t_n\}$ then the grafting $\{X_1 \mapsto t_{1F}, ..., X_n \mapsto t_{nF}\}$ is a solution to the problem $a_F =^?_{\lambda\sigma} b_F$. ∎

We want now to prove the converse of this proposition, i.e., if the problem $a_F =^?_{\lambda\sigma} b_F$ has a solution then the problem $a =^?_{\beta\eta} b$ also has a solution. To apply Proposition 3.5, we need to show that if $a_F =^?_{\lambda\sigma} b_F$ has a solution, then it also has a solution in the image of the precooking translation. To constructively prove this statement, we use the completeness of the system **Unif**, i.e., the fact that if there exists a solution to $a_F =^?_{\lambda\sigma} b_F$, then this problem has a solved form. From this solved form we build a solution in the image of the precooking translation. One of the difficulties is the following. If we solve the flexible–flexible equations by the grafting of Proposition 4.6 then we construct a grafting that is not in the image of the precooking translation. For instance, consider the context $\Gamma = A.nil$ and the variables X and Y with type A and context Γ. Then for the equation $X =^?_{\lambda\sigma} Y$ (which is not flexible–flexible in the sense of Definition 4.5 because it is solved, but allows us to describe the problem), if we take the grafting $X \mapsto Z[\uparrow]$, $Y \mapsto Z[\uparrow]$ we do not get a solution in the image of the precooking translation. In this case, we must take, of course, $X \mapsto Z$, $Y \mapsto Z$. More generally, instead of taking the grafting $X \mapsto Z_T[\uparrow^{|\Gamma_X|}]$ as solution of a flexible–flexible equation, we take the grafting $X \mapsto Z_T[\uparrow^{|\Gamma_X| - |\Gamma|}]$, where Γ is the context in which the initial problem is typed and Z_T is a metavariable of context Γ and type T. Thus, in order to be able to apply Proposition 3.5 we must show that if the variable X has an occurrence in a term of the form $X[a_1 \cdots a_p . \uparrow^n]$ then we have $p \leqslant |\Gamma_X| - |\Gamma|$. We prove now that this proposition is an invariant of the system **Unif**.

DEFINITION 5.2. Let Γ and Δ be two contexts, Γ is an extension of Δ if it has the form $\Gamma = A_1 \cdots A_n . \Delta$. It is a strict extension if $n \neq 0$.

For a context Γ, a σ-normal term a is called *Γ-stable* if, for every variable X occurring in a subterm of a of the form $X[b_1 \cdots b_p . \uparrow^n]$, Γ_X is an extension of Γ and $p \leqslant |\Gamma_X| - |\Gamma|$.

PROPOSITION 5.3. *Let Γ be a given context.*

1. *If a is a Γ-stable term well typed in a context $A_1 \cdots A_i . \Gamma$ and $p \leqslant i$, then the σ-normal form of $a[\mathtt{1} \cdots \mathtt{p} . \uparrow^{p+1}]$ is Γ-stable.*

2. *If a is a Γ-stable term well typed in a context $A_1 \cdots A_i . \Gamma$ and if $b_1, ..., b_p$ are Γ-stable terms and $p \leqslant i$, then the σ-normal form of $a[b_1 \cdots b_p . \uparrow^n]$ is Γ-stable.*

3. *If a is a σ-normal term well typed in a context $A_1 \cdots A_i . \Gamma$ and Γ-stable, then any σ-normal term a' obtained by reducing a is Γ-stable. Thus, the normal form of a is Γ-stable. The long normal form of a is also Γ-stable.*

4. *If a and b are σ-normal terms Γ-stable and well typed in the contexts $A_1 \cdots A_i.\Gamma$ and $B_1 \cdots B_j.\Gamma$ then the σ-normal form of $\{X \mapsto b\}\, a$ is Γ-stable.*

Proof. 1. By induction on the structure of a.

- If $a = \lambda a'$ then $a[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}] = \lambda(a'[\mathtt{1}.\mathtt{2} \cdots \mathtt{p+1}.\uparrow^{p+2}])$. The term a' is well typed in $B.A_1 \cdots A_i.\Gamma$ and is Γ-stable, thus, by induction hypothesis $a'[\mathtt{1}.\mathtt{2} \cdots \mathtt{p+1}.\uparrow^{p+2}]$ is Γ-stable. Therefore $a[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}]$ is Γ-stable.
- If $a = (a_1\, a_2)$ or a is a de Bruijn index, the case is easy.
- If $a = X[c_1 \cdots c_q.\uparrow^m]$ in this case we have $q \leqslant |\Gamma_X| - |\Gamma|$. We get the term

$$X[c_1 \cdots c_q.\uparrow^m][\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}];$$

i.e.,

$$X[c_1[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}] \cdots c_q[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}].(\uparrow^m \circ (\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}))].$$

The terms c_i are well typed in the context $A_1 \cdots A_i.\Gamma$; thus by the induction hypothesis, the σ-normal forms of the terms

$$c_1[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}], ..., c_q[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}];$$

are Γ-stables.

If $m \geqslant p$ we get

$$X[c_1[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}] \cdots c_q[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}].\uparrow^{m+1}]$$

as $q \leqslant |\Gamma_X| - |\Gamma|$ and the σ-normal forms of the terms $c_1[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}], ...,$ $c_q[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}]$ are Γ-stables, the σ-normal form of $a[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}]$ is Γ-stable.

If $m < p$ we get the term

$$X[c_1[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}] \cdots c_q[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}].\mathtt{m+1} \cdots \mathtt{p}.\uparrow^{p+1}].$$

The length of this substitution is $r = q + p - m$. The term $a = X[c_1 \cdots c_q.\uparrow^m]$ is well typed in the context $A_1 \cdots A_i.\Gamma$; thus $|\Gamma| + i = |\Gamma_X| - q + m$. Thus,

$$r = q + p - m = p + |\Gamma_X| - |\Gamma| - i$$

as $p \leqslant i$ we have $r \leqslant |\Gamma_X| - |\Gamma|$. Then the σ-normal form of $a[\mathtt{1} \cdots \mathtt{p}.\uparrow^{p+1}]$ is Γ-stable.

2. By induction on the structure of a.

- If $a = \lambda a'$ then $a[b_1 \cdots b_p.\uparrow^n] = \lambda a'[\mathtt{1}.b_1[\uparrow] \cdots b_p[\uparrow].\uparrow^{n+1}]$. The term a' is well typed in the context $B.A_1 \cdots A_i.\Gamma$ and is Γ-stable. By the previous remark, the terms $\mathtt{1}, b_1[\uparrow], ..., b_p[\uparrow]$ are also Γ-stable. Thus, by the induction hypothesis the σ-normal form of $a'[\mathtt{1}.b_1[\uparrow] \cdots b_p[\uparrow].\uparrow^{n+1}]$ is Γ-stable. Therefore $a[b_1 \cdots b_p \uparrow^n]$ is Γ-stable.

- If $a = (a_1\, a_2)$ or a is a de Bruijn index, the case is easy.
- If $a = X[c_1 \cdots c_q . \uparrow^m]$ in this case we have $q \leqslant |\Gamma_X| - |\Gamma|$. We get the term

$$X[c_1 \cdots c_q . \uparrow^m][b_1 \cdots b_p . \uparrow^n];$$

i.e.,

$$X[c_1[b_1 \cdots b_p . \uparrow^n] \cdots c_q[b_1 \cdots b_p . \uparrow^n].(\uparrow^m \circ (b_1 \cdots b_p . \uparrow^n))].$$

The terms c_i are well typed in the context $A_1 \cdots A_i . \Gamma$; thus by the induction hypothesis, the σ-normal forms of the terms $c_1[b_1 \cdots b_p . \uparrow^{p+1}]$, ..., $c_q[b_1 \cdots b_p . \uparrow^{p+1}]$ are Γ-stables.

If $m \geqslant p$ we get

$$X[c_1[b_1 \cdots b_p . \uparrow^n] \cdots c_q[b_1 \cdots b_p . \uparrow^n] . \uparrow^{n+m-p}].$$

As $q \leqslant |\Gamma_X| - |\Gamma|$ and the σ-normal form of $c_1[b_1 \cdots b_p . \uparrow^n]$, ..., $c_q[b_1 \cdots b_p . \uparrow^n]$ are Γ-stable, the term $a[b_1 \cdots b_p \uparrow^n]$ is Γ-stable.

If $m < p$ we get the term

$$X[c_1[b_1 \cdots b_p . \uparrow^n] \cdots c_q[b_1 \cdots b_p . \uparrow^n] . b_{m+1} \cdots b_p . \uparrow^n].$$

The length of this substitution is $r = q + p - m$. The term $a = X[c_1 \cdots c_q . \uparrow^m]$ is well typed in the context $A_1 \cdots A_i . \Gamma$; thus $|\Gamma| + i = |\Gamma_X| - q + m$. Thus

$$r = q + p - m = p + |\Gamma_X| - |\Gamma| - i$$

as $p \leqslant i$ we have $r \leqslant |\Gamma_X| - |\Gamma|$. Therefore the σ-normal form of $a[b_1 \cdots b_p . \uparrow^n]$ is Γ-stable.

3. When we reduce a β-redex $(\lambda_C a)b$ occurring under j abstractions of type $B_1, ..., B_j$, the term a is well typed in a context $C.B_1 \cdots B_j A_1 \cdots A_i \Gamma$. This term reduces to $a[b.id]$. Thus, by the previous point the σ-normal form of $a[b.id]$ is Γ-stable.

When we reduce a η-redex $\lambda_C(a\, 1)$ occurring under j abstractions of type $B_1, ..., B_j$, the term a is well typed in a context $C.B_1 \cdots B_j A_1 \cdots A_i \Gamma$. This term reduces to the σ-normal form of the term $a[Z.id]$ where Z is a fresh variable of type C in the context $B_1 \cdots B_j . A_1 \cdots A_j . \Gamma$; by the previous point the σ-normal form of this term is Γ-stable.

When we expand a term a in $\lambda(a[\uparrow]\, 1)$, the σ-normal form of the term $a[\uparrow]$ is Γ-stable and thus the term $\lambda(a[\uparrow]\, 1)$ is Γ-stable.

4. By induction over the structure of a we show that the σ-normal form of $\{X \mapsto b\}\, a$ is Γ-stable.

• if $a = X[a_1 \cdots a_p . \uparrow^n]$ then $\{X \mapsto b\} a = b[\{X \mapsto b\}\ a_1 \cdots \{X \mapsto b\}\ a_p . \uparrow^n]$. By the induction hypothesis, the σ-normal forms a'_1, ..., a'_p of the terms $\{X \mapsto b\}\ a_1$, ..., $\{X \mapsto b\}\ a_p$ are Γ-stable.

As the term $a = X[a_1 \cdots a_p . \uparrow^n]$ is Γ-stable, we have $p \leqslant |\Gamma_X| - |\Gamma|$. As X and b are well typed in the same context we have $\Gamma_X = B_1 \cdots B_j . \Gamma$; thus $p \leqslant j$ and the σ-normal form of $b[a'_1 \cdots a'_p . \uparrow^n]$ is Γ-stable. Therefore the σ-normal form of $\{X \mapsto b\}\ a$ is also Γ-stable.

• If $a = Y[a_1 \cdots a_p . \uparrow^n]$ then $\{X \mapsto b\}\ a = Y[\{X \mapsto b\}\ a_1 \cdots \{X \mapsto b\}\ a_p . \uparrow^n]$. By the induction hypothesis, the σ-normal forms a'_1, ..., a'_p of the terms $\{X \mapsto b\}\ a_1$, ..., $\{X \mapsto b\}\ a_p$ are Γ-stable. Thus the σ-normal form of $\{X \mapsto b\}\ a$ is Γ-stable.

• if a is a de Bruijn index, an application, or an abstraction, we apply the induction hypothesis. ∎

Proposition 5.4 (Invariants). *Let $a =^?_{\beta\eta} b$ be a unification problem in $\Lambda_{DB}(\mathscr{X})$ well typed in a context Γ. Consider a derivation of $a_F =^?_{\lambda\sigma} b_F$ in* **Unif**. *We have the following invariants*:

• *if an equation of the derived problem is well typed in a context Δ then Δ has the form $A_1 \cdots A_n \Gamma$,*

• *for every variable Y of the derived problem, its context Γ_Y has the form $B_1 \cdots B_p . \Gamma$,*

• *for every subterm $Y[a_1 \cdots a_p . \uparrow^n]$ of the derived problem we have $p \leqslant |\Gamma_Y| - |\Gamma|$.*

Proof. By induction on the structure of the derivation, using Proposition 5.3 for the rules **Normalise** and **Replace**.

As an immediate consequence of the previous proposition, all the solved forms issued from a precooked problem in context Γ are Γ-stables.

Proposition 5.5. *Let Γ be a context, a a σ-normal term, and T a type. If a is Γ-stable and every variable X in a is such that $\Gamma_X = \Gamma$ then a is in the image of the precooking translation.*

Proof. Every occurrence u of a variable X belongs to a subterm of the form $X[a_1 \cdots a_p . \uparrow^n]$. As a is Γ-stable we have $p \leqslant |\Gamma_X| - |\Gamma|$. As $\Gamma_X = \Gamma$ we have $p = 0$. Then as the term $X[\uparrow^n]$ is well typed in some context $B_1 \cdots B_{|u|} . \Gamma$ and the term X is well typed in Γ we have $n = |u|$; thus a is in the image of the precooking translation. ∎

Proposition 5.6. *Let $a =^?_{\lambda\sigma} b$ be a unification problem in $\Lambda_{DB}(\mathscr{X})$. From any solved form of $a_F =^?_{\lambda\sigma} b_F$ verifying the invariants above, we can construct a solution to $a =^?_{\beta\eta} b$.*

Proof. Call P a solved form of $a_F =^?_{\lambda\sigma} b_F$ and Γ a context in which the initial problem is well typed. Let θ be the grafting associating to every nonsolved variable X: $\Gamma_X \vdash T$ of P, the term $Z_T[\uparrow^{|\Gamma_X| - |\Gamma|}]$, where Z_T is a fresh variable of sort $\Gamma \vdash T$:

$$\theta : X : \Gamma_X \vdash T \mapsto Z_T[\uparrow^{|\Gamma_X| - |\Gamma|}] \qquad \text{with} \quad Z : \Gamma \vdash T.$$

Since the term $Z_T[\uparrow^{|\Gamma_X|-|\Gamma|}]$ is Γ-stable (just apply the definition), by application of Proposition 5.3, the problem $\theta(P)$ is Γ-stable. Furthermore, θ preserves the solved variables of the system.

All the nonsolved variables occurring in the $\lambda\sigma$-normal form of $\theta(P)$ are the Z_T and these variables are well typed in Γ. Such a variable occurs in a subterm $Z_T[a_1 \cdots a_p . \uparrow^n]$ and we have $p \leqslant |\Gamma_Z| - |\Gamma|$, thus $p = 0$ since $\Gamma_Z = \Gamma$ by definition.

The flexible–flexible equations appearing in the $\lambda\sigma$-normal form of $\theta(P)$ have necessarily the form $Z_T[\uparrow^m] =^?_{\lambda\sigma} Z_U[\uparrow^n]$. By definition of an equation, the terms $Z_T[\uparrow^m]$ and $Z_U[\uparrow^n]$ are well typed with the same type and in the same context, so we have $T = U$ and $m = n$. Therefore, the flexible–flexible equations of the normal form of $\theta(P)$ relate identical terms. The grafting θ is thus a solution of the flexible–flexible equations of the solved form P.

Consider the grafting ϕ that binds every variable X to the term c, where $X =^?_{\lambda\sigma} c$ is a solved equation of $\theta(P)$. The grafting ϕ is a solution of $\theta(P)$, thus $\phi \circ \theta$ is a solution of P and thus of $a_F =^?_{\lambda\sigma} b_F$.

Consider a variable X that occurs in the initial problem $a_F =^?_{\lambda\sigma} b_F$ and in the solved form P. Then either no solved equation of the form $X =^?_{\lambda\sigma} c$ occurs in P and $\phi \circ \theta(X)$ is some Z_T or $\phi \circ \theta(X)$ is $\theta(c)$. In both cases, the term $\phi \circ \theta(X)$ is Γ-stable (since c is Γ-stable as well as $\theta(c)$) and every variable Z occurring in this term is such that $\Gamma_Z = \Gamma$.

As the variable X occurs in the problem $a_F =^?_{\lambda\sigma} b_F$ we have by definition $\Gamma_X = \Gamma$. Then the term $\phi \circ \theta(X)$ is Γ-stable and every variable Z occurring in $\phi \circ \theta(X)$ is such that $\Gamma_Z = \Gamma$ thus by Proposition 5.5, $\phi \circ \theta(X)$ is in the image of the precooking translation. Therefore $a =^?_{\beta\eta} b$ has a solution. ▮

Remark. The variables Z_T remaining in the solution defined in the last proof have the sort $\Gamma \vdash T$. Provided that there is a term of every atomic type in Γ, this sort is not empty. This contrasts with the solution built in Lemma 4.6 where the variables Z_T have the sort $nil \vdash T$ which is empty.

COROLLARY 5.7. *Let $a =^?_{\beta\eta} b$ be a unification problem in $\Lambda_{DB}(\mathcal{X})$. If the unification problem $a_F =^?_{\lambda\sigma} b_F$ has a solution then $a =^?_{\beta\eta} b$ has a solution.*

Proof. If the problem $a_F =^?_{\lambda\sigma} b_F$ has a solution then it has a solved form by the system **Unif**. This solved form verifies the invariants above and the problem $a =^?_{\beta\eta} b$ has a solution. ▮

THEOREM 5.8. *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.*

An immediate consequence of the undecidability of higher order unification [25, 21] and of the last theorem is that equational unification is undecidable in the theory $\lambda\sigma$.

5.2. *Translating Back Equations*

We now show that for any unification problem P, derived from a problem $a_F =^?_{\lambda\sigma} b_F$, we can reconstruct a problem Q in the image of the precooking translation that has the same solutions as P. Therefore Q may be translated back to λ-calculus into a higher unification problem R. Now, the solutions of P and Q are the precooking of the solutions of R. This result will be used for two purposes. First, solved forms of P are translated back to λ-calculus, giving a description of the set of solutions of R as solved forms. Then we will use this translation to simulate the algorithm of [26].

DEFINITION 5.9. The system **Unif** is extended with the two following rules:

Anti-Exp-λ $P \rightarrow \exists Y (P \wedge X =^?_{\lambda\sigma} (Y[\uparrow]\, 1))$
if $X \in \mathcal{V}ar(P)$ such that $\Gamma_X = A.\Gamma'_X$
where $Y \in \mathcal{X}$, $Y \notin \mathcal{V}ar(P)$ and
$T_Y = A \rightarrow T_X, \Gamma_Y = \Gamma'_X$

Anti-Dec-λ $P \wedge a =^?_{\lambda\sigma} b \rightarrow P \wedge \lambda_A a =^?_{\lambda\sigma} \lambda_A b$
if $a =^?_{\lambda\sigma} b$ is well-typed in a context $\Delta = A.\Delta'$

PROPOSITION 5.10. *The system* **Unif**, *augmented by these rules, remains sound and complete.*

Proof. Soundness is trivial, completeness of **Anti-Dec-λ** also.

For the completeness of **Anti-Exp-λ**, consider a grafting θ solution of the problem P, take $\theta' Y = \lambda(\theta X)$ and $\theta' Z = \theta Z$ if $Z \neq Y$ then θ' is a solution of P and (denoting $\theta'(X)$ by a)

$$\theta'(X =^?_{\lambda\sigma} (Y[\uparrow]\, 1)) = (a =^?_{\lambda\sigma} ((\lambda a)[\uparrow]\, 1)) = (a =^?_{\lambda\sigma} a);$$

thus θ' is a solution of the system $P \wedge (X =^?_{\lambda\sigma} (Y[\uparrow]\, 1))$ and θ is a solution to the system $\exists Y (P \wedge (X =^?_{\lambda\sigma} (Y[\uparrow]\, 1)))$. ▮

For $a_F =^?_{\lambda\sigma} b_F$ an initial well-typed problem in the context Γ, we call **Back** the strategy of the **Anti-*** and **Replace** rules application consisting in applying the rule **Anti-Exp-λ** only to variables X such that Γ_X is a strict extension of Γ and the **Anti-Dec-λ** only to equations whose context is a strict extension of Γ, **Replace** being applied eagerly.

PROPOSITION 5.11. *Let $a =^?_{\beta\eta} b$ be a higher order unification problem and P be an equational problem derived from $a_F =^?_{\lambda\sigma} b_F$ by the rules of* **Unif**. *When applying the strategy* **Back** *on P, the invariants of Proposition* 5.4 *are verified.*

Proof. The two first invariants are kept as we use only the rules **Anti-Exp-λ** and **Anti-Dec-λ** on variables and equations whose context is a strict extension of Γ. The third invariant is kept as the subterms $X[a_1 \cdots a_p . \uparrow^n]$ of the problem obtained by the rules above are already terms of this form in the initial problem or are X or $Y[\uparrow]$ which both verify $p = 0$. ▮

PROPOSITION 5.12. *Let $a =^?_{\beta\eta} b$ be a higher order unification problem and P be an equational problem derived from $a_F =^?_{\lambda\sigma} b_F$ by using the rules of* **Unif**. *The system resulting of the normalisation with the strategy* **Back** *of the system P is the precooking of a problem in λ-calculus.*

Proof. As asserted by the invariant of the system **Unif**, every context Γ_X is an extension of Γ and every equation is well typed in an extension of Γ. Thus we can apply the rules **Anti-Dec-λ** and **Anti-exp-λ** then **Replace** to get rid of every variable whose context is not Γ and every equation whose context is not Γ. We obtain a Γ-stable problem such that all equations are well typed in the context Γ, for every variable X occurring in the problem, $\Gamma_X = \Gamma$. We can conclude by Proposition 5.5 that this problem is the the pre-cooking of a problem in λ-calculus. ▮

We call *λ-solved form* any solved system of equations in λ-calculus in the sense of [46].

COROLLARY 5.13. *Let $a =^?_{\beta\eta} b$ a higher order unification problem such that $a_F =^?_{\lambda\sigma} b_F$ can be rewritten by the system* **Unif** *to a disjunction of systems that has one of its constitutive systems P solved. Let Q be the system resulting of the normalisation with the strategy* **Back** *of the system P and $R = F^{-1}(Q)$. Then R is a λ-solved form and the solutions of R are solutions of $a =^?_{\beta\eta} b$. If we apply the trivial solution of* [46] *to solve the flexible–flexible equations, we get back the solution built in the proof of Proposition* 5.6.

Proof. Let θ be a substitution in λ-calculus. Applying the previous results, if θ is a solution of R then θ_F is a solution of R_F, θ_F is a solution of Q, θ_F is a solution of P, θ_F is a solution of $a_F =^?_{\lambda\sigma} b_F$, and θ is a solution of $a =^?_{\beta\eta} b$. ▮

THEOREM 5.14 (Description of the solutions). *Let $a =^?_{\beta\eta} b$ be a higher order unification problem such that $a_F =^?_{\beta\eta} b_F$ is well typed in context Γ. Any solution θ of $a =^?_{\beta\eta} b$ can be obtained as the solution of a system in λ-solved form resulting from the application of the* **Unif** *rules followed by normalisation using the* **Back** *strategy and back-cooking.*

Proof. Let θ be a substitution in λ-calculus. By application of the previous results, θ is a solution of R if and only if θ_F is a solution of R_F, if and only if θ_F is a solution of Q, if and only if θ_F is a solution of P, if and only if θ_F is a solution of $a_F =^?_{\lambda\sigma} b_F$, if and only if θ is a solution of $a =^?_{\beta\eta} b$. ▮

This can be depicted in the following way:

$$\begin{tikzcd}
a =^?_{\beta\eta} b \arrow[d, "F"'] & & R \\
a_F =^?_{\lambda\sigma} b_F \arrow[r, "\text{Unif}"] & P \arrow[r, "\text{Back}"] & Q \arrow[u, "F^{-1}"']
\end{tikzcd}$$

where P is one of the solved forms of $a_F =^?_{\lambda\sigma} b_F$ and R one of the λ-solved form of the initial problem $a =^?_{\beta\eta} b$.

Remark. When using the **Occur-Check** rule, we can notice that we get an algorithm that restricts to syntactic first order unification when the input problem is a first order unification problem.

5.3. Relation with Other Unification Algorithms

Huet's algorithm can be seen as a particular strategy for the system **Unif** extended by the two rules above. Indeed, a simplification of an equation consists in applying **Dec-λ** n times, then **Dec-app** once, and finally **Anti-Dec-λ** n times to the equations obtained by **Dec-app**. For instance, from the equation

$$\lambda \cdots \lambda(\mathtt{k}\, a_1 \cdots a_p) =^?_{\lambda\sigma} \lambda \cdots \lambda(\mathtt{k}\, b_1 \cdots b_p),$$

we get $(\mathtt{k}\, a_1 \cdots a_p) =^?_{\lambda\sigma} (\mathtt{k}\, b_1 \cdots b_p)$ by applying **Dec-λ** n times, then $a_1 =^?_{\lambda\sigma} b_1, ...,$ $a_n =^?_{\lambda\sigma} b_n$ using **Dec-app**, then $\lambda \cdots \lambda a_1 =^?_{\lambda\sigma} \lambda \cdots \lambda b_1, ...,$ $\lambda \cdots \lambda a_p =^?_{\lambda\sigma} \lambda \cdots \lambda b_p$ by applying **Anti-Dec-λ** n times to each equation.

In the same way, applying an elementary substitution consists in applying **Exp-λ** n times, then **Exp-app** once, and finally applying **Anti-Exp-λ** n times to each new variable H_i. For instance, if we have a variable X of type $T_1 \to \cdots \to T_n \to U$, we get $X \mapsto \lambda \cdots \lambda Y$ by applying **Exp-λ** n times, then $X \mapsto \lambda \cdots \lambda(\mathtt{k}\, H_1 \cdots H_p)$ by applying **Exp-app** to Y and finally $X \mapsto \lambda \cdots \lambda(\mathtt{k}\,(K_1[\uparrow^n]\, \mathtt{n} \cdots \mathtt{1}) \cdots (K_p[\uparrow^n]\, \mathtt{n} \cdots \mathtt{1}))$ by applying **Anti-Exp-λ** n times to each H_i. This term is the precooking of $\lambda \cdots \lambda(\mathtt{k}\,(K_1\, \mathtt{n} \cdots \mathtt{1}) \cdots (K_p\, \mathtt{n} \cdots \mathtt{1}))$ which is Huet's elementary substitution.

When we apply this strategy, the equations we obtain and the graftings we build are always the precooking of λ-equations and λ-substitutions.

5.4. Dependence Constraints

In some extension of higher order unification (for instance in unification with mixed prefix [37]) unification problems come with more scoping constraints than the ones present in the equations. For instance we may want to restrict the term substituted to some variable X in such a way that some de Bruijn index of the context (for instance $\mathtt{2}$) does not appear in it.

Such problems can be expressed very easily in $\lambda\sigma$. For instance, we may express that $\mathtt{2}$ is forbidden in X just by adding the equation $X =^?_{\lambda\sigma} Y[\mathtt{1}.\uparrow^2]$.

6. EXAMPLES

Let us show in detail how the results of Theorem 5.14 could be applied on two examples.

6.1. A Simple Example

Let us first solve the problem $\lambda_A\, y.(X\, a) =^?_{\beta\eta} \lambda_A\, y.a$ with $a : A$, $X : A \to A$. This equation is encoded in de Bruijn terms, using the context $\Gamma = A.nil$ into $\lambda(X\, \mathtt{2}) =^?_{\beta\eta} \lambda\mathtt{2}$ and then precooked into:

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

Applying the rule **Dec-λ** we get

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

and then with the rule **Exp-λ**,

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

where $\Gamma_Y = A.\Gamma$ and $T_Y = A$. The rule **Replace** is then applied to get

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

and applying **Normalise** results in

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

Applying the rule **Exp-app** yields

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

which reduces using the rule **Replace** into

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

that is normalised by rule **Normalise** into

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

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

6.2. A More Elaborated Example

We solve now the classical equation

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

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

Applying the rules in **Unif**, we get the following derivations, where the variables Y, H_1, H_2 are existentially quantified:

$$\{(2(X\,1)) =^?_{\lambda\sigma} (X(2\,1)) \qquad \to_{\textbf{Exp-}\boldsymbol{\lambda}} \begin{cases}(2(X\,1)) =^?_{\lambda\sigma} (X(2\,1))\\ X =^?_{\lambda\sigma} \lambda Y \qquad (Y : A\cdot\Gamma \vdash A)\end{cases}$$

$\to_{\textbf{Replace}}$

$$\begin{cases}(2(\lambda Y\,1)) =^?_{\lambda\sigma} (\lambda Y(2\,1))\\ X =^?_{\lambda\sigma} \lambda Y\end{cases} \qquad \to_{\textbf{Normalise}} \begin{cases}(2\;Y[1.id]) =^?_{\lambda\sigma} Y[(2\,1).id]\\ X =^?_{\lambda\sigma} \lambda Y\end{cases}$$

$\to_{\textbf{Exp-app}}$

$$\begin{cases}(2\;Y[1.id]) =^?_{\lambda\sigma} Y[(2\,1).id]\\ Y =^?_{\lambda\sigma} 1\\ X =^?_{\lambda\sigma} \lambda Y\end{cases} \vee \begin{cases}(2\;Y[1.id]) =^?_{\lambda\sigma} Y[(2\,1).id]\\ Y =^?_{\lambda\sigma} (3\;H_1) \qquad (H_1 : \Gamma_Y \vdash A)\\ X =^?_{\lambda\sigma} \lambda Y\end{cases}$$

$\to_{\textbf{Replace}}$

$$\begin{cases}(2\;1[1.id]) =^?_{\lambda\sigma} 1[(2\,1).id]\\ Y =^?_{\lambda\sigma} 1\\ X =^?_{\lambda\sigma} \lambda 1\end{cases} \vee \begin{cases}(2\;(3\;H_1)[1.id]) =^?_{\lambda\sigma} (3\;H_1)[(2\,1).id]\\ Y =^?_{\lambda\sigma} (3\;H_1)\\ X =^?_{\lambda\sigma} \lambda Y\end{cases}$$

$\to_{\textbf{Normalise}}$

$$\begin{cases}(2\,1) =^?_{\lambda\sigma} (2\,1)\\ Y =^?_{\lambda\sigma} 1\\ X =^?_{\lambda\sigma} \lambda 1\end{cases} \vee \begin{cases}(2\;(2\;H_1[1.id])) =^?_{\lambda\sigma} (2\;H_1[(2\,1).id])\\ Y =^?_{\lambda\sigma} (3\;H_1)\\ X =^?_{\lambda\sigma} \lambda(3\;H_1)\end{cases}$$

We get a first solved form, $X =^?_{\lambda\sigma} \lambda\,1$, and we continue with the second system:

$\to_{\textbf{Dec-app1}}$

$$\begin{cases}(2\;H_1[1.id]) =^?_{\lambda\sigma} H_1[(2\,1).id]\\ Y =^?_{\lambda\sigma} (3\;H_1)\\ X =^?_{\lambda\sigma} \lambda(3\;H_1)\end{cases}$$

$\to_{\textbf{Exp-app}}$

$$\begin{cases}(2\;H_1[1.id]) =^?_{\lambda\sigma} H_1[(2\,1).id]\\ H_1 =^?_{\lambda\sigma} 1\\ Y =^?_{\lambda\sigma} (3\;H_1)\\ X =^?_{\lambda\sigma} \lambda(3\;H_1)\end{cases} \vee \begin{cases}(2\;H_1[1.id]) =^?_{\lambda\sigma} H_1[(2\,1).id]\\ H_1 =^?_{\lambda\sigma} (3\;H_2) \qquad (H_2 : \Gamma_Y \vdash A)\\ Y =^?_{\lambda\sigma} (3\;H_1)\\ X =^?_{\lambda\sigma} \lambda(3\;H_1)\end{cases}$$

$\to_{\textbf{Replace}}$

$$\begin{cases}(2\;1[1.id]) =^?_{\lambda\sigma} 1[(2\,1).id]\\ H_1 =^?_{\lambda\sigma} 1\\ Y =^?_{\lambda\sigma} (3\,1)\\ X =^?_{\lambda\sigma} \lambda(3\,1)\end{cases} \vee \begin{cases}(2\;(3\;H_2)[1.id]) =^?_{\lambda\sigma} (3\;H_2)[(2\,1).id]\\ H_1 =^?_{\lambda\sigma} (3\;H_2)\\ Y =^?_{\lambda\sigma} (3(3\;H_2))\\ X =^?_{\lambda\sigma} \lambda(3(3\;H_2))\end{cases}$$

$\to_{\textbf{Normalise}}$

$$\begin{cases}(2\,1) =^?_{\lambda\sigma} (2\,1)\\ H_1 =^?_{\lambda\sigma} 1\\ Y =^?_{\lambda\sigma} (3\,1)\\ X =^?_{\lambda\sigma} \lambda(3\,1)\end{cases} \vee \begin{cases}(2(2\;H_2[1.id])) =^?_{\lambda\sigma} (1\;H_2[(2\,1).id])\\ H_1 =^?_{\lambda\sigma} (3\;H_2)\\ Y =^?_{\lambda\sigma} (3(3\;H_2))\\ X =^?_{\lambda\sigma} \lambda(3(3\;H_2))\end{cases}$$

We get another solved form, $X =^?_{\lambda\sigma} \lambda(\mathtt{3}\ \mathtt{1})$, 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 precooking them back to λ-terms we get for the first $X =^?_{\beta\eta} \lambda x.x$ and for the second $X =^?_{\beta\eta} \lambda x.(f\,x)$ which are clearly two solutions of the initial problem.

7. CONCLUSION

It was well known that higher order unification was some kind of equational unification for the theory $\beta\eta$. But standard first order equational unification algorithms could not be used for higher order unification, because substitution in higher order algebras is different from substitution in first order ones (grafting).

We have introduced and used the concepts based on the existence of two different kinds of variables: reduction and unification variables. By using substitution and grafting acting on these two kinds of variables, we have enabled the two calculi to interact explicitly. The precooking transformation has then allowed us to relate in a precise and faithful way the λ- and $\lambda\sigma$-calculi. Based on these ideas, we have shown that higher order unification problems could be translated into first order ones. We came up with a new algorithm for higher order unification based on a unification algorithm for the equational theory of $\lambda\sigma$. In our higher order unification algorithm, the separation between substitutions initiated by reduction and substitutions of unification variables permits an avoidance of the functional encoding of scoping constraints which was one of the burdens of previous algorithms. By using a language with explicit substitutions, our algorithm remains close to the one of Huet, in particular each of them can be simulated in terms of the other and the description of solutions are the same in both cases.

The fundamental role of the substitution calculus for unification has also been addressed in the theory of instantiation of [48] as well as in the unification algebra framework of [45]. One of the main differences between the present work and these approaches is that we consider a substitution calculus, entirely devoted to the evaluation of λ-calculus, as part of the framework and thus we design unification for this recent equational theory.

Another reduction of higher order unification in a first order framework has been given by [16] using combinatory logic and an extension of narrowing for handling extensionality. But this reduction inherits the defaults of the translation of λ-calculus on combinatory logic: coming back to λ-calculus is rather intricate.

We hope that the new framework we propose that allowed us to understand higher order unification as first order equational unification 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 [42]. This is a way to reduce higher order equational unification to first order [31], different from [17] which is based on combinatory logic. Also, we have shown in [20] that the framework could be fruitfully used for studying decidable subproblems of unification-like patterns [36].

The general concepts and algorithms presented in this paper can be adapted to any calculus of explicit substitutions provided that it is weakly terminating and confluent on substitution-closed terms. Nevertheless, determining the potential head variables in the rule **Exp-app** in a richer calculus such as that presented in [11] may be more tricky. With calculi that are only confluent on closed terms, narrowing can simply be used to perform unification [7]. But in the case of such a calculus, translating back solved forms to λ-calculus is rather intricate.

Our work has also to be carried with a precise analysis of the unification 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, and the experimental implementation [6] of the unification rules which has been realized using the ELAN logical framework [30], indeed uses only head normal forms. A major continuation of this work is its extension to unification in richer λ-calculi, such as the calculi of Barendregt's cube [5]. In this case, the functional expression of scoping constraints leads to technical difficulties [18] that may be simplified using explicit substitutions. Finally, this work suggests that higher order logic itself should be expressed using a calculus with explicit substitutions instead of ordinary λ-calculus. Then, higher order resolution is equational resolution in this theory. This program has been completed in [49, 50].

ACKNOWLEDGMENTS

This work has been supported partly by the French Inter-PRC operation "Mécanisation du raisonnement", the Esprit basic research actions Confer and Types and the Esprit working group CCL. We acknowledge helpful comments from Martin Abadi, Peter Borovansky, Daniel Briaud, César Muñoz, Alejandro Ríos, and anonymous referees.

Final manuscript received March 1, 1999

REFERENCES

1. Abadi, M., Cardelli, L., Curien, P. L., and Lévy, J. J. (1991), Explicit substitutions, *J. Functional Programming* **1**, 375–416.
2. Andrews, P. B. (1971), Resolution in type theory, *J. Symbolic Logic* **36**, 414–432.
3. Andrews, P. B. (1986), "An Introduction to Mathematical Logic and Type Theory: To Truth through Proof," Academic Press, New York.
4. Barendregt, H. P. (1984), The lambda-calculus, its syntax and semantics, *in* "Studies in Logic and the Foundation of Mathematics," second ed., Elsevier Science North-Holland, Amsterdam.
5. Barendregt, H. P. (1992), Lambda calculi with types, *in* "Handbook of Logic in Computer Science" (S. Abramsky, D. Gabbay, and T. Maibaum, Eds.), Clarendon Press, Oxford.
6. Borovanský, P. (1995), Implementation of higher-order unification based on calculus of explicit substitutions, *in* "Proceedings of the SOFSEM'95: Theory and Practice of Informatics" (M. Bartošek, J. Staudek, and J. Wiedermann, Eds.), Lecture Notes in Computer Science, Vol. 1012, pp. 363–368, Springer-Verlag, Berlin.
7. Briaud, D. (1996), Higher order unification as a typed narrowing, *in* "Proceedings of UNIF'96" (K. Schulz and S. Kepser, Eds.), CIS-Universität München, pp. 131–138. [CIS-Bericht-96-91]
8. Chen, H., Hsiang, J., and Kong, H. C. (1990), On finite representations of infinite sequences of terms, *in* "Proceedings of the 2nd International Workshop on Conditional and Typed Rewriting

Systems" (S. Kaplan and M. Okada, Eds.), Lecture Notes in Computer Science, Vol. 516, pp. 100–114, Springer-Verlag, Berlin.

9. Church, A. (1940), A formulation of the simple theory of types, *J. Symbolic Logic* **5**, 56–68.
10. Curien, P.-L. (1993), "Categorical Combinators, Sequential Algorithms and Functional Programming," 2nd ed., Birkhäuser, Basel.
11. Curien, P.-L., Hardin, T., and Lévy, J. J. (1996), Confluence properties of weak and strong calculi of explicit substitutions, *J. Assoc. Comput. Mach.* **43**, 362–397.
12. Curien, P.-L., and Rios, A. (1991), Un résultat de complétude pour les substitutions explicites, *C. R. Acad. Sci. Paris* **312**, 471–476.
13. Dauchet, M. (1992), Simulation of Turing machines by a regular rule, *Theoret. Comput. Sci.* **103**, 409–420.
14. de Bruijn, N. (1972), Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem, *Indag. Math.* **34**, 381–392.
15. Dershowitz, N., and Jouannaud, J.-P. (1990), Rewrite Systems, *in* "Handbook of Theoretical Computer Science" (J. van Leeuwen, Ed.), Chap. 6, pp. 244–320, Elsevier Science North-Holland, Amsterdam.
16. Dougherty, D. J. (1993), Higher-order unification via combinators, *Theoret. Comput. Sci.* **114**, 273–298.
17. Dougherty, D. J., and Johann, P. (1995), A combinatory logic approach to higher-order E-unification, *Theoret. Comput. Sci.* **139**, 207–242.
18. Dowek, G. (1993), A complete proof synthesis method for the cube of type systems, *J. Logic Comput.* **3**, 287–315.
19. Dowek, G., Hardin, T., and Kirchner, C. (1995), Higher-order unification via explicit substitutions, extended abstract, *in* "Proceedings of LICS'95, San Diego, June 1995" (D. Kozen, Ed.), pp. 366–374.
20. Dowek, G., Hardin, T., Kirchner, C., and Pfenning, F. (1996), Unification via explicit substitutions: The case of higher-order patterns, *in* "Proceedings of JICSLP'96, Bonn, September 1996" (M. Maher, Ed.), MIT Press, Cambridge, MA.
21. Goldfarb, D. (1981), The undecidability of the second order unification problem, *Theoret. Comput. Sci.* **13**, 225–230.
22. Goubault-Larrecq, J. (January 1997), "A Proof of Weak Termination of the Simply-Typed $\lambda\sigma$-Calculus," Technical Report 3090, INRIA.
23. Herbrand, J. (1930), Recherches sur la théorie de la démonstration, *Travaux Soc. Sci. Lett. Varsovie*, **Classe III**, 33.
24. Hintermeier, C., Kirchner, C., and Kirchner, H. (1994), Dynamically-typed computations for order-sorted equational presentations—extended abstract—, *in* "Proc. 21st International Colloquium on Automata, Languages, and Programming" (S. Abiteboul and E. Shamir, Eds.), Lecture Notes in Computer Science, Vol. 820, pp. 450–461, Springer-Verlag, Berlin.
25. Huet, G. (1973), The undecidability of unification in third order logic, *Inform. and Control* **22**, 257–267.
26. Huet, G. (1975), A unification algorithm for typed lambda calculus, *Theoret. Comput. Sci.* **1**, 27–57.
27. Hullot, J.-M. (1980), Canonical forms and unification, *in* "Proceedings 5th International Conference on Automated Deduction, Les Arcs, France" (W. Bibel and R. Kowalski, Eds.), Lecture Notes in Computer Science, Vol. 87, pp. 318–334, Springer-Verlag, Berlin.
28. Hussmann, H. (1985), Unification in conditional equational theories, *in* "Proceedings of the EUROCAL Conference, Linz, Austria" (B. Buchberger, Ed.), Lecture Notes in Computer Science, Vol. 204, pp. 543–553, Springer-Verlag, Berlin.
29. Jouannaud, J.-P., and Kirchner, C. (1991), Solving equations in abstract algebras: a rule-based survey of unification, *in* "Computational Logic. Essays in honor of Alan Robinson" (J.-L. Lassez and G. Plotkin, Eds.), Chap. 8, pp. 257–321, The MIT Press, Cambridge, MA.

30. Kirchner, C., Kirchner, H., and Vittek, M. (1995), Designing constraint logic programming languages using computational systems, *in* "Principles and Practice of Constraint Programming" (P. Van Hentenryck and V. Saraswat, Eds.), Chap. 8, pp. 131–158, The Newport Papers, The MIT Press, Cambridge, MA.

31. Kirchner, C., and Ringeissen, C. (1997), Higher-order equational unification via explicit substitutions, *in* "Algebraic and Logic Programming ALP-HOA," Lecture Notes in Computer Science, Vol. 1298, pp. 61–75, Springer-Verlag, Berlin.

32. Kirchner, H., and Hermann, M. (1990), Meta-rule synthesis from crossed rewrite systems, *in* "Proceedings 2nd International Workshop on Conditional and Typed Rewriting Systems, Montreal" (S. Kaplan and M. Okada, Eds.), Lecture Notes in Computer Science, Vol. 516, pp. 143–154, Springer-Verlag, Berlin.

33. Krivine, J.-L. (1993), "Lambda Calculus, Types and Models," Ellis Horwood, Chichester.

34. Martelli, A., and Montanari, U. (1982), An efficient unification algorithm, *ACM Trans. Programming Languages Systems* **4**, 258–282.

35. Middeldorp, A., and Hamoen, E. (1994), Completeness results for basic narrowing, *Appl. Algebra Eng. Commun. Comput.* **5**, 213–253.

36. Miller, D. (1991), A logic programming language with lambda-abstraction, function variables, and simple unification, *in* "Extensions of Logic Programming: International Workshop, Tübingen, Germany, December 1989" (P. Schroeder-Heister, Ed.), Lecture Notes in Computer Science, Vol. 475, pp. 253–281, Springer-Verlag, Berlin.

37. Miller, D. (1992), Unification under a mixed prefix, *J. Symbolic Comput.* **14**, 321–358.

38. Muñoz, C. (1997), A left linear variant of $\lambda\sigma$, *in* "Proceedings 6th International Joint Conference ALP'97-HOA'97, Southampton, UK," Lecture Notes in Computer Science, Vol. 1298, Springer-Verlag, Berlin.

39. Nadathur, G. (May 1996), "A Fine-Grained Notation for Lambda Terms and Its Use in Intensional Operations," Tech. Report TR-96-13, Department of Computer Science, University of Chicago.

40. Nadathur, G., and Wilson, D. S. (1990), A representation of lambda terms suitable for operations on their intensions, *in* "Proceedings of the 1990 ACM Conference on Lisp and Functional Programming," pp. 341–348, ACM Press, New York.

41. Nadathur, G., and Wilson, D. S. (January 1997), "A Notation for Lambda Terms: A Generalization of Environments," Tech. Report CS-1997-01, Department of Computer Science, Duke University.

42. Pagano, B. (1998), X.R.S.: explicit reduction systems, a first-order calculus for higher-order calculi, *in* "Conference on Automated Deduction" (C. Kirchner and H. Kirchner, Eds.), Lecture Notes in Artificial Intelligence, Vol. 1421, pp. 66–80.

43. Plotkin, G. (1972), Building-in equational theories, *Mach. Intelligence* **7**, 73–90.

44. Ríos, A. (1993), "Contributions à l'étude des λ-calculs avec des substitutions explicites," Thèse de Doctorat d'Université, U. Paris VII.

45. Siekmann, J., and Schmidt-Schauß, M. (1988), "Unification Algebras: an Axiomatic Approach to Unification, Equation Solving and Constraint Solving," SEKI report SR-88-23, Universität Kaiserslautern.

46. Snyder, W., and Gallier, J. (1989), Higher order unification revisited: Complete sets of tranformations, *J. Symbolic Comput.* **8**, 101–140. [Special Issue on Unification, Part Two]

47. Werner, A. (1995), Normalizing narrowing for weakly terminating and confluent systems, *in* "Proceedings of the first international conference on Principles and Practice of Constraint Programming, Cassis, France, September 1995" (U. Montanari and F. Rossi, Eds.), Lecture Notes in Computer Science, Vol. 976, pp. 415–430, Springer-Verlag, Berlin.

48. Williams, J. G. (1991), "Instanciation Theory. On the Foundations of Automated Deduction," Lecture Notes in Artificial Intelligence, Vol. 518, Springer-Verlag, Berlin.

49. Dowek, G., Hardin, T., and Kirchner, C. (1998), "Theorem Proving Module," Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique.

50. Dowek, G., Hardin, T., and Kirchner, C. (1999), HOL–lambda–sigma: An intensional first-order expression of higher-order logic, *in* "Rewriting Techniques and Applications" (P. Naremdran and M. Rusinowitch, Eds.), Lecture Notes in Computer Science, Vol. 1631, pp. 317–331, Springer-Verlag, Berlin.