Higher Order Unification via Explicit Substitutions

Gilles Dowek, Thérèse Hardin, Claude Kirchner

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Higher Order Unification via Explicit Substitutions

Gilles DOWEK , ThérŁse HARDIN , Claude KIRCHNER

N˚ 2709

Novembre 1995

PROGRAMME 2

ISSN 0249-6399

Higher Order Unification via Explicit Substitutions

Gilles DOWEK* , Thérèse HARDIN** , Claude KIRCHNER***

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projets COQ,PARA,PROTHEO

Rapport de recherche n2709 — Novembre 1995 — 42 pages

Abstract: Higher order unification is equational unification for $\beta\eta$-conversion. But it is not first order equational unification, as substitution has to avoid capture. 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 achived 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. The difference is at the variable level. 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 these two kinds of variables and such that reduction and grafting commute. For that, 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 a 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 specialized algorithm for a 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 step is decomposed in elementary ones, giving 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

(Résumé : tsvp)

*INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France. E-mail: Gilles.Dowek@inria.fr
**LITP, Université Paris 6, 4 Place Jussieu, 75252 Paris Cedex 05, France and INRIA-Rocquencourt. E-mail:Therese.Hardin@litp.ibp.fr
***INRIA Lorraine & CRIN, 615, rue du Jardin Botanique, BP 101, 54602 Villers-lès-Nancy Cedex, France. E-mail: Claude.Kirchner@loria.fr.

Unité de recherche INRIA Lorraine
Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : (33) 83 59 30 30 – Télécopie : (33) 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : (33) 87 20 35 00 – Télécopie : (33) 87 76 39 77

Unification d'Ordre Supérieur via les Substitutions Explicites

Résumé : L'unification d'ordre supérieur consiste à unifier modulo $\beta\eta$-conversion. Ce n'est donc pas de l'unification équationnelle modulo $\beta\eta$-conversion car les substitutions doivent éviter les captures. Les méthodes (comme la surréduction) permettant d'unifier modulo une théorie équationnelle sont basées sur l'opération de greffe (qui consiste à substituer sans renomer) et ne peuvent donc pas s'appliquer directement à l'unification d'ordre supérieur. Ce papier a pour but de réduire l'unification d'ordre supérieur à l'unification du premier ordre modulo une théorie équationnelle appropriée.

Ce résultat est obtenu en remplacant l'opération de substitution par l'opération de greffe, ce qui soulève deux problèmes. Tout d'abord, un problème d'unification peut avoir des solutions par greffe sans en avoir par substitution. Par ailleurs, les algorithmes d'unification équationelle reposent sur le fait que l'opération de greffe et la réduction commutent. Mais greffe et $\beta\eta$-réduction ne commutent pas dans le λ-calcul et la réduction d'une équation peut changer ses solutions. L'origine de cette difficulté provient de l'interaction entre les substitutions initialisées par la $\beta\eta$-réduction et celles initialisées par le processus d'unification. La difference est au niveau des variables. Deux types de variables sont impliqués: celle nécessaires à la $\beta\eta$-réduction et celles nécessaires pour l'unification. Par conséquent, nous devons élaborer un calcul qui distingue ces deux types de variables et pour lequel réduction et greffe commute. Pour ce faire, l'application d'une substitution à une variable de réduction dans une variable d'unification doit être retardée jusqu'à ce que cette variable soit instanciée. Cette séparation et ce délai nous sont fournis grâce à un calcul de substitutions explicites.

L'unification dans un tel calcul peut alors être réalisée par des algorithmes connus tels que la surréduction, mais nous présentons un algorithme spécialisé plus efficace.

Enfin nous montrons comment relier l'unification dans le λ-calcul avec l'unification équationnelle modulo un calcul avec substitutions explicites.

Nous aboutissons donc à un nouvel algorithme d'unification d'ordre supérieur qui élimine les lourdeurs des algorithmes antérieurs, en particulier en ce qui concerne la gestion des portées par la fonctionnalité. L'algorithme de Huet peut être obtenu comme une stratégie particulière du nôtre car toutes ses étapes peuvent être décomposées en étapes élémentaires, donnant ainsi une description plus atomique du processus d'unification. Enfin, les formes résolues d'un problème d'ordre supérieur s'obtiennent facilement à partir des formes résolues du problème calculées dans le calcul de substitutions explicites.

Mots-clé : Substitution explicite, unification d'ordre supérieur

Contents

Introduction

In the original presentation of higher order logic, β and η-conversion were defined as axioms of the theory [Chu40, And86]. The use of these axioms in a proof searching method leads to replacing 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, $\beta\eta$-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 [And71]: this is the so-called higher order unification, or unification in simply typed λ-calculus [Hue75].

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 [Plo72] 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, if we consider a problem where a variable X occurs and θ is a solution to this problem, the normal form of the term θX is the term:

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

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 we do in 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:

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

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:

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

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. Secondly, 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 internalization of the substitution calculus was already required for describing implementations of λ-calculi and has motivated the development of $\lambda\sigma$-calculus [ACCL91, CHLar] 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 keep names instead.

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 internalized in the expression of the problem. We define a translation (that we call pre-cooking) of λ-calculus into $\lambda\sigma$-calculus such that a problem $a =^?_{\beta\eta} b$ 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 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 optimized narrowing.

Last we show that in contrast with the reduction of higher order unification to equational unification in a combinatory language [Dou93], the unification algorithm for $\lambda\sigma$ can simulate the algorithm of [Hue75], i.e. 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 can in fact be skipped.

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 type information in the calculi and we make explicit a first order presentation of the typed $\lambda\sigma$-calculus. After introducing the standard notions of equational unification and the appropriate notion of solved forms, section 3 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. The last section then shows how the previous unification procedure can be applied to solve the problem of higher order unification thanks to the "pre-cooking" transformation. The paper ends with detailled examples and conclusions.

This paper is the full version, including full motivations and proofs, of the extended abstract which appeared in the LICS'95 proceedings [DHK95]. We have chosen to make this version quite detailled since we believe 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 $\lambda\sigma$-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 emphasize 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 $X, Y, \ldots$) and a set $\mathcal{F}$ of operators. The set of variables of a term a is denoted $\mathcal{V}ar(a)$.

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 $a_1, \ldots, a_k \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $f \in \mathcal{F}$, then $\bar{\theta}(f(a_1, \ldots, a_k)) = f(\bar{\theta}(a_1), \ldots, \bar{\theta}(a_k))$.

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

$$\mathcal{D}om(\theta) = \{x | x \in \mathcal{X} \text{ and } \theta(x) \neq x\},$$

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

$$\mathcal{R}an(\theta) = \bigcup_{x \in \mathcal{D}om(\theta)} \mathcal{V}ar(\theta(x)).$$

The set of all variables involved in θ is $\mathcal{V}ar(\theta) = \mathcal{D}om(\theta) \cup \mathcal{R}an(\theta)$.

Notation: Let θ be a valuation binding the variables $X_1, \ldots, X_n$ to the terms $a_1, \ldots, a_n$, we write the grafting associated to θ: $\bar{\theta} = \{X_1 \mapsto a_1, \ldots, X_n \mapsto a_n\}$. 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 then the infix notation for the term in prefix notation $\_\ \_(a,b)$) and an infinite number of unary operators called abstractors $\lambda x.\_$ indexed on $\mathcal{V}$.

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 $\mathcal{T}(\{\lambda x.\_, \_\ \_\}, \mathcal{V})$. 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 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 uncorrect. 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, in the spirit of [Bar84]. For this purpose, we first need to introduce the appropriate notion of renaming:

Definition 1.2 Let V be a set of variables. The application α^V, structurally defined as follows:

1. $\alpha^V(x) = x$,
2. $\alpha^V(a\ b) = (\alpha^V(a)\ \alpha^V(b))$,

3. $\alpha^V(\lambda x.a) = \lambda x.\alpha^V(a)$ if $x \notin V$,
4. $\alpha^V(\lambda x.a) = \lambda y.\{x \mapsto y\}\alpha^V(a)$ if $x \in V$, where y is a "fresh" variable, i.e. not occurring in a nor in V,

renames consistently all the bound variables of a λ-term outside V.

We can now define the usual substitution operation:

Definition 1.3 Let $a \in \Lambda(\mathcal{V})$. The set of *free variables* of a is written $FV(a)$. For a valuation θ binding the variables $x_1, \ldots, x_n$ to the terms $a_1, \ldots, a_n$, the *substitution* extending θ and written $\overline{\theta} = \{x_1/a_1, \ldots, x_n/a_n\}$, is defined by:

1. $\overline{\theta}x = \theta x$,
2. $\overline{\theta}(a\ b) = (\overline{\theta}a\ \overline{\theta}b)$,
3. $\overline{\theta}(\lambda y.a) = \lambda z.(\overline{\theta}\{y \mapsto z\}\alpha^{\{y\}\cup\mathcal{V}ar(\theta)}(a))$ where z is a fresh variable, i.e. a variable such that $\theta z = z$, z does not occur in a and for every $x \in FV(a)$, $z \notin FV(\theta x)$.

Bound variable renaming (α-conversion) has to be done before performing the substitution under a λ. This is combined with the grafting 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 [Kri93]).

Notice that $\{x_1/a_1, \ldots, x_n/a_n\}$ is the simultaneous substitution of the variables x_1, ..., x_n by the terms $a_1, \ldots, a_n$ and not the composition of $\{x_1/a_1\}, \ldots, \{x_n/a_n\}$. Indeed:

$$\{x/y, y/x\}(f\ x\ y) = (f\ y\ x) \text{ and } \{x/y\}\{y/x\}(f\ x\ y) = (f\ y\ y).$$

Definition 1.4 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:

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

1.3 Unification variables

At a 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 be either mere elements of $\mathcal{V}$ or be considered as metavariables, i.e. external to the β-process. If we take the first choice (see [Hue75]), the definition 1.3 applies directly, but we need to keep at a meta-level a distinction between constants and true unification variables during the unification process.

Here we favor the second choice, i.e. we have two syntactical categories, a first one for bound variables and constants represented by $\mathcal{V}$ and a second one for unification variables, called metavariables and represented by $\mathcal{X}$. Although this choice is not the more economical, it eases the understanding of higher order unification as equational unification. A third choice can be envisaged using the substitution variables of the $\lambda\sigma$-calculus, we detailed this in the Section 4.3.

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

Definition 1.5 $\Lambda(\mathcal{V}, \mathcal{X})$, the set of open λ-terms, is inductively defined as:

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

where $x, v \in \mathcal{V}$ and $X \in \mathcal{X}$.

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

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

$$\theta(X) = X \text{ if } X \in \mathcal{X}.$$

Let us turn now to the substitution of meta-variables. Is it possible to reduce it to a $\mathcal{X}$-grafting? With the separation of name spaces, the first drawback of $\mathcal{X}$-grafting disappears: variables bound by a λ are elements of $\mathcal{V}$, thus they remain unchanged by an $\mathcal{X}$-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 1.6 Let θ be a valuation (i.e. a function from $\mathcal{X}$ to $\Lambda(\mathcal{V}, \mathcal{X})$), the *substitution* written $\bar{\theta}$ is the extension of the valuation such that:

1. $\bar{\theta}(X) = \theta(X)$,
2. $\bar{\theta}(v) = v$ if $v \in \mathcal{V}$,
3. $\bar{\theta}(a\ b) = (\bar{\theta}(a)\ \bar{\theta}(b))$,
4. $\bar{\theta}(\lambda y.a) = \lambda z.\bar{\theta}(\{y \mapsto z\}\alpha^{\{y\}\cup\mathcal{V}ar(\theta)}(a))$ 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 1.1 Substitution and reduction commute.

Proof: For β-reduction we have $((\lambda x.a)\ b) \to^{\beta} \{x/b\}a$ and $\theta((\lambda x.a)\ b) = (\theta(\lambda x.a)\ \theta b) = (\lambda z.\theta(\{x \mapsto z\}\alpha^{\{x\}\cup\mathcal{V}ar(\theta)}(a))\ \theta b) \to^{\beta} \{z/\theta(b)\}(\theta(\{x \mapsto z\}\alpha^{\{x\}\cup\mathcal{V}ar(\theta)}(a))$. So we have to prove the equality: $\theta(\{x/b\}a) = \{z/\theta(b)\}(\theta(\{x \mapsto z\}\alpha^{\{x\}\cup\mathcal{V}ar(\theta)}(a))$ but this is an application of the *substitution lemma* [Bar84] on commutation of substitutions.

For η-reduction we have $\lambda x.(a\ x) \to a$ and $\theta(\lambda x.(a\ x)) = \lambda z.\theta(\{x \mapsto z\}\alpha^{\{x\}\cup\mathcal{V}ar(\theta)}(a\ x)) = \lambda z.\theta(\{x \mapsto z\}\alpha^{\{x\}\cup\mathcal{V}ar(\theta)}(a)\ z)$ 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 names management may be avoided with another formulation of λ-calculus: the de Bruijn notation [dB72]. We give here a presentation slightly different from the original one, as we add metavariables.

The intuitive idea of 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 $\Lambda_{DB}(\mathcal{X})$ 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 or equal to 1[1] and $X \in \mathcal{X}$.

Notice 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.

[1] Some authors use 0 as the first de Bruijn number, see [Cur93] for example.

Definition 1.8 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 a. If $\mathtt{p} \leq |u|$, then $\mathtt{p}$ is said *bound* in a, otherwise it is a *free* number of a.

In the original notation of de Bruijn, free variables of a term a are ordered into a list $(x_0 \ldots x_n)$, called a *referential* and written $\mathcal{R}$. 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 $\lambda x_0 \ldots \lambda x_n.a$. Here only $\mathcal{V}$-variables (i.e. bound variables and constants, but not unification variables) are recorded in the referential $\mathcal{R}$.

Definition 1.9 Let $\mathcal{R}$ be a referential. Let $a \in \Lambda(\mathcal{V}, \mathcal{X})$ such that all the free $\mathcal{V}$-variables of a are declared in $\mathcal{R}$. The de Bruijn translation of a, written $tr(a, \mathcal{R})$, is defined by:

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

So, during the de Bruijn translation, the referential is incremented when crossing a λ. For example, a closed λ-term (i.e. without free $\mathcal{V}$-variables) may be translated in de Bruijn notation, according to an empty referential. Note that this translation does not correspond to a bijection between $\mathcal{V}$ 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 $\lambda\, (\lambda\, (\mathtt{2}\ (\lambda\, \mathtt{1}\ \mathtt{3})\ \mathtt{1}))$. Now, let u be an occurrence of a number $\mathtt{p}$ in a term $tr(a, \mathcal{R})$. If $\mathtt{p} \leq |u|$, $\mathtt{p}$ is an occurrence of a $\mathcal{V}$-variable which is bound in a, otherwise it represents the constant, 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 $\mathcal{V}$.

Suppose that $(\lambda\, a)\ b)$ is expressed in a referential $\mathcal{R}$, then b is also expressed in $\mathcal{R}$ but a is expressed in $x.\mathcal{R}$, where x is a name for the variable bound by the λ. The term $a\{\mathtt{1}/b\}$ is expressed in $\mathcal{R}$ and is obtained from a by replacing the indices referring to x in $x.\mathcal{R}$ 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 $\mathcal{R}$ have still to refer to the same variable. Therefore free indices in b must be incremented by $|u|$. Also, as a is expressed in $x.\mathcal{R}$ and $a\{\mathtt{1}/b\}$ is expressed in $\mathcal{R}$, all the free indices of a, which do not refer to the first variable of $x.\mathcal{R}$, 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 $a \in \Lambda_{DB}(\mathcal{X})$. The term a^+, called *lift* of a, is defined by $a^+ = \mathrm{lft}(a, 0)$ where $\mathrm{lft}(a, i)$ is inductively defined by:

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

Lemma 1.1 Let a be a term of $\Lambda(\mathcal{V}, \mathcal{X})$, expressed in a referential $\mathcal{R}$. Let v be any variable of $\mathcal{V}$, not belonging to $\mathcal{R}$. Then, $tr(a, \mathcal{R})^+ = tr(a, v.\mathcal{R})$

Proof: It suffices to prove the following equality:

$$\mathrm{lft}(tr(a, x_1 \ldots x_i.\mathcal{R}), i) = tr(a, x_1 \ldots x_i.v \cdot \mathcal{R})$$

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

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

Definition 1.11 The substitution by b at the λ-height $(n-1)$ in a, written $\{\mathtt{n}/b\}a$ is defined by induction as follows:

$$\begin{array}{lcll}
\{\mathtt{n}/b\}(a_1\ a_2) & = & (\{\mathtt{n}/b\}a_1\ \{\mathtt{n}/b\}a_2) & \\
\{\mathtt{n}/b\}X & = & X & \\
\{\mathtt{n}/b\}\,\lambda\,a & = & \lambda\,(\{\mathtt{n+1}/b^+\}a) & \\
\{\mathtt{n}/b\}\mathtt{m} & = & \mathtt{m}-\mathtt{1} \quad \text{if} \quad m > n & (\mathtt{m} \in FV(a)) \\
 & & b \quad \text{if} \quad m = n & (\ \mathtt{m}\ \text{bound by the}\ \lambda\ \text{of the Beta-redex}) \\
 & & \mathtt{m} \quad \text{if} \quad m < n & (\mathtt{m} \in BV(a))
\end{array}$$

Definition 1.12 The β-reduction is defined by:

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

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 $\mathtt{n}+\mathtt{1}$ 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 unclear. The only simultaneous substitution which can be easily defined is the substitution of an initial segment of the natural numbers $\{\mathtt{1}/a_1,\mathtt{2}/a_2,\ldots,\mathtt{n}/a_n\}$. In this case all the other indices in the term have to be decremented by $\mathtt{n}$. Notice that such a substitution is better represented as a list $a_1,\ldots,a_n$ 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 $\mathcal{R}$ be a referential containing the constants of a. As a is coded by $tr(a,\mathcal{R})$ and $\lambda\,x.(a\ x)$ by $tr(\lambda x.(a\ x),\mathcal{R}) = \lambda(tr(a,x\cdot\mathcal{R})\ \mathtt{1}) = \lambda(tr(a,\mathcal{R})^+\ \mathtt{1})$, we get the following definition for η:

Definition 1.13 The η-reduction in $\Lambda_{DB}(\mathcal{X})$ is defined by the rule:

$$\lambda\,(a\ \mathtt{1}) \ \rightarrow b \text{ if } \exists b \in \Lambda_{DB}(\mathcal{X}) \text{ such that } a = b^+.$$

Proposition 1.2 Let $a \in \Lambda_{DB}(\mathcal{X})$, 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^+: $\mathrm{lft}(\mathtt{m},i)$ is never equal to $\mathtt{i}+\mathtt{1}$. For the only if part, it suffices to build, from a, the term b by replacing any free number $\mathtt{n}$ of a by $\mathtt{n}-\mathtt{1}$. □

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

Definition 1.14 Let θ be a valuation from $\mathcal{X}$ to $\Lambda_{DB}(\mathcal{X})$, 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 1.3 Let $a \in \Lambda(\mathcal{V},\mathcal{X})$ and $\theta = \{X_1/a_1,\ldots,X_n/a_n\}$ be a term and a substitution of the λ-calculus with names. Let $\theta' = \{X_1/tr(a_1,\mathcal{R}),\ldots,X_n/tr(a_n,\mathcal{R})\}$ then, $tr(\theta(a),\mathcal{R}) = \theta'(tr(a,\mathcal{R}))$.

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

$$\begin{array}{rcl} tr(\{X/c\}(\lambda\, v.b), \mathcal{R}) & = & tr(\lambda\, w.\{X/c\}\{v \mapsto w\}\alpha^{\{v,w\}\cup\mathcal{V}ar(c)}(b), \mathcal{R}) \\ & = & \lambda\,(tr(\{X/c\}\{v \mapsto w\}\alpha^{\{v,w\}\cup\mathcal{V}ar(c)}(b), w.\mathcal{R})) \\ \{X/tr(c,\mathcal{R})\}tr(\lambda\, v.b, \mathcal{R}) & = & \{X/tr(c,\mathcal{R})\}(\lambda\,(tr(b, v.\mathcal{R}))) \\ & = & \lambda\,(\{X/tr(c,\mathcal{R})^+\}(tr(b, v.\mathcal{R}))) \end{array}$$

We have $tr(b, v.\mathcal{R}) = tr(\{v \mapsto w\}\alpha^{\{v,w\}\cup\mathcal{V}ar(c)}(b), w.\mathcal{R})$ and, as w is a fresh variable, $tr(c,\mathcal{R})^+ = tr(c, w.\mathcal{R})$. So, we can apply the induction hypothesis, which allows to conclude as follows:

$$\begin{array}{l} \{X/tr(c, w.\mathcal{R})\}(tr(\{v \mapsto w\}\alpha^{\{v,w\}\cup\mathcal{V}ar(c)}(b), w.\mathcal{R})) = \\ tr(\{X/c\}(\{v \mapsto w\}\alpha^{\{v,w\}\cup\mathcal{V}ar(c)}(b)), w.R). \end{array}$$

□

1.5 $\lambda\sigma$-calculus

The $\lambda\sigma$-calculi [ACCL91, CHLar] 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 [ACCL91] and we try to give an intuitive presentation of it in the following.

Internalizing operations as operators Let us consider an algebra $\mathcal{A}$ 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 internalizing the operation F and rules rewriting expressions containing the symbol f into other expressions that eventually do not contain the symbol f. For instance:

$$f(0) \to 0$$

$$f(S(X))) \to S(S(f(X))).$$

It can be shown that any computable function can be expressed this way, even using a linear regular lonesome rule [Dau92].

λ-calculus is such an internalization 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 internalizes operations only half-way. Indeed application is internalized, but substitution and variable renaming (or lifting) are still external operations. $\lambda\sigma$-calculus goes one step further by internalizing also substitutions and lifting.

A First Try When internalizing the operation $\{n/b\}a$ as a term $a[n/b]$ 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 internalizing the operation lft. Then, introducing meta-variables makes this system non confluent. Indeed:

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

and:

$$((\lambda((\lambda X)\ Y))\ Z) \to ((\lambda X)\ Y)[1/Z] \to (\lambda X[2/Z\uparrow]\ Y[1/Z]) \to X[2/Z\uparrow][1/Y[1/Z]].$$

The term $\lambda((\lambda X)\ Y)\ Z$ reduces to $X[1/Y][1/Z]$ and to $X[2/Z\uparrow][1/Y[1/Z]]$ 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 $a_1.a_2.\ldots.a_n.id$ 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 internalizes the lifting operator a^+, can be seen as the infinite simultaneous substitution $\mathtt{2.3.4}.\ldots$. We also introduce a composition operator $\circ$. Finally, the index $\mathtt{n+1}$ is often written $\mathtt{1}[\uparrow \circ \ldots \circ \uparrow]$ 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 [ACCL91] and [CHLar].

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

Definition 1.15 Let $\mathcal{X}$ be a set of term metavariables, and $\mathcal{Y}$ be a set of substitution metavariables. The set $\mathcal{T}_{\lambda\sigma}(\mathcal{X}, \mathcal{Y})$ 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 $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$.

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

$\mathtt{1}$	:	$\rightharpoonup$ `term`
$(\_\ \_)$	: `term term`	$\rightharpoonup$ `term`
$(\lambda\ \_)$	: `term`	$\rightharpoonup$ `term`
$\_[\_]$	: `term substitution`	$\rightharpoonup$ `term`
id	:	$\rightharpoonup$ `substitution`
$\uparrow$	:	$\rightharpoonup$ `substitution`
$\_.\_$	: `term substitution`	$\rightharpoonup$ `substitution`
$\_ \circ \_$	: `substitution substitution`	$\rightharpoonup$ `substitution`

Notice 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 sort information using the arrow $\rightharpoonup$. The set of variables of a term a of sort `term` is denoted by $\mathcal{TV}ar(a)$.

Except if explicitly mentioned, we assume in all the paper that all the terms considered do not contain any substitution variable.

Notation: Terms like $((((a\ a_1)\ a_2)\ldots)\ a_n)$ are written as usual $(a\ a_1\ \ldots\ a_n)$.

The $\lambda\sigma$-calculus is defined as the term rewriting system defined in Figure 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 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 schematization community [KH90, CHK90]), so we rather express it as the conditional rewrite rule **Eta**.

Remark: Notice if $a = b[\uparrow]$ then $b = a[c.id]$ where c can be any term. In the following, c is assumed to be a fresh variable.

The main properties of $\boldsymbol{\lambda\sigma}$ are:

1. The term rewriting system $\boldsymbol{\lambda\sigma}$ is locally confluent on any $\lambda\sigma$-term, open or closed [ACCL91].

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

Figure 1: $\boldsymbol{\lambda\sigma}$ – The $\lambda\sigma$-term rewriting system

2. $\boldsymbol{\lambda\sigma}$ is confluent on substitution-closed terms (i.e. on terms without substitution variable) [Río93].

3. $\boldsymbol{\lambda\sigma}$ is *not* confluent on open terms (i.e. terms with term and substitution variables) [CHLar].

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

Proposition 1.4 [Río93] Any $\lambda\sigma$-term in normal form for $\boldsymbol{\lambda\sigma}$ is of one of the following forms:

1. $\lambda\, a$,

2. $(a\ b_1 \ldots b_n)$, where 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. $a_1 \ldots a_p \cdot \uparrow^n$, where $a_1, \ldots, a_p$ are normal terms 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 $\mathcal{V}$ (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. Notice that, in particular, the condition $a =_\sigma b[\uparrow]$ is stronger than $a = b^+$ as $X = X^+$ but there exists no term b such that $X =_\sigma b[\uparrow]$.

The fact that the application of a substitution to a metavariable can be suspended until the metavariable is instantiated will be used to code substitution of variables in $\mathcal{X}$ by $\mathcal{X}$-grafting and explicit lifting. Thus a notion of $\mathcal{X}$-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 1.5 $\mathcal{X}$-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 $A_1.A_2.\ldots.A_n.nil$ associates the type A_i to the index i.

Definition 2.1 The syntax of simply typed λ-calculus using de Bruijn indices is:

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

and the typing rules are:

$$(var1) \qquad A.\Gamma \vdash \mathtt{1} : A$$

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

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

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

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:

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

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

Proposition 2.1 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:

$$\text{If } C_i \ldots C_1.\Gamma \vdash b : B \text{ then } C_i \ldots C_1.C.\Gamma \vdash \mathrm{lft}(b, i) : B.$$

This is done by induction on the structure of b.

(a) If $b = X$. Then $\mathrm{lft}(b, i) = X$.

(b) If $b = \mathtt{n}$. If $n > i$ then the type of $\mathtt{n}$ in $C_i \ldots C_1.\Gamma$ is the one of $\mathtt{n-i}$ in Γ. The type of $\mathrm{lft}(\mathtt{n}, i) = \mathtt{n+1}$ is the one of $\mathtt{n+1-(i+1)}$ in Γ. If $n \leq i$ 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 $D.C_i \ldots C_1.\Gamma \vdash c : E$. By induction hypothesis $D.C_i \ldots C_1.C.\Gamma \vdash \mathrm{lft}(c, i+1) : E$ thus $C_i \ldots C_1.C.\Gamma \vdash \mathrm{lft}(\lambda_D.c, i) : C$.

□

2.2 Typed $\lambda\sigma$-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 $s \triangleright \Gamma$ to express the fact that the substitution s has type Γ.

Definition 2.2 [CR91, ACCL91] Syntax:

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

Typing rules

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

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

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

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

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

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

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

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

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

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

and:

$$nat.nil \vdash id \triangleright nat.nil$$

thus:

$$nat.nil \vdash \mathtt{1}.\ id \triangleright nat.nat.nil.$$

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 $\boldsymbol{\lambda\sigma}$ the relevant typing informations. Below are the rules from $\boldsymbol{\lambda\sigma}$ where type information have been added, the other rules are unchanged:

Beta	$(\lambda_A\, a)b$	$\rightarrow$	$a[b.id]$
Abs	$(\lambda_A\, a)[s]$	$\rightarrow$	$\lambda_A(a[1.(s \circ \uparrow)])$

Notice that the typed version of σ has the same properties (termination and confluence) than the untyped one. One can also clearly type the **eta** rule. The resulting term rewriting system is also called $\boldsymbol{\lambda\sigma}$.

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, to type metavariables.

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

Proposition 2.2 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 $\Delta \vdash s \triangleright \Delta'$ we have $\Delta \vdash \{X \mapsto b\}s \triangleright \Delta'$.

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

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 $b = (Y\ \lambda X\ X)$ can be typed in any context. Indeed if for instance, a were typable 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 2.3 [Río93] This typed $\lambda\sigma$ calculus is weakly normalizing and confluent.

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

Definition 2.3 (η-long normal form) Let a be a $\lambda\sigma$-term of type $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$ 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 $a = (\mathtt{k}\ b_1 \ldots b_p)$ then $a' = \lambda_{A_1} \ldots \lambda_{A_n}(\mathtt{k+n}\ c_1 \ldots c_p\ \mathtt{n'}\ \ldots \mathtt{1'})$, where c_i is the η-long normal form of the normal form of $b_i[\uparrow^n]$.
3. If $a = (X[s]\ b_1 \ldots b_p)$ then $a' = \lambda_{A_1} \ldots \lambda_{A_n}(X[s']\ c_1 \ldots c_p\ \mathtt{n'}\ \ldots \mathtt{1'})$, where c_i is the η-long normal form of the normal form of $b_i[\uparrow^n]$ and if $s = d_1 \ldots d_q.\ \uparrow^k$ then $s' = e_1 \ldots e_q.\ \uparrow^{k+n}$ where e_i is the η-long form of $d_i[\uparrow^n]$.

This definition is well-founded as we are now proving using an induction based on the lexicographic ordering on the 3-uple 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. $|(n\ a_1 \ldots a_p)| = 1 + |a_1| + \ldots + |a_p|$
3. $|(X[s]\ a_1 \ldots a_n)| = 1 + |a_1| + \ldots + |a_n|$

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

Proof: By induction on the structure of a. □

Lemma 2.2 The definition 2.3 is correct and well-founded.

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

Definition 2.4 The *long normal form* of a term is the η-long form of its $\beta\eta$-normal form.

Proposition 2.4 Two terms are $\beta\eta$-equivalent if and only if they have the same long normal form.

Remark: Notice that in $\lambda\sigma$-calculus, the reduction of a η-redex may create a σ-redex. For instance, the term $X[\lambda\ (2\ 1).\uparrow]$ reduces to $X[1.\uparrow]$ 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.

Notice also that substitutions also could be written in extensional forms, by using a kind of surjective pairing id could be rewritten in $1.\uparrow$ when type permits. But we do not make these expansions in this work.

2.3 A first order specification of typed $\lambda\sigma$-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 only sense 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 like syntax.

```
module SortLangage [BaseType]
sort Type Context TermSort SubstitutionSort;
op
 _    : BaseType          → Type;
 →    : Type Type         → Type;
 nil :                    → Context;
 _._ : Type Context       → Context;
 ⊢    : Context Type      → TermSort;
 ⊢    : Context Context   → SubstitutionSort;
end of module
```

We use the elements in `TermSort` and `SubstitutionSort` for sorting our $\lambda\sigma$-terms. Notice 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 $\mathtt{1}, \_\ \_, \lambda, \_[\_], id, \uparrow, ., \circ$ by the appropriate type and context. We assign to each variable a sort $\Gamma \vdash A$. Notice 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 the definition 2.2, but here we simply use the usual rules for typing many-sorted terms with the assigned ranks.

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

A sort $\Gamma \vdash A$ is naturally interpreted as the set of all the ground terms that have type A in the context Γ. Notice 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 formalize the same first order signature by using a polymorphic order-sorted signature as in [HKK94], 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 $(1_{A\to B}^\Gamma \quad 1_A^\Gamma)$ 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 $(1_{A\to B}^{A.\Gamma} \quad 1_A^\Gamma[\uparrow_{A\to B}^{A.\Gamma}])$ 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 2.5 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 typing derivation. □

Proposition 2.6 The sorted reduction system is weakly normalizing and confluent.

3 Unification in the $\lambda\sigma$-calculus

We are now considering the problem of solving equation systems on *typed* $\lambda\sigma$-terms modulo the equational theory $\boldsymbol{\lambda\sigma}$. We are restricting the class of problems that we are solving to those consisting only of terms build on substitution-closed $\lambda\sigma$-terms. This is due to a technical fact: $\boldsymbol{\lambda\sigma}$ is not confluent on terms with substitution variables. Moreover this restriction is enough for our purpose, as our final goal is to provide an alternative tool for unification in the λ-calculus.

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 normalizing 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 $\boldsymbol{\lambda\sigma}$ to perform (conditional) narrowing or better basic conditional narrowing [Hul80, Hus85, MH94, Wer95]. However, we shall design a much more efficient algorithm using properties of the rewriting system $\boldsymbol{\lambda\sigma}$.

3.1 Equational unification

Let us first recall briefly the unification notions we will need in the following. For details see [JK91].

Definition 3.1 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 nor universal quantifier whose atoms are $\mathbb{T}, \mathbb{F}$ 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:

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

When $|J| = 1$ the problem is called a *system*. Variables $\overrightarrow{w}$ in a system $P = \exists \overrightarrow{w} \bigwedge_{i \in I} s_i =^?_{\mathcal{A}} t_i$ 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 3.2 A *$\mathcal{A}$-unifier* of an $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification system:

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

is a grafting σ such that:

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

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

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

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

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

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

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

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

In order to find 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 schematas. For example, considering a given signature $\mathcal{F} = \{f, g, a\}$, the transformation rule:

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

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 Figure 2, where the connectors $\vee$ and $\wedge$ are assumed to be associative and commutative and where $\mathbb{F}$ denotes a unification problem without solution (the empty disjunction) and $\mathbb{T}$ 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 [Her30, MM82] and is fully developed in [JK91]. We can summarize it as follows. First choose the intended solved forms that characterize which equational problems actually correspond to most general unifiers or more generally to the desired simplified form of equational problems (think to 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 [KKV95].

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

Figure 2: **Simplif**, simplification rules for unification problems

3.2 Transformation rules for $\lambda\sigma$-unification

The standard unification results are not directly applicable since on one hand the many-sorted term rewriting system consists in an infinite number of rules (see Section 2.3) and on the other hand the term rewriting system $\boldsymbol{\lambda\sigma}$ is not strongly normalizing [Mel95]. However it is weakly normalizing and the technique from [Wer95] could be used 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.

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

Definition 3.4 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 $\mathcal{TV}ar(P)$. 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 normalized. This is done by the rule **Normalize** presented in the set of transformations rules **Unif** given in Figure 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-$\boldsymbol{\lambda}$**. In the same way, an equation $(\mathtt{n}\ a_1\ \ldots\ a_p) =^?_{\lambda\sigma} (\mathtt{n}\ b_1\ \ldots\ b_p)$ can be simplified to $a_1 =^?_{\lambda\sigma} b_1, \ldots, a_p =^?_{\lambda\sigma} b_p$ by the rule **Dec-App** and an equation of the form $(\mathtt{n}\ a_1\ \ldots\ a_p) =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q)$ with $n \neq m$ can be simplified to the unsatisfiable problem $\mathbb{F}$ by the rule **Dec-Fail**, as it has no solution.

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

When the type of a variable X is $A \rightarrow B$, a step towards the solution $\{X \mapsto \lambda a\}$ is done by performing, using the rule **Exp-$\boldsymbol{\lambda}$**, 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 $A \rightarrow A$ 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 $\{Y \mapsto (Z[s]\ a_1\ \ldots\ a_k)\}$ or $\{Y \mapsto (\mathtt{n}\ a_1\ \ldots\ a_k)\}$. A grafting of the form $\{Y \mapsto (Z[s]\ a_1\ \ldots\ a_k)\}$ is obviously not

a solution, as the normal form of the term $(Z[s]\ a_1\ \ldots\ a_k)[\mathtt{1}.id]$ is not $\mathtt{1}$. Thus all the solutions are of the form $\{Y \mapsto (\mathtt{n}\ a_1\ \ldots\ a_k)\}$. A step towards such a solution is done by performing the grafting $\{Y \mapsto (\mathtt{n}\ H_1\ \ldots\ H_k)\}$ where $H_1, \ldots, H_k$ 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 $(\mathtt{n}\ H_1\ \ldots\ H_k)[1.id]$ would be $\mathtt{n}-1$ and thus different from $\mathtt{1}$.

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

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 to describe the solutions of unification problems as problems in solved forms and to let the unification rules be transformation rules preserving the solutions.

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

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

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

Solved: $X =^?_{\lambda\sigma} a$ where the variable X does not 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 solved.

Flex-flex: $X[a_1 \ldots a_p.\ \uparrow^n] =^?_{\lambda\sigma} Y[a'_1 \ldots a'_{p'}.\ \uparrow^{n'}]$, where $X[a_1 \ldots a_p.\ \uparrow^n]$ and $Y[a'_1 \ldots a'_{p'}.\ \uparrow^{n'}]$ are long normal terms, the type of the variables X and Y is atomic and the equation is not solved.

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

Lemma 3.1 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 flex-flex equations. Consider such an equation $X[a_1 \ldots a_p.\ \uparrow^n] =^?_{\lambda\sigma} Y[a'_1 \ldots a'_{p'}.\ \uparrow^{n'}]$ of type T in a context Δ. We have:

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

Thus $\Gamma_X = A_1 \ldots A_p.\Gamma$ and $\Delta = B_1 \ldots B_n.\Gamma$ 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 non solved variable X of type T to the term $Z_T[\uparrow^{|\Gamma_X|}]$.

When we apply this grafting to the flex-flex 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 the section 4 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 Figure 3. Additional unification rules given in Figure 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 Figure 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 [JK91], prove the following results:

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

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

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

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

Exp-App $P \wedge X[a_1 \ldots a_p.\uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q)$
$\rightarrow$
$P \quad \wedge X[a_1 \ldots a_p.\uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q)$
$\quad \wedge \bigvee_{r \in R_p \cup R_i} \exists H_1, \ldots, H_k,\ X =^?_{\lambda\sigma} (\mathtt{r}\ H_1\ \ldots\ H_k)$
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 $P \wedge X =^?_{\lambda\sigma} a$
$\rightarrow$
$\{X \mapsto a\}(P) \wedge X =^?_{\lambda\sigma} a$
if $X \in \mathcal{TV}ar(P), X \notin \mathcal{TV}ar(a)$ and $a \in \mathcal{X} \Rightarrow a \in \mathcal{TV}ar(P)$

Normalize $P \wedge a =^?_{\lambda\sigma} b$
$\rightarrow$
$P \wedge a' =^?_{\lambda\sigma} b'$
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

Figure 3: **Unif**, the basic rules for unification in $\lambda\sigma$

- 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 3.2 The transformation by the rules in **Unif** of a well typed equation gives rise only to well typed equations, $\mathbb{T}$ and $\mathbb{F}$.

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

Lemma 3.3 Any solved problem is normalized for **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 non solved 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 some where else in P, in which case **Replace** applies.
If a or a' are not in long normal form, then we apply the rule **Normalize**.
The other cases where a and a' are long normal terms — which allows to infer that if a is not a λ-abstraction, then it has an atomic type — are summarized in the following table, using the result of Proposition 1.4 to describe the form of the terms involved and assuming that $\vec{b}$ and $\vec{b'}$ consists both in at least one element. Notice 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**
$X'[s']$				Flex-Flex	Flex-Flex or **Replace**
X'					**Replace**

□

Lemma 3.4 Any rule **r** in **Unif** is correct i.e.:

$$P \to^{\mathbf{r}} P' \Rightarrow \mathcal{U}_{\lambda\sigma}(P') \subseteq \mathcal{U}_{\lambda\sigma}(P).$$

Proof: Let us check it for all the rules:
Normalize: clear.
Dec-λ and **Dec-App**: Since grafting is defined as a congruence on $\lambda\sigma$-terms, the result follows.
Dec-Fail: The set inclusion is trivial here.
Exp-*: This is trivial by definition of the rules.
Replace: The proof works like in the standard first order case.

□

Lemma 3.5 Any rule **r** in **Unif** is complete i.e.:

$$P \to^{\mathbf{r}} P' \Rightarrow \mathcal{U}_{\lambda\sigma}(P) \subseteq \mathcal{U}_{\lambda\sigma}(P').$$

Proof: Let us check it for all the rules:

Normalize: 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 $\boldsymbol{\lambda\sigma}$ could be applied on top of these terms, we necessary have $\theta(a) =_{\lambda\sigma} \theta(b)$.

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

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

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

$$\begin{array}{rll} \theta(\mathtt{n}\ a_1\ \ldots\ a_p) & =_{\lambda\sigma_\tau} & \theta(\mathtt{n}\ b_1\ \ldots\ b_p) \quad \Leftrightarrow \\ (\mathtt{n}\ \theta(a_1)\ \ldots\ \theta(a_p)) & =_{\lambda\sigma_\tau} & (\mathtt{n}\ \theta(b_1)\ \ldots\ \theta(b_p)) \quad \Leftrightarrow \\ \forall 1 \leq i \leq p,\ \theta(a_i) & =_{\lambda\sigma_\tau} & \theta(b_i). \end{array}$$

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

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

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

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

Exp-λ: If θ is a $\lambda\sigma$-unifier of P and if $X : \Gamma \vdash A \to B \in \mathcal{TV}ar(P)$, 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 $X[a_1 \ldots a_p.\uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q)$ is unifiable by θ, the unifier should be such that $\theta(X) = (\mathtt{r}\ c_1 \ldots c_s)$. Thus it verifies:

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

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

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

In this case θ is clearly solution of $\exists H_1, \ldots, H_k,\ X =^?_{\lambda\sigma} (\mathtt{r}\ H_1\ \ldots\ H_k)$.

Or $r > p$ in which case the equation becomes:

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

and it has a solution if and only if $n + r - p = m$, thus if $r = m - n + p$, at the condition that $r > p \Leftrightarrow m - n + p > p \Leftrightarrow m \geq n + 1$, which gives the condition asserted in rule **Exp-App.**
So in all the cases, the grafting θ is solution of:

$$P \wedge X[a_1 \ldots a_p.\uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q) \wedge \bigvee_{r \in R_p \cup R_i} \exists H_1, \ldots, H_k,\ X =^?_{\lambda\sigma} (\mathtt{r}\ H_1\ \ldots\ H_k).$$

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

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

3.3 A complete strategy for λσ-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 time 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:

Normalize 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 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 non solved 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 $SysToSubst(R)$ the canonical grafting associated to R. For example if $R = (X =^?_{\lambda\sigma} a)$ then $SysToSubst(R) = \{X \mapsto a\}$.

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

Definition 3.6 For a system of equations $P = (Q, R)$ and a $\lambda\sigma$-normalized grafting θ solution of P, we define the transformations $(Q, R, \theta) \longrightarrow^{\mathbf{r}} (Q', R', \theta')$ as follows:

1. $(Q, R, \theta) \longrightarrow^{\mathbf{Normalize}} (Q', R', \theta)$
   where Q' and R' are the normalized forms of Q and R as defined in **Unif**.

2. $(Q \wedge \lambda_A a =^?_{\lambda\sigma} \lambda_A b, R, \theta) \longrightarrow^{\mathbf{Dec\text{-}\lambda}} (Q', R', \theta)$
   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. $(Q \wedge (\mathtt{n}\ a_1\ \ldots\ a_p) =^?_{\lambda\sigma} (\mathtt{n}\ b_1\ \ldots\ b_p), R, \theta) \longrightarrow^{\mathbf{Dec\text{-}App}} (Q', R', \theta)$
   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. $(Q \wedge X =^?_{\lambda\sigma} b, R, \theta) \longrightarrow^{\mathbf{Replace}} (\{X \mapsto b\}(Q), R \wedge X =^?_{\lambda\sigma} b, \theta)$

5. (Q, R, θ)
   $\longrightarrow^{\mathbf{Exp\text{-}\lambda R}}$
   $(\{X \mapsto \lambda_A Y\}Q, R \ \wedge\ X =^?_{\lambda\sigma} \lambda_A Y, \theta - \{X \mapsto \lambda_A a\} + \{Y \mapsto a\})$
   when **Exp-λ** is applicable on $Q \wedge R$.

6. $(Q \wedge X[a_1 \ldots a_p.\uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q), R, \theta)$
   $\longrightarrow^{\mathbf{Exp\text{-}AppR}}$
   $(\{X \mapsto (\mathtt{r}\ H_1\ \ldots\ H_k)\}(Q \wedge X[a_1 \ldots a_p.\uparrow^n] =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q),$
   $R \wedge X =^?_{\lambda\sigma} (\mathtt{r}\ H_1\ \ldots\ H_k),$
   $\theta - \{X \mapsto (\mathtt{r}\ c_1\ \ldots\ c_k)\} + \{H_i \mapsto c_i\})$
   for one of the $r \in R_p \cup R_i$ and when **Exp-App** is applicable on $Q \wedge R$.

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

Lemma 3.6 The transformations **Exp-λR** and **Exp-AppR** are well defined.

Proof: What we need to prove here is that the transformations made on the grafting part θ make sense.

- For the rule **Exp-λR**, since the transformation **Exp-λ** has been shown to preserve the solutions, this means that θ is also $\lambda\sigma$-solution of the equation $X =^?_{\lambda\sigma} \lambda_A Y$ and thus this means that the instantiation of X by θ should be of the form $\{X \mapsto \lambda_A a\}$ (since θ is assumed to be $\lambda\sigma$-normalized). So the transformation is well defined.
- For the rule **Exp-AppR**, since the transformation **Exp-App** has been shown to preserve the solutions, this means that θ is also $\lambda\sigma$-solution of the equation $X =^?_{\lambda\sigma} (\mathtt{r}\ H_1\ \ldots\ H_k)$ and consequently $\theta(X) = (\mathtt{r}\ c_1\ \ldots\ c_k)$ for some $c_i (i = 1..k)$. Thus the transformation is also well defined in this case. □

Lemma 3.7 For a system (Q, R) having for solution the $\lambda\sigma$-normalized 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 size 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 **Normalize**. 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 κ_i is the length of the shortest $\lambda\sigma$-normalizing derivation of $a_i =^?_{\lambda\sigma} b_i$.

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 [DJ90]).

We check now that each application of one of the rules **Dec-***, **Replace** and **Normalize** 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 increase κ_i since the normalization derivation of a subterm is always equal or smaller than the derivation of its context term, and it always decreases the size of the equation to which it is applied.

Normalize may decrease the number of solved variables but always decreases the size of one of the κ_i.

In order to prove the termination of the application of 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 Figure 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 to build the solutions:

Lemma 3.8 If θ is $\lambda\sigma$-solution of the system Q and if $(Q, R, \theta) \longrightarrow^{\mathbf{r}} (Q', R', \theta')$ then θ' is $\lambda\sigma$-solution of Q' and:

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

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'. Notice that the equality modulo $\lambda\sigma$ is introduced by possible normalization 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:
  $\theta(X) = (\theta \circ SysToSubst(R))(X) = \theta(X) = \lambda_A a$, and
  $(\theta' \circ SysToSubst(R'))(X) = \theta'(\lambda_A Y) = \lambda_A a$.
- 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 3.9 Starting from the problem $P_0 = (Q_0, R_0)$ having the $\lambda\sigma$-normalized $\lambda\sigma$-solution θ_0, and applying the rules defined in **UStrat** leads to a finite derivation:

$$(Q_0, R_0, \theta_0) \longrightarrow (Q_1, R_1, \theta_1) \longrightarrow \ldots \longrightarrow (Q_n, R_n, \theta_n)$$

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

Occur-Check	$P \wedge X =^?_{\lambda\sigma} a$ $\rightarrow$ $\mathbb{F}$ if $X \in \mathcal{TV}arR(a)$
Merge	$P \wedge X =^?_{\lambda\sigma} a \wedge X =^?_{\lambda\sigma} b \quad \rightarrow \quad P \wedge X =^?_{\lambda\sigma} a \wedge a =^?_{\lambda\sigma} b$

Figure 4: **Unif**+, more rule for unification

Proof: Notice first that because of the definition of R_0, $\theta_0 = \theta_0 \circ SysToSubst(R_0)$.

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

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

where $Q_n \wedge R_n$ should be a solved form (otherwise this would not be the end of the derivation by Lemma 3.3) and $\mathcal{V}ar(P_0) \supseteq \mathcal{V}ar(P_1) \supseteq \ldots \supseteq \mathcal{V}ar(P_n)$ since the set of variables of the unification problems (thus excluding the existentially bounded variables) could only decrease, due to the **Normalize** rule. □

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

Theorem 3.1 *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 having one of its one constitutive system solved, then the problem P is $\lambda\sigma$-unifiable and a solution to P is the solution constructed in Lemma 3.1 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.*

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

Remarks:

— 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.

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

— In order to fail more often (notice that the algorithm loops on equations like $X = (1\ X)$), one can introduce an occur-check rule as described in Figure 4.

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

- $X \in \mathcal{TV}arR(\lambda\, a) \Leftrightarrow X \in ns\mathcal{TV}arR(a)$,
- $X \in \mathcal{TV}arR((\mathtt{n}\ a_1, \ldots, a_n)) \Leftrightarrow X \in \bigcap_{i=1}^{n} ns\mathcal{TV}arR(a_i)$,
- $X \in ns\mathcal{TV}arR((X[s]\ a_1, \ldots, a_n))$.

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

— 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 normalizing rule. This allows in particular to obtain solved forms that describe the set of open $\lambda\sigma$-unifiers.

3.4 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 normalizing, we consider only normalized equations (as in normalized 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 $\lambda a =^?_{\lambda\sigma} \lambda b$ to $a =^?_{\lambda\sigma} b$. In the same way, a term of the form $(\mathtt{n}\ a_1\ \ldots\ a_p)$ whose type is atomic reduces only to terms of the form $(\mathtt{n}\ a'_1\ \ldots\ a'_p)$ where a_i reduces on a'_i, we can simplify equations of the form $(\mathtt{n}\ a_1\ \ldots\ a_p) = (\mathtt{n}\ b_1\ \ldots\ b_p)$ to $a_1 = b_1, \ldots, a_p = b_p$ and $(\mathtt{n}\ a_1\ \ldots\ a_p) = (\mathtt{m}\ b_1\ \ldots\ b_q)$ with $n \neq m$ to failure.
- As our system is normalizing, 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 $(\mathtt{r}\ a_1\ \ldots\ a_k)$ where k is the arity of $\mathtt{r}$, we consider only graftings of the form $X = \lambda Y$ if X has a functional type and $X = (\mathtt{r}\ H_1\ \ldots\ H_k)$ is X has an atomic type. When we have an equation of the form $X[a_1\ \ldots\ a_p.\ \uparrow^n] = (\mathtt{m}\ b_1\ \ldots\ b_q)$ where X has an atomic type, and we take the grafting $X = (\mathtt{r}\ H_1\ \ldots\ H_k)$ we know that if r is different form $1, \ldots, p$ and $m - n + p$, we can anticipate the failure of the grafting $X = (\mathtt{r}\ H_1\ \ldots\ H_k)$.
- At last the flexible-flexible equations always have solutions.

Notice 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 normalization and the simplification of rigid contexts are optimizations to narrowing that are of course applicable to other theories.

3.5 Dropping the η-rule

Higher order unification algorithms come in two flavors 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 permit to determine 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:

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

we loose completeness if we apply only the rule **Exp-$\boldsymbol{\lambda}$** 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 is of atomic type:

Exp-App
$$\begin{array}{l} P \wedge ((X[a_1 \ldots a_p.\ \uparrow^n]\ e_1\ \ldots\ e_l) =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q)) \\ \to \\ \quad P \quad \wedge \quad (X[a_1 \ldots a_p.\ \uparrow^n]\ e_1\ \ldots\ e_l) =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q) \\ \qquad \wedge \quad \bigvee_{r \in R_p \cup R_i} \exists H_1, \ldots, H_k,\ X =^?_{\lambda\sigma} (\mathtt{r}\ H_1\ \ldots\ H_k) \end{array}$$
if X is not solved
where $H_1, \ldots, H_k$ are variables of the appropriate type not occurring in P with the contexts $\Gamma_{H_i} = \Gamma_X$, R_p is the subset of $\{1, \ldots, p\}$ 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 $\emptyset$

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

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

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

$$\textbf{Dec-App-}\boldsymbol{\lambda} \quad P \wedge ((\mathtt{n}\ a_1\ \ldots\ a_p) =^?_{\lambda\sigma} \lambda b) \quad \rightarrow \quad \mathbb{F}$$

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

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

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

$$\begin{array}{ll}\textbf{Exp-App2} & P \wedge ((X[a_1 \ldots a_p.\uparrow^n]\ e_1\ \ldots\ e_l) =^?_{\lambda\sigma} \lambda b) \\ & \rightarrow \\ & \quad P \quad \wedge \quad (X[a_1 \ldots a_p.\uparrow^n]\ e_1\ \ldots\ e_l) =^?_{\lambda\sigma} \lambda b \\ & \quad \wedge \quad \bigvee_{r \in R_p} \exists H_1, \ldots, H_k,\ X =^?_{\lambda\sigma} (\mathtt{r}\ H_1\ \ldots\ H_k) \\ & \text{if } X \text{ is not solved} \\ & \text{where } H_1, \ldots, H_k \text{ are variables of the appropriate type not occurring in} \\ & \quad P \text{ with the contexts } \Gamma_{H_i} = \Gamma_X \ R_p \text{ is the subset of } \{1, \ldots, p\} \text{ such} \\ & \quad \text{that } (\mathtt{r}\ H_1\ \ldots\ H_k) \text{ has the right type,}\end{array}$$

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

4 Application to unification in λ-calculus

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 $\lambda(X[\uparrow]) =^?_{\beta\eta} \lambda 2$. We extend this remark to any unifier and, by defining a suitable translation a_F of a λ-term $a \in \Lambda_{DB}(\mathcal{X})$, we reduce higher-order unification to equational unification: the higher-order problem $a =^?_{\beta\eta} b$ has a solution if and only if the 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.

4.1 Pre-cooking: definition and properties

In order to sort variables in the next definition and results, we assign to each variable X in a term a, the context of a and the variable type. This is needed since when grafting a variable by a term containing bound variables, the context in which these variables are bound will be the one of a.

Definition 4.1 Let $a \in \Lambda_{DB}(\mathcal{X})$ 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 pre-cooking of a from $\Lambda_{DB}(\mathcal{X})$ to $\mathcal{T}_{\lambda\sigma}(\mathcal{X})$ 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. $F((a\ b), n) = F(a, n)F(b, n)$,

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

Proposition 4.1 If $\Gamma \vdash a : T$ in $\Lambda_{DB}(\mathcal{X})$, then $\Gamma \vdash a_F : T$ in $\lambda\sigma$-calculus.

Proof: We prove the following more general result.

If $A_1 \dots A_n, \Gamma \vdash a : T$ in λ-calculus, if every variable of a is associated to the context Γ, then $A_1 \dots A_n, \Gamma \vdash F(a, n) : T$.

This is done by induction on the structure of a, for all n.

1. $a = \lambda_B.b$; Then $T = B \to C$ and $B.A_1 \dots A_n, \Gamma \vdash b : C$. So $B.A_1 \dots A_n, \Gamma \vdash F(b, n+1) : C$ and $A_1 \dots A_n, \Gamma \vdash F(\lambda\, b, n) = \lambda\,(F(b, n+1)) : B \to C$.
2. $a = X$; By definition, in λ-calculus, $\Gamma \vdash X : T$. By hypothesis, in $\lambda\sigma$-calculus, $\Gamma \vdash X : T$ and $A_1 \dots A_n, \Gamma \vdash X[\uparrow^n] = F(X, n) : T$.
3. The cases $a = \mathtt{n}$ and $a = (a_1\ a_2)$ are easy.

□

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

Proposition 4.2 Let $a, b_1, \dots, b_p$ be λ-terms in de Bruijn notation. Then:

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

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

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

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:

$$F(\mathrm{lft}^i(b, k), i+k) = F(b, k)[1\ \dots\ k.\ \uparrow^{i+k}].$$

- If $b = \mathtt{m}$ and $m \leq k$ then $\mathrm{lft}^i(\mathtt{m}, k) = \mathtt{m}$ thus $F(\mathrm{lft}^i(b, k), i+k) = \mathtt{m}$ and $F(b, k)[\mathtt{1} \dots \mathtt{k}.\ \uparrow^{i+k}] = \mathtt{m}$ also.
- If $b = \mathtt{m}$ and $m > k$ then $\mathrm{lft}^i(\mathtt{m}, k) = \mathtt{m} + \mathtt{i}$ thus $F(\mathrm{lft}^i(b, k), i+k) = \mathtt{m} + \mathtt{i}$ and $F(b, k)[\mathtt{1} \dots \mathtt{k}.\ \uparrow^{i+k}] = \mathtt{m} + \mathtt{i}$ also.
- if $b = X$, we get $X[\uparrow^{i+k}]$ for the two members.
- if $b = \lambda\, c$, then $\mathrm{lft}(b, k) = \lambda\,(\mathrm{lft}(c, k+1))$, so $\mathrm{lft}^i(b, k) = \lambda\,(\mathrm{lft}^i(c, k+1))$. Then, $F(\mathrm{lft}^i(b, k), i+k) = \lambda\,(F(\mathrm{lft}^i(c, k+1), i+k+1))$ and $F(b, k)[\mathtt{1} \dots \mathtt{k}.\ \uparrow^{i+k}] = \lambda\,(F(c, k+1)[\mathtt{1} \dots \mathtt{k+1}.\ \uparrow^{i+k+1}])$ and we conclude using the induction hypothesis.
- $b = (c\ d)$. By induction.

□

Proposition 4.3 Let a be a λ-term.

1. If $a \xrightarrow{\beta} b$, then $a_F \xrightarrow{\lambda\sigma} b_F$.
2. If a is $\beta\eta$-normal then a_F is $\lambda\sigma$-normal.
3. If $a \xrightarrow{\eta} b$, then $a_F \xrightarrow{\eta} b_F$.

4. $a =_{\beta\eta} b$ if and only if $a_F =_{\lambda\sigma} b_F$.

Proof: 1. We show that if $a \stackrel{\beta}{\to} b$, then for every n, $F(a,n) \stackrel{\lambda\sigma}{\to} F(b,n)$ 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:

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

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

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

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

$$F(b, n+r)[\mathtt{1}\ldots\mathtt{r}.\uparrow^{r+1}] = F(\mathrm{lft}(b,r), n+r+1).$$

This follows from an easy induction on b.

2. We have $(\lambda(b^+\ 1))_F = \lambda(F(b^+,1)\ 1)$, 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

$$F(\mathrm{lft}^i(b,k), i+k) = F(b,k)[\mathtt{1}\ldots\mathtt{k}.\uparrow^{i+k}]$$

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 pre-cooking translation is injective we get $a' = b'$ and thus $a =_{\beta\eta} b$.

□

Proposition 4.4 Let a and b be two λ-terms. There exists terms $N_1, \ldots, N_p$ such that $a\{X_1/N_1, \ldots, X_p/N_p\} =_{\beta\eta} b\{X_1/N_1, \ldots, X_p/N_p\}$ if and only if there exists terms $M_1, \ldots, M_p$ *in the image of the pre-cooking translation* such that $\{X_1 \mapsto M_1, \ldots, X_p \mapsto M_p\}a_F =_{\lambda\sigma} \{X_1 \mapsto M_1, \ldots, X_p \mapsto M_p\}b_F$.

Proof: By propositions 4.2 and 4.3 □

4.2 Grafting-unification and substitution-unification

We now precisely relate unification in λ-calculus and unification in $\lambda\sigma$-calculus.

Proposition 4.5 Let $a =^?_{\beta\eta} b$ be a unification problem in $\Lambda_{DB}(\mathcal{X})$. If the problem $a =^?_{\beta\eta} b$ has a solution, then the problem $a_F =^?_{\lambda\sigma} b_F$ also has a solution.

Proof: By proposition 4.4, if the problem $a =^?_{\beta\eta} b$ has a solution $\{X_1/t_1, \ldots, X_n/t_n\}$ then the grafting $\{X_1 \mapsto t_{1F}, \ldots, X_n \mapsto t_{nF}\}$ 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. 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 pre-cooking 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 pre-cooking translation. One of the difficulties is the following. If we solve the flexible-flexible equations by the grafting of proposition 3.1 then we construct a grafting that is not in the image of the pre-cooking 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 flex- flex in the sense of Definition 3.5 because it is solved, but allows 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 pre-cooking 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 flex-flex 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. Thus, in order to be able to apply the proposition 4.4 we must show that if the variable X has an occurrence in a term of the form $X[a_1 \dots a_p.\ \uparrow^n]$ then we have $p \leq |\Gamma_X| - |\Gamma|$. We prove now that this proposition is an invariant of the system **Unif**.

Definition 4.2 Let Γ and Δ be two contexts, Γ is an extension of Δ if it has the form $\Gamma = A_1 \dots A_n.\Delta$. 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 $X[b_1 \dots b_p.\ \uparrow^n]$, Γ_X is an extension of Γ and $p \leq |\Gamma_X| - |\Gamma|$.

Proposition 4.6 Let Γ be a given context.

1. If a is a Γ-stable term well-typed in a context $A_1 \dots A_i.\Gamma$ and $p \leq i$, then the σ-normal form of $a[1 \dots p.\ \uparrow^{p+1}]$ is Γ-stable.
2. If a is a Γ-stable term well-typed in a context $A_1 \dots A_i.\Gamma$ and if $b_1, \dots, b_p$ are Γ-stable terms and $p \leq i$, then the σ-normal form of $a[b_1 \dots b_p.\ \uparrow^n]$ is Γ-stable.
3. If a is a σ-normal term well-typed in a context $A_1 \dots A_i.\Gamma$ 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 $A_1 \dots A_i.\Gamma$ and $B_1 \dots B_j.\Gamma$ 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 $a[1 \dots p.\ \uparrow^{p+1}] = \lambda(a'[1.2 \dots p+1.\ \uparrow^{p+2}])$. The term a' is well-typed in $B.A_1 \dots A_i.\Gamma$ and is Γ-stable, thus, by induction hypothesis $a'[1.2 \dots p+1.\ \uparrow^{p+2}]$ is Γ-stable. Therefore $a[1 \dots p.\ \uparrow^{p+1}]$ is Γ-stable.
- if $a = (a_1\ a_2)$ or a is a de Bruijn index, the case is easy.
- If $a = X[c_1 \dots c_q.\ \uparrow^m]$ in this case we have $q \leq |\Gamma_X| - |\Gamma|$. We get the term

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

i.e.

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

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

$$c_1[1\ \dots\ p.\ \uparrow^{p+1}], \dots, c_q[1\ \dots\ p.\ \uparrow^{p+1}]$$

are Γ-stables.
If $m \geq p$ we get

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

as $q \leq |\Gamma_X| - |\Gamma|$ and the σ-normal forms of the terms $c_1[1\ \dots\ p.\ \uparrow^{p+1}], \dots, c_q[1\ \dots\ p.\ \uparrow^{p+1}]$ are Γ-stables, the σ-normal form of $a[1 \dots p.\ \uparrow^{p+1}]$ is Γ-stable.
If $m < p$ we get the term

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

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

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

as $p \leq i$ we have $r \leq |\Gamma_X| - |\Gamma|$. Then all the terms of this grafting are Γ-stable.

2. By induction on the structure of a.
   - If $a = \lambda a'$ then $a[b_1 \dots b_p.\uparrow^n] = \lambda a'[1.b_1[\uparrow] \dots b_p[\uparrow].\uparrow^{n+1}]$ the terms $a', 1, b_1[\uparrow], \dots, b_p[\uparrow]$ are well-typed in $B.A_1 \dots A_i.\Gamma$ and by the previous point they are Γ-stable, thus, by induction hypothesis $a'[1.b_1[\uparrow] \dots b_p[\uparrow].\uparrow^{n+1}]$ is Γ-stable. Therefore $a[b_1 \dots b_p \uparrow^n]$ is Γ-stable.
   - if $a = (a_1\ a_2)$ or a is a de Bruijn index, the case is easy.
   - If $a = X[c_1 \dots c_q.\uparrow^m]$ in this case we have $q \leq |\Gamma_X| - |\Gamma|$. We get the term

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

     i.e.

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

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

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

     As $q \leq |\Gamma_X| - |\Gamma|$ and the σ-normal form of $c_1[b_1\ \dots\ b_p.\uparrow^n], \dots, c_q[b_1\ \dots\ b_p.\uparrow^n]$ are Γ-stable, the term $a[b_1 \dots b_p \uparrow^n]$ is Γ-stable.
     If $m < p$ we get the term

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

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

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

     as $p \leq i$ we have $r \leq |\Gamma_X| - |\Gamma|$. Then all the terms of this grafting are Γ-stable. Therefore $a[b_1 \dots b_p.\uparrow^n]$ is Γ-stable.
3. When we reduce a β-redex $(\lambda_C a)\ b$ occurring under j abstractions of type $B_1, \dots, B_j$, the term a is well-typed in a context $C.B_1 \dots .B_j A_1 \dots .A_i\Gamma$. 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 $B_1, \dots, B_j$, the term a is well-typed in a context $C.B_1 \dots .B_j A_1 \dots .A_i\Gamma$. 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 $B_1 \dots B_j.A_1 \dots A_j.\Gamma$, 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 $a = X[a_1 \dots a_p.\uparrow^n]$ then $\{X \mapsto b\}a = b[\{X \mapsto b\}a_1 \dots \{X \mapsto b\}a_p.\uparrow^n]$. By induction hypothesis, the σ-normal forms $a'_1, \dots, a'_p$ of the terms $\{X \mapsto b\}a_1, \dots, \{X \mapsto b\}a_p$ are Γ-stable. As the term $a = X[a_1 \dots a_p.\uparrow^n]$ is Γ-stable, we have $p \leq |\Gamma_X| - |\Gamma|$. As X and b are well-typed in the same context we have $\Gamma_X = B_1 \dots B_j.\Gamma$, thus $p \leq j$ and the σ normal form of $b[a'_1 \dots a'_p.\uparrow^n]$ is Γ-stable. Therefore the σ-normal form of $\{X \mapsto b\}a$ is also Γ-stable.
   - if $a = Y[a_1 \dots a_p.\uparrow^n]$ then $\{X \mapsto b\}a = Y[\{X \mapsto b\}a_1 \dots \{X \mapsto b\}a_p.\uparrow^n]$. By induction hypothesis, the σ-normal forms $a'_1, \dots, a'_p$ of the terms $\{X \mapsto b\}a_1, \dots, \{X \mapsto b\}a_p$ 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 4.7 (Invariants) Let $a =^?_{\beta\eta} b$ be a unification problem in $\Lambda_{DB}(\mathcal{X})$ 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 $A_1 \dots A_n\Gamma$,
- for every variable Y of the derived problem, its context Γ_Y has the form $B_1 \dots B_p.\Gamma$,

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

Proof: By induction on the structure of the derivation, using the proposition 4.6 in the third case for the **Replace** rule. □

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

Proposition 4.8 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 pre-cooking translation.

Proof: Every occurrence u of a variable X belongs in a subterm of the form $X[a_1 \dots a_p.\ \uparrow^n]$. As a is Γ-stable we have $p \leq |\Gamma_X| - |\Gamma|$. As $\Gamma_X = \Gamma$ we have $p = 0$. Then as the term $X[\uparrow^n]$ is well-typed in some context $B_1 \dots B_{|u|}.\Gamma$ and the term X is well-typed in Γ we have $n = |u|$, thus a is in the image of the pre-cooking translation. □

Proposition 4.9 Let $a =^?_{\beta\eta} b$ be a unification problem in $\Lambda_{DB}(\mathcal{X})$. 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 non-solved 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 $\Gamma \vdash T$:

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

Since the term $Z_T[\uparrow^{|\Gamma_X|-|\Gamma|}]$ is stable (just apply the definition), by application of Proposition 4.6, the problem $\theta(P)$ is Γ-stable. Furthermore, θ preserves the solved variables of the system.

All the non-solved 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 $Z_T[a_1\ \dots\ a_p.\ \uparrow^n]$ and we have $p \leq |\Gamma_Z| - |\Gamma|$, 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 flex-flex 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 θ) and every variable Z occurring in this term if 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 4.8, $\phi \circ \theta(X)$ is in the image of the pre-cooking 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 build in Lemma 3.1 where the variables Z_T have the sort $nil \vdash T$ which is empty.

Corollary 4.1 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 4.1 *Let $a =^?_{\beta\eta} b$ be a unification problem in $\Lambda_{DB}(\mathcal{X})$. The equational problem $a_F =^?_{\lambda\sigma} b_F$ has a solution if and only if the higher order problem $a =^?_{\beta\eta} b$ has a solution.*

An immediate consequence of the undecidability of higher-order unification [Hue73, Gol81] and of the last theorem is that equational unification is undecidable in the theory $\lambda\sigma$.

4.3 On the use of substitution variables

Before showing how to relate solutions of problems in $\lambda\sigma$ and in λ-calculus, we should answer to a very natural question: "why not using substitution variables in order to encode unification in lambda calculus into the $\lambda\sigma$-calculus?" The answer is indeed a priori surprising: such an approach leads also to the formalism presented in the previous section. Let us detail why.

A direct way to encode unification in $\lambda\sigma$-calculus is to consider ground terms, i.e. where the free variables are encoded by additional indices, and to set the unification problem as follows.

For two terms a and b, find a grafting γ of the substitution variable Y such that:

$$\gamma(a[Y]) =_{\lambda\sigma} \gamma(b[Y]),$$

which amongs to solve in the algebra $\mathcal{T}_{\lambda\sigma}(\mathcal{Y})$ the equation:

$$a[Y] =^?_{\lambda\sigma} b[Y],$$

modulo the equational theory $\lambda\sigma$.

The image of Y by a grafting should necessarily be of the form:

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

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 $a_1, \dots, a_p$ such that:

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

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

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

Let us now concentrate on one side of the equation, for example the one concerning a. Let us consider an index $\mathtt{r}$ of a free variable in a. By $\lambda\sigma$-reduction of $a[X_1 \dots X_p \cdot \uparrow^n]$ and assuming that the λ-height of $\mathtt{r}$ in a is k, we get:

$$\mathtt{r}[1.(1.(\dots \circ \uparrow) \circ \uparrow)],$$

which $\lambda\sigma$-normalizes, for the appropriate i, to $X_i[\uparrow^k]$.

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

Notice that the previous discussion provides 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.

4.4 Translating back equations

We now come back to higher-order unification by showing in this section that for any unification problem P, deriving from a problem $a_F =^?_{\lambda\sigma} b_F$, we can reconstruct a problem Q in the image of the pre-cooking 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 pre-cooking 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 [Hue75].

Let us extend the system **Unif** with the two rules given in Figure 5.

Proposition 4.10 The system **Unif**, augmented by the **Anti** 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 $\exists Y\ (P \wedge (X =^?_{\lambda\sigma} (Y[\uparrow]\ 1)))$. □

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 $\Delta = A.\Delta'$

Figure 5: **Anti**, the basic rules for unification in $\lambda\sigma$

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 4.11 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 4.7 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 $X[a_1\ \ldots\ a_p.\uparrow^n]$ 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 4.12 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 normalization with the strategy **Back** of the system P is the the pre-cooking 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 4.8 that this problem is the the pre-cooking of a problem in λ-calculus. □

We call *λ-solved form* any solved systems of equation in λ-calculus in the sense of [SG89].

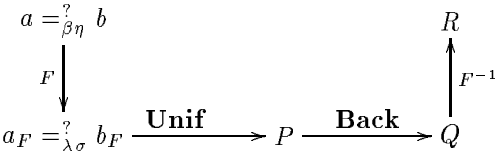
Corollary 4.2 Let $a =^?_{\beta\eta} b$ a higher-order unification problem such that $a_F =^?_{\lambda\sigma} b_F$ can be rewritten by the system **Unif** to a disjunction of systems that has one of its constitutive systems P solved. Let Q be the system resulting of the normalization with the strategy **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 [SG89] to solve the flexible-flexible equations, we get back the solution built in the proof of the Proposition 4.9.

Proof: Let θ be a substitution in λ-calculus, 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 4.2 *(Description of the solutions)*
Let $a =^?_{\beta\eta} b$ be a higher-order unification problem such that $a_F =^?_{\lambda\sigma} b_F$ 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 normalization using the* **Back** *strategy and back-cooking.*

Proof: Let θ be a substitution in λ-calculus, 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:



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 restrict to syntactic first order unification when the input problem is a first order unification problem.

4.5 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, at last **Anti-Dec-λ** n times to the equations obtained by **Dec-app**. For instance from the equation:

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

we get $(\mathtt{k}\ a_1\ \ldots\ a_p) = (\mathtt{k}\ b_1\ \ldots\ b_p)$ by applying **Dec-λ** n times, then $a_1 = b_1, \ldots, a_n = b_n$ using **Dec-app**, then $\lambda \ldots \lambda a_1 = \lambda \ldots \lambda b_1, \ldots, \lambda \ldots \lambda a_p = \lambda \ldots \lambda b_p$ 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, at last applying **Anti-Exp-λ** n times to each new variable H_i. For instance if we have a variable X of type $T_1 \to \ldots \to T_n \to U$. We get $X \mapsto \lambda \ldots \lambda Y$ by applying **Exp-λ** n times, then $X \mapsto \lambda \ldots \lambda(\mathtt{k}\ H_1\ \ldots\ H_p)$ by applying **Exp-app** to Y and at last $X \mapsto \lambda \ldots \lambda(\mathtt{k}\ (K_1[\uparrow^n]\ n\ \ldots\ 1)\ \ldots\ (K_p[\uparrow^n]\ n\ \ldots\ 1))$ by applying **Anti-Exp-λ** n times to each H_i. This term is the pre-cooking of $\lambda \ldots \lambda(\mathtt{k}\ (K_1\ n\ \ldots\ 1)\ \ldots\ (K_p\ n\ \ldots\ 1))$ which is Huet's elementary substitution.

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

4.6 Dependence constraints

In some cases unification problems come with more scoping constraints than the ones present in the equations. For instance we may want restrict the term substituted to some variable X in such a way that some de Bruijn index of the context (for instance 2) does not appear in it (see for instance [Mil92]).

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

4.7 Examples

A simple example

In order to illustrate the approach of higher order unification presented in this paper, let us 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 $\lambda(X\ 2) =^?_{\beta\eta} \lambda 2$ and then pre-cooked into $\lambda(X[\uparrow]\ 2) =^?_{\lambda\sigma} \lambda 2$. 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 **Normalize** results in:

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

Applying the rule **Exp-app** yields:

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

which reduces using the rule **Replace** into:

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

that is normalized by rule **Normalize** into:

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

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

A more elaborated example

We solve 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 pre-cooking using the variable X with context Γ and type $A \to A$, yields $(2\ (X\ 1)) =^?_{\lambda\sigma} (X\ (2\ 1))$.

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

$$\{(2\ (X\ 1)) =^?_{\lambda\sigma} (X\ (2\ 1)) \quad \longrightarrow_{\textbf{Exp-}\lambda} \exists Y \begin{cases} (2\ (X\ 1)) =^?_{\lambda\sigma} (X\ (2\ 1)) \\ X =^?_{\lambda\sigma} \lambda Y \end{cases} \quad (Y : A \cdot \Gamma \vdash A)$$

$$\longrightarrow_{\textbf{Replace}}$$

$$\exists Y \begin{cases} (2\ (\lambda Y\ 1)) =^?_{\lambda\sigma} (\lambda Y\ (2\ 1)) \\ X =^?_{\lambda\sigma} \lambda Y \end{cases} \quad \longrightarrow_{\textbf{Normalize}} \exists Y \begin{cases} (2\ Y[1.id]) =^?_{\lambda\sigma} Y[(2\ 1).id] \\ X =^?_{\lambda\sigma} \lambda Y \end{cases}$$

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

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

$$\longrightarrow_{\textbf{Replace}}$$

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

$$\longrightarrow_{\textbf{Normalize}}$$

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

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

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

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

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

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

$$\longrightarrow_{\textbf{Replace}}$$

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

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

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

Now if we consider the two previous solved forms, they are both in the image of F, and pre-cooking them back to 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.

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 algorithm could not be used for higher order unification, because substitution in higher-order algebras is different from substitution in first order ones (grafting).

In this paper we have shown that higher-order unification problems could be translated into first order ones. We come up with a new algorithm for higher-order unification based on an 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 to avoid the encoding of scoping constraints by $\beta\eta$-reduction, 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 [Wil91] as well as in the unification algebra framework of [SSS88]. One of the main differences between the present work with 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 [Dou93] 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, this may be a way to reduce higher-order equational unification to first order. Also, the framework seems to be very promising for studying decidable subproblems of unification like patterns [Mil91].

This work has to be carried with a precise analysis of the algorithm, in order to define strategies as lazy as possible. For example, there is no need to compute $\lambda\sigma$-normal forms at every step, and the experimental implementation [Bor95] of the unification rules which has been realized using the ELAN logical framework [KKV95], 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 [Bar92]. In this case, the functional expression of scoping constraints leads to technical difficulties [Dow93] that may be simplified using explicit substitutions. At last, 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 would be equational resolution in this theory.

Acknowledgements

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 comments from Martin Abadi, Peter Borovansky, Daniel Briaud, César Muñoz and anonymous referees of the conference version.

References

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

[And71] P. Andrews. Resolution in type theory. *Journal of Symbolic Logic*, 36:414–432, 1971.

[And86] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Academic Press, New York, 1986.

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics.* Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.

[Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science.* Clarendon Press, 1992.

[Bor95] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitution. In *Proceedings of SOFSEM'95*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[CHK90] H. Chen, J. Hsiang, and H. C. Kong. On finite representations of infinite sequences of terms. In S. Kaplan and M. Okada, editors, *Proceedings of the 2nd International Workshop on Conditional and Typed Rewriting Systems*, volume 516 of *Lecture Notes in Computer Science*, pages 100–114. Springer-Verlag, 1990.

[CHLar] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, to appear. also as 1992 INRIA report 1617.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CR91] P.-L. Curien and A. Rios. Un résultat de complétude pour les substitutions explicites. *Compte-rendus de l'Académie des Sciences de Paris*, 312(I):471–476, 1991.

[Cur93] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Birkhäuser, 1993. 2nd edition.

[Dau92] M. Dauchet. Simulation of Turing machines by a regular rule. *Theoretical Computer Science*, 103:409–420, 1992.

[dB72] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[DHK95] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions, extended abstract. In D. Kozen, editor, *Proceedings of LICS'95*, pages 366–374, San Diego, June 1995.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Dou93] D. J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114:273–298, 1993.

[Dow93] G. Dowek. A complete proof synthesis method for the cube of type systems. *Journal of Logic and Computation*, 3(3):287–315, 1993.

[Gol81] D. Goldfarb. The undecidability of the second order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Her30] J. Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Soc. des Sciences et des Lettres de Varsovie, Classe III*, 33(128), 1930.

[HKK94] C. Hintermeier, C. Kirchner, and H. Kirchner. Dynamically-typed computations for order-sorted equational presentations –extended abstract–. In S. Abiteboul and E. Shamir, editors, *Proc. 21st International Colloquium on Automata, Languages, and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 450–461. Springer-Verlag, 1994.

[Hue73] G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257–267, 1973.

[Hue75] G. Huet. A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[Hul80] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag, July 1980.

[Hus85] H. Hussmann. Unification in conditional equational theories. In B. Buchberger, editor, *Proceedings of the EUROCAL Conference, Linz (Austria)*, volume 204 of *Lecture Notes in Computer Science*, pages 543–553. Springer-Verlag, 1985.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[KH90] H. Kirchner and M. Hermann. Meta-rule synthesis from crossed rewrite systems. In S. Kaplan and M. Okada, editors, *Proceedings 2nd International Workshop on Conditional and Typed Rewriting Systems, Montreal (Canada)*, volume 516 of *Lecture Notes in Computer Science*, pages 143–154. Springer-Verlag, June 1990.

[KKV95] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. The MIT press, 1995.

[Kri93] J.-L. Krivine. *Lambda calculus, types and models.* Ellis Horwood, 1993.

[Mel95] P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In M. Dezani, editor, *Int. Conf. on Typed Lambda Calculus and Applications*, 1995.

[MH94] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computation*, 5(3 & 4):213ff, 1994.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen, Germany, December 1989*, pages 253–281. LNCS 475, 1991.

[Mil92] D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, October 1992.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[Plo72] G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[Río93] A. Ríos. *Contributions à l'étude des λ-calculs avec des substitutions explicites.* Thèse de Doctorat d'Université, U. Paris VII, 1993.

[SG89] W. Snyder and J. Gallier. Higher order unification revisited: Complete sets of tranformations. *Journal of Symbolic Computation*, 8(1 & 2):101–140, 1989. Special issue on unification. Part two.

[SSS88] J. Siekmann and M. Schmidt-Schauß. Unification algebras: an axiomatic approach to unification, equation solving and constraint solving. SEKI report SR-88-23, Universität Kaiserslautern, December 1988.

[Wer95] A. Werner. Normalizing narrowing for weakly terminating and confluent systems. In U. Montanari and F. Rossi, editors, *Proceedings of the first international conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes in Computer Science*, pages 415–430, Cassis, France, September 1995. Springer-Verlag.

[Wil91] J. G. Williams. *Instanciation Theory. On the Foundations of Automated Deduction*, volume 518 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1991.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399