Unification via Explicit Substitutions: The Case of Higher-Order Patterns

Gilles Dowek, Thérèse Hardin, Claude Kirchner, Frank Pfenning

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Unification via Explicit Substitutions: The Case of Higher-Order Patterns

Gilles Dowek , Thérèse Hardin , Claude Kirchner , Frank Pfenning

No 3591

Décembre 1998

THÈME 2

ISSN 0249-6399

Unification via Explicit Substitutions: The Case of Higher-Order Patterns

Gilles Dowek* , Thérèse Hardin† , Claude Kirchner‡ , Frank Pfenning§

Thème 2 — Génie logiciel
et calcul symbolique
Projets COQ,PARA,PROTHEO

Rapport de recherche n ° 3591 — Décembre 1998 — 33 pages

Abstract: In [6] we have proposed a general higher-order unification method using a theory of explicit substitutions and we have proved its completeness. In this paper, we investigate the case of higher-order patterns as introduced by Miller. We show that our general algorithm specializes in a very convenient way to patterns. We also sketch an efficient implementation of the abstract algorithm and its generalization to constraint simplification, which has yielded good experimental results at the core of a higher-order constraint logic programming language.

Key-words: Explicit substitutions, higher order unification, pattern unification, constraint simplification, higher-order logic programming.

(Résumé : tsvp)

* INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France. Gilles.Dowek@inria.fr.
† LIP6, Université Paris 6, 4 Place Jussieu, 75252 Paris Cedex 05, France Therese.Hardin@lip6.fr.
‡ LORIA & INRIA, 615, rue du Jardin Botanique, B.P. 101, 54602 Villers-lès-Nancy Cedex, France. Claude.Kirchner@loria.fr.
§ Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, U.S.A. fp@cs.cmu.edu.

Unit´e de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
T´el´ephone : 01 39 63 55 11 - International : +33 1 39 63 55 11
T´el´ecopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Unification d'Ordre Supérieur via les Substitutions Explicites : Le cas des patterns

Résumé : Nous avons proposé dans [6] une procédure d'unification d'ordre supérieur utilisant un calcul de substitutions explicites. Dans cet article, nous étudions le cas des patterns d'ordre supérieur introduits par Miller. Nous montrons que l'algorithme général se spécialise de façon tout à fait fructueuse dans le cas des patterns. Nous donnons aussi les éléments d'une implémentation efficace de l'algorithme abstrait et de sa généralisation au cas de la simplification de système d'équations ne comportant pas que des patterns. Cette implémentation a donné de bons résultats expérimentaux comme moteur d'un langage de programmation logique d'ordre supérieur contraint.

Mots-clé : Substitution explicite, unification d'ordre supérieur, unification de patterns, simplification de contraintes, programmation logique d'ordre supérieur.

Introduction

Typed λ-calculi of various sorts are used pervasively in logical frameworks and their implementations (e.g., λProlog [19], Isabelle [22], or Elf [25]) and in general reasoning systems such as Coq or Nuprl. Unlike functional programming languages, these implementations require access to the body of λ-terms for such operations as substitution, normalization, matching, or unification. Their efficient implementation is therefore a central problem in theorem proving and higher-order logic programming.

While at present there is no conclusive evidence, it seems that calculi of explicit substitutions have many potential benefits in the implementation of such operations as unification or constraint simplification in typed λ-calculi. They permit the expensive operation of substitution to be postponed until necessary and, furthermore, they allow multiple successive substitutions to be combined. As shown in [6], Huet's unification algorithm for simply typed lambda terms is a specific instance of first-order equational unification in a calculus of explicit substitutions.

In practice, modified versions of Huet's algorithm have performed well, despite the general undecidability of the problem. Miller observed that many problems fell within a decidable fragment [17], usually now referred to as *higher-order patterns*. Unification in this fragment is decidable and unitary, even for very rich type theories such as the Calculus of Constructions [24].

In this paper we show first that the general algorithm for higher-order unification with explicit substitutions can be cleanly specialized to the case of patterns, providing new insights on why patterns behave so nicely. In this pattern unification only some special substitutions called *pattern substitutions* are involved and such substitutions are injective and thus have an inverse.

An empirical analysis of existing higher-order logic programs [15, 16] suggested that a *static* restriction of higher-order logic programming to patterns is too severe from the programmer's point of view. However, almost all dynamically arising equations fall within the pattern fragment. The need to use the power of pattern unification, while keeping the possibility to handle more general unification problem, has led to the development of a constraint simplification algorithm [23] which solves all equations between patterns and postpones other constraints. This constraint handling can then be used in a deduction-with-constraints process [11]. We show also that the unification algorithm for patterns can be generalized to a constraint simplification algorithm for the unrestricted case. This algorithm (augmented, at present without proof, to the case of dependent types) has been implemented in SML as the core of MLF, a version of the logical framework LF with a module layer [9] and in

the system Twelf [26]. It has shown good computational properties, although we have not undertaken a systematic empirical study.

The paper is organized as follows. Section 1 describes $\lambda\sigma$, the calculus of explicit substitutions that we are using, together with its fundamental properties. In section 2 we study the properties of $\lambda\sigma$-patterns and of pattern substitutions. These properties allow us to prove that the transformation rules of pattern unification problems given in Section 3 are correct and complete and compute a most general unifier, when it exists. Then in Section 4, we describe an efficient way to implement these transformations in a logical framework context. We finally conclude with related and future work.

For the basic concepts and definitions of λ-calculus, calculus of substitutions, unification and term rewriting, we refer to [6].

A preliminary version of this paper appeared in [7].

1 Explicit substitutions

The $\lambda\sigma$-calculus is a first-order rewriting system, introduced to provide an explicit treatment of substitutions initiated by β-reductions. Here, we shall use the $\lambda\sigma$-calculus described in [1], in its typed version [1, 5], but similar free calculi with explicit substitutions can be used in the same way provided they are confluent and weakly terminating on the free algebra generated by term variables (here also called meta-variables). In this setting the reduction variables of the λ-calculus are encoded by de Bruijn indices while the unification variables are coded by meta-variables.

In $\lambda\sigma$-calculus, the term $a[s]$ (read as "the image of a under the explicit substitution s") represents the term a on which the substitution s has to be applied. The simplest substitutions are lists of terms build with the constructors "$\cdot$" (cons) and "id" (nil). Applying the substitution $(a_1 \cdots a_p \cdot id)$ to a term a replaces the de Bruijn indices $1, \ldots, p$ in a by the terms $a_1, \ldots, a_p$ and decrements accordingly by p the remaining de Bruijn indices. The composition operation transforms a sequence of substitutions into a single simultaneous one which ensures confluence. The substitution "$\uparrow$" increments de Bruijn indices: $n+1$ is expressed by $1[\uparrow^n]$ and $\uparrow^0$ is the substitution id by convention. For example, the term $((\lambda X)\ Y)[s]$ reduces to $X[Y.id][s]$ but also to $X[1.(s \circ \uparrow)][Y[s].id]$. Using composition these two terms reduce to $X[Y[s].s]$ allowing to recover confluence.

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

$$
\begin{array}{lrcl}
\textbf{Types} & A & ::= & K \mid A \to B \\
\textbf{Contexts} & \Gamma & ::= & nil \mid A.\Gamma \\
\textbf{Terms} & a & ::= & \mathtt{1} \mid \mathcal{X} \mid (a\ b) \mid \lambda_A.a \mid a[s] \\
\textbf{Substitutions} & s & ::= & id \mid \uparrow \mid a : A \cdot s \mid s \circ t
\end{array}
$$

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

$$
\begin{array}{llll}
(var) & A.\Gamma \vdash 1 : A & (lambda) & \dfrac{A.\Gamma \vdash b : B}{\Gamma \vdash \lambda_A b : A \to B} \\
(app) & \dfrac{\Gamma \vdash a : A \to B \quad \Gamma \vdash b : A}{\Gamma \vdash (a\ b) : B} & (clos) & \dfrac{\Gamma \vdash s : \Gamma' \quad \Gamma' \vdash a : A}{\Gamma \vdash a[s] : A} \\
(id) & \Gamma \vdash id : \Gamma & (shift) & A.\Gamma \vdash \uparrow : \Gamma \\
(cons) & \dfrac{\Gamma \vdash a : A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash a : A \cdot s\ :\ A.\Gamma'} & (comp) & \dfrac{\Gamma \vdash s'' : \Gamma'' \quad \Gamma'' \vdash s' : \Gamma'}{\Gamma \vdash s' \circ s''\ :\ \Gamma'} \\
 & & (metavar) & \Gamma_X \vdash X : T_X
\end{array}
$$

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

The typing rules are described in Figure 2. The typing judgment $\Gamma \vdash a : A$ defines when a term a has type A in a context Γ. Similarly, $\Gamma' \vdash s : \Gamma$ defines when a substitution s maps terms constructed over Γ to terms over Γ'. Each meta-variable X carries its own unique context Γ_X and type A_X such that $\Gamma_X \vdash X : A_X$ (rule *metavar*). When $\Gamma \vdash a : A$, we say that the pair $< \Gamma, A >$ is the *sort* of the term a and we also write $a : (\Gamma \vdash A)$. Similarily when $\Gamma \vdash s : \Delta$ we say that the pair $< \Gamma, \Delta >$ is the sort of the substitution s and we write $s : (\Gamma \vdash \Delta)$. For the sake of brevity we often omit type labels in λ-abstractions and substitutions.

The reduction rules defining the semantics of this typed calculus are given in Figure 3. The whole set of rules is called $\lambda\sigma$ and the whole set except the rules **Beta** and **Eta** is called σ. A term of the form $(\lambda a\ b)$ reduces in a first step to the term

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

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

$a[b.id]$ by the rule **Beta**. Then, the explicit substitution $b.id$ is propagated through the term a by the rules of the system σ.

It is shown in [28] that the typed $\lambda\sigma$-calculus is confluent and weakly terminating. For later use, let us also mention the notion of *long normal form* of a $\lambda\sigma$-term which is the η-long form of its $\beta\eta$-normal form (or equivalently of its β-normal form).

The normal form of a $\lambda\sigma$-term has one of the following forms:

$$\lambda a, \quad (\mathtt{n}\ a_1\ \ldots\ a_p), \quad (X[a_1 \ldots a_p.\uparrow^n]\ b_1\ \ldots\ b_q), \quad (X\ b_1\ \ldots\ b_q).$$

By convention, we consider that the later form is a particular case of the third with $p = n = 0$.

2 $\lambda\sigma$-patterns

2.1 Definitions

Let us first recall the usual notion of pattern in the simply typed λ-calculus.

Definition 1 A *higher-order pattern*, or *λ-pattern* for short, is a simply typed lambda term whose free variables X are only applied to a sequence of distinct bound variables, i.e. $(X\ x_1 \ldots x_p)$.

As usual, we assume throughout the paper, that a variable is denoted by its name x even if its actual form in a term must be η-expanded.

Definition 2 A $\lambda\sigma$-term is a *$\lambda\sigma$-pattern* if all its subterms of the form $(X[s]\ b_1\ \ldots\ b_q)$ are such that $s = a_1\ \ldots\ a_p.\uparrow^n$ where $a_1, \ldots, a_p, b_1, \ldots, b_q$ are distinct de Bruijn indices lower than or equal to n. A $\lambda\sigma$-pattern is called *atomic* if all its meta-variables have atomic type. In such a term all the subterms of the form $(X[s]\ b_1\ \ldots\ b_q)$ are such that $q = 0$.

In our approach to unification, we translate equations in the simply-typed λ-calculus to equations the $\lambda\sigma$-calculus [6]. Crucial in the translation is the proper treatment of free variables, since higher-order variables and capture-avoiding substitutions are replaced by meta-variables and *grafting*, i.e., first-order replacement denoted by $(X \mapsto a)$. This gap is bridged through the so-called *pre-cooking* of a term which raises the free variables to their proper context, which, in our presentation, is initially the outermost context. This pre-cooking of a λ-term a, written a_F, is defined as $a_F = F(a, 0)$ where:

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

As shown in [6], pre-cooking is a homomorphism from λ-calculus to $\lambda\sigma$-calculus.

Proposition 1 Let a be a λ-term, a is a λ-pattern if and only if its pre-cooked form a_F is a $\lambda\sigma$-pattern.

Proof: By definition of a λ-pattern, a free variable F occurs only under the form $(F\ x_1 \ldots x_p)$. The pre-cooked form of such a subterm is by definition of the form $(F[\uparrow^n]\ a_1 \ldots a_p)$ where the a_i are all distinct de Bruijn indices which are all lower than or equal to n because $x_1 \ldots x_p$ are bound in the term. □

Definition 3 A *pattern substitution* is a $\lambda\sigma$-substitution whose $\lambda\sigma$-normal form $a_1 \dots a_p.\ \uparrow^n$ is such that all its de Bruijn indices a_i are distinct and less than or equal to n.

2.2 Stability

The $\lambda\sigma$-patterns are stable under substitution composition and under grafting.

Proposition 2 If s and t are two pattern substitutions, then the σ-normal form of $1.(s \circ \uparrow)$ and $s \circ t$ are pattern substitutions.

Proof: Let $s = a_1 \dots a_p.\ \uparrow^n$ and $t = b_1 \dots b_q.\ \uparrow^m$.

$$\begin{aligned} 1.(s \circ \uparrow) &= 1.((a_1 \dots a_p.\ \uparrow^n) \circ \uparrow) \\ &= 1.(a_1[\uparrow] \dots a_p[\uparrow].\ \uparrow^{n+1}) \\ &= 1.(a_1 + 1) \dots (a_p + 1).\ \uparrow^{n+1} \end{aligned}$$

which is clearly a pattern substitution.

– If $n < q$, then

$$\begin{aligned} s \circ t &= (a_1 \dots a_p.\ \uparrow^n) \circ (b_1 \dots b_q.\ \uparrow^m) \\ &= a_1[t] \dots a_p[t].(\uparrow^n \circ (b_1 \dots b_q.\ \uparrow^m)) \\ &= a_1[t] \dots a_p[t].b_{n+1} \dots b_q.\ \uparrow^m \end{aligned}$$

since $a_i \leq n < q$, $a_i[t] = b_{a_i}$ then as $a_i \leq n$, $b_{a_1}, \dots, b_{a_p}, b_{n+1} \dots b_q$ are distinct de Bruijn indices and they are lower than or equal to m. Thus $s \circ t$ is a pattern substitution.

– If $n \geq q$, then

$$\begin{aligned} s \circ t &= (a_1 \dots a_p.\ \uparrow^n) \circ (b_1 \dots b_q.\ \uparrow^m) \\ &= a_1[t] \dots a_p[t].(\uparrow^n \circ (b_1 \dots b_q.\ \uparrow^m)) \\ &= a_1[t] \dots a_p[t].\ \uparrow^{(n-q)+m} \end{aligned}$$

$a_i[t] = b_{a_i}$ if $a_i \leq q$ and $a_i - q + m$ when $a_i > q$ thus all the $a_i[t]$ are all different de Bruijn indices. They are and lower than or equal to $(n - q) + m$ and thus $s \circ t$ is a pattern substitution.

It may be the case that the normal form of $a_p[t]$ is the de Bruijn index $(n-q)+m$ in this case the substitution $s \circ t$ can still be reduced to $a_1[t] \dots a_{p-1}[t].\ \uparrow^{(n-q)+m-1}$, in this case the indices $a_1[t]$, $a_{p-1}[t]$ are still distinct and they are lower than

or equal to $(n-q)+m$ and different from $(n-q)+m$ thus they are lower than or equal to $(n-q)+m-1$.

The same holds if again $a_{p-1}[t] = (n-q)+m-1$, etc. □

Proposition 3 The image of an atomic pattern a under a pattern substitution s is an atomic pattern.

Proof: By induction on the structure of the normal form of a.

- If $a = \lambda b$ then $a[s] = (\lambda b)[s] = \lambda(b[1.(s \circ \uparrow)])$. By the previous lemma, the normal form of the substitution $1.(s \circ \uparrow)$ is a pattern substitution and thus by the induction hypothesis the normal form of the term $b[1.(s \circ \uparrow)]$ is an atomic pattern. Thus the normal form of $a[s]$ is an atomic pattern.
- If $a = (\mathtt{m}\ b_1\ \ldots\ b_p)$ then $a[s] = (\mathtt{m}[s]\ b_1[s]\ \ldots\ b_p[s])$. By induction hypothesis the normal form of $b_1[s], \ldots, b_p[s]$ are atomic patterns. Thus the normal form of $a[s]$ is an atomic pattern.
- If $a = X[t]$ then $a[s] = X[t \circ s]$. By the previous lemma, the normal form of $t \circ s$ is a pattern substitution. Thus the normal form of $a[s]$ is an atomic pattern.

□

Proposition 4 Atomic patterns are stable under pattern grafting, i.e. for any atomic patterns a and b, the normal form of $(X \mapsto b)a$ is an atomic pattern.

Proof: By induction on the structure of a.

- If $a = \lambda a'$ then $(X \mapsto b)a = \lambda(X \mapsto b)a'$, by induction hypothesis the normal form of $(X \mapsto b)a'$ is an atomic pattern. Thus the normal form of $(X \mapsto b)a$ is an atomic pattern.
- If $a = (\mathtt{m}\ a'_1\ \ldots\ a'_p)$ then $(X \mapsto b)a = (\mathtt{m}\ (X \mapsto b)a'_1\ \ldots\ (X \mapsto b)a'_p)$. By induction hypothesis the normal forms of the terms $(X \mapsto b)a'_i$ are atomic patterns and thus the normal form of $(X \mapsto b)a$ is an atomic pattern.
- If $a = Y[s]$ $(Y \neq X)$ then $(X \mapsto b)a = Y[s]$ and thus it is an atomic pattern.
- If $a = X[s]$ then $(X \mapsto b)a = b[s]$ and its normal form is an atomic pattern by the previous lemma.

□

As mentioned above the properties of pattern unification rely on algebraic properties of pattern substitutions such as injectivity. We focus now on the properties of these substitutions.

2.3 Injectivity

Proposition 5 Let $a_1, \ldots, a_p$ be numbers and $s = a_1 \ldots a_p.\uparrow^n$ be a pattern substitution. Consider the substitution $\overline{s} = e_1 \ldots e_n.\uparrow^p$ where by definition:

- for all i such that $1 \leq i \leq p$, $e_{a_i} = i$,
- otherwise $e_i = Z$ for some new variable Z.

Then we have $s \circ \overline{s} =_\sigma id$.

Proof: First notice that the e_j are well defined since all the a_i are assumed to be distinct.
$s \circ \overline{s} = (a_1 \ldots a_p.\uparrow^n) \circ (e_1 \ldots e_n.\uparrow^p) = (a_1[\overline{s}] \ldots a_p[\overline{s}].\uparrow^n \circ \overline{s})$. But since $p \leq n$, $a_i[\overline{s}] = e_{a_i} = i$, thus $s \circ \overline{s} = (1 \ldots p.\uparrow^p) = id$. □

For example a right inverse of the pattern substitution $s = (1.3.4.\uparrow^5)$ is $\overline{s} = (1.W.2.3.Z.\uparrow^3)$ since $s \circ \overline{s} = (1.3.4.\uparrow^5) \circ (1.W.2.3.Z.\uparrow^3) =_\sigma (1.2.3.\uparrow^3) =_\sigma id$.

Notice that, as in this example, a right inverse of a pattern substitution is not itself a pattern substitution in general.

As composition is to be read from left to right, we have the following corollary.

Corollary 1 Any pattern substitution is injective.

2.4 Image

Pattern substitutions are injective, thus an equation of the form $X[s] =^?_{\lambda\sigma} b$ has at most one solution. But they need not be surjective, thus such an equation may have no solution. In this section we show that given a pattern substitution s and a term a we can decide if a is in the image of s and if so we can compute the preimage of a under s.

Definition 4 A de Bruijn index k is said to *occur* in a normal term (resp. a normal substitution):

- λa, if $k+1$ occurs in a,
- $(\mathtt{n}\ a_1\ \ldots\ a_p)$, if $k = n$ or k occurs in some a_i,
- $(X[s]\ a_1\ \ldots\ a_p)$, if k occurs in s or in some a_i,
- $a_1 \ldots a_p.\uparrow^n$, if k occurs in some a_i or if $k > n$.

For example, the index 1 occurs in the term $\lambda 2$, but the index 2 does not. Any index occurs in $(X\ a_1\ \ldots\ a_p)$ (to be read as $(X[\uparrow^0]\ a_1\ \ldots\ a_p)$).

Only a finite number of indices do not occur in a substitution s and one can compute this finite set. Then for each index, one can check if it occurs in a term a. Thus one can decide if all the indices occurring in a occur in s.

Proposition 6 (Inversion lemma) Let s be a pattern substitution, a be a normal term (resp. t be a normal substitution). There exists a term a' (resp. a substitution t') such that $a = a'[s]$ (resp. $t = t' \circ s$) if and only if every index occurring in a (resp. t) occurs in s.

Proof: By induction over the structure of a (resp. t).

- If $a = \lambda b$ then
  there exists a', $a = a'[s]$
  if and only if there exists b', $a = (\lambda b')[s]$ (because the long normal form of a' is of the form $\lambda b'$),
  if and only if there exists b', $b = b'[1.(s \circ \uparrow)]$,
  if and only if every index occurring in b occurs in $1.(s \circ \uparrow)$,
  if and only if for every index k such that $k+1$ occurs in b, $k+1$ occurs in $1.(s \circ \uparrow)$,
  if and only if for every k such that $k+1$ occurs in b, k occurs in s,
  if and only if every index occurring in a occurs in s.
- If $a = (m\ b_1\ \ldots\ b_q)$ then
  there exists $a', a = a'[s]$
  if and only if there exists $m', b'_1, \ldots, b'_q$, $a = (m'\ b'_1\ \ldots\ b'_q)[s]$,
  if and only if there exists $m', b'_1, \ldots, b'_q$, $m = m'[s]$ and $b_i = b'_i[s]$,
  if and only if m occurs in s and every index occurring in b_i occurs in s,
  if and only if every index occurring in a occurs in s.
- If $a = (Y[t]\ b_1\ \ldots\ b_q)$ then
  there exists a' such that $a = a'[s]$

if and only if there exists $t', b'_1, \ldots, b'_q$ such that $a = (Y[t']\ b'_1\ \ldots\ b'_q)[s]$,
if and only if there exists $t', b'_1, \ldots, b'_q$ such that $t = t' \circ s$ and $b_i = b'_i[s]$,
if and only for every index occurring in t, $b_1\ \ldots\ b_q$, occurs in s,
if and only if every index occurring in a occurs in s.

- If $t = b_1 \ldots b_q.\ \uparrow^m$ then let s be $e_1 \ldots e_p.\ \uparrow^n$.
  There exists t' such that $t = t' \circ s$
  if and only if there exists t' of the form $c_1 \ldots c_{q+n+r}.\ \uparrow^{p+l}$ such that $t = t' \circ s$,
  (i.e. expanding the substitution t' enough, we can assume without loss of generality that it has more that $q+n$ terms and that the exponent of the shift is bigger than p)
  if and only if there exists $c_1, \ldots, c_{q+n+r}$, l such that $c_1[s] = b_1, \ldots, c_q[s] = b_q, c_{q+1}[s] = m+1, \ldots, c_{q+n+r}[s] = m+n+r, \uparrow^{l+n} = \uparrow^{m+n+r}$,
  if and only if every index occurring in b_i occurs in $s, m+1, \ldots, m+n+r$ occur in s,
  if and only if every index occurring in t occurs in s.

□

For instance, the term (2 3) is in the image of $\uparrow$ because 1 does not occur in it. Thus the equation $X[\uparrow] =^?_{\lambda\sigma}$ (2 3) has a solution ($X = (1\ 2)$). But the term (1 3) is not in the image of $\uparrow$ because 1 occurs in it. Thus the equation $X[\uparrow] =^?_{\lambda\sigma}$ (1 3) has no solution.

Proposition 7 Let s be a pattern substitution, a be a normal term (resp. t be a normal substitution) such that the normal form of $a[s]$ is an atomic pattern (resp. the normal form of $t \circ s$ is a pattern substitution) then a is an atomic pattern (resp. t is a pattern substitution).

Proof: By induction over the structure of a (resp. t).

- If $a = \lambda b$ then $a[s] = \lambda b[1.(s \circ \uparrow)]$. As $a[s]$ is an atomic pattern, then $b[1.(s \circ \uparrow)]$ is an atomic pattern, thus b is an atomic pattern and a is an atomic pattern.
- If $a = (m\ b_1\ \ldots\ b_q)$ then $a[s] = (m[s]\ b_1[s]\ \ldots\ b_q[s])$. As $a[s]$ is an atomic pattern, then $b_1[s], \ldots, b_q[s]$ are atomic patterns, thus $b_1, \ldots, b_q$ are atomic patterns and a is an atomic pattern.

- If $a = X[t]$ then $a[s] = X[t \circ s]$. As $a[s]$ is an atomic pattern, then $t \circ s$ is a pattern substitution, thus t is a pattern substitution and a is an atomic pattern.
- If $t = b_1 \dots b_q.\uparrow^m$ then let $a_1 ... a_p.\uparrow^n$ be the substitution s. We have

$$t \circ s = b_1[s] \dots b_q[s].a_{m+1} \dots a_p.\uparrow^n$$

if $m < p$, and

$$t \circ s = b_1[s] \dots b_q[s].\uparrow^{(m-p)+n}$$

if $m \geq p$.

In both cases, since $t \circ s$ is a pattern substitution, $b_1[s] \dots b_q[s]$ are distinct indexes lower than or equal to n. Thus b_1, ..., b_q are de Bruijn indices and they are distinct.

If some b_i is greater than m, say $b_i = m + r$ then we would get $i[t] = b_i = (q+r)[t]$. Thus $i[t \circ s] = (q+r)[t \circ s]$ and this would contradict injectivity.

□

Proposition 8 Let s be a pattern substitution, a be a normal term (resp. t be a normal substitution). If there exists a term a' (resp. a substitution t') such that $a = a'[s]$ (resp. $t = t' \circ s$) then $a' = a[\overline{s}]$ (resp. $t' = t \circ \overline{s}$).

Proof: If $a = a'[s]$ then $a[\overline{s}] = a'[s][\overline{s}] = a'[s \circ \overline{s}] = a'[id] = a'$.
If $t = t' \circ s$ then $t \circ \overline{s} = t' \circ s \circ s' = t' \circ id = t'$. □

2.5 Fixpoints

As we shall see below, when we have an equation of the form $X =^?_{\lambda\sigma} t$, then three cases may occur. When X has no occurrence in t, X can be replaced by t everywhere. If X occurs in t but not at top level, then we shall see that the equation has no solution. The third case is that of equations of the form $X =^?_{\lambda\sigma} X[s]$. So, in this section we give a characterization of the fixpoints of a pattern substitution s.

Proposition 9 Let s be a pattern substitution, a be a normal term (resp. t be a normal substitution). We have $a[s] = a$ (resp. $t \circ s = t$) if and only if every k occurring in a (resp. t) is a fixpoint of s (i.e. is such that $k[s] = k$).

Proof: By induction on the structure of a (resp. t).

- if $a = \lambda b$ then
  $a[s] = a$
  if and only if $b[1.s \circ \uparrow] = b$,
  if and only if every k occurring in b is such that $k[1.s \circ \uparrow] = k$,
  if and only if every k such that $k+1$ occurs in b is such that $(k+1)[1.s \circ \uparrow] = k+1$,
  if and only if every k such that $k+1$ occurs in b is such that $k[s] = k$,
  if and only if every index occurring in a is a fixpoint of s.
- If $a = (n\ b_1\ \ldots\ b_p)$ then
  $a[s] = a$
  if and only if $n[s] = n$ and $b_i[s] = b_i$,
  if and only if $n[s] = n$ and every index occurring in some b_i is a fixpoint of s,
  if and only if every index occurring in a is a fixpoint of s.
- If $a = (X[t]\ b_1\ \ldots\ b_p)$ then
  $a[s] = a$
  if and only if $b_i[s] = b_i$, $t \circ s = t$,
  if and only if every index occurring in some b_i is a fixpoint of s and every index occurring in t is a fixpoint of s,
  if and only if every index occurring in a is a fixpoint of s.
- If $t = b_1 \ldots b_p.\ \uparrow^n$ then
  $t \circ s = t$
  if and only if $b_1[s] \ldots b_p[s].(\uparrow^n \circ s) = b_1 \ldots b_p.\ \uparrow^n$,
  if and only if $b_1[s] = b_1$, ..., $b_p[s] = b_p$ and $\uparrow^n \circ s = \uparrow^n$,
  if and only if all the indices occurring in $b_1, \ldots, b_p$ are fixpoints of s and for every r, $r[\uparrow^n \circ s] = r[\uparrow^n]$,
  if and only if all the indices occurring in $b_1, \ldots, b_p$ are fixpoints of s and for every r, $(r+n)[s] = r+n$,
  if and only if every index occurring in t is a fixpoint of s.

□

3 Pattern unification

3.1 Transformation rules

Let us first define the equation systems on which we are working now.

Definition 5 The language of (unification) *constraints* is defined by [10]:

$$\mathcal{C} \quad ::= \quad (\Gamma \vdash a =^{?}_{\lambda\sigma} b) \mid \mathbb{T} \mid \mathbb{F} \mid (\mathcal{C}_1 \wedge \mathcal{C}_2) \mid (\exists X.\mathcal{C})$$

Here $\mathbb{T}$ represents the constraint that is always satisfied, $\mathbb{F}$ the unsatisfiable constraint, a and b are any $\lambda\sigma$-terms of the same type in a context Γ, and $X \in \mathcal{X}$. The set of term variables of a constraint P (resp. of a term a) is denoted $\mathcal{V}ar(P)$ (resp. $\mathcal{V}ar(a)$). A constraint is sometimes called a *system*.

We take propositional constraint simplification (such as $\mathcal{C} \wedge \mathbb{T} \twoheadrightarrow \mathcal{C}$) for granted and generally assume that constraints are maintained in prenex form. For simplicity, we often omit the leading existential quantifiers and the context Γ associated with equations.

Definition 6 A *$\lambda\sigma$-pattern unification problem* is any unification constraint, built only on $\lambda\sigma$-patterns.

As usual, a $\lambda\sigma$-unifier (also called a solution) of a constraint P is a grafting making the two sides of every equation of P equal modulo $\lambda\sigma$. The set of $\lambda\sigma$-unifiers of a system P is denoted $\mathcal{U}_{\lambda\sigma}(P)$. In our setting, the solution of a $\lambda\sigma$-pattern unification is the result of the simplification of the initial problem to a *solved form*.

Definition 7 A *$\lambda\sigma$-pattern solved form* is any conjunction of equations:

$$\exists \vec{Z}.\ X_1 =^{?}_{\lambda\sigma} a_1 \wedge \cdots \wedge X_n =^{?}_{\lambda\sigma} a_n$$

such that $\forall 1 \leq i \leq n, X_i \in \mathcal{X}$, a_i is a pattern, and:

$$\begin{array}{lll} (i) & \forall 1 \leq i < j \leq n & X_i \neq X_j, \\ (ii) & \forall 1 \leq i, j \leq n & X_i \notin \mathcal{V}ar(a_j), \\ (iii) & \forall 1 \leq i \leq n & X_i \notin \vec{Z}, \\ (iv) & \forall Z \in \vec{Z}, \exists 1 \leq j \leq n & Z \in \mathcal{V}ar(a_j). \end{array}$$

Note that flexible-flexible equations (see [6]) do not occur in $\lambda\sigma$-pattern solved forms. A solved form therefore trivially determines a unique grafting which is its unifier. In order to state the transformation rules, we need the following definitions and properties. When a de Bruijn index has an occurrence in a term, we can consider two cases. First, when there is no variable on the path from the root of the term to the index, this index cannot be removed by any substitution. In this case, it is said to have a rigid occurrence. Otherwise the occurrence is said to be flexible.

Definition 8 A de Bruijn index k is said to have a *rigid occurrence* in an atomic pattern

- λa if $k+1$ has a rigid occurrence in a,
- $(\mathtt{n}\ a_1\ \ldots\ a_p)$ if $k = n$ or k has a rigid occurrence in some a_i,
- $X[s]$ never.

Definition 9 A de Bruijn index k is said to have a *flexible occurrence* in an atomic pattern:

- λa if $k+1$ has a flexible occurrence in a,
- $(\mathtt{n}\ a_1\ \ldots\ a_p)$ if k has a flexible occurrence in some a_i,
- $X[a_1\ \ldots\ a_p.\uparrow^n]$ if some a_i is k or if $k > n$.

Proposition 10 If k has a flexible occurrence in a then a has a subterm of the form $X[a_1\ \ldots\ a_p.\uparrow^n]$ under l abstractions and some a_i is equal to $k+l$. We say that k has a flexible occurrence in a, *in the* i^{th} *argument of the variable* X.

Given a $\lambda\sigma$-pattern unification problem, the transformation rules described in Figures 4 and 5 allow normalizing it into a solved form when it has solutions, or into $\mathbb{F}$ otherwise.

The main differences between pattern unification and $\lambda\sigma$-unification lie in the treatment of flexible-rigid equations (see the rule **Exp-app** of [6]). When we have an equation of the form $X[s] =^?_{\lambda\sigma} b$, then if b is in the image of s, we can invert the substitution and we get the equation $X =^?_{\lambda\sigma} b[\overline{s}]$ (rule **Invert**). If the term b is not in the image of s there is an index occurring in this term but not in the substitution s. If this index has a rigid occurrence, then the equation has no solution (rule **Pruning1**). When it is flexible, we can eliminate it by substituting the variable on its path (**Pruning2**). This allows us to bypass the **Exp-app** rule and also to solve all flexible-flexible equations. We are reduced this way to consider only equations of the form $X =^?_{\lambda\sigma} b$.

Three cases may then occur. When X has no occurrence in b, X is replaced by b everywhere (rule **Replace**). If X occurs in b but not at top level, then the equation has no solution (rule **Occur-check**). The third case is that of equations of the form $X =^?_{\lambda\sigma} X[s]$ (rule **Same-variable**). Notice that in this case, since the terms X and $X[s]$ have the same type, the substitution s has the form $a_1 ... a_n.\uparrow^n$.

Since there is no transformation introducing disjunctions, the unitary character of $\lambda\sigma$-pattern unification becomes trivial once the completeness and correctness of the rules is shown together with a terminating strategy of the rules application.

3.2 Examples

3.2.1 An succeeding example

Let us show how these rules act on a simple example. Consider the pattern equation:

$$\lambda x\ \lambda y\ \lambda z\ (F\ z\ y) =^?_{\beta\eta} \lambda x\ \lambda y\ \lambda z\ (z\ (G\ y\ x))$$

In de Bruijn notation this equation is:

$$\lambda\lambda\lambda(F\ 1\ 2) =^?_{\beta\eta} \lambda\lambda\lambda(1\ (G\ 2\ 3))$$

Translating this equation by precooking yields:

$$\lambda\lambda\lambda(F[\uparrow^3]\ 1\ 2) =^?_{\lambda\sigma} \lambda\lambda\lambda(1\ (G[\uparrow^3]\ 2\ 3))$$

Using **Dec-λ** this equation simplifies to:

$$(F[\uparrow^3]\ 1\ 2) =^?_{\lambda\sigma} (1\ (G[\uparrow^3]\ 2\ 3))$$

Using **Exp-λ** four times we instantiate F by $\lambda\lambda X$ and G by $\lambda\lambda Y$. So we get:

$$X[2.1.\uparrow^3] =^?_{\lambda\sigma} (1\ (Y[3.2.\uparrow^3]))$$

Now consider the substitution $2.1.\uparrow^3$. The only de Bruijn number not occurring in it is 3. This numbers occurs in the right-hand side in $Y[3.2.\uparrow^3]$. Thus with the rule **Pruning2** we add the equation $Y =^?_{\lambda\sigma} Z[\uparrow]$. With the rule **Replace** we get:

$$X[2.1.\uparrow^3] =^?_{\lambda\sigma} (1\ (Z[2.\uparrow^3]))$$

Now 3 has no more occurrences in the right-hand side, thus the rule **Invert** yields:

$$X =^?_{\lambda\sigma} (1\ (Z[2.\uparrow^3]))[2.1.Z'.\uparrow^2]$$

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

This gives:

$$F =^?_{\lambda\sigma} \lambda\lambda(2\ (Z[1.\uparrow^2]))$$

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

Dec-app1
$P \wedge (\mathtt{n}\ a_1\ \ldots\ a_p) =^?_{\lambda\sigma} (\mathtt{n}\ b_1\ \ldots\ b_p)$
$\mapsto\!\!\!\!\twoheadrightarrow$
$P \wedge (\bigwedge_{i=1..p} a_i =^?_{\lambda\sigma} b_i)$

Dec-app2
$P \wedge (\mathtt{n}\ a_1\ \ldots\ a_p) =^?_{\lambda\sigma} (\mathtt{m}\ b_1\ \ldots\ b_q)$
$\mapsto\!\!\!\!\twoheadrightarrow$
$\mathbb{F}$
if $n \neq m$

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

Occur-check
$P \wedge X =^?_{\lambda\sigma} a$
$\mapsto\!\!\!\!\twoheadrightarrow$
$\mathbb{F}$
if $X \in \mathcal{V}ar(a)$ and a has not the form $X[s]$

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

Normalize
$P \wedge a =^?_{\lambda\sigma} b$
$\mapsto\!\!\!\!\twoheadrightarrow$
$P \wedge a' =^?_{\lambda\sigma} b'$
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 4: **PatternUnif**: Rules for pattern unification in $\lambda\sigma$: Part 1

Pruning1

$P \wedge X[s] =^?_{\lambda\sigma} b$
$\mapsto\!\!\!\!\rightarrow$
$\mathbb{F}$
if b is an atomic pattern, s is a pattern substitution and some index k has no occurrence in s but a rigid occurrence in b

Pruning2

$P \wedge X[s] =^?_{\lambda\sigma} b$
$\mapsto\!\!\!\!\rightarrow$
$\exists Z(P \wedge X[s] =^?_{\lambda\sigma} b \wedge Y =^?_{\lambda\sigma} Z[1 \ldots i-1.\uparrow^i])$
if b is an atomic pattern, s is a pattern substitution and some index k has no occurrence in s and a flexible occurrence in b in the i^{th} argument of the variable Y

Invert

$P \wedge X[s] =^?_{\lambda\sigma} b$
$\mapsto\!\!\!\!\rightarrow$
$P \wedge X =^?_{\lambda\sigma} b[\overline{s}]$
if b is an atomic pattern, s is a pattern substitution and X does not occur in b, all the de Bruijn indices occurring in b occur in s

Same-variable

$P \wedge X =^?_{\lambda\sigma} X[a_1 \ldots a_n.\uparrow^n]$
$\mapsto\!\!\!\!\rightarrow$
$\exists Y(P \wedge X =^?_{\lambda\sigma} Y[i_1 \ldots i_r.\uparrow^n])$
where $a_1 \ldots a_n.\uparrow^n$ is a pattern substitution and $i_1 \ldots i_r$ are the indices such that $a_i = i$

Figure 5: **PatternUnif**: Rules for pattern unification in $\lambda\sigma$: Part 2

$$G =^?_{\lambda\sigma} \lambda\lambda(Z[\uparrow])$$

As shown below, we can translate these solutions back in λ-calculus. With **Anti-Exp-λ** we get $Z =^?_{\lambda\sigma} (H[\uparrow]\ 1)$ and:

$$F =^?_{\lambda\sigma} \lambda\lambda(2\ (H[\uparrow^2]\ 1))$$

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

which is the pre-cooking of:

$$F =^?_{\lambda\sigma} \lambda\lambda(2\ (H\ 1))$$

$$G =^?_{\lambda\sigma} \lambda\lambda(H\ 2)$$

i.e.:

$$F =^?_{\beta\eta} \lambda u\ \lambda v\ (u\ (H\ v))$$
$$G =^?_{\beta\eta} \lambda u\ \lambda v\ (H\ u)$$

3.2.2 An example failing by the rule Pruning1

$$\lambda x\ \lambda y\ \lambda z\ (F\ z\ y) =^?_{\beta\eta} \lambda x\ \lambda y\ \lambda z\ (x\ (G\ y\ x))$$

we get:

$$X[2.1.\uparrow^3] =^?_{\lambda\sigma} (3\ (Y[3.2.\uparrow^3]))$$

3 has now a rigid occurrence in the right-hand side and the system fails by the rule **Pruning1**.

3.2.3 An example failing by the rule Occur check

$$\lambda x\ \lambda y\ \lambda z\ (F\ x\ y\ z) =^?_{\beta\eta} \lambda x\ \lambda y\ \lambda z\ (z\ (F\ z\ y\ x))$$

we get:

$$X =^?_{\lambda\sigma} (1\ (X[3.2.1.\uparrow^3]))$$

and the system fails by occur check.

3.3 Correctness and completeness

We can now show that the **PatternUnif** rules are correct and complete.

Proposition 11 Any rule **r** in **PatternUnif** is correct (i.e. $P \mapsto\!\!\!\!\!\rightarrow^{\mathbf{r}} P' \Rightarrow \mathcal{U}_{\lambda\sigma}(P') \subseteq \mathcal{U}_{\lambda\sigma}(P)$) and complete (i.e. $P \mapsto\!\!\!\!\!\rightarrow^{\mathbf{r}} P' \Rightarrow \mathcal{U}_{\lambda\sigma}(P) \subseteq \mathcal{U}_{\lambda\sigma}(P')$).

Proof: This needs to be proved only for the rules in Figure 5 since the other have been proved correct and complete in [6] in the more general case of any $\lambda\sigma$-unification problem.

Pruning1 is obviously correct. It is complete because if k has a rigid occurrence in b then for any grafting θ, k has a rigid occurrence in θb. Thus by the inversion lemma θb is not in the image of s and the problem $X[s] = b$ has no solutions.

Pruning2 is obviously correct. Let us prove it is complete.

Let θ be a grafting such that $(\theta X)[s] = \theta b$ and let $Y[c_1 \dots c_p.\ \uparrow^n]$ be the subterm of b under l abstractions such that $c_i = k + l$. Then i does not occur in θY.

Indeed, if i occurred in θY then $k + l$ would occur in $(\theta Y)[c_1 \dots c_q.\ \uparrow^n]$ thus (rigid path) k would occur in θb, and by the inversion lemma θb would not be in the image of s.

Thus by the inversion lemma θY is in the image of the substitution $1.2 \dots i - 1.\ \uparrow^i$. Let d be the term such that $\theta Y = d[1.2 \dots i - 1.\ \uparrow^i]$ then $\theta \cup \{< Z, d >\}$ is a solution to the generated problem.

Invert is correct by Proposition 5. Let us prove it is complete. If $\theta X = \theta(b[\overline{s}])$ then $\theta(X[s]) = \theta(b[\overline{s}][s])$. By proposition 8, b is the image of s thus $b[\overline{s}][s] = b$ and $\theta(X[s]) = \theta b$.

Same-variable is obviously correct, let us prove it is complete.

Let θ be a grafting solution to the initial problem.

We have $\theta X = (\theta X)[a_1 \dots a_n.\ \uparrow^n]$. Thus by the proposition 9, all the indices occurring in θX are among $i_1, \dots, i_r, n + 1, n + 2, \dots$ Thus by the inversion lemma, θX is in the image of $i_1 \dots i_r.\ \uparrow^n$. Let c such that $\theta X = c[i_1 \dots i_r.\ \uparrow^n]$, the grafting $\theta \cup \{< Y, c >\}$ is a solution to the problem. □

3.4 A unification algorithm

We should now show how the rules in **PatternUnif** can be used in order to solve a $\lambda\sigma$-pattern unification problem, i.e., reduce it to a $\lambda\sigma$-pattern solved form. We first check that atomic pattern unification problems are stable under the unification transformations, i.e. applying a unification rule of **PatternUnif** (except the rules **Anti-***) on an atomic pattern problem yields an atomic problem after normalization. Then one can prove that an atomic pattern unification problem is in normal form for the rules in **PatternUnif** (except the rules **Anti-***), if it is either $\mathbb{F}$ or a in solved form.

Proposition 12 An atomic pattern unification problem is in normal form for the rules in **PatternUnif** , if it is either $\mathbb{F}$ or a $\lambda\sigma$-pattern solved form.

Proof: Solved forms are obviously normal.

Conversely consider an atomic pattern unification problem. If it does not have the form $X[s] =^?_{\lambda\sigma} b$ or $X =^?_{\lambda\sigma} b$ then one of the **Dec-*** rule applies. If it has

the form $X[s] =^?_{\lambda\sigma} b$ then one of the rules **Pruning1**, **Pruning2** or **Invert** applies. If it has the form $X =^?_{\lambda\sigma} b$ then either X has an occurrence in b and one of the rules **Occur check** or **Same-variable** applies. If X has no occurrence in b then either the rule **Replace** applies or the equation is solved. □

Proposition 13 The above strategy in the application of the **PatternUnif** rules is terminating on atomic pattern problems.

Proof: After any application of **Pruning2** or **Same-variable** the rule **Replace** is applicable, after any application of **Invert** and **Normalize** one of the rules **Occur-check**, **Same-variable** or **Replace** is applicable. Thus on a unsolved form one of these sequences can be applied.

- **Dec-***
- **Pruning1**
- **Pruning2;Replace**
- **Invert;Normalize;Occur-check**
- **Invert;Normalize;Same-variable;Replace**
- **Invert;Normalize;Replace**.
- **Same-variable;Replace**
- **Replace**

At each application, the sum of the sizes of the contexts of the unsolved variables decreases, but for the rule **Dec-*** that keeps the sum of the sizes of the contexts of the unsolved variables and decreases the size of the problem. □

The main result becomes now clear since any problem either has no solution or can be transformed into an equivalent solved problem with no disjunction:

Proposition 14 Unification of atomic $\lambda\sigma$-patterns is decidable and unitary.

Proof: The **PatternUnif** rules are correct and complete, and using the strategy described above, their application on any $\lambda\sigma$-pattern unification problem terminate. This yields either failure when the initial problem has no solution or a unique $\lambda\sigma$-pattern solved form. □

For non-atomic pattern problems the application of the rules (**Exp-λ**; **Replace**; **Normalize**) is trivially terminating and yields an atomic pattern problem. Thus:

Theorem 1 *Unification of $\lambda\sigma$-patterns is decidable and unitary.*

If the problem we start with is the pre-cooking of some problem in λ-calculus in a context Γ, then we may want to translate back the solved problem in λ-calculus. This can be done with the rules **Anti-***.

But we need to prove the following invariants of transformations.

Proposition 15 The **PatternUnif** rules preverve the following properties:

1. the context of the $\lambda\sigma$-pattern unification problem has the form $A_1 \dots A_n.\Gamma$,
2. for every variable X, the context Γ_X has the form $B_1 \dots B_p.\Gamma$,
3. for every subterm $X[a_1 \cdots a_p.\uparrow^n]$ of a derived problem we have $p \leq |\Gamma_X| - |\Gamma|$.

Proof:

- For the rule **Pruning1** the conservation of invariants is trivial.
- For the rule **Pruning2**, the variable Y occurs in a subterm of the form $Y[a_1 \dots a_p.\uparrow^n]$ thus we have $\Gamma_Y = A_1 \dots A_p.B_1 \dots B_q\Gamma$ The context of Z is $A_1 \dots A_{i-1}A_{i+1}A_pB_1 \dots B_q\Gamma$. and in $Z[1 \dots (i-1).\uparrow^i]$ we have $i-1 \leq p-1 \leq (p-1)+q$.
- For the rule **Invert**, as the equation $X[s] =^?_{\lambda\sigma} b$ verifies the invariant we have $\Gamma_X = A_1 \dots A_p.C_1 \dots C_q\Gamma$ and thus the context of the equation is $\Delta = B_1 \dots B_n.C_1 \dots C_q\Gamma$. The substitution $\overline{s} = e_1 \dots e_n.\uparrow^p$ is such that every e_i is well typed in Γ_X and is either a de Bruijn index or a variable. Then we have $n \leq n+q$ thus, by the case 2 of proposition 4.6 of [6] the normal form of $b[\overline{s}]$ verifies the invariant.
- For the rule **Same-variable** we have $\Gamma_X = A_1 \dots A_nB_1 \dots B_q\Gamma$. The context of Y is $A_{i_1} \dots A_{i_r}B_1 \dots B_q.\Gamma$. and in $Z[i_1 \dots i_r.\uparrow^n]$ we have $r \leq n \leq n+q$.
- The other rules are proved to verify this invariant in [6].

□

So we are now in shape to state a similar theorem as for full unification in $\lambda\sigma$-calculus.

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

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

Figure 6: **AntiRules**

Theorem 2 *The most general unifier of a higher-order pattern unification problem* $a =^?_{\beta\eta} b$ *can be obtained by pre-cooking followed by the application of the* **PatternUnif** *rules under the strategy we have described and ended by the* **Anti-*** *rules given in Figure 6 and back-cooking.*

In comparison with the standard presentation of pattern unification, our presentation does not need mixed prefixes to simplify abstractions. Here we just reap the benefit of the encoding of scopes constraints in the calculus itself [6]. Similarly, the raising step is merely our **Anti-Exp-λ** rule. Most flexible-rigid and flexible-flexible cases can be uniformly reduced to inversion of a pattern substitution, with the case of two identical head variables being the only exception (rule **Same-variable**).

4 Towards an efficient implementation

The literal interpretation of the algorithm in Section 3.4 is still quite impractical. For example, pruning is done one variable at a time, and so the right-hand side of a variable/term equation would have to be traversed and copied many times. In this section we present an algorithm which is close to the actual efficient implementation of pattern unification in the MLF (modular LF) language [9] and in the system Twelf [26].

Since we generalize pattern unification to a constraint simplification algorithm which does not require the pattern restriction, we use ξ and ζ to range over pattern substitutions, later to be distinguished from arbitrary substitutions.

4.1 Atomic weak head normal forms

The rule **Normalize** is not practical, since it is in general too expensive to reduce the whole term into normal form, perhaps only to discover later that the two terms we unify disagree in their top-level constructor. Instead, we transform the term only into *weak head-normal form* (see also [2]).

Definition 10 A $\lambda\sigma$-term a is in *atomic weak head-normal form* if it has the form:

- $\lambda_A a$, where a is arbitrary,
- $(\mathtt{n}\ a_1\ \ldots\ a_p)$ where $a_1, \ldots, a_p$ are arbitrary,
- $X[s]$ where s is arbitrary.

Proposition 16 If every meta-variable in a has atomic type, then a has an atomic weak head-normal form.

Proof: Head reduction is a complete strategy for well-typed terms. □

If a contains meta-variables with non-atomic type, we can obtain an equivalent term (with respect to unification) by instantiating functional variables with new variables of lower type (see rule **Exp-λ** in Figure 4). This instantiation must be recorded as an equation, so converting a term to atomic weak head-normal form may introduce new constraints.

So as not to clutter the notation, we assume a global constraints store $\mathcal{CS}$ to which new equations may be added during the transformations. In its simplest form (e.g., when all terms are patterns), $\mathcal{CS}$ represents a substitution. We write $\widehat{a}$ for the atomic weak head-normal form of a obtained by successive head reductions as modeled in the $\lambda\sigma$-calculus. As noted, this operation may add equations to the constraints store.

4.2 Constraints transformations

In this section we postulate (and maintain) that in an equation all terms are in atomic weak head-normal form unless the equation is solved, that is, of the form $X =^{?}_{\lambda\sigma} b$ where X does not occur in b. Furthermore, solved variables are immediately propagated so that X is replaced by b throughout the other constraints, i.e., **Replace** is applied eagerly. In the implementation, this is modeled by an efficient destructive update of a pointer associated with X in the manner familiar from Prolog implementations.

AA	$a_1\ a_2 =^?_{\lambda\sigma} b_1\ b_2$	$\rightarrow$	$a_1 =^?_{\lambda\sigma} b_1 \wedge \widehat{a_2} =^?_{\lambda\sigma} \widehat{b_2}$	
LL	$\lambda_A a =^?_{\lambda\sigma} \lambda_A b$	$\rightarrow$	$\widehat{a} =^?_{\lambda\sigma} \widehat{b}$	
LT	$\lambda_A a =^?_{\lambda\sigma} b$	$\rightarrow$	$\widehat{a} =^?_{\lambda\sigma} \widehat{(b[\uparrow]1)}$	if b is not of the form $\lambda_A b_1$
TL	$b =^?_{\lambda\sigma} \lambda_A a$	$\rightarrow$	$\widehat{a} =^?_{\lambda\sigma} \widehat{(b[\uparrow]1)}$	if b is not of the form $\lambda_A b_1$
NN $=$	$n =^?_{\lambda\sigma} n$	$\rightarrow$	$\mathbb{T}$	
NN $\neq$	$n =^?_{\lambda\sigma} m$	$\rightarrow$	$\mathbb{F}$	if $n \neq m$
NA	$n =^?_{\lambda\sigma} b_1\ b_2$	$\rightarrow$	$\mathbb{F}$	
AN	$b_1\ b_2 =^?_{\lambda\sigma} n$	$\rightarrow$	$\mathbb{F}$	
VV	$X[\xi] =^?_{\lambda\sigma} X[\zeta]$	$\rightarrow$	$X =^?_{\lambda\sigma} Y[\xi \cap \zeta]$	
VT	$X[\xi] =^?_{\lambda\sigma} b$	$\rightarrow$	$X =^?_{\lambda\sigma} b[\xi]^{-1}$	if $b[\xi]^{-1}$ exists and X not rigid in $NF(b)$
TV	$b =^?_{\lambda\sigma} X[\xi]$	$\rightarrow$	$X =^?_{\lambda\sigma} b[\xi]^{-1}$	if $b[\xi]^{-1}$ exists and X not rigid in $NF(b)$
OC1	$X[\xi] =^?_{\lambda\sigma} b$	$\rightarrow$	$\mathbb{F}$	if neither **VV** nor **VT** applies
OC2	$b =^?_{\lambda\sigma} X[\xi]$	$\rightarrow$	$\mathbb{F}$	if neither **VV** nor **TV** applies

Figure 7: Practical transformations

The **PatternUnif** rules can be implemented by the transformation rules described in Figure 7 with the auxiliary operations $\xi \cap \zeta$ and $a[\zeta]^{-1}$ which are explained below.

The main differences between the **PatternUnif** rules and the implemented ones are the following. The η-expansion is done lazily by the rules **LT** and **TL**. The decomposition of applications is done incrementally, using the fact that applications are always rigid terms since meta-variables are assumed to be atomic. The rule **Same-variable** is replaced by the computation of the substitution $\xi \cap \zeta$. Inverting a substitution and applying the inverse to a term is now a single operation. Pruning is replaced by adding equations during this operation. We also omit the context Γ which could be reconstructed easily.

The rules **VT**, **TV**, **OC1** and **OC2** contain side conditions on occurrences of a meta-variable X in a term. For the pattern fragment, any meta-variable occurrence is rigid, so simply the occurrence or absence of X from the normal form of b is in question. These rules remain valid under the generalization away from patterns in section 5.4, but only if we restrict the occurrences to be rigid. In the actual implementation this is not checked separately (which would require expensive traversal and possibly normalization of b), but simultaneously with the computation of $b[\xi]^{-1}$ in rules **VT** and **TV**.

Given two pattern substitutions ξ and ζ with the same domain and co-domain, we define $\xi \cap \zeta$ by:

$$\begin{array}{rcl} i.\xi \cap i.\zeta & = & 1.((\xi \cap \zeta) \circ \uparrow) \\ i.\xi \cap j.\zeta & = & (\xi \cap \zeta) \circ \uparrow \quad \text{if } i \neq j \\ i.\xi \cap \uparrow^n & = & i.\xi \cap (n+1).\uparrow^{n+1} \\ \uparrow^m \cap j.\zeta & = & (m+1).\uparrow^{m+1} \cap j.\zeta \\ \uparrow^n \cap \uparrow^n & = & id \end{array}$$

Note that $\uparrow^n \cap \uparrow^m$ for $n \neq m$ cannot arise, since the substitutions are required to have the same domain and co-domain.

4.3 Inverting substitutions

The properties from Section 2 show that the right inverse of a pattern substitution exists. The proof of this property is constructive and implicitly defines an algorithm to compute this inverse. In this section we write out another version of this algorithm in a form amenable to an efficient implementation.

Assume we have:

$$\Gamma \vdash b : A$$
$$\Gamma \vdash \xi : \Delta$$

where b is in atomic weak head-normal form. We define $b[\xi]^{-1}$ so that $\Delta \vdash b[\xi]^{-1} : A$ and $(b[\xi]^{-1})[\xi] = b$ if $b[\xi]^{-1}$ exists and b contains no free variables. If b does contain free variables, then the inversion may generate some additional constraints (as in the pruning rules). In that case we have $(b[\xi]^{-1})[\xi] = b$ only modulo the solution to the generated constraints. We need to define the operation mutually recursively with $\zeta \circ \xi^{-1}$, where:

$$\Gamma \vdash \zeta : \Gamma'$$
$$\Gamma \vdash \xi : \Delta$$

and then $\Delta \vdash \zeta \circ \xi^{-1} : \Gamma'$ and $(\zeta \circ \xi^{-1}) \circ \xi = \zeta$ if $\zeta \circ \xi^{-1}$ exists.

The application of an inverse substitution is described in Figure 8. In the final case, $\zeta' = \zeta \mid \xi$ is the *pruning substitution* which guarantees that $(\zeta' \circ \zeta) \circ \xi^{-1}$ exists. Such a pruning substitution always exists but we must take care to construct it such

A	$(a_1\ a_2)[\xi]^{-1}$	$=$	$(a_1[\xi]^{-1})\ (\widehat{a_2}[\xi]^{-1})$
L	$(\lambda_A a)[\xi]^{-1}$	$=$	$\lambda_A(\widehat{a}[1.(\xi\circ\uparrow)]^{-1})$
NK $>$	$n[\uparrow^k]^{-1}$	$=$	$n-k$ if $n>k$
NK $\leq$	$n[\uparrow^k]^{-1}$		fails if $n\leq k$
ND $=$	$n[n.\xi]^{-1}$	$=$	1
ND $\neq$	$n[m.\xi]^{-1}$	$=$	$(n[\xi]^{-1})[\uparrow]$ if $n\neq m$
D	$(a.\zeta)\circ\xi^{-1}$	$=$	$a[\xi]^{-1}.(\zeta\circ\xi^{-1})$
UD1	$\uparrow^m\circ(a.\xi)^{-1}$	$=$	$(\uparrow^m\circ\xi^{-1})\circ\uparrow$ if $m\geq a$
UD2	$\uparrow^m\circ(a.\xi)^{-1}$	$=$	$((m+1).\uparrow^{m+1})\circ(a.\xi)^{-1}$ if $m<a$
UU1	$\uparrow^m\circ(\uparrow^n)^{-1}$	$=$	$\uparrow^{m-n}$ if $m\geq n$
UU2	$\uparrow^m\circ(\uparrow^n)^{-1}$	$=$	$(m+1).\uparrow^{m+1}\circ(\uparrow^n)^{-1}$ if $m<n$
V+	$(Y[\zeta])[\xi]^{-1}$	$=$	$Y[\zeta\circ\xi^{-1}]$ if $\zeta\circ\xi^{-1}$ exists
V−	$(Y[\zeta])[\xi]^{-1}$	$=$	$Y'[(\zeta'\circ\zeta)\circ\xi^{-1}]$ if $\zeta\circ\xi^{-1}$ does not exist, adding constraint $Y=^?_{\lambda\sigma} Y'[\zeta']$ where $\zeta'=\zeta\mid\xi$ and Y' is a new meta-variable

Figure 8: Inverse Computations

that no solutions are lost. It is defined by:

$$
\begin{array}{rcl}
(b.\zeta)\mid\xi & = & 1.((\zeta\mid\xi)\circ\uparrow) \quad \text{if } b[\xi]^{-1} \text{ exists}\\
(b.\zeta)\mid\xi & = & (\zeta\mid\xi)\circ\uparrow \quad \text{if } b[\xi]^{-1} \text{ does not exist}\\
\uparrow^m\mid(a.\xi) & = & \uparrow^m\mid\xi \quad \text{if } m\geq a\\
\uparrow^m\mid(a.\xi) & = & (m+1).\uparrow^{m+1}\mid(a.\xi) \quad \text{if } m<a\\
\uparrow^m\mid\uparrow^n & = & (m+1).\uparrow^{m+1}\mid\uparrow^n \quad \text{if } m<n\\
\uparrow^m\mid\uparrow^n & = & id \quad \text{if } m\geq n
\end{array}
$$

4.4 Constraint simplification: the general case

Extensive practical experience with languages like λProlog and Elf has shown that a static restriction of the language to employ only patterns in their terms is too severe. An empirical study confirming this observation can be found in [15]. These experiments suggest an operational model, whereby equations which lie entirely within the pattern fragment of the typed λ-calculus should be solved immediately with the pattern unification algorithm. Others (including certain flex-rigid and flexible-flexible equations) should be postponed as constraints in the hope that they will be instantiated further by solutions to other constraints. Such a general scheme of deduction with constraints has been studied in first order logic with equality [11]

and has shown a fundamental improvement in the theorem proving techniques [13]. In our situation, this technique avoids all branching during constraint simplification (as opposed to Huet's algorithm, for example). The resulting *pattern simplification algorithm* is described in [23] in the framework of the Elf programming language. The implementation [25] is practical, but not nearly as efficient as unification would be for a language such as Prolog. Some of these inefficiencies can be addressed using a calculus of explicit substitutions.

While not all problems have been solved, we believe that such a treatment is an important and essential step forward. The ideas below should also apply to Nadathur's environment calculus [20, 21, 18], which was designed for an efficient implementation of Huet's unification algorithm in the context of λProlog.

If we relax the pattern restriction, atomic weak head-normal forms can now also be of the form $X[s]$, where s is *not* a pattern substitution. Throughout the remainder of this section we use s for substitutions which are not pattern substitutions.

The constraint transformation rules from Section 4.2 do not change, except that now there are additional cases where no rules apply. These remaining constraints have the form $X[s] =^{?}_{\lambda\sigma} a$ where:

- $a = n$,
- $a = a_1\ a_2$,
- $a = Y[t]$, or
- $a = Y[\zeta]$ such that $s \circ \zeta^{-1}$ does not exist or would generate unsolved constraints.

In the rules for inverse application of substitutions in Section 4.3, we have to consider versions of the rules $\mathbf{V}+$, $\mathbf{V}-$, and $\mathbf{D}$ where ζ may not be a pattern substitution, as described in Figure 9.

One could safely omit $\mathbf{V2}+$ and unconditionally apply $\mathbf{V2}\ =$, but in practice this sometimes leaves constraints with an obvious principal solution.

This generalized algorithm still always terminates. If it succeeds without unsolved constraints, the induced answer substitution is guaranteed to be a principal solution. If it fails there is no solution. If some constraints remain there may or may not be a solution, and the algorithm is therefore not as complete as Huet's. However, the remaining constraints and the original constraints have the same set of solutions (restricted to the original variables), and we are thus free to employ other algorithms to determine satisfiability of remaining constraints at the end if we wish. The algorithm is also obviously still sound and complete for unification problems consisting entirely of patterns.

D2	$(a.t) \circ \xi^{-1}$	$=$	$a[\xi]^{-1}.(t \circ \xi^{-1})$	
V2+	$(Y[t])[\xi]^{-1}$	$=$	$Y[t \circ \xi^{-1}]$	if $t \circ \xi^{-1}$ exists without introducing new unsolved constraints
V2 $=$	$(Y[t])[\xi]^{-1}$	$=$	Y'	if $t \circ \xi^{-1}$ does not exist or would add unsolved constraints, adding constraint $Y[t] =^{?}_{\lambda\sigma} Y'[\xi]$ for a new meta-variable Y'.

Figure 9: General inverse computation

In practice it has proved superior to Huet's algorithm in many cases, primarily because the latter makes non-deterministic choices which may have to be undone and backtracked later. Furthermore, we have the guarantee that any produced solution without remaining constraints is indeed a most general solution, which is important, for example, in type reconstruction.

5 Related and future work

Nadathur [20, 21, 18] has independently developed a calculus quite similar to explicit substitutions and sketched an implementation of Huet's algorithm for higher-order unification in it. Another solution has been adopted in Prolog/Mali [4]. As far as we know, nobody has already considered this question for patterns or constraints. Qian [27] gives a linear algorithm for higher-order pattern unification, but to our knowledge it has never been implemented and its practicality remains open. He also does not consider more general constraints.

One question we plan to consider in future work is if an explicit substitution calculus might be exposed at the level of the programming language, rather than remain confined to the implementation. This raises a number of tantalizing possibilities as pointed out by Duggan [8]. Another point of interest will be to investigate equational patterns unification. This has very useful applications but could probably not be consider as a simple specialisation of the general case developped in [12], as shown in [3].

Besides MLF [9] and Twelf [26], the ALF system [14] also employs an explicit substitution calculus with dependent types and makes it available to the user, but its operational properties (such as unification) have not yet been fully investigated.

Acknowledgements. We would like to thank Robert Harper and Carsten Schürmann for many useful and stimulating discussions and Peter Borovansky for his implementation of pattern unification in ELAN. This research has been partly supported by the HCM European network Console and by NSF grant CCR-9303383.

References

[1] Martin Abadi, Lucas Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] Peter Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In Moroslav Bartošek, Jan Staudek, and Jiří Wiedermann, editors, *Proceedings of the SOFSEM'95: Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 363–368. Springer-Verlag, 1995.

[3] Alexandre Boudet and Evalyne Contejean. AC-unification of higher-order patterns. In Gert Smolka, editor, *Third International Conference on Principles and Practice of Constraint Programming*, 1997.

[4] Pascal Brisset. *Compilation de Lambda-Prolog.* Thèse de Doctorat, Université de Rennes I, 1992.

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

[6] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions, extended abstract. In Dexter Kozen, editor, *Proceedings of LICS'95*, pages 366–374, San Diego, June 1995. Rapport de Recherche INRIA 2709, 1995.

[7] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In Michael Maher, editor, *Proceedings of JICSLP'96*, Bonn (Germany), September 1996. The MIT press.

[8] Dominic Duggan. Logical closures. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 114–129, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.

[9] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 8:5–31, 1996.

[10] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis 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.

[11] Claude Kirchner, Hélène Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on Automatic Deduction.

[12] Claude Kirchner and Christophe Ringeissen. Higher-Order Equational Unification via Explicit Substitutions. In *Proceedings 6th International Joint Conference ALP'97-HOA'97, Southampton (UK)*, volume 1298 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 1997.

[13] Christopher Lynch. Paramodulation without duplication. In Dexter Kozen, editor, *Proceedings of LICS'95*, San Diego, June 1995.

[14] Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[15] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[16] Spiro Michaylov and Frank Pfenning. Higher-order logic programming as constraint logic programming. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 221–229, Newport, Rhode Island, April 1993. Brown University.

[17] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[18] Gopalan Nadathur. A notation for lambda terms II: Refinements and applications. Technical Report CS-1994-01, Department of Computer Science, Duke University, January 1994.

[19] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[20] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 341–348, 1990.

[21] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms I: A generalization of environments (revised). Technical Report CS-1994-03, Department of Computer Science, Duke University, January 1994.

[22] Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 673–676, Saratoga Springs, NY, 1992. Springer-Verlag LNAI 607. System abstract.

[23] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[24] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[25] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[26] Frank Pfenning and Carsten Schürmann. Twelf user's guide, 1.2 edition. Technical Report CMU-CS-1998-173, Carnegie Mellon University, September 1998.

[27] Zhenyu Qian. Linear unification of higher-order patterns. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 391–405, Orsay, France, April 1993. Springer-Verlag LNCS 668.

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

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e 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)
http://www.inria.fr
ISSN 0249-6399