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

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

Thérèse Hardin
LITP and INRIA-Rocquencourt
Université Paris 6, 4 Place Jussieu
75252 Paris Cedex 05, France
Therese.Hardin@litp.ibp.fr

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

Frank Pfenning[1]
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
fp@cs.cmu.edu

Abstract

Following the general method and related completeness results on using explicit substitutions to perform higher-order unification proposed in [5], we investigate in this paper 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.

1 Introduction

Typed λ-calculi of various sorts are used pervasively in logical frameworks and their implementations (e.g., λProlog [16], Isabelle [19], or Elf [22]) 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 [5], 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 [14], usually now referred to as *higher-order patterns*. Unification on this fragment is decidable and unitary, even for very rich type theories such as the Calculus of Constructions [21].

In this paper we show that the general algorithm for higher-order unification with explicit substitutions can be cleanly specialized to the case of patterns, bringing a

[1]currently on sabbatical at Technical University Darmstadt, Germany, with support from the Alexander von Humboldt Foundation.

new view on the fundamental reasons why patterns behave so nicely. This behaviour can be explained in algebraic terms as the fact that applying a pattern substitution to a term is injective and thus has an inverse.

An empirical analysis of existing higher-order logic programs [12, 13] 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. This led to the development of a constraint simplification algorithm [20] which solves all equations between patterns and postpones other constraints. This constraint handling can then be used in a deduction-with-constraints process [10]. We show in this paper 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 [8]. It has shown good computational properties, although we have not undertaken a systematic empirical comparison.

The remainder of the paper is organized as follows. The next section describes $\lambda\sigma$, the explicit calculus of substitution that we are using, together with its fundamental properties. In section 3 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 4 are correct and complete and compute a most general unifier, when it exists. Then in Section 5, 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 on lambda calculus, calculus of substitution, unification and term rewriting, we refer to [5]. For lack of space, almost all proofs and several developments are missing from this version of the paper. They can be found in [6].

2 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, 4], 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 $a[s]$ represents the term a on which the substitution s has to be applied. The simplest substitutions are list 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 X,

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

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

$$
\begin{array}{llll}
(var) & A.\Gamma \vdash 1 : A & (lambda) & \dfrac{A.\Gamma \vdash b : B}{\Gamma \vdash \lambda_A b : A \to B} \\[2ex]
(app) & \dfrac{\Gamma \vdash a : A \to B \quad \Gamma \vdash b : A}{\Gamma \vdash (a\ b) : B} & (clos) & \dfrac{\Gamma \vdash s : \Gamma' \quad \Gamma' \vdash a : A}{\Gamma \vdash a[s] : A} \\[2ex]
(id) & \Gamma \vdash id : \Gamma & (shift) & A.\Gamma \vdash\ \uparrow : \Gamma \\[2ex]
(cons) & \dfrac{\Gamma \vdash a : A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash a : A \cdot s\ :\ A.\Gamma'} & (comp) & \dfrac{\Gamma \vdash s'' : \Gamma'' \quad \Gamma'' \vdash s' : \Gamma'}{\Gamma \vdash s' \circ s''\ :\ \Gamma'} \\[2ex]
 & & (metavar) & \Gamma_X \vdash X : T_X
\end{array}
$$

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

Figure 1 gives the syntax of $\lambda\sigma$-terms. Notice that we do not have substitution variables in the calculus we consider here.

The typing judgment $\Gamma \vdash a : A$ defines when a term a has type A in 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*). 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 rule **Beta** is called σ.

It is shown in [24] 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.

The normal form of a $\lambda\sigma$-term has one of the following forms: λa, $(\mathtt{n}\ a_1\ \dots\ a_p)$, $(X[a_1 \dots a_p.\ \uparrow^n]\ b_1\ \dots\ b_q)$, $(X\ b_1\ \dots\ b_q)$. By convention, we consider that the later form is a particular case of the third with $p = n = 0$.

3 $\lambda\sigma$-patterns

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 is only applied to a sequence of distinct bound variables, i.e. $(X\ x_1 \dots x_p)$.

Beta	$(\lambda_A.a)b$	$\rightarrow$	$a[b.id]$
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
Eta	$\lambda_A.(a\ \mathtt{1})$	$\rightarrow$	b if $a =_\sigma b[\uparrow]$

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

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

Equations in the simply-typed λ-calculus are translated to equations in the $\lambda\sigma$-calculus [5]. Crucial in the translation is the proper treatment of free variables, since higher-order variables and capture-avoiding substitution are replaced by meta-variables and grafting, i.e., first-order replacement. This gap is bridged through the so-called *pre-cooking* of a term which raises the free variables to their proper 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 [5], pre-cooking is an homomorphism from λ-calculus to $\lambda\sigma$-calculus.

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

Proposition 3 A λ-term a is a λ-pattern if and only if its pre-cooked form a_F is a $\lambda\sigma$-pattern.

Proposition 4 Let s and t be pattern substitutions and a and b be atomic patterns. Then

- the σ-normal form of $1.(s \circ \uparrow)$ and $s \circ t$ are pattern substitutions,
- the normal form of $a[s]$ is an atomic pattern,
- $(X \mapsto b)a$ is an atomic pattern.

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

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 6 Any pattern substitution is injective.

Pattern substitutions are injective, but they need not be surjective. We now 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 7 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 occuring in a occur in s.

Proposition 8 (Inversion) 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. Then a' is an atomic pattern (resp. t' is a pattern substitution) and $a' = a[\overline{s}]$ (resp. $t' = t \circ \overline{s}$).

For instance, the term (2 3) is in the image of $\uparrow$ because 1 does not occur in it, but (1 3) is not in the image of $\uparrow$ because 1 occurs in it.

4 Pattern unification

4.1 Transformation rules

Definition 9 The language of (unification) *constraints* is defined by [9]:

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

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 context Γ, and $X \in \mathcal{X}$. The set of term variables of a constraint (also called a system) P or of a term a is denoted $\mathcal{V}ar(P)$ (resp. $\mathcal{V}ar(a)$).

We take propositional constraint simplification (such as $\mathcal{C} \wedge \mathbb{T} \mapsto\!\!\!\rightarrow \mathcal{C}$) 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 10 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 terms of all the equations of P equal modulo $\lambda\sigma$. 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 11 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 flex-flex equations (see [5]) 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.

Definition 12 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 13 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.

Proposition 14 If k has a flexible occurrence in a then a has a subterm of the form $X[a_1\ \ldots\ a_p.\uparrow^n]$ at depth l and some a_i is equal to $k+l$. We say that k has a flexible occurence 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 **Exp-app** rule of [5]. This rule, that implements for $\lambda\sigma$-unification the imitation and projection steps of Huet's algorithm, is mainly replaced by the rules **Pruning1**, **Pruning2** and **Invert**. This is due to the fact that when starting from a $\lambda\sigma$-pattern unification problem, if we reach an equation of the form: $X[s] =^?_{\lambda\sigma} a$, the substitution s is a pattern substitution and is thus right invertible. This allows us to bypass the **Exp-app** rule and also to solve all flex-flex equations except the ones with the same head variable, for which the **Same-variable** rule is applied.

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.

Let us show how these rules act on 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 indices this equation is $\lambda\lambda\lambda(F\ 1\ 2) =^?_{\lambda\sigma} \lambda\lambda\lambda(1\ (G\ 2\ 3))$, after pre-cooking, becomes

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

Using **Dec-$\boldsymbol{\lambda}$** this equation simplifies to $(F[\uparrow^3]\ 1\ 2) =^?_{\lambda\sigma} (1\ (G[\uparrow^3]\ 2\ 3))$. Using **Exp-$\boldsymbol{\lambda}$** 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 number occurs in the right member 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 no longer occurs on the right-hand side. Thus **Invert** is applicable and yields $X =^?_{\lambda\sigma} (1\ (Z[2.\uparrow^3]))[2.1.\uparrow]$ that is $X =^?_{\lambda\sigma} (2\ (Z[1.\uparrow^2]))$. This gives $F =^?_{\lambda\sigma} \lambda\lambda(2\ (Z[1.\uparrow^2]))$, $G =^?_{\lambda\sigma} \lambda\lambda(Z[\uparrow])$. With **Anti-Exp-$\boldsymbol{\lambda}$** we get $Z =^?_{\lambda\sigma} (H[\uparrow]\ 1)$ and:

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

which is the pre-cooking of $F =^?_{\lambda\sigma} \lambda\lambda(2\ (H\ 1))$, $G =^?_{\lambda\sigma} \lambda\lambda(H\ 2)$ i.e. $F =^?_{\beta\eta} \lambda u\ \lambda v\ (u\ (H\ v))$, $G =^?_{\beta\eta} \lambda u\ \lambda v\ (H\ u)$.

Proposition 15 Any rule $\mathbf{r}$ in **PatternUnif** is correct (i.e. $P \mapsto\!\!\!\!\to^{\mathbf{r}} P' \Rightarrow \mathcal{U}_{\lambda\sigma}(P') \subseteq \mathcal{U}_{\lambda\sigma}(P)$) and complete (i.e. $P \mapsto\!\!\!\!\to^{\mathbf{r}} P' \Rightarrow \mathcal{U}_{\lambda\sigma}(P) \subseteq \mathcal{U}_{\lambda\sigma}(P')$).

4.2 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

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

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

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

Exp-$\boldsymbol{\lambda}$ P
$\mapsto\!\!\!\twoheadrightarrow$
$\exists Y : (A \cdot \Gamma \vdash B),\ \ P \wedge X =^?_{\lambda\sigma} \lambda_A Y$
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 $X \in \mathcal{V}ar(a)$

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

Anti-Exp-$\boldsymbol{\lambda}$ P
$\mapsto\!\!\!\twoheadrightarrow$
$\exists Y\ (P \wedge X =^?_{\lambda\sigma} (Y[\uparrow]\ 1))$
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-$\boldsymbol{\lambda}$ $P \wedge a =^?_{\lambda\sigma} b$
$\mapsto\!\!\!\twoheadrightarrow$
$P \wedge \lambda_A a =^?_{\lambda\sigma} \lambda_A b$
if $a =^?_{\lambda\sigma} b$ is well-typed in a context $\Delta = A.\Delta'$

Figure 4: **PatternUnif**: Rules for pattern unification in $\lambda\sigma$: Part 1

Pruning1 $P \wedge X[s] =^?_{\lambda\sigma} b$

$\mapsto\!\!\!\!\twoheadrightarrow$

$\mathbb{F}$

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 $P \wedge X[s] =^?_{\lambda\sigma} b$

$\mapsto\!\!\!\!\twoheadrightarrow$

$\exists Z(P \wedge X[s] =^?_{\lambda\sigma} b \wedge Y =^?_{\lambda\sigma} Z[1 \dots i-1. \uparrow^i])$

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 $P \wedge X[s] =^?_{\lambda\sigma} b$

$\mapsto\!\!\!\!\twoheadrightarrow$

$P \wedge X =^?_{\lambda\sigma} b[\overline{s}]$

if b is an atomic pattern, s is a pattern substitution and X does not occur in b, all the de Bruijn indices occuring in b occur in s

Same-variable $P \wedge X =^?_{\lambda\sigma} X[a_1 \dots a_n. \uparrow^n]$

$\mapsto\!\!\!\!\twoheadrightarrow$

$\exists Y(P \wedge X =^?_{\lambda\sigma} Y[i_1 \dots i_r. \uparrow^n])$

where $a_1 \dots a_n. \uparrow^n$ is a pattern substitution and $i_1 \dots i_r$ are the indices such that $a_i = i$

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

first check that atomic pattern unification problems are stable under the unification transformations, i.e. applying a unification rule of **PatternUnif** 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**, if it is either $\mathbb{F}$ or a in solved form.

After any application of **Pruning2** or **Same-variable** the rule **Replace** is applicable, after any application of **Invert** one of the rules **Occur-check**, **Same-variable** or **Replace** is applicable. Thus on an unsolved form one of these sequences can be applied: (**Dec-***), (**Pruning1**), (**Pruning2**; **Replace**; **Normalize**), (**Invert**; **Normalize**; **Occur-check**), (**Invert**; **Normalize**; **Same-variable**; **Replace**; **Normalize**), (**Invert**; **Normalize**; **Replace**; **Normalize**), (**Same-variable**; **Replace**; **Normalize**), (**Replace**; **Normalize**). 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. Thus we get:

Proposition 16 The above strategy in the application of the **PatternUnif** rules is terminating on atomic pattern problems. Thus unification of atomic $\lambda\sigma$-patterns is decidable and unitary.

For non-atomic pattern problems the application of the rules (**Exp-$\boldsymbol{\lambda}$**; **Replace**; **Normalize**) is trivially terminating and yields an atomic pattern problem. Thus:

Theorem 17 *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-*** of [5]. As shown in the full paper, starting from a pattern unification problem in the simply typed λ-calculus, pre-cooking followed by the application of the **PatternUnif** rules under the strategy we have described and ended by the **Anti-*** rules, provides a correct and complete way of solving pattern unification.

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 [5]. Similarly, the raising step is merely our **Anti-Exp-$\boldsymbol{\lambda}$** 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**).

5 Towards an efficient implementation

The literal interpretation of the algorithm in Section 4.2 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 [8].

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.

5.1 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 18 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 19 If every meta-variable in a has atomic type, then a has an atomic weak head-normal form.

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-$\boldsymbol{\lambda}$** 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 reduction as modelled in the $\lambda\sigma$-calculus. As noted, this operation may add equations to the constraints store.

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

Figure 6: Practical transformations

5.2 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 equations are immediately applied in that X is replaced by b throughout the other constraints, i.e., **Replace** is applied eagerly. In the implementation, this is modelled by an efficient destructive update of a pointer associated with X in the manner familiar from Prolog implementations.

The **PatternUnif** rules can be implemented by the transformation rules described in Figure 6 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,

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

Figure 7: Inverse Computations

we define $\xi\cap\zeta$ by:

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

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.

5.3 Inverting substitutions

The properties from Section 3 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:

$$
\begin{array}{c}
\Gamma\vdash b:A \\
\Gamma\vdash\xi\rhd\Delta
\end{array}
$$

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:

$$
\begin{array}{c}
\Gamma\vdash\zeta\rhd\Gamma' \\
\Gamma\vdash\xi\rhd\Delta
\end{array}
$$

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

The application of an inverse substitution is described in Figure 7. 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 that no solutions are lost. It is defined by:

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

5.4 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 [12]. 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 flex-flex 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 [10] and has shown a fundamental improvement in the theorem proving techniques. 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 [20] in the framework of the Elf programming language. The implementation [22] 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.

We modify the previous algorithm by computing inverses of general terms $(Y[t])[\xi]^{-1}$ (where t is not a pattern substitution) if they can be computed without introducing any additional constraints. Otherwise appropriate equations are added to a global constraints store. Constraints are awakened and reconsidered when the meta-variable at the head of a term is instantiated.

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. A more detailed description can be found in [6].

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.

6 Related and future work

Perhaps most closely related is work by Nadathur [17, 18, 15] who 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 [3]. As far as we know, nobody has considered this question for patterns or constraints. Qian [23] 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 interesting 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 [7].

Besides MLF [8], the ALF system [11] 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 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] P. 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] P. Brisset. *Compilation de LambdaProlog*. PhD thesis, Rennes I, March 1992.

[4] 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.

[5] 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.

[6] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. Rapport technique, INRIA, Rocquencourt, France, 1996. Forthcoming.

[7] 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.

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

[9] J.-P. 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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