Higher-Order Unification: A structural relation between Huet's method and the one based on explicit substitutions [⋆]

Flávio L. C. de Moura [a,∗], Mauricio Ayala-Rincón [b,1], Fairouz Kamareddine [c]

[a] *Departamento de Ciência da Computação, Universidade de Brasília, Brasília D.F., Brasil*

[b] *Departamento de Matemática, Universidade de Brasília, Brasília D.F., Brasil*

[c] *School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland*

Abstract

We compare two different styles of Higher-Order Unification (HOU): the classical HOU algorithm of Huet for the simply typed λ-calculus and HOU based on the $\lambda\sigma$-calculus of explicit substitutions. For doing so, first, the original Huet algorithm for the simply typed λ-calculus with names is adapted to the language of the λ-calculus in de Bruijn's notation, since this is the notation used by the $\lambda\sigma$-calculus. Afterwards, we introduce a new structural notation called *unification tree*, which eases the presentation of the subgoals generated by Huet's algorithm and its behaviour. The unification tree notation will be important for the comparison between Huet's algorithm and unification in the $\lambda\sigma$-calculus whose derivations are presented into a structure called *derivation tree*. We prove that there exists an important structural correspondence between Huet's HOU and the $\lambda\sigma$-HOU method: for each (sub-)problem in the unification tree there exists a counterpart in the derivation tree. This allows us to conclude that the $\lambda\sigma$-HOU is a generalization of Huet's algorithm and that solutions computed by the latter are always computed by the former method.

Key words: Higher-Order Unification, Calculi of Explicit Substitutions.

⋆ Work supported by CNPq grant 471791/2004-0

∗ Corresponding author. Supported by Brazilian CAPES Foundation.
Email addresses: `flavio@cic.unb.br` (Flávio L. C. de Moura), `ayala@unb.br` (Mauricio Ayala-Rincón), `fairouz@macs.hw.ac.uk` (Fairouz Kamareddine).

1 Partially supported by Brazilian CNPq Council.

1 Introduction

More than thirty years ago, G. Huet (Hue75; Hue02) gave the most successful and largely used Higher-Order Unification (HOU) algorithm. HOU is undecidable (Gol81), Huet's algorithm is in fact a semi-decision procedure because it always finds the solutions to unifiable problems but may loop if the problem does not have a solution. The kernel of Huet's algorithm consists of two procedures called SIMPL and MATCH used for dealing with the so called rigid-rigid, and flexible-rigid equations, respectively, and its practical success is based on the observation that flexible-flexible equations always have solutions and consequently (for deciding whether a problem is or is not unifiable) it is not necessary to explicitly present all possible unifiers. Huet's algorithm behaves well in practice, and as a consequence, many of the modern computational systems which use HOU, such as λProlog and Isabelle/HOL, are based on this algorithm. In addition, this algorithm has been extended to several higher-order equational theories (Dow01) and specialised to treat reducts of practical interest such as the case of higher-order patterns (Nip91).

The most promising alternative for treating HOU problems is based on explicit substitutions calculi and was developed over the $\lambda\sigma$-calculus almost ten years ago (DHK00). This alternative method has been shown to be of general applicability for other calculi of explicit substitutions like the λs_e-calculus (ARK01). Calculi of explicit substitutions are essentially formal mechanisms attempting to solve an important drawback of the λ-calculus: the implicitness of substitution, that is the basic operation on which the computational functionality of the λ-calculus is founded. The formal basis of some programming languages is founded on explicit substitutions; for instance, λProlog is founded on the suspension calculus of explicit substitutions. As a matter of fact, real programming languages are based on some *ad-hoc* (and mostly obscure) explicit implementation of the substitution operation. When substitution is made explicit, it allows one to include HOU mechanisms at a lower level; that is, directly over the associated language of explicit substitutions instead of implementing HOU mechanisms, as usual, as strategies in a higher level of the implementation based on Huet's algorithm. The importance of a precise knowledge of the style of explicit substitutions used in the implementation of programming languages has been made evident recently in (LNQ04). In that work, the efficiency of different implementations of λProlog over the system Teyjus was tested for several programs. Simple changes in the way explicit substitutions are treated over the suspension calculus were shown to imply great changes in the performance of the language.

Essentially, HOU via calculi of explicit substitutions consists of, firstly, translating HOU problems presented in the language of the simply typed λ-calculus (in de Bruijn's notation) to the language of the explicit substitutions calculus;

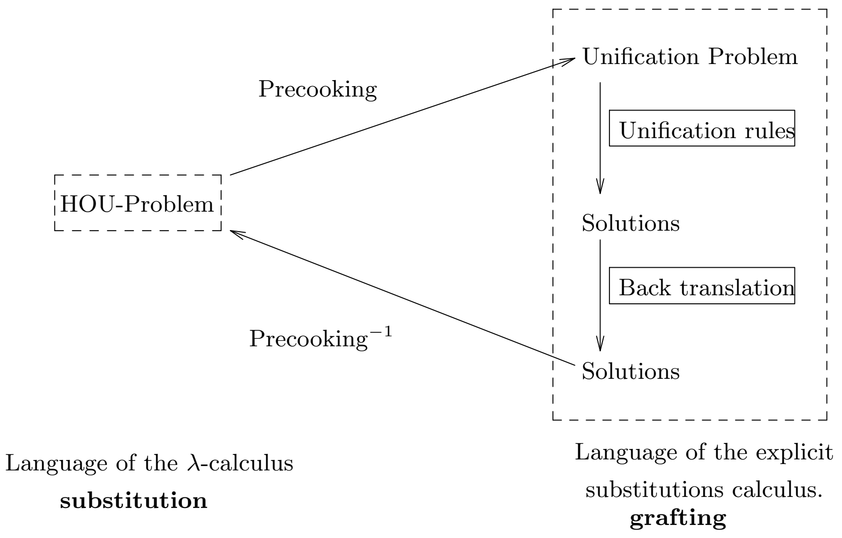


Figure 1. HOU via calculi of Explicit substitutions.

this process is known as a *precooking translation.* Afterwards, precooked problems are resolved as first-order unification problems modulo the equational theory which defines the calculus of explicit substitutions and, finally, the solutions are translated back to the language of the original problem (see Fig. 1, that has been taken from (ARK03)). Therefore, the main advantage of the use of explicit substitutions to solve HOU problems is that the substitution operation becomes a first order substitution (called *grafting*) and the higher-order substitutions which are solutions of the original problem can be obtained by applying the inverse of the precooking translation to the generated graftings, i.e., to the solutions of the precooked version of the original problem.

In (DHK00), it has been noted that the $\lambda\sigma$-HOU algorithm is a generalisation of Huet's method. In this paper, we refine this result by establishing a structural relation between sub-problems in the λ-calculus and in the $\lambda\sigma$-calculus in the following way: we introduce a new notation called *unification tree* which clarifies the presentation of Huet's algorithm and eases the presentation of subgoals generated by Huet's algorithm because each step of the algorithm is represented by an arc in the tree. This notation is independent of the grammar used and can be used for both λ-calculi with names or in de Bruijn's notation. In a similar way, applications of the unification procedure in the simply typed $\lambda\sigma$-calculus are represented as trees, called *derivation trees.*

We prove that for a given unification problem P in the simply typed λ-calculus in de Bruijn's notation, each subgoal (or derived problem) generated in the unification tree of P has a counterpart in the derivation tree of its precooking translation P_F in the $\lambda\sigma$-calculus, i.e., there exists a structural relation between the unification tree of P and the derivation tree of P_F. From this, we establish a relation between the solutions of derived problems of P and derived problems of P_F.

In section 2 we present the simply typed λ-calculus with names, Huet's algorithm and the unification tree notation. In section 3 we define the simply typed λ-calculus and we introduce Huet's algorithm in de Bruijn's notation. A detailed description of Huet's algorithm is given and the relevant aspects that differ from the presentation with names are emphasised with examples. In section 4 we briefly present the $\lambda\sigma$-HOU method and we formalise the relation between unification in the simply typed λ-calculus and in the simply typed $\lambda\sigma$-calculus by relating unification trees and derivation trees. Some of the presented examples were generated with the system SUBSEXPL (MAK06). Finally, in the last section, we conclude and give directions for future work.

2 Background

Since the λ-calculus with names is clearer and easier for humans than the λ-calculus in de Bruijn's notation, we start the next subsection with a general presentation of the simply typed λ-calculus with names and of Huet's algorithm. For this presentation we use standard notations and suppose familiarity with basic notions on rewriting theory (BN98), type theory (Hin97) and λ-calculus (Bar84).

2.1 Simply typed λ-calculus with Names

We assume two infinite denumerable sets $\mathcal{V}$ (of variables) and $\mathcal{X}$ (of meta-variables). We let $x, y, z, \ldots$ range over $\mathcal{V}$ and $X, Y, Z, \ldots$ range over $\mathcal{X}$. The λ-terms (firstly without types) are built inductively defined by:

$$a \ ::= \ x \mid X \mid a\ a \mid \lambda_x.a$$

We use $a, b, c, d, e, u, \ldots$ to range over λ-terms.

Remark 1 *Parenthesis are used to avoid ambiguities and we assume that applications are left associative; i.e.,* $(a_1\ a_2 \ldots a_n)$ *means* $((\ldots(a_1\ a_2)\ldots)\ a_n)$ *and, abstractions are right associative, i.e.,* $\lambda_{x_1}\lambda_{x_2}\ldots\lambda_{x_n}.u$ *is interpreted as* $\lambda_{x_1}.(\lambda_{x_2}.(\ldots(\lambda_{x_n}.u)\ldots))$*. Moreover, an application has higher priority than an abstraction. In this way,* $\lambda_x.a\ b$ *means* $\lambda_x.(a\ b)$*.*

Remark 2 *The separation of constants and bound variables on one side and meta-variables (also known as unification variables) on the other side is important to distinguish between the substitutions generated by β-reductions and those generated by the unification procedure. In fact, bound variables and constants are not concerned with the unification process and the meta-variables will play the role of the unification variables.*

In the λ-calculus with names, terms are interpreted modulo α-conversion, which means that the names of bound variables used in abstractions are irrelevant. For example, $\lambda_x.x\ z$ and $\lambda_y.y\ z$ represent the same λ-term. Free and bound occurrences are defined as usual; for instance, in the term $(\lambda_y.y\ z)\ y$, z and the second occurrence of y are free while the first occurrence of y is bound.

The basic operations of the λ-calculus are β-reduction and η-reduction[2]. The former implements the applications of functional terms over arguments and the latter represents functional equivalence. These operations are "implicitly" defined by:

$$(\lambda_x.a)\ b \rightarrow a\{x/b\} \qquad (\beta)$$

$$\lambda_x.a\ x \rightarrow a, \text{ if } x \text{ does not occur free in } a. \qquad (\eta)$$

In (β), "$a\{x/b\}$" represents the term obtained from a by substituting all its free occurrences of x by b. Implicitness of the definition of β-reduction is a consequence of this pseudo-definition of substitution. And this is the main drawback of the λ-calculus, when it has to be used for concrete implementations. In fact, for implementing the λ-calculus one has to decide how to implement substitutions and this is done usually by *ad-hoc* mechanisms, which are adjusted during the implementation process. Calculi of explicit substitutions attack this problem by formalizing, in different styles, the notion of substitution, which make these formalisms close to concrete implementations.

η-reduction stands for functional equivalence and this can be understood by noticing that, whenever it applies, for any term b, it holds that $a\ b$ and $(\lambda_x.a\ x)\ b$ coincide since $(\lambda_x.a\ x)\ b \rightarrow_\beta (a\ x)\{x/b\} = a\ b$.

Notations used for rewriting concepts of the λ-calculus with names as well as for any other rewriting system in this work are the standard ones from rewriting theory (see (BN98; Hin97)). Let a be a λ-term, a *β-redex* in a is a sub-term of a which is an instance of the left hand side of the β-reduction rule. The right hand side of an instance of the β-reduction rule is called a *β-contractum*. Supposing the term obtained by replacing in a the β-redex by its contractum is the term b, we write $a \rightarrow_\beta b$. A term without β-redexes is said to be in *β-normal form* or β-nf for short. The inverse of the binary relation $\rightarrow_\beta$ is denoted by ${}_\beta\!\leftarrow$ and its reflexive transitive closure by $\rightarrow^*_\beta$. The symmetric closure of $\rightarrow_\beta$ which is the relation $\rightarrow_\beta \cup {}_\beta\!\leftarrow$ is denoted by $\leftrightarrow_\beta$ and its reflexive transitive closure, called *β-conversion*, by $=_\beta$. A β-nf of a term a is a term b such that b is a β-nf and $a \rightarrow^*_\beta b$. Similarly, we define η-redexes, η-contractum, the notations $\rightarrow_\eta$, η-conversion, η-nf, etc. Also for the relation $\rightarrow_\beta \cup \rightarrow_\eta$, denoted as $\rightarrow_{\beta\eta}$, the same notations are used.

[2] We will use the word "reduction" for both the β and η rewriting rules (usually called β- and η-contraction) and the rewriting relation generated from these rules.

It is well known that the adequate environment for higher-order unification is the simply typed λ-calculus. In the following we present the simply typed version of the λ-calculus with names. We assume that there exists an infinite set $\mathbb{T}$ of type variables (*atomic types*). Types are inductively defined by:

$$A \ ::= \ K \mid A \to A$$

where K ranges over the set $\mathbb{T}$. We say that A is the *target type* of the type $A_1 \to \ldots \to A_n \to A$, where $n \geq 0$. We follow the Church approach for typing terms. In this approach, differently to the approach of Curry (also known as type assignment theory), typed λ-terms are inductively defined by:

$$a \ ::= \ x \mid X \mid a\ a \mid \lambda_{x:A}.a$$

A type assignment is an expression of the form $a : A$, where a is a λ-term and A is a type. Type contexts, or just contexts, are used to store the type information of the constants occurring in a term and are defined as finite sets of type assignments. We use $\Gamma, \Delta, \ldots$ to denote contexts. A context Γ is said to be *consistent* if each variable in Γ has no more than one assignment. We assume contexts to be consistent and use the following typing rules:

(var) $\dfrac{}{\Gamma \cup \{x : A\} \vdash x : A}$ if $\Gamma \cup \{x : A\}$ is consistent.

(meta) $\dfrac{}{\Gamma \vdash X : A}$ where Γ is any context.

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

(lambda) $\dfrac{(\Gamma - x) \cup \{x : A\} \vdash a : B}{\Gamma \vdash (\lambda_{x:A}.a) : A \to B}$ if Γ is consistent with $x : A$

The *type judgement* $\Gamma \vdash a : A$ is said to be *derivable* if it can be deduced from the above typing rules. In the rule (lambda), the notation $\Gamma - x$ means that the assignment to x in Γ (if it exists) is removed and, the condition "Γ is consistent with $x : A$" means that either Γ contains $x : A$ or Γ contains no assignment to x at all. In the former case, we say that x is discharged from Γ and in the latter case that x is discharged vacuously from Γ. Note that this typing system allows weakening.

The rule (meta) implies that the type of a meta-variable is independent from the context, which is necessary for placing repetitions of meta-variables at different levels of abstraction in λ-terms. For instance, consider the type judgement $\vdash \lambda_{z:A\to(A\to A)\to A}.z\ Y\ \lambda_{x:A}.Y : (A \to (A \to A) \to A) \to A$. This judgement is derivable from the above typing rules after applying the (meta) rule twice for obtaining the judgements $\vdash Y : A$ and $x : A \vdash Y : A$.

A λ-term a is called *well typed* if and only if there exists a context Γ and a type A, such that $\Gamma \vdash a : A$ is derivable. It is well known that the λ-calculus restricted to well typed terms is closed under sub-terms and $\beta\eta$-reduction. Moreover, it is strongly terminating, which means that every $\beta\eta$-reduction starting from a well typed λ-term is finite.

2.2 Huet's Algorithm

In the next paragraphs we give a general overview of Huet's algorithm (Hue75). Roughly speaking, Huet's algorithm is a semi-decision procedure for unification in the simply typed λ-calculus. It is a *semi-decision* algorithm because it always finds solutions to unifiable unification problems but may loop if the unification problem has no solution. We start with some relevant definitions.

Definition 3 (Structure of nfs) *If a is well typed and in β-nf, then it has the form:*

$$\lambda_{x_1:A_1} \ldots \lambda_{x_n:A_n}.h\ e_1 \ldots e_p$$

where $n, p \geq 0$, h is a constant, a bound variable or a meta-variable, called the head *of a, and $e_1, \ldots, e_p$ are λ-terms in β-nf, called the* arguments *of h. We call $\lambda_{x_1:A_1}, \ldots, \lambda_{x_n:A_n}$ the* external abstractors *of a and $\lambda_{x_1:A_1} \ldots \lambda_{x_n:A_n}.h$ its* heading*.*

Definition 4 *A λ-term in β-nf is* rigid *if its head is a constant or a bound variable. Otherwise, the term is* flexible*, i.e., if its head is a meta-variable.*

Definition 5 (η-long nf (Hin97)) *A well typed λ-term a in β-nf is in η-long normal form, written η-lnf, if every variable occurrence in a is followed by the longest sequence of arguments allowed by its type; i.e., if each component of the form $(u\ e_1 \ldots e_p)$ with $p \geq 0$ that is not in function position has an atomic type.*

From now on, we write "λ-terms" to mean "well typed λ-terms" and, we assume that terms are always in η-lnf.

Definition 6 (Unification Problem) *A* unification problem *P in the simply typed λ-calculus is a conjunction of equations of the form $a =^? b$, where a and b are two λ-terms of the same type, all terms of the problem in the same context, say Γ. In this case, we say that P is well typed in context Γ. The equation $a =^? b$ is called* rigid-rigid *(resp.* flexible-flexible*) if both a and b are rigid (resp. flexible) terms, and* flexible-rigid *if a is flexible and b is rigid or vice-versa. An equation of the form $a =^? a$ is called* trivial*.*

The requirement for a general unique context in unification problems arises from the necessity to give the same assignments for constant names occurring in different equations of the unification problem, as seen by the following example.

Example 7 *Let* $\Gamma = \{x : A, f : A \to A\}$*. Consider the unification problem:*

$$X(f\ x) =^? f\ x \wedge X(f\ x) =^? f(X\ x).$$

Notice that the type judgements $\Gamma \vdash x : A$*,* $\Gamma \vdash f : A \to A$ *and* $\Gamma \vdash X : A \to A$ *are derivable. Consequently, all terms involved in the equations of this problem have type* A *in context* Γ*.*

HOU is undecidable (Gol81), nevertheless Huet (Hue75) developed a semi-decision algorithm that finds a solution if it exists and may loop if it does not. This semi-decision algorithm, known as Huet's algorithm, is based on two procedures called SIMPL and MATCH. The procedure SIMPL is used for simplifying rigid-rigid equations while the procedure MATCH incrementally generates substitutions for flexible-rigid equations that will compose the solutions of the original problem. Flexible-flexible equations always have solutions and, by this reason Huet's algorithm does not need to deal with them. This is why Huet's algorithm is also known as a pre-unification algorithm. In the following we give some intuition on how it works.

Let Γ be a context and P a unification problem well typed in context Γ. The first step of Huet's algorithm is a simplification step, i.e., an application of SIMPL that consists in "breaking" rigid-rigid equations (that have the same heads) into "smaller equations" that need to be solved. For instance, suppose that P is a unification problem containing the following rigid-rigid equation well typed in context Γ:

$$\lambda_{x_1:A_1} \ldots \lambda_{x_n:A_n}.h_1\ e^1_1 \ldots e^1_p =^? \lambda_{y_1:A_1} \ldots \lambda_{y_n:A_n}.h_2\ e^2_1 \ldots e^2_p \qquad (1)$$

where $n, p \geq 0$, $e^1_1 \ldots e^1_p, e^2_1 \ldots e^2_p$ are terms in η-lnf, and h_1 and h_2 represent either the same constant (in which case $h_1 = h_2$) or the same bound variable, i.e., $h_1 = x_i$ and $h_2 = y_i$, for some $1 \leq i \leq n$ (see Example 8). Notice that the number of external abstractors in (1) must be the same because by definition, the terms in the left and right hand side of the equation have the same type and are in η-lnf. An application of SIMPL to P will replace the equation (1) by the following conjunction of equations:

$$\begin{array}{c} \lambda_{x_1:A_1} \ldots \lambda_{x_n:A_n}.e^1_1 =^? \lambda_{y_1:A_1} \ldots \lambda_{y_n:A_n}.e^2_1 \\ \wedge \ldots \wedge \\ \lambda_{x_1:A_1} \ldots \lambda_{x_n:A_n}.e^1_p =^? \lambda_{y_1:A_1} \ldots \lambda_{y_n:A_n}.e^2_p \end{array} \qquad (2)$$

which are well typed in context Γ.

The application of SIMPL to rigid-rigid equations with different heads returns a failure status because the current problem is not unifiable. This simplification step is repeated for all rigid-rigid equations of the current unification problem and, as a consequence, a simplified problem contains only flexible-rigid and/or flexible-flexible equations. Trivial equations are automatically eliminated during the whole process.

Example 8 *Let $\Gamma = \{w : A, u : A \to B, v : A \to A\}$ be a context, X a meta-variable of type $A \to B$ and consider the unification problem composed by the sole rigid-rigid equation $\lambda_{y:B\to B}.y\ (X\ w) =^{?} \lambda_{x:B\to B}.x\ (u\ (v\ w))$ which is well typed in context Γ. An application of SIMPL to this problem generates the following simplified unification problem $\lambda_{y:B\to B}.X\ w =^{?} \lambda_{x:B\to B}.u\ (v\ w)$ which is well typed in context Γ.*

For each simplified unification problem containing at least one flexible-rigid equation, Huet's algorithm calls the procedure MATCH that receives as input a flexible-rigid equation and returns a finite set Σ of substitutions for the head of the flexible term of the given equation. The substitutions generated by MATCH are based on two rules called *imitation* and *projection*. To explain how these rules work, let Γ be a context, and consider the following flexible-rigid equation:

$$\lambda_{x_1:A_1} \ldots \lambda_{x_n:A_n}.X\ e^1_1 \ldots e^1_{p_1} =^{?} \lambda_{y_1:A_1} \ldots \lambda_{y_n:A_n}.h\ e^2_1 \ldots e^2_{p_2} \qquad (3)$$

well typed in context Γ, where:
- $n, p_1, p_2 \geq 0$;
- X is a meta-variable of type $B_1 \to \ldots \to B_{p_1} \to A$ (A atomic);
- h is either a bound variable or a constant of type $C_1 \to \ldots \to C_{p_2} \to A$ (A atomic);
- if $p_1 \neq 0$ then e^1_i is a λ-term in η-lnf of type B_i for all $1 \leq i \leq p_1$;
- if $p_2 \neq 0$ then e^2_j is a λ-term in η-lnf of type C_j for all $1 \leq j \leq p_2$.

2.2.1 Imitation Rule.

The imitation rule generates a substitution that replaces X, the head of the flexible term, by another term whose head corresponds to the head of the rigid term of the current equation, i.e., by a term with head h (consider the equation (3)). In this sense it tries to imitate the term on the right hand side of the equation. Imitation is possible only if the head of the rigid term of the considered equation is a constant due to the fact that variable capture is forbidden in the λ-calculus. Then, if h is a constant, the imitation substitution generated is given by:

$$X/\lambda_{z_1:B_1} \ldots \lambda_{z_{p_1}:B_{p_1}}.h\ (H_1\ z_1 \ldots z_{p_1}) \ldots (H_{p_2}\ z_1 \ldots z_{p_1}) \qquad (4)$$

where, if $p_2 > 0$ then H_i is a fresh meta-variable of type $B_1 \to \ldots \to B_{p_1} \to C_i$ for each $1 \leq i \leq p_2$. Of course, if h has an atomic type, i.e., if $p_2 = 0$ then no meta-variable is introduced by the previous substitution and, the imitation substitution is given by $X/\lambda_{z_1:B_1} \dots \lambda_{z_{p_1}:B_{p_1}}.h$.

Example 9 *Consider the flexible-rigid equation generated in Example 8. An imitation substitution is possible because the head u of the rigid term is a constant. This imitation substitution is given by $X/\lambda_{z:A}.u\ (H_1\ z)$ where H_1 is a fresh meta-variable of type $A \to A$. Note that the above substitution is not a solution of the original problem but, it is part of a possible solution. In fact, substitutions generated by Huet's algorithm are incrementally generated in the sense that each application of MATCH determines part of the solution. At the end of the unification process, the composition of all the substitutions along a success branch will contain a solution of the original problem (see Fig. 2).*

2.2.2 Projection Rule.

A projection is a substitution generated when the head h of the rigid term is either a bound variable or a constant. A projection means that the head X of the flexible term is "projected" over its arguments. Considering equation (3), X can be projected over the arguments that have the same target type as X. Since X has of type $B_1 \to \dots \to B_{p_1} \to A$, suppose that e_i^1 has type $B_i = D_1 \to \dots \to D_q \to A$ for some $i = 1, \dots, p_1$ and where $q \geq 0$. In this case, the projection substitution is given by $X/\lambda_{z_1:B_1} \dots \lambda_{z_{p_1}:B_{p_1}}.z_i\ (H_1\ z_1 \dots z_{p_1}) \dots (H_q\ z_1 \dots z_{p_1})$ where, if $q > 0$ then H_j is a fresh meta-variables of type $B_1 \to \dots \to B_{p_1} \to C_j$ for all $1 \leq j \leq q$.

As a last remark, notice that there exists at most one possible imitation and p_1 possible projections for a given flexible-rigid equation. In case no substitution is generated, i.e., if Σ is the empty set then Huet's algorithm stops reporting a failure status because the current unification problem (and therefore the original unification problem) is not unifiable.

Example 10 *Consider again the flexible-rigid equation generated in Example 8. In this case, no projection is possible because the target type of X is B and the target type of its sole argument w is A.*

Calls of SIMPL and MATCH are synchronised by the main procedure of Huet's algorithm. The main procedure receives a unification problem and, if it contains a rigid-rigid equation, it calls SIMPL. In case the original problem does not contain a rigid-rigid equation or after a possible application of SIMPL to it, the main procedure will look for a flexible-rigid equation in the current problem. If such an equation exists, the procedure MATCH is applied to this equation. Otherwise, it is a conjunction of flexible-flexible equations and, in this case, the algorithm stops and reports a success status. After an application of MATCH, either terminals or new unification problems are generated and in the latter case, this process is repeated for each of the new generated

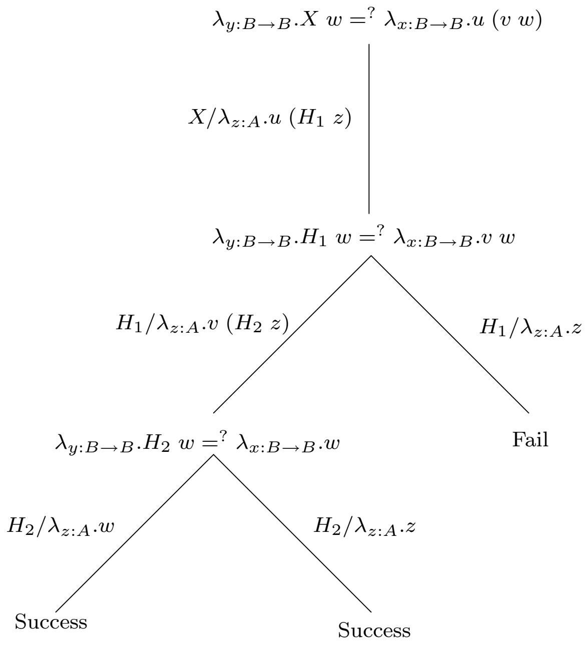

$\lambda_{y:B\to B}.X\ w =^? \lambda_{x:B\to B}.u\ (v\ w)$
$X/\lambda_{z:A}.u\ (H_1\ z)$
$\lambda_{y:B\to B}.H_1\ w =^? \lambda_{x:B\to B}.v\ w$
$H_1/\lambda_{z:A}.v\ (H_2\ z)$
$H_1/\lambda_{z:A}.z$
$\lambda_{y:B\to B}.H_2\ w =^? \lambda_{x:B\to B}.w$
Fail
$H_2/\lambda_{z:A}.w$
$H_2/\lambda_{z:A}.z$
Success
Success

Figure 2. A Matching Tree

unification problems.

Since HOU is undecidable, there exist unification problems for which Huet's algorithm does not terminate (cf. (Hue75)). The application of Huet's algorithm can be seen into a tree structure, called *matching tree*, presented in (Hue75). The matching tree is a tree whose nodes are labelled with simplified unification problems or terminals (Success or Fail) and linked to a finite number of successors by arcs labelled with substitutions. The next example shows a matching tree for the problem presented in Example 8.

Example 11 *A matching tree for the problem presented in Example 8 is given in Fig. 2. The root of the tree contains the simplified version of the original problem and the arc starting in it corresponds to an imitation substitution generated after an application of MATCH. The following node contains the simplified problem obtained after the application of this substitution. A new application of MATCH to this new unification problem generates two substitutions: an imitation that leads to two success nodes and, a projection that leads to a fail node. The solutions of the original problem are obtained by composing the substitutions generated along a success node. In this case, the solutions are given by* $X/\lambda_{z:A}.u\ (v\ w)$ *and* $X/\lambda_{z:A}.u\ (v\ z)$.

Example 12 *(Continuing example 7) Notice that the sole solution of the unification problem* $X(f\ x) =^? f\ x \wedge X(f\ x) =^? f(X\ x)$ *is the identity function:* $X/\lambda_{z:A}.z$. *The solutions for the second equation include the identity function and all compositions of* f: $X/\lambda_{z:A}.f\ z, X/\lambda_{z:A}.f(f\ z), X/\lambda_{z:A}.f(f(f\ z)), \ldots$

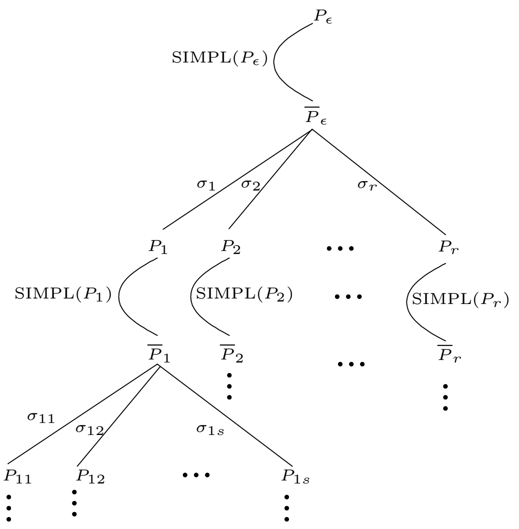


Figure 3. A Unification Tree

2.3 Unification Tree Notation

In this subsection, we introduce the *unification tree* notation for giving a systematic presentation of Huet's algorithm over the simply typed λ-calculus. Using this structure we can exhibit the connection between the two main procedures of Huet's algorithm naturally. This clarifies the description and simplifies the comparison between explicit substitutions based HOU procedures and Huet's method.

The unification tree notation derives from Huet's matching tree (Hue75) by adding new arcs for applications of SIMPL and labels for the unification problems and substitutions. These labels provide information about the position of the unification problems and of the substitutions in the unification tree (see Fig. 3).

A unification tree $\mathcal{A}(P)$ for a given unification problem P is built as follows:

(1) Label P with the subscript ϵ (the empty position), i.e., P_ϵ. This subscript means that this problem is in the root of the unification tree.
(2) For a node labelled with P_α, its child node is written $\overline{P_\alpha}$ whenever it is obtained by an application of SIMPL. This step is represented by a curly line in the unification tree since the subscript remains the same after a simplification step.
(3) For a node labelled with P_α containing the flexible-rigid equation eq, call $\sigma_{\alpha 1}, \sigma_{\alpha 2}, \ldots, \sigma_{\alpha k}$ $(k > 0)$ the substitutions generated by an application of MATCH to eq. The children nodes of P_α, written $P_{\alpha 1}, \ldots, P_{\alpha k}$ are defined by $P_{\alpha i} := \overline{P_\alpha}\sigma_{\alpha i}$, for all $1 \leq i \leq k$.

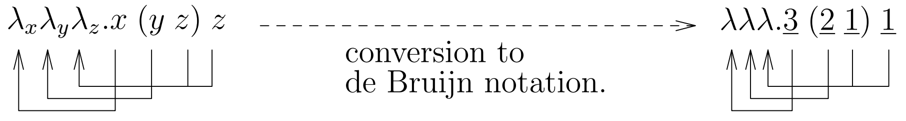
Using this notation, it is straightforward to see for instance that, for a given higher-order unification problem P, a substitution with label σ_{12315} is generated (by an application of MATCH) from a unification problem with label P_{1231}. The solutions of unification problems can be easily computed by composing the generated substitutions from the root of the unification tree to a success leaf. For instance, if P_{1223} is a success node but P_{122} is not, then the substitution solution corresponding to this success path is given by the composition $\sigma_1\sigma_{12}\sigma_{122}\sigma_{1223}$. In subsection 3.2, we describe Huet's algorithm in de Bruijn's notation using unification tree notation.

3 The Simply Typed λ-calculus and Huet's Algorithm with de Bruijn's indexes

3.1 Simply Typed λ-calculus in de Bruijn's Notation

In this subsection, we present the simply typed λ-calculus in de Bruijn's notation (dB72). The philosophy of de Bruijn's notation is based on the fact that the link between a bound variable and the corresponding λ in a term, which binds this x (we also say that x is bound by λ_x), could also be indicated by the binding height of an occurrence. To do so, bound variables and constants are represented by positive integers called *de Bruijn indexes*, which range over $\mathbb{N} = \{1, 2, \ldots\}$ and free variables (or meta-variables) are represented by capital letters $X, Y, Z, \ldots$, which range over the set $\mathcal{X}$. Meta-variables were not used in the original presentation of de Bruijn, but this separation of variables in two different classes is important for two reasons. First, for a better understanding of the unification methods presented here because we keep a clear distinction between the substitutions generated by the unification procedure and the ones generated by β-reductions, and second because we continue with a grammar for terms that is similar to the one used in the presentation of the λ-calculus with names (see section 2.2) which permits a better comprehension of the similarities and differences between the two approaches.

Rewriting λ-terms with names to de Bruijn's notation is an easy task. Consider, for instance, the closed term $\lambda_x\lambda_y\lambda_z.x\ (y\ z)\ z$. Translating this term to de Bruijn's notation consists in replacing each variable by the number that corresponds to the height of the abstractor that binds it:


$\lambda_x\lambda_y\lambda_z.x\ (y\ z)\ z$
conversion to
de Bruijn notation.
$\lambda\lambda\lambda.\underline{3}\ (\underline{2}\ \underline{1})\ \underline{1}$

Contexts for the λ-calculus in de Bruijn's notation are represented by a list of types. In the presentation with names, contexts were just finite sets of assignments. They now need to be ordered because constants, that we also call *free de Bruijn indexes*, refer to a specific position in the context.

Well typed λ-terms in the λ-calculus with names can be translated to de Bruijn's notation by fixing a referential containing its constants. So, suppose we want to write $a = \lambda_{xyz}.y\ (X\ u\ z)\ v$ in the referential u, v. To do so, we consider the term a in the scope of the abstractors $\lambda v \lambda u$, i.e., $\lambda_{vuxyz}.y\ (X\ u\ z)\ v$ which gives $\lambda\lambda\lambda.\underline{2}\ (X\ \underline{4}\ \underline{1})\ \underline{5}$. Note that, using the referential v, u we get $\lambda\lambda\lambda.\underline{2}\ (X\ \underline{5}\ \underline{1})\ \underline{4}$. In general, to convert a λ-term with names to its counterpart in de Bruijn notation we need to create a referential containing all its free variables and, as shown in the above example, different referentials lead to different de Bruijn λ-terms. Notice that meta-variables remain unchanged during this translation. For typed terms, such a referential corresponds to an "ordered" context.

Definition 13 *The set of untyped λ-terms in de Bruijn's notation is defined inductively by:*

$$a ::= \underline{n} \mid X \mid a\ a \mid \lambda.a \qquad \textit{where } n \in \mathbb{N} \textit{ and } X \in \mathcal{X}.$$

We define the syntax of simply typed λ-calculus in de Bruijn's notation by:

Types $A ::= K \mid A \to A$ *where* $K \in \mathbb{T}$.

Contexts $\Gamma ::= nil \mid A \cdot \Gamma$

Terms $a ::= \underline{n} \mid X \mid a\ a \mid \lambda_A.a$ *where* $n \in \mathbb{N}$ *and* $X \in \mathcal{X}$.

We write $\Lambda_{dB}(\mathcal{X})$ for the set of simply typed λ-terms in de Bruijn's notation. The typing rules are as follows:

$$\textit{(var)} \quad \frac{}{A \cdot \Gamma \vdash \underline{1} : A} \qquad \textit{(var+)} \quad \frac{\Gamma \vdash \underline{n} : B}{A \cdot \Gamma \vdash \underline{n+1} : B}$$

$$\textit{(lambda)} \quad \frac{A \cdot \Gamma \vdash a : B}{\Gamma \vdash \lambda_A.a : A \to B} \qquad \textit{(app)} \quad \frac{\Gamma \vdash a : A \to B \quad \Gamma \vdash b : A}{\Gamma \vdash\ (a\ b) : B}$$

In addition, to each meta-variable X we associate a unique type A and, we assume that for each type there exists an infinite number of meta-variables with that type. We add the following type rule for meta-variables:

$$\textit{(meta)} \quad \frac{}{\Gamma \vdash X : A} \qquad \textit{where } \Gamma \textit{ is any context.}$$

As in the λ-calculus with names, the type of meta-variables is independent from its context. Nevertheless, the type of λ-terms (that contain constants) depends on the context. In addition, if a is a λ-term, we write a^{Γ}_{A} as a short hand for the type judgement $\Gamma \vdash a : A$.

Definition 14 (Extension of contexts(DHK00)) *Let $n \geq 0$, $A_1, \ldots, A_n$ be types and Γ and Δ be two contexts. We say that Γ is an* extension *of Δ if it has the form $\Gamma = A_1 \cdot \ldots \cdot A_n \cdot \Delta$. It is a strict extension if $n \neq 0$.*

The β-reduction for λ-terms in de Bruijn's notation is given by:

$$(\lambda_A.a)\ b \to a\{\underline{1}/b\} \qquad (\beta)$$

We say that a λ-term a, in de Bruijn's notation, is in β-normal form (β-nf for short) if a does not have a sub-term of the form $(\lambda_A.b)\ c$. This definition of β-reduction requires specific rules for propagating the substitution $\{\underline{1}/b\}$ over the term a. This is done by the following definition:

Definition 15 *Let $\underline{n}, a, b$ be well typed λ-terms in de Bruijn's notation such that $\underline{n}$ and a are two λ-terms with the same type. The substitution of a for $\underline{n}$ in b, written $b\{\underline{n}/a\}$, is defined by induction over the structure of b as follows:*

(a) $X\{\underline{n}/a\} = X$.

(b) $\underline{m}\{\underline{n}/a\} = \begin{cases} \underline{m}, & \text{if } m < n; \\ a, & \text{if } m = n; \\ \underline{m-1}, & \text{if } m > n. \end{cases}$

(c) $(c\ d)\{\underline{n}/a\} = c\{\underline{n}/a\}\ d\{\underline{n}/a\}$

(d) $(\lambda_A.c)\{\underline{n}/a\} = \lambda_A.c\{\underline{n+1}/a^+\}$

This definition of a (higher-order) substitution is specific for β-reduction: in fact, in item (b), if $m > n$ then $\underline{m}\{\underline{n}/a\}$ is equal to $\underline{m-1}$ because this substitution subsumes that a λ disappeared after an application of a β-reduction and, hence all the constants of the current term need to be decremented by one because now they are under the scope of one less abstractor. When the substitution $\{\underline{n}/a\}$ is propagated inside an abstraction, the term a^+ is generated. It is called the *lift* of a and is given by the following definition:

Definition 16 (The lift(DHK00)) *Let $a \in \Lambda_{dB}(\mathcal{X})$, $i \geq 0$. The term a^+, called the lift of a, is defined by $a^+ = a^{+0}$, where a^{+i} is inductively defined by:*

(a) $X^{+i} = X$, *for* $X \in \mathcal{X}$;

(b) $\underline{n}^{+i} = \begin{cases} \underline{n+1}, & \text{if } n > i; \\ \underline{n}, & \text{if } n \leq i; \end{cases}$

(c) $(a\ b)^{+i} = a^{+i}\ b^{+i}$;

(d) $(\lambda_A.a)^{+i} = \lambda_A.a^{+(i+1)}$.

Notice that we have defined β-reduction as well as the notion of higher-order substitution for β-reduction without the decoration with types and contexts. Nevertheless, since we are interested in higher-order unification, it is important to keep in mind how this information is manipulated. The decorated version of β-reduction is given by:

$$(\lambda_A.a_B^{A\cdot\Gamma})\ b_A^{\Gamma} \to a_B^{A\cdot\Gamma}\{\underline{1}_A^{A\cdot\Gamma}/b_A^{\Gamma}\} \qquad (\beta)$$

This definition says that each free occurrence of $\underline{1}_A^{A\cdot\Gamma}$ in $a_B^{A\cdot\Gamma}$ must be replaced by b_A^{Γ}, and that is the reason why $\underline{1}_A^{A\cdot\Gamma}$ and $a_B^{A\cdot\Gamma}$ must have the same context. One important point about a substitution generated by β-reductions is that it always has the form $\{\underline{n}/b\}$ where $\underline{n}$ and b are terms of the same type but with different contexts. In fact, the free occurrences of the de Bruijn index $\underline{n}$ in the term a are in the scope of the abstractor that will be removed after the application of the (β), but the term b is not in the scope of this abstractor. In what follows, we present the decorated version of definitions 15 and 16.

Remark 17 *Let Δ be a context and let $A, B, A_1, \ldots, A_n$ be types. The decorated version of Definition 15 is given in what follows. Assuming that the considered substitution was originated by a β-redex whose abstractor was of type A_1, we have that:*

(a) $X_A^{A_n\cdot\ldots\cdot A_1\cdot\Delta}\{\underline{n}_{A_1}^{A_n\cdot\ldots\cdot A_1\cdot\Delta}/a_{A_1}^{A_n\cdot\ldots\cdot A_2\cdot\Delta}\} = X_A^{A_n\cdot\ldots\cdot A_2\cdot\Delta}$*, i.e., meta-variables are not affected by the substitutions generated by β-reduction, but the context of the resulting term is given by the context of the term a given in the substitution.*

(b) It is divided in three sub-cases:

- *If $m < n$ then $\underline{m}$ represents a bound de Bruijn index and must remain unchanged:*

$$\underline{m}_A^{A_n\cdot\ldots\cdot A_1\cdot\Delta}\{\underline{n}_{A_1}^{A_n\cdot\ldots\cdot A_1\cdot\Delta}/a_{A_1}^{A_n\cdot\ldots\cdot A_2\cdot\Delta}\} = \underline{m}_A^{A_n\cdot\ldots\cdot A_2\cdot\Delta}$$

- *If $m = n$ then*

$$\underline{n}_{A_1}^{A_n\cdot\ldots\cdot A_1\cdot\Delta}\{\underline{n}_{A_1}^{A_n\cdot\ldots\cdot A_1\cdot\Delta}/a_{A_1}^{A_n\cdot\ldots\cdot A_2\cdot\Delta}\} = a_{A_1}^{A_n\cdot\ldots\cdot A_2\cdot\Delta}$$

- *If $m > n$ then the de Bruijn index $\underline{m}$ represents a constant whose scope contains one less abstractor (the one that originated the β-reduction was eliminated) and, then the first element of the context (whose type is exactly the type of the eliminated abstractor) is removed:*

$$\underline{m}_A^{A_n\cdot\ldots\cdot A_1\cdot\Delta}\{\underline{n}_{A_1}^{A_n\cdot\ldots\cdot A_1\cdot\Delta}/a_{A_1}^{A_n\cdot\ldots\cdot A_2\cdot\Delta}\} = \underline{m-1}_A^{A_n\cdot\ldots\cdot A_2\cdot\Delta}$$

(c) Trivial

(d) After propagating a substitution inside an abstractor of a term, the index $\underline{n}$ that defines the substitution and the term in the substitution as well as their contexts need to be updated:
$((\lambda_B.b_A^{B\cdot A_n\cdot\ldots\cdot A_1\cdot\Delta})_{B\to A}^{A_n\cdot\ldots\cdot A_1\cdot\Delta}\{\underline{n}_{A_1}^{A_n\cdot\ldots\cdot A_1\cdot\Delta}/a_{A_1}^{A_n\cdot\ldots\cdot A_2\cdot\Delta}\})_{B\to A}^{A_n\cdot\ldots\cdot A_2\cdot\Delta} =$
$(\lambda_B.(b_A^{B\cdot A_n\cdot\ldots\cdot A_1\cdot\Delta}\{\underline{n+1}_{A_1}^{B\cdot A_n\cdot\ldots\cdot A_1\cdot\Delta}/(a^+)_{A_1}^{B\cdot A_n\cdot\ldots\cdot A_2\cdot\Delta}\}))_{B\to A}^{A_n\cdot\ldots\cdot A_2\cdot\Delta}$.
In this way, the lift increases the context of the terms in the substitutions with the type of the abstractor that binds them.

The lift of Definition 16 is motivated by item (d) above. In fact, the lift of a term is necessary only when a substitution is propagated inside an abstraction whose type is essential to determine the resulting context. In the following, *we assume that B is the type of the abstraction that originated the lift. The decorated version of the lift is given by:*

(a) *For all $i \geq 0$, $((X_A^{A_1 \cdot \ldots \cdot A_i \cdot \Delta})^{+i})_A^{A_1 \cdot \ldots \cdot A_i \cdot B \cdot \Delta} = X_A^{A_1 \cdot \ldots \cdot A_i \cdot B \cdot \Delta}$.*

(b) *We analyse each case separately:*

- *If $n > i$ then the index $\underline{n}$ represents a constant that is in the scope of i abstractors and which has now been inserted (by the substitution) in the scope of a new abstractor of type B. Therefore it needs to be updated and, its context must contain the type information concerning this new abstractor:*

$$((\underline{n}_{A_n}^{A_1 \cdot \ldots \cdot A_i \cdot \Delta})^{+i})_{A_n}^{A_1 \cdot \ldots \cdot A_i \cdot B \cdot \Delta} = \underline{n+1}_{A_n}^{A_1 \cdot \ldots \cdot A_i \cdot B \cdot \Delta}$$

- *If $n \leq i$ then $\underline{n}$ represents a bound variable and must remain unchanged but the resulting context depends on the lift:*

$$((\underline{n}_{A_n}^{A_1 \cdot \ldots \cdot A_i \cdot \Delta})^{+i})_{A_n}^{A_1 \cdot \ldots \cdot A_i \cdot B \cdot \Delta} = \underline{n}_{A_n}^{A_1 \cdot \ldots \cdot A_i \cdot B \cdot \Delta}$$

(c) *Trivial.*

(d) *To propagate a lift inside an abstraction, it is necessary to include the type of the abstraction that originated the lift:*
$(((\lambda_A.a_C^{A \cdot A_1 \cdot \ldots \cdot A_i \cdot \Delta})_{A \to C}^{A_1 \cdot \ldots \cdot A_i \cdot \Delta})^{+i})_{A \to C}^{A_1 \cdot \ldots \cdot A_i \cdot B \cdot \Delta} =$
$(\lambda_A.(a^{i+1})_C^{A \cdot A_1 \cdot \ldots \cdot A_i \cdot B \cdot \Delta})_{A \to C}^{A_1 \cdot \ldots \cdot A_i \cdot B \cdot \Delta}$.

Whenever it is possible we avoid the decorated notation for the sake of clarity.

The η-reduction for the λ-calculus in de Bruijn's notation is defined as follows:

$$\lambda_A.a\ \underline{1} \to b \text{ if } a = b^+ \qquad (\eta)$$

and its version decorated with types and contexts is given by:

$$\lambda_A.a_B^{A \cdot \Gamma}\ \underline{1}_A^{A \cdot \Gamma} \to b_B^{\Gamma} \text{ if } a_B^{A \cdot \Gamma} = ((b_B^{\Gamma})^+)_B^{A \cdot \Gamma} \qquad (\eta)$$

The above definition of η-reduction tries to capture the operational semantics of the η-reduction of the λ-calculus with names, but in fact it fails because it does not show how to construct the term b from a. However, implementations of the η-reduction based on detection of occurrences of the index $\underline{1}$ in a are adequate (AMK05).

As mentioned before, the separation of the free variables (meta-variables) on one side and the bound variables and constants (de Bruijn indexes) on the other side allow us to distinguish between the substitutions generated by β-reductions from the ones generated by the unification procedure. The next definition formalises the notion of substitution generated by the unification procedure, i.e., substitutions for meta-variables:

Definition 18 *Let θ be a valuation (i.e., a function) from $\mathcal{X}$ to $\Lambda_{dB}(\mathcal{X})$. The* substitution θ' *extending the valuation θ is defined by:*

(a) $X\theta' = X\theta$ $\qquad$ (b) $\underline{n}\theta' = \underline{n}$

(c) $(a\ b)\theta' = a\theta'\ \ b\theta'$ $\qquad$ (d) $(\lambda_A.a)\theta' = \lambda_A.(a\theta'^{+})$

where $\theta'^{+} := \{X_1/a_1^{+}, \ldots, X_n/a_n^{+}\}$ *when* $\theta' = \{X_1/a_1, \ldots, X_n/a_n\}$.

The main difference between the substitution generated by β-reduction and the one generated by the unification procedure is that the latter always replaces a meta-variable for a term, say X/a, where X and a are λ-terms with the same type and context. Here again, contexts need to be updated when propagated over abstractions: for instance, the decorated version of item (d) is given by $(\lambda_B.c_C^{B\cdot\Delta})\{X_A^{\Gamma}/a_A^{\Gamma}\} = \lambda_B.(c_C^{B\cdot\Delta}\{X_A^{B\cdot\Gamma}/(a^{+})_A^{B\cdot\Gamma}\})$.

The next example clarifies the process of propagating substitutions and the notion of lifting.

Example 19 *Let $\Gamma = (A \to A) \to A$ be a context and, X be a meta-variable of type $(A \to A) \to A$ in context Γ. In this example, we show how the substitution $\{X^{\Gamma}_{(A\to A)\to A}/\underline{1}^{\Gamma}_{(A\to A)\to A}\}$ can be propagated over the λ-term $\lambda_{A\to A}.(X\ \lambda_A.(X\ \underline{2}))$ that has type $(A \to A) \to A$ in context Γ. Due to lack of space, we only decorate the sub-terms that are relevant while propagating the substitution:*

$$(\lambda_{A\to A}.(X\ \lambda_A.(X\ \underline{2})))^{\Gamma}_{(A\to A)\to A}\{X^{\Gamma}_{(A\to A)\to A}/\underline{1}^{\Gamma}_{(A\to A)\to A}\} =$$
$$\lambda_{A\to A}.(X\ \lambda_A.(X\ \underline{2}))_A^{A\to A\cdot\Gamma}\{X^{A\to A\cdot\Gamma}_{(A\to A)\to A}/\underline{2}^{A\to A\cdot\Gamma}_{(A\to A)\to A}\} =$$
$$\lambda_{A\to A}.(\underline{2}^{A\to A\cdot\Gamma}_{(A\to A)\to A}\ (\lambda_A.(X\ \underline{2}))^{A\to A\cdot\Gamma}_{A\to A}\{X^{A\to A\cdot\Gamma}_{(A\to A)\to A}/\underline{2}^{A\to A\cdot\Gamma}_{(A\to A)\to A}\}) =$$
$$\lambda_{A\to A}.(\underline{2}^{A\to A\cdot\Gamma}_{(A\to A)\to A}\ \lambda_A.(X\ \underline{2})_A^{A\cdot A\to A\cdot\Gamma}\{X^{A\cdot A\to A\cdot\Gamma}_{(A\to A)\to A}/\underline{3}^{A\cdot A\to A\cdot\Gamma}_{(A\to A)\to A}\}) =$$
$$\lambda_{A\to A}.(\underline{2}^{A\to A\cdot\Gamma}_{(A\to A)\to A}\ \lambda_A.(\underline{3}^{A\cdot A\to A\cdot\Gamma}_{(A\to A)\to A}\ \underline{2}^{A\cdot A\to A\cdot\Gamma}_{A\to A})).$$

In this example, the lift was used twice: once in the second line (top-down) and once in the fourth line.

In the following we define the *updating functions* that are used in the definition of η-long normal forms for λ-terms in de Bruijn's notation.

Definition 20 *The* updating functions $U_k^i : \Lambda_{dB}(\mathcal{X}) \to \Lambda_{dB}(\mathcal{X})$, *for $k \geq 0$ and $i \geq 1$ are defined inductively by:*

(a) $U_k^i(X) = X,$ *for* $X \in \mathcal{X}$ $\qquad$ (b) $U_k^i(a\ b) = U_k^i(a)\ U_k^i(b)$

(c) $U_k^i(\lambda_A.a) = \lambda_A.U_{k+1}^i(a)$ $\qquad$ (d) $U_k^i(\underline{n}) = \begin{cases} \underline{n+i-1}, & \text{if } n > k \\ \underline{n}, & \text{if } n \leq k \end{cases}$

In the λ-calculus, η-long forms play an important role. Definition 21 and Proposition 23 were adapted from (DHK00):

Definition 21 (η-long nf) *Let $a \in \Lambda_{dB}(\mathcal{X})$ be a λ-term in de Bruijn's notation of type $A_1 \to \ldots \to A_m \to B$ (B atomic) in context Γ and in β-nf. The* η-long normal form *(or η-lnf for short) a' of a, is inductively defined by:*

- *if $a = \lambda_A.b$ then $a' = \lambda_A.b'$.*
- *if $a = \underline{n}\ b_1 \ldots b_q$, with $q \geq 0$, then $a' = \lambda_{A_1} \ldots \lambda_{A_m}.\underline{n+m}\ c_1 \ldots c_q\ \underline{m}' \ldots \underline{1}'$, where $c_1, \ldots, c_q$ are the η-lnf of the β-nf of $U_0^{m+1}(b_1), \ldots, U_0^{m+1}(b_q)$, resp.*
- *if $a = X\ b_1 \ldots b_q$, with $q \geq 0$, then $a' = \lambda_{A_1} \ldots \lambda_{A_m}.X\ c_1 \ldots c_q\ \underline{m}' \ldots \underline{1}'$, where $c_1, \ldots, c_q$ are the η-lnf of the β-nf of $U_0^{m+1}(b_1), \ldots, U_0^{m+1}(b_q)$, resp.*

The next definition is needed to prove that the definition of η-lnf is well founded.

Definition 22 *The size $|a|$ of a λ-term $a \in \Lambda_{dB}(\mathcal{X})$ is inductively defined by:*

- *if $a = \underline{n}$ or $a = X$ then $|a| = 1$;*
- *if $a = b\ c$ then $|a| = 1 + |b| + |c|$;*
- *if $a = \lambda_A.b$ then $|a| = 1 + |b|$.*

Proposition 23 *The definition of η-lnf for λ-terms in de Bruijn's notation is well founded.*

PROOF. The proof is by induction based on the lexicographic order on the triple consisting of the number of occurrences of meta-variables, the size of the λ-term and the size of its type. The size of a type is defined as usual: if A is atomic then $|A| = 1$ and if B and C are types then $|B \to C| = max(1 + |B|, |C|)$.

In the case $a = \lambda_A.b$ we have that the number of meta-variables remain unchanged and the size of the term decreases. When $a = \underline{n}\ b_1 \ldots b_q$ and $q = 0$ the number of meta-variables and the size of the term remain unchanged but the size of the type decreases. If $q \neq 0$ then the number of meta-variables remains unchanged and the size of the term decreases. When $a = X\ b_1 \ldots b_q$ the number of meta-variables decreases. □

Example 24 *Consider the type judgement $A \to A{\cdot}nil \vdash \underline{1} : A \to A$. The η-lnf of the de Bruijn index $\underline{1}$ in this type judgement, in a first step, corresponds to the η-lnf of $A \to A \cdot nil \vdash \lambda_A.\underline{2}\ \underline{1}' : A \to A$. But the η-lnf of a de Bruijn index of an atomic type is the index itself, and therefore, the η-lnf of the original term is given by $A \to A \cdot nil \vdash \lambda_A.\underline{2}\ \underline{1} : A \to A$.*

Example 25 *A more interesting case is to calculate the η-lnf of $(A \to A) \to A \cdot nil \vdash \underline{1} : (A \to A) \to A$. According to the definition it corresponds to the η-lnf of $(A \to A) \to A \cdot nil \vdash \lambda_{A \to A}.\underline{2}\ \underline{1}' : (A \to A) \to A$. Now the problem is reduced to calculating the η-lnf of the λ-term $\underline{1}'$ that has type $A \to A$ in context*

$A \to A \cdot (A \to A) \to A \cdot nil$. Following the previous example, we have that the η-lnf of $A \to A \cdot (A \to A) \to A \cdot nil \vdash \underline{1}' : A \to A$ is given by $A \to A \cdot (A \to A) \to A \cdot nil \vdash \lambda_A.\underline{2}\ \underline{1} : A \to A$. Therefore, we have that the η-lnf of the original term is given by $(A \to A) \to A \cdot nil \vdash \lambda_{A\to A}.\underline{2}\ \lambda_A.\underline{2}\ \underline{1} : (A \to A) \to A$.

3.2 Huet's algorithm in de Bruijn's notation

The definitions of normal forms, flexible and rigid terms for de Bruijn's notation are a straightforward adaptation from those given in subsection 2.2. For the definition of a unification problem in de Bruijn notation one needs to observe that contexts now have order; they are represented by lists of types. In order to convert a unification problem from the λ-calculus with names to the λ-calculus in de Bruijn notation one needs to set an order for contexts. This can be done in many different ways, and in the following we explain how we perform this transformation through an example:

Example 26 *Let $\Gamma = \{x : A, y : B, z : A \to B\}$ be a context, X a meta-variable of type $A \to B$ and P_Λ be the unification problem given by:*

$$\lambda_{u:A}.X\ u =^? \lambda_{u:A}.y\ \wedge X\ x =^? z\ x$$

which is well typed in Γ. In order to convert P_Λ to de Bruijn's notation, we need first to convert the context Γ into a list of types. To do so, we simply get the types of all the elements in Γ in any order and build a list with these types (for simplicity we keep the name Γ for the resulting context): $\Gamma = A \cdot B \cdot A \to B \cdot nil$. Note that in a certain sense, the generated context corresponds to the referential *cited in Section 3.1 that allows us to convert terms with free variables. In this way, the unification problem P_Λ in de Bruijn notation, written $P_{\Lambda_{dB}}$, is given by: $\lambda_A.X\ \underline{1} =^? \lambda_A.\underline{3}\ \wedge X\ \underline{1} =^? \underline{3}\ \underline{1}$.*

In the next subsections we present the procedures SIMPL and MATCH of Huet's algorithm in de Bruijn's notation using the unification tree notation.

3.2.1 The procedure SIMPL

It receives as argument a unification problem P_α containing at least one rigid-rigid equation (otherwise it is already a simplified problem) and returns either a terminal (Success or Fail) or an equivalent (simplified) unification problem, written $\overline{P_\alpha}$, containing at least one flexible-rigid equation. In the following we give a description of SIMPL.

Procedure SIMPL

`INPUT`: A unification problem P_α with at least one rigid-rigid equation.

`OUTPUT`: Either a terminal (Success or a Fail) or an equivalent unification problem $\overline{P_\alpha}$ without rigid-rigid equations and containing at least one flexible-rigid equation.

`WHILE` there exists a rigid-rigid equation in P_α, say:

$$\lambda_{A_1} \dots \lambda_{A_n}.h_1\ e^1_1 \dots e^1_{p_1} =^? \lambda_{A_1} \dots \lambda_{A_n}.h_2\ e^2_1 \dots e^2_{p_2} \wedge P' \tag{5}$$

where $n, p_1, p_2 \geq 0$ and h_1 and h_2 are de Bruijn indexes `DO`
If h_1 and h_2 are different de Bruijn indexes then stop and report a failure status. Otherwise, replace the equation (5) (in which $p_1 = p_2$ because the terms have the same type) by the conjunction

$$\lambda_{A_1} \dots \lambda_{A_n}.e^1_1 =^? \lambda_{A_1} \dots \lambda_{A_n}.e^2_1 \ \wedge \dots \wedge \ \lambda_{A_1} \dots \lambda_{A_n}.e^1_{p_1} =^? \lambda_{A_1} \dots \lambda_{A_n}.e^2_{p_1}$$

in P_α and call $\overline{P_\alpha}$ the resulting problem.
`DONE`.
`IF` there exists a flexible-rigid equation in $\overline{P_\alpha}$ `THEN` return $\overline{P_\alpha}$ `ELSE` stop and report a success status.

3.2.2 The Procedure MATCH

The procedure MATCH takes a flexible-rigid equation as input and returns a finite set of substitutions, Σ. As explained for the notation with names, it is based on the *imitation* and *projection* rules detailed in the following.

3.2.3 The Imitation Rule.

Consider the following flexible-rigid equation well typed in context Γ:

$$\lambda_{A_1} \dots \lambda_{A_n}.X\ e^1_1 \dots e^1_{p_1} =^? \lambda_{A_1} \dots \lambda_{A_n}.\underline{h}\ e^2_1 \dots e^2_{p_2} \tag{6}$$

where:

- $n, p_1, p_2 \geq 0$;
- X is a meta-variable of type $B_1 \to \dots \to B_{p_1} \to A$ (A atomic);
- $\underline{h}$ is a de Bruijn index of type $C_1 \to \dots \to C_{p_2} \to A$ (A atomic);
- if $p_1 \neq 0$ then e^1_i is a λ-term in η-lnf of type B_i for all $1 \leq i \leq p_1$;
- if $p_2 \neq 0$ then e^2_j is a λ-term in η-lnf of type C_j for all $1 \leq j \leq p_2$.

In order to avoid variable capture, an imitation substitution is generated only if $\underline{h}$ is a constant, i.e., $h > n$. In this case, the imitation substitution is given by:

$$X/\lambda_{B_1} \dots \lambda_{B_{p_1}}.\underline{p_1 + h - n}\ (X_1\ \underline{p_1} \dots \underline{1}) \dots (X_{p_2}\ \underline{p_1} \dots \underline{1})$$

where X_i is a fresh meta-variable of type $B_1 \to \ldots \to B_{p_1} \to C_i$ in context Γ, for all $1 \leq i \leq p_2$.

3.2.4 The Projection Rule.

For each argument of X (in equation (6)) that have the same target type as X, a projection is generated. In this way, if e_i^1 has a type of the form $B_i = D_1 \to \ldots \to D_q \to A$, for some $1 \leq i \leq p_1$, then the generated projection substitution is given by:

$$X/\lambda_{B_1} \ldots \lambda_{B_{p_1}} . \underline{p_1 - i + 1}\, (H_1\, \underline{p_1} \ldots \underline{1}) \ldots (H_q\, \underline{p_1} \ldots \underline{1})$$

where H_j is a fresh meta-variable of type $B_1 \to \ldots \to B_{p_1} \to D_j$ for all $1 \leq j \leq q$.

In the following we give an algorithmic description of the procedure MATCH.

Procedure MATCH

`INPUT`: A flexible-rigid equation *eq*.
`OUTPUT`: A set Σ of substitutions for the head of the flexible term.

(1) Apply the imitation and the projection rules to *eq* non-deterministically and call Σ the set of generated substitutions.

3.2.5 The Main Procedure

The main procedure of Huet's algorithm non-deterministically and successively calls the procedures SIMPL and MATCH.

Main Procedure

`INPUT`: A unification problem P_ϵ.
`OUTPUT`: A success status if the original problem is unifiable or a failure status if the original problem is not unifiable. The algorithm may not terminate in the latter case.

(1) If $P_{i_1 \ldots i_k}$ contains a rigid-rigid equation then apply SIMPL and go to the next step, else if it contains a flexible-rigid equation then rename $P_{i_1 \ldots i_k}$ to $\overline{P}_{i_1 \ldots i_k}$ and go to the next step, else go to step 4.
(2) Let *eq* be a flexible-rigid equation in $\overline{P}_{i_1 \ldots i_k}$. Apply MATCH to *eq* and call $\Sigma_{i_1 \ldots i_k}$ the generated set of substitutions and go to step 3.
(3) If $\Sigma_{i_1 \ldots i_k}$ is the empty set then stop and report a failure status, else let $\Sigma_{i_1 \ldots i_k} = \{\sigma_{i_1 \ldots i_k 1}, \ldots, \sigma_{i_1 \ldots i_k r}\}$ where $r > 0$ and, for each substitution

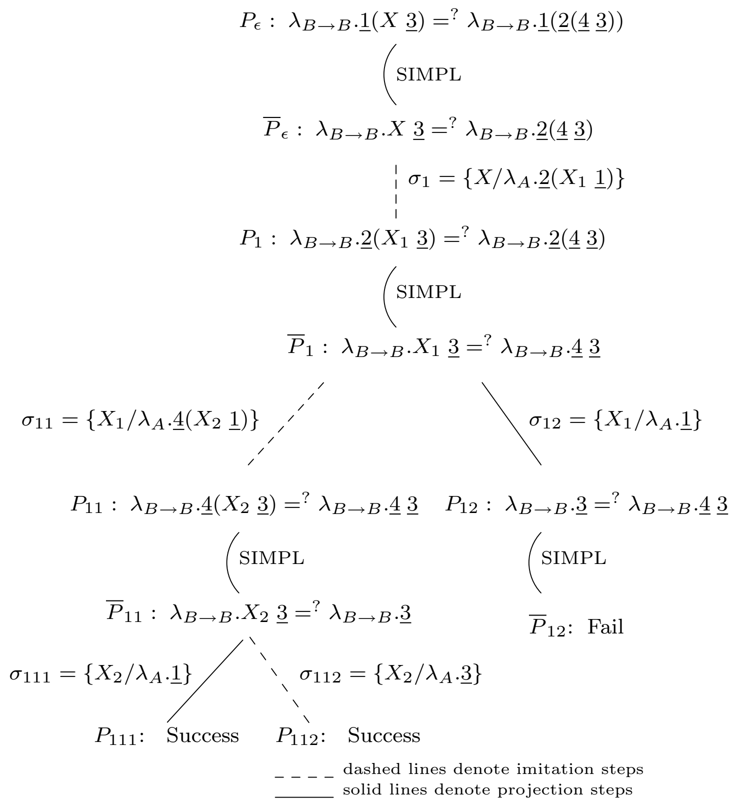

$P_\epsilon : \lambda_{B\to B}.\underline{1}(X\ \underline{3}) =^? \lambda_{B\to B}.\underline{1}(\underline{2}(\underline{4}\ \underline{3}))$
SIMPL
$\overline{P}_\epsilon : \lambda_{B\to B}.X\ \underline{3} =^? \lambda_{B\to B}.\underline{2}(\underline{4}\ \underline{3})$
$\sigma_1 = \{X/\lambda_A.\underline{2}(X_1\ \underline{1})\}$
$P_1 : \lambda_{B\to B}.\underline{2}(X_1\ \underline{3}) =^? \lambda_{B\to B}.\underline{2}(\underline{4}\ \underline{3})$
SIMPL
$\overline{P}_1 : \lambda_{B\to B}.X_1\ \underline{3} =^? \lambda_{B\to B}.\underline{4}\ \underline{3}$
$\sigma_{11} = \{X_1/\lambda_A.\underline{4}(X_2\ \underline{1})\}$
$\sigma_{12} = \{X_1/\lambda_A.\underline{1}\}$
$P_{11} : \lambda_{B\to B}.\underline{4}(X_2\ \underline{3}) =^? \lambda_{B\to B}.\underline{4}\ \underline{3}$
$P_{12} : \lambda_{B\to B}.\underline{3} =^? \lambda_{B\to B}.\underline{4}\ \underline{3}$
SIMPL
SIMPL
$\overline{P}_{11} : \lambda_{B\to B}.X_2\ \underline{3} =^? \lambda_{B\to B}.\underline{3}$
$\overline{P}_{12}$: Fail
$\sigma_{111} = \{X_2/\lambda_A.\underline{1}\}$
$\sigma_{112} = \{X_2/\lambda_A.\underline{3}\}$
P_{111}: Success
P_{112}: Success
dashed lines denote imitation steps
solid lines denote projection steps

Figure 4. Unification tree example.

$\sigma_{i_1 \dots i_k j} \in \Sigma_{i_1 \dots i_k}$ call $P_{i_1 \dots i_k j} := P_{i_1 \dots i_k} \sigma_{i_1 \dots i_k j}$ the new unification problem and go to step 1.

(4) Stop and report a success status. The corresponding solution assuming that the current node is at position $i_1 \dots i_k$ is given by the composition: $\sigma_{i_1} \sigma_{i_1 i_2} \dots \sigma_{i_1 i_2 \dots i_{k-1}} \sigma_{i_1 i_2 \dots i_k}$

Example 27 *Consider the unification problem given in Example 8. First of all, we need to rewrite its terms in de Bruijn's notation. To do so, we choose the referential $u : A \to B, w : A, v : A \to A$. This referential corresponds to the context $\Gamma = A \to B \cdot A \cdot A \to A \cdot nil$. As explained in Section 3.1 this corresponds to considering the terms of the equation:*

$$\lambda_{y:B\to B}.y\ (X\ w) =^? \lambda_{x:B\to B}.x\ (u\ (v\ w))$$

under the scope of the abstractors $\lambda_{v:A\to A}\lambda_{w:A}\lambda_{u:A\to B}$ and we get:

$$\lambda_{B\to B}.\underline{1}(X\ \underline{3}) =^? \lambda_{B\to B}.\underline{1}(\underline{2}(\underline{4}\ \underline{3}))$$

is well typed in context Γ.

Figure 4 shows a unification tree generated for this problem. The solutions are given by the compositions of the substitutions through a path whose leaf is a success node, i.e., this node corresponds to a problem containing at most *a finite number of flexible-flexible equations. Note that, after composing, the terms need to be normalised. For instance, one can compute the composition $\sigma_1\sigma_{11}\sigma_{112}$ by first applying the usual composition of substitutions:*

$$\{X/\lambda_A.\underline{2}((\lambda_A.\underline{5}((\lambda_A.\underline{1})\ \underline{1}))\ \underline{1}), X_1/\lambda_A.\underline{4}((\lambda_A.\underline{1})\ \underline{1}), X_2/\lambda_A.\underline{1}\}$$

and then applying β-reduction:

$$\{X/\lambda_A.\underline{2}(\underline{4}\ \underline{1}), X_1/\lambda_A.\underline{4}\ \underline{1}, X_2/\lambda_A.\underline{1}\}.$$

In the same way, one computes the substitution:

$$\sigma_1\sigma_{11}\sigma_{112} = \{X/\lambda_A.\underline{2}(\underline{4}\ \underline{3}), X_1/\lambda_A.\underline{4}\ \underline{3}, X_2/\lambda_A.\underline{3}\}.$$

The solutions to the original problem are given by the substitutions for the meta-variables that appear in it: $X/\lambda_A.\underline{2}(\underline{4}\ \underline{3})$ and $X/\lambda_A.\underline{2}(\underline{4}\ \underline{1})$.

4 The Relation between HOU in the λ-calculus and in the $\lambda\sigma$-calculus

In this section, we relate, HOU à la Huet and HOU in the $\lambda\sigma$-calculus. In (DHK00), Dowek, Hardin and Kirchner prove that a unification problem in the simply typed λ-calculus has a solution if and only if its precooked image has a solution. In this paper we go a step further and show that, for each derived problem P_α of a given problem P, there exists a derived problem P^*_B of the precooked image of P that preserves solutions in the following sense: if the substitution σ is a solution to P_α then the grafting σ_F is a solution to P^*_B. We start with a brief presentation of the $\lambda\sigma$-calculus.

4.1 The $\lambda\sigma$-calculus

The λ-calculus is based on a notion of substitution that belongs to a meta-language. Such a notion is necessary because the substitution process adopts renaming of bound variables in order to avoid variable capture. A natural solution to define a substitution which belongs to the language itself is to extend the language of the λ-calculus by incorporating explicit operators for the substitution. The first mechanism that "explicited" the substitution operation was the $\lambda\sigma$-calculus (ACCL91) that we briefly present in the following.

Definition 28 *The syntax of the simply typed $\lambda\sigma$-calculus is given by:*

Types	$A ::= K \mid A \to A$	*where* $K \in \mathbb{T}$
Contexts	$\Gamma ::= nil \mid A \cdot \Gamma$	
Terms	$a ::= \underline{1} \mid X \mid a\ a \mid \lambda_A.a \mid a[s]$	*where* $X \in \mathcal{X}$
Substitutions	$s ::= id \mid \uparrow \mid a \cdot s \mid s \circ s$	

The set of well typed $\lambda\sigma$-terms with meta-variables is denoted by $\Lambda_{\lambda\sigma}(\mathcal{X})$. Substitutions are lists of terms in the $\lambda\sigma$-calculus and hence the type of a substitution must be a list of types, i.e., a context. If s is a substitution and Γ and Δ are contexts then we write $\Gamma \vdash s \triangleright \Delta$ to represent that the substitution s has type Δ in context Γ. The typing rules for the $\lambda\sigma$-calculus are as follows:

$$\textit{(var)}\ \frac{}{A \cdot \Gamma \vdash \underline{1} : A} \qquad \textit{(lambda)}\ \frac{A \cdot \Gamma \vdash a : B}{\Gamma \vdash \lambda_A.a : A \to B}$$

$$\textit{(app)}\ \frac{\Gamma \vdash a : A \to B \quad \Gamma \vdash b : A}{\Gamma \vdash\ (a\ b) : B} \qquad \textit{(clos)}\ \frac{\Gamma \vdash s \triangleright \Gamma' \quad \Gamma' \vdash a : A}{\Gamma \vdash a[s] : A}$$

$$\textit{(id)}\ \frac{}{\Gamma \vdash id \triangleright \Gamma} \qquad \textit{(shift)}\ \frac{}{A \cdot \Gamma \vdash \uparrow \triangleright \Gamma}$$

$$\textit{(cons)}\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash s \triangleright \Gamma'}{\Gamma \vdash a \cdot s \triangleright A \cdot \Gamma'} \qquad \textit{(comp)}\ \frac{\Gamma \vdash s'' \triangleright \Gamma'' \quad \Gamma'' \vdash s' \triangleright \Gamma'}{\Gamma \vdash s' \circ s'' \triangleright \Gamma'}$$

In addition, to each meta-variable X we associate a unique type T_X and a unique context Γ_X. We add the following type rule for meta-variables:

$$\textit{(meta)}\ \frac{}{\Gamma_X \vdash X : T_X}$$

In contrast to the (*meta*) rule of the simply typed λ-calculus (in de Bruijn's notation), the (*meta*) rule for the $\lambda\sigma$-calculus shows that the types of $\lambda\sigma$-terms are not independent from the contexts. This is necessary because unification in the $\lambda\sigma$-calculus uses grafting instead of substitution; and we would like grafting and typing to be compatible in the $\lambda\sigma$-calculus. This restriction over meta-variables avoids, for example, the replacement of the two occurrences of X in the $\lambda\sigma$-term $(X\ \lambda_A.X)$ by the same $\lambda\sigma$-term (see Remark 36 for further details).

The rewriting rules of the $\lambda\sigma$-calculus are given in Table 1.

In this calculus, when a substitution s is applied to a term a we internalise this as $a[s]$. Simultaneous substitutions are represented as lists of terms with the usual operator *cons* (written as "$\cdot$") and an operator for the empty list (written *id* which represents the identity substitution) and the operator $\uparrow$ which

(Beta)	$(\lambda.a)\ b$	$\longrightarrow$	$a[b \cdot id]$
(App)	$(a\ b)[s]$	$\longrightarrow$	$a[s]\ b[s]$
(Abs)	$(\lambda.a)[s]$	$\longrightarrow$	$\lambda.a[\underline{1} \cdot (s\circ \uparrow)]$
(Clos)	$(a[s])[t]$	$\longrightarrow$	$a[s \circ t]$
(VarCons)	$\underline{1}[a \cdot s]$	$\longrightarrow$	a
(Id)	$a[id]$	$\longrightarrow$	a
(Assoc)	$(s \circ t) \circ u$	$\longrightarrow$	$s \circ (t \circ u)$
(Map)	$(a \cdot s) \circ t$	$\longrightarrow$	$a[t] \cdot (s \circ t)$
(IdL)	$id \circ s$	$\longrightarrow$	s
(IdR)	$s \circ id$	$\longrightarrow$	s
(ShiftCons)	$\uparrow \circ (a \cdot s)$	$\longrightarrow$	s
(VarShift)	$\underline{1} \cdot \uparrow$	$\longrightarrow$	id
(SCons)	$\underline{1}[s] \cdot (\uparrow \circ s)$	$\longrightarrow$	s
(Eta)	$\lambda.a\ \underline{1}$	$\longrightarrow$	b if $a =_\sigma b[\uparrow]$

Table 1
The $\lambda\sigma$-rewriting system with η-conversion

represents the infinite substitution $\underline{2} \cdot \underline{3} \cdot \ldots$. The notation $\uparrow^n$ is a shorthand for the composition $\underbrace{\uparrow \circ (\uparrow \circ \ldots \circ \uparrow)}_{n \text{ times}}$. Although the $\lambda\sigma$-calculus codifies the de Bruijn index $\underline{n}$ as $\underline{1}[\uparrow^{n-1}]$, for the sake of clarity, we will follow (DHK00) in not adopting such a codification.

The notion of normal form for $\lambda\sigma$-expressions is given in what follows:

Proposition 29 ($\lambda\sigma$-normal form(Río93)) *Any $\lambda\sigma$-term in* normal form *is of one of the following forms:*

(1) $\lambda_A.a$, where a is in normal form.
(2) $a\ b_1 \ldots b_q$, where a and b_i are in normal form and a is either $\underline{1}$, $\underline{1}[\uparrow^n]$, X or $X[s]$ where s is a substitution in nf and different from id.
(3) $a_1 \cdot \ldots \cdot a_p \cdot \uparrow^n$, where $a_1, \ldots, a_p$ are $\lambda\sigma$-terms in nf and $a_p \neq \underline{n}$.

$\lambda\sigma$-terms in η-lnf are given in what follows:

Definition 30 (η-lnf (DHK00)) *Let a be a $\lambda\sigma$-term of type $A_1 \to \ldots \to A_n \to B$ in context Γ and in $\lambda\sigma$-nf. The η-*lnf *of a, written as a', is given by:*

(1) If $a = \lambda_A.b$ then $a' = \lambda_A.b'$.

(2) If $a = \underline{k}\ b_1 \dots b_q$ then $a' = \lambda_{A_1} \dots \lambda_{A_n}.\underline{k+n}\ c_1 \dots c_q\ \underline{n}' \dots \underline{1}'$, where c_i is the η-lnf of the normal form of $b_i[\uparrow^n]$.

(3) If $a = X[s]\ b_1 \dots b_q$ then $a' = \lambda_{A_1} \dots \lambda_{A_n}.X[s']\ c_1 \dots c_q\ \underline{n}' \dots \underline{1}'$, where c_i is the η-lnf of the normal form of $b_i[\uparrow^n]$ and if $s = d_1 \cdot \dots \cdot d_r \cdot \uparrow^k$ then $s' = e_1 \cdot \dots \cdot e_r \cdot \uparrow^{k+n}$ where e_i is the η-lnf of the nf of $d_i[\uparrow^n]$.

Remark 31 *Definition 30 is shown to be well founded in (DHK00), from where we should note that "in the $\lambda\sigma$-calculus, the reduction of an η-redex may create a σ-redex. For instance, the term $X[\lambda_A.(\underline{2}\ \underline{1})\cdot \uparrow]$ reduces to $X[\underline{1}\cdot \uparrow]$ then to $X[id]$ then to X. Thus to compute the η-lnf we need to reduce all the redexes (including the η ones) before expanding the term."*

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

A unification problem in the $\lambda\sigma$-calculus is written as a disjunction of existentially quantified conjunctions of the form $\bigvee_{j\in J} \exists \overrightarrow{w}_j \bigwedge_{i\in I_j} a_i^j =^?_{\lambda\sigma} b_i^j$ where a_i^j and b_i^j are $\lambda\sigma$-terms of the same type and well typed in the same context Γ_i^j. In order to follow the typing discipline given by the *(meta)* rule of the $\lambda\sigma$-typing system (see Definition 28), for each meta-variable all its occurrences in the unification problem must be typed with the same type in the same context. If $|J| = 1$ then the unification problem is called a *unification system.*

In the unification method over the $\lambda\sigma$-calculus, the solutions are given by the solved forms which are defined as follows:

Definition 32 ($\lambda\sigma$-solved form(DHK00)) *A unification system P is in* $\lambda\sigma$-solved form *if it is a conjunction of nontrivial equations of the following forms:*

- **Solved***: $X =^?_{\lambda\sigma} a$, where the meta-variable X does not appear anywhere else in P and a is in η-lnf. Such an equation is said to be* solved *in P and the variable X is also said to be solved.*
- **Flexible-flexible***: $X[a_1 \cdot \dots \cdot a_p \cdot \uparrow^n] =^?_{\lambda\sigma} Y[b_1 \cdot \dots \cdot b_q \cdot \uparrow^m]$, where $X[a_1 \cdot \dots \cdot a_p \cdot \uparrow^n]$ and $Y[b_1 \cdot \dots \cdot b_q \cdot \uparrow^m]$ are in η-lnf and the equation is not solved.*

Since we are interested in relating a unification algorithm in the simply typed λ-calculus and one in the $\lambda\sigma$-calculus, it is important to know how to translate unification problems from one language to the other. In Example 26 we explained how a unification problem from the λ-calculus with names can be converted to de Bruijn's notation; and the conversion from de Bruijn'notation to the language of the simply typed $\lambda\sigma$-calculus is done by the *precooking* translation defined as follows:

Definition 33 (Precooking (DHK00)) *The* precooking *translates terms in $\Lambda_{dB}(\mathcal{X})$ to $\Lambda_{\lambda\sigma}(\mathcal{X})$ converts a terms $a \in \Lambda_{dB}(\mathcal{X})$ such that $\Gamma \vdash a : A$ in the $\lambda\sigma$-term $a_F = f(a,0)$ where $f(a,n)$, for all $n \geq 0$, is given by:*

(a) $f((\lambda_B.a), n) = \lambda_B(f(a, n+1))$ $\qquad$ (b) $f(\underline{k}, n) = \underline{1}[\uparrow^{k-1}]$

(c) $f(a\ b, n) = f(a, n)\ f(b, n)$ $\qquad$ (d) $f(X, n) = X[\uparrow^n]$

In addition, to all occurrences of each meta-variable X of type B in a we associate the same type B and the context Γ in a_F.

Note that the precooking defined above is injective and hence its inverse is well defined. This remark will be important when unification solutions in the language of the $\lambda\sigma$-calculus need to be translated back to the language of the simply typed λ-calculus. Of course, some $\lambda\sigma$-terms cannot be translated back to the language of the simply typed λ-calculus by the inverse of the precooking translation. In addition, the precooking is a type preserving function as stated by the next proposition:

Proposition 34 (Context and type preservation(DHK00))
If $\Gamma \vdash a : A$ in $\Lambda_{dB}(\mathcal{X})$, then $\Gamma \vdash a_F : A$ in the $\Lambda_{\lambda\sigma}(\mathcal{X})$.

The next example shows how the unification problem presented in Example 26 is converted to the $\lambda\sigma$-calculus according to the precooking translation.

Example 35 *Consider the unification problem $P_{\Lambda_{dB}}$ presented in Example 26. Applying the precooking translation to the terms of $P_{\Lambda_{dB}}$, we get the unification problem $\lambda_A.X[\uparrow]\ \underline{1} =^? \lambda_A.\underline{3}\ \wedge X\ \underline{1} =^? \underline{3}\ \underline{1}$ which is well typed in context $\Gamma = A \cdot B \cdot A \rightarrow B \cdot nil$. Observe that contexts remain unchanged and the sole difference between $P_{\Lambda_{dB}}$ and its precooking translation is that the occurrence of the meta-variable X in the first equation now appears as $X[\uparrow]$ meaning that it is in the scope of one abstractor.*

Now we are ready to point out the fundamental importance of the precooking translation and how it deals with the differences of the (*meta*) rules in the λ-calculus and in the $\lambda\sigma$-calculus.

Remark 36 *In this remark, we want to emphasise an important difference between the* (*meta*) *rule of the λ-calculus in de Bruijn's notation and the one of the $\lambda\sigma$-calculus. In the λ-calculus in de Bruijn's notation the types of meta-variables are independent from the contexts. In fact, we can type the same meta-variable in different levels of abstraction: for instance, for a given context Γ, the λ-term $X\ \lambda_A.(X\ \lambda_A.\underline{1})$ is well typed in Γ, where X is a meta-variable* *of type $(A \to A) \to A$. The type of this term can be deduced as follows:*

$$\dfrac{\boxtimes \qquad \dfrac{\dfrac{\boxdot \qquad \dfrac{\dfrac{}{A \cdot A \cdot (A \to A) \to A \cdot nil \vdash \underline{1} : A}\,(var)}{A \cdot (A \to A) \to A \cdot nil \vdash \lambda_A.\underline{1} : A \to A}\,(lambda)}{A \cdot (A \to A) \to A \cdot nil \vdash (X\ \lambda_A.\underline{1}) : A}\,(app)}{(A \to A) \to A \cdot nil \vdash \lambda_A.(X\ \lambda_A.\underline{1}) : A \to A}\,(lambda)}{(A \to A) \to A \cdot nil \vdash X\ \lambda_A.(X\ \lambda_A.\underline{1}) : A}\,(app)$$

where $\boxdot$ corresponds to:

$$\dfrac{}{A \cdot (A \to A) \to A \cdot nil \vdash X : (A \to A) \to A}\,(meta)$$

and $\boxtimes$ corresponds to:

$$\dfrac{}{(A \to A) \to A \cdot nil \vdash X : (A \to A) \to A}\,(meta)$$

Nevertheless, seen as a $\lambda\sigma$-term, $X\ \lambda_A.(X\ \lambda_A.\underline{1})$ is not well typed; although it is well-formed! In fact, in the type derivation above we need to use $(meta)$ twice for the meta-variable X under different contexts which is allowed in the λ-calculus but not in the $\lambda\sigma$-calculus. In the $\lambda\sigma$-calculus each meta-variable has a unique context and hence, we cannot type the same meta-variable at different levels of abstraction. This means that in the $\lambda\sigma$-calculus the type of meta-variables depends on the context. This seems to be a severe restriction but this is necessary because in the $\lambda\sigma$-calculus one uses grafting instead of substitution and, for instance, the application of the grafting $\{X \mapsto \underline{1}\}$ to the $\lambda\sigma$-term $X\ \lambda_A.(X\ \lambda_A.\underline{1})$ leads to $\underline{1}\ \lambda_A.(\underline{1}\ \lambda_A.\underline{1})$ which is not correct due to variable capture. The precooking translation is the key idea to solve this problem. In fact, the term that corresponds to $X\ \lambda_A.(X\ \lambda_A.\underline{1})$ in the $\lambda\sigma$-calculus is its precooked version given by $X\ \lambda_A.(X[\uparrow]\ \lambda_A.\underline{1})$ and which is well typed in the $\lambda\sigma$-calculus:

$$\dfrac{\boxtimes \qquad \dfrac{\dfrac{\boxdot \qquad \dfrac{\dfrac{}{A \cdot A \cdot (A \to A) \to A \cdot nil \vdash \underline{1} : A}\,(var)}{A \cdot (A \to A) \to A \cdot nil \vdash \lambda_A.\underline{1} : A \to A}\,(lambda)}{A \cdot (A \to A) \to A \cdot nil \vdash (X[\uparrow]\ \lambda_A.\underline{1}) : A}\,(app)}{(A \to A) \to A \cdot nil \vdash \lambda_A.(X[\uparrow]\ \lambda_A.\underline{1}) : A \to A}\,(lambda)}{(A \to A) \to A \cdot nil \vdash X\ \lambda_A.(X[\uparrow]\ \lambda_A.\underline{1}) : A}\,(app)$$

where $\boxtimes$ corresponds to:

$$\dfrac{}{(A \to A) \to A \cdot nil \vdash X : (A \to A) \to A}\,(meta)$$

and $\boxdot$ *corresponds to:*

$$\dfrac{\boxplus \qquad \dfrac{}{(A \to A) \to A \cdot nil \vdash X : (A \to A) \to A}\,(meta)}{A \cdot (A \to A) \to A \cdot nil \vdash X[\uparrow] : (A \to A) \to A}\,(clos)$$

where $\boxplus$ *corresponds to:*

$$\dfrac{}{A \cdot (A \to A) \to A \cdot nil \vdash\uparrow \rhd (A \to A) \to A \cdot nil}\,(shift)$$

This example shows that the precooking translation performs the adequate adjustments to $\lambda\sigma$-terms which allow the use of grafting instead of substitution.

The unification rules for the $\lambda\sigma$-calculus are given in Table 2 which is taken from (DHK00). This set of rules is called **Unif** and is assumed to be applied in a "fair" way: this means that applications of **Exp-**λ (the rule that introduces fresh meta-variables with simpler types) are always followed by applications of **Replace** to avoid infinite applications of **Exp-**λ. In fact, since an application of **Exp-**λ adds a new flexible-flexible equation and does not change anything else in the current problem, it could be applied *ad infinitum*.

A *derivation tree* is a tree that represents an application of the $\lambda\sigma$-HOU method. Formally, it is a tree with a unification system labelling its nodes. Moreover, the arcs that link the nodes are labelled with the unification rules presented in Table 2. Disjunctions of unification systems are represented as "or" branches as usual in tree descriptions. Figure 5 gives an example of a derivation tree.

Example 37 *Consider again the context $\Gamma = A \to B \cdot A \cdot A \to A \cdot nil$ and the unification problem $\lambda_{B\to B}.\underline{1}(X\ \underline{3}) =^? \lambda_{B\to B}.\underline{1}(\underline{2}(\underline{4}\ \underline{3}))$. A unification tree for this problem is given in Figure 4. Applying the precooking, we get:*

$$\lambda_{B\to B}.\underline{1}(X[\uparrow]\ \underline{3}) =^?_{\lambda\sigma} \lambda_{B\to B}.\underline{1}(\underline{2}(\underline{4}\ \underline{3})) \qquad (7)$$

which is well typed in context Γ. A derivation tree for this system is presented in Figures 5, 6, 7, 9 and 8. Note that this derivation tree and the unification tree of Figure 4 have a similar structure: both have exactly one fail node and two success nodes. The solutions to equation (7) are given by the graftings $\{X \mapsto \lambda_A.(\underline{2}(\underline{4}\ \underline{1}))\}$ and $\{X \mapsto \lambda_A.(\underline{2}(\underline{4}\ \underline{3}))\}$ that correspond, respectively, to the substitutions $\{X/\lambda_A.\underline{2}(\underline{4}\ \underline{1})\}$ and $\{X/\lambda_A.\underline{2}(\underline{4}\ \underline{3})\}$ given in Example 27.

Dec-λ	$\dfrac{P \wedge \lambda_A.e_1 =^?_{\lambda\sigma} \lambda_A.e_2}{P \wedge e_1 =^?_{\lambda\sigma} e_2}$
Dec-App	$\dfrac{P \wedge (\underline{n}\ e^1_1 \dots e^1_p) =^?_{\lambda\sigma} (\underline{n}\ e^2_1 \dots e^2_p)}{P \wedge e^1_1 =^?_{\lambda\sigma} e^2_1 \wedge \dots \wedge e^1_p =^?_{\lambda\sigma} e^2_p}$
Dec-Fail	$\dfrac{P \wedge (\underline{n}\ e^1_1 \dots e^1_{p_1}) =^?_{\lambda\sigma} (\underline{m}\ e^2_1 \dots e^2_{p_2})}{Fail}$, if $m \neq n$.
Exp-λ	$\dfrac{P}{\exists(A \cdot \Gamma \vdash Y : B), P \wedge X =^?_{\lambda\sigma} \lambda_A.Y}$ if $(\Gamma \vdash X : A \to B) \in \mathcal{TV}ar(P)$, $Y \notin \mathcal{TV}ar(P)$, and X is not a solved variable.
Exp-App	$\dfrac{P \wedge X[a_1 \cdot \ldots \cdot a_p \cdot \uparrow^n] =^?_{\lambda\sigma} (\underline{m}\ b_1 \ldots b_q)}{P \wedge X[a_1 \cdot \ldots \cdot a_p \cdot \uparrow^n] =^?_{\lambda\sigma} (\underline{m}\ b_1 \ldots b_q) \wedge \bigvee_{r \in R_p \cup R_i} \exists H_1 \ldots \exists H_k : X =^?_{\lambda\sigma} (\underline{r}\ H_1 \ldots H_k)}$ if X has an atomic type and is not solved; where $H_1, \dots, H_k$ are fresh variables of appropriate types, not occurring in P, with the contexts $\Gamma_{H_i} = \Gamma_X$, R_p is the subset of $\{1, \dots, p\}$ such that $(\underline{r}\ H_1 \dots H_k)$ has the right type, $R_i =$if $m \geq n + 1$ then $\{m - n + p\}$ else $\emptyset$.
Normalise	$\dfrac{P \wedge e_1 =^?_{\lambda\sigma} e_2}{P \wedge e'_1 =^?_{\lambda\sigma} e'_2}$ if e_1 or e_2 is not in η-lnf. where e'_1 (resp. e'_2) is the η-lnf of e_1 (resp. e_2) if e_1 (resp. e_2) is not a solved variable and e_1 (resp. e_2) otherwise.
Replace	$\dfrac{P \wedge X =^?_{\lambda\sigma} t}{\{X \mapsto t\}(P) \wedge X =^?_{\lambda\sigma} t}$ if $X \in \mathcal{TV}ar(P), X \notin \mathcal{TV}ar(t)$ and if t is a meta-variable then $t \in \mathcal{TV}ar(P)$.

Table 2
Unification Rules for the $\lambda\sigma$-calculus (DHK00)

4.3 A Structural Relation Between HOU in the λ-calculus and in the $\lambda\sigma$-calculus

Here we establish a relation between HOU in the λ-calculus and in the $\lambda\sigma$-calculus that refines a result established by Dowek, Hardin and Kirchner in (DHK00). We start with the definition of the *pseudo-precooking* that extends the usual notion of precooking by combining it with some unification rules.

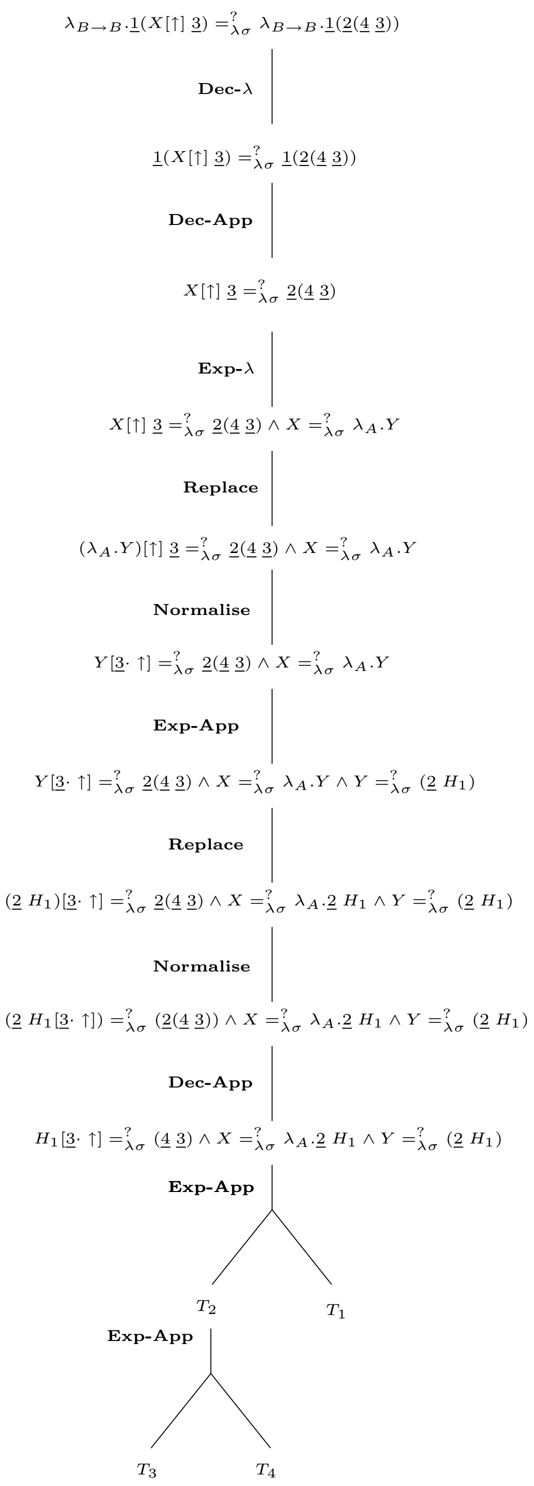

$\lambda_{B\to B}.\underline{1}(X[\uparrow]\ \underline{3}) =^?_{\lambda\sigma} \lambda_{B\to B}.\underline{1}(\underline{2}(\underline{4}\ \underline{3}))$
Dec-λ
$\underline{1}(X[\uparrow]\ \underline{3}) =^?_{\lambda\sigma} \underline{1}(\underline{2}(\underline{4}\ \underline{3}))$
Dec-App
$X[\uparrow]\ \underline{3} =^?_{\lambda\sigma} \underline{2}(\underline{4}\ \underline{3})$
Exp-λ
$X[\uparrow]\ \underline{3} =^?_{\lambda\sigma} \underline{2}(\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.Y$
Replace
$(\lambda_A.Y)[\uparrow]\ \underline{3} =^?_{\lambda\sigma} \underline{2}(\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.Y$
Normalise
$Y[\underline{3}\cdot\uparrow] =^?_{\lambda\sigma} \underline{2}(\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.Y$
Exp-App
$Y[\underline{3}\cdot\uparrow] =^?_{\lambda\sigma} \underline{2}(\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.Y \wedge Y =^?_{\lambda\sigma} (\underline{2}\ H_1)$
Replace
$(\underline{2}\ H_1)[\underline{3}\cdot\uparrow] =^?_{\lambda\sigma} \underline{2}(\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ H_1 \wedge Y =^?_{\lambda\sigma} (\underline{2}\ H_1)$
Normalise
$(\underline{2}\ H_1[\underline{3}\cdot\uparrow]) =^?_{\lambda\sigma} (\underline{2}(\underline{4}\ \underline{3})) \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ H_1 \wedge Y =^?_{\lambda\sigma} (\underline{2}\ H_1)$
Dec-App
$H_1[\underline{3}\cdot\uparrow] =^?_{\lambda\sigma} (\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ H_1 \wedge Y =^?_{\lambda\sigma} (\underline{2}\ H_1)$
Exp-App
T_2
T_1
Exp-App
T_3
T_4

Figure 5. Example 37: Derivation Tree of $\lambda_{B\to B}.\underline{1}(X\ \underline{3}) =^? \lambda_{B\to B}.\underline{1}(\underline{2}(\underline{4}\ \underline{3}))$

Definition 38 (Pseudo-Precooking) *The* pseudo-precooking *translates any term a such that $\Gamma \vdash a : A$, from $\Lambda_{dB}(\mathcal{X})$ to $\Lambda_{\lambda\sigma}(\mathcal{X})$ into the term $\overline{a} = p(a, 0)$, where $p(a', n)$ is recursively defined for any sub-term a' of a within the scope of $n \geq 0$ abstractors in a by:*

$$H_1[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} (\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ H_1 \wedge Y =^?_{\lambda\sigma} (\underline{2}\ H_1) \wedge H_1 =^?_{\lambda\sigma} \underline{1}$$

Replace

$$\underline{1}[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} (\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ \underline{1} \wedge Y =^?_{\lambda\sigma} (\underline{2}\ \underline{1}) \wedge H_1 =^?_{\lambda\sigma} \underline{1}$$

Normalise

$$\underline{3} =^?_{\lambda\sigma} (\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ \underline{1} \wedge Y =^?_{\lambda\sigma} (\underline{2}\ \underline{1}) \wedge H_1 =^?_{\lambda\sigma} \underline{1}$$

Dec-Fail

Fail

Figure 6. Example 37: subtree T_1 of Fig. 5

$$H_1[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} (\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ H_1 \wedge Y =^?_{\lambda\sigma} (\underline{2}\ H_1) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ H_2)$$

Replace

$$(\underline{4}\ H_2)[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} (\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (\underline{4}\ H_2) \wedge Y =^?_{\lambda\sigma} (\underline{2}\ (\underline{4}\ H_2)) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ H_2)$$

Normalise

$$(\underline{4}\ H_2[\underline{3}\cdot \uparrow]) =^?_{\lambda\sigma} (\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (\underline{4}\ H_2) \wedge Y =^?_{\lambda\sigma} (\underline{2}\ (\underline{4}\ H_2)) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ H_2)$$

Dec-App

$$H_2[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} \underline{3} \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (\underline{4}\ H_2) \wedge Y =^?_{\lambda\sigma} (\underline{2}\ (\underline{4}\ H_2)) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ H_2)$$

Figure 7. Example 37: subtree T_2 of Fig. 5

- *If* $a' = \lambda_{A_1} \dots \lambda_{A_m}.b$ *then* $p(\lambda_{A_1} \dots \lambda_{A_m}.b, n) = \lambda_{A_1} \dots \lambda_{A_m}.p(b, n+m)$*;*
- *If* $a' = (\underline{k}\ a_1 \dots a_m)$ *then* $p(\underline{k}\ a_1 \dots a_m, n) = \underline{1}[\uparrow^{k-1}]\ p(a_1, n) \dots p(a_m, n)$*;*
- *If* $a' = (X\ a_1 \dots a_m)$ *then, supposing that* $B_n \cdot \dots \cdot B_1 \cdot \Gamma \vdash a' : A'$*, we have:*
  - *if* $m \geq 1$*, then* $p(a', n) = Y[p(a_m, n) \cdot \dots \cdot p(a_1, n) \cdot \uparrow^n]$*, where for all* $1 \leq i \leq m$, $B_n \cdot \dots \cdot B_1 \cdot \Gamma \vdash a_i : A_i$ *and* Y *is a fresh meta-variable with type* A' *and context* $A_m \cdot \dots \cdot A_1 \cdot \Gamma$*.*
  - *if* $m = 0$*, then* $p(a', n) = X[\uparrow^n]$*, and the new type judgement for* X *is* $\Gamma \vdash X : A'$*.*

The pseudo-precooking is a function that takes a well typed λ-term a in β-normal form and returns a well typed $\lambda\sigma$-term $\overline{a}$. Intuitively, $\overline{a}$ can be obtained from a_F after normalisation w.r.t. the rules **Exp-λ**, **Replace** and **Normalise** applied to the unification equation that contains a_F. This intuition is formalised by Lemma 41.

$$H_2[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} \underline{3} \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (\underline{4}\ H_2) \wedge Y =^?_{\lambda\sigma} (\underline{2}\ (\underline{4}\ H_2)) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ H_2) \wedge H_2 =^?_{\lambda\sigma} \underline{3}$$

Replace

$$\underline{3}[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} \underline{3} \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (\underline{4}\ \underline{3}) \wedge Y =^?_{\lambda\sigma} (\underline{2}\ (\underline{4}\ \underline{3})) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ \underline{3}) \wedge H_2 =^?_{\lambda\sigma} \underline{3}$$

Normalise

$$X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (\underline{4}\ \underline{3}) \wedge Y =^?_{\lambda\sigma} (\underline{2}\ (\underline{4}\ \underline{3})) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ \underline{3}) \wedge H_2 =^?_{\lambda\sigma} \underline{3}$$

Success

Figure 8. The subtree T_4 of Fig. 5.

$$H_2[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} \underline{3} \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (\underline{4}\ H_2) \wedge Y =^?_{\lambda\sigma} (\underline{2}\ (\underline{4}\ H_2)) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ H_2) \wedge H_2 =^?_{\lambda\sigma} \underline{1}$$

Replace

$$\underline{1}[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} \underline{3} \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (\underline{4}\ \underline{1}) \wedge Y =^?_{\lambda\sigma} (\underline{2}\ (\underline{4}\ \underline{1})) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ \underline{1}) \wedge H_2 =^?_{\lambda\sigma} \underline{1}$$

Normalise

$$X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (\underline{4}\ \underline{1}) \wedge Y =^?_{\lambda\sigma} (\underline{2}\ (\underline{4}\ \underline{1})) \wedge H_1 =^?_{\lambda\sigma} (\underline{4}\ \underline{1}) \wedge H_2 =^?_{\lambda\sigma} \underline{1}$$

Success

Figure 9. Example 37: subtree T_3 of Fig. 5

Example 39 *In the unification tree of Fig. 4, take the derived problem $\overline{P}_\epsilon$:*

$$\lambda_{B\to B}.X\ \underline{3} =^? \lambda_{B\to B}.\underline{2}(\underline{4}\ \underline{3}) \tag{8}$$

whose pseudo-precooking translation is given by $\lambda_{B\to B}.Z[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} \lambda_{B\to B}.\underline{2}(\underline{4}\ \underline{3})$ which can be found in Fig. 5 after the first application of **Normalise** *up to the renaming of meta-variables and without the external abstractors. In fact, external abstractors are usually removed by applications of* **Dec-λ** *at the beginning of the derivation.*

This pseudo-precooking translation can be obtained from the precooking trans-

Anti-Exp-λ	$$\frac{P}{\exists Y(P \wedge X =^?_{\lambda\sigma} (Y[\uparrow]\ \underline{1}))}$$ if $X \in Var(P)$ such that $\Gamma_X = A \cdot \Gamma'_X$ where $Y \in \mathcal{X}, Y \notin Var(P)$ and $T_y = A \to T_X, \Gamma_Y = \Gamma'_X$
Anti-Dec-λ	$$\frac{P \wedge a =^?_{\lambda\sigma} b}{P \wedge \lambda_A.a =^?_{\lambda\sigma} \lambda_A.b}$$ if $a =^?_{\lambda\sigma} b$ is well typed in context $\Delta = A \cdot \Delta'$.

Table 3
Back

lation of equation (8) as follows:

$$\cfrac{\cfrac{\cfrac{\lambda_{B\to B}.X[\uparrow]\ \underline{3} =^?_{\lambda\sigma} \lambda_{B\to B}.\underline{2}(\underline{4}\ \underline{3})}{\lambda_{B\to B}.X[\uparrow]\ \underline{3} =^?_{\lambda\sigma} \lambda_{B\to B}.\underline{2}(\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.Z}\ \textbf{Exp-}\lambda}{\lambda_{B\to B}.(\lambda_A.Z)[\uparrow]\ \underline{3} =^?_{\lambda\sigma} \underline{2}(\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_{B\to B}.\lambda_A.Z}\ \textbf{Replace}}{\lambda_{B\to B}.Z[\underline{3}\cdot \uparrow] =^?_{\lambda\sigma} \lambda_{B\to B}.\underline{2}(\underline{4}\ \underline{3}) \wedge X =^?_{\lambda\sigma} \lambda_A.Z}\ \textbf{Normalise}$$

Notice that applications of **Exp-λ** introduce new equations, but these new equations are ignored by the pseudo-precooking because we are interested only in the structure of particular equations obtained in the $\lambda\sigma$-calculus; the information “lost” from these equations is, in some sense, stored in the context of the equation and can be recovered after the application of the strategy **Back** defined in Table 3 (cf. (DHK00)). The strategy **Back** in a certain way undoes the work done by the rules **Exp-λ** and **Dec-λ**.

The next proposition shows that the pseudo-precooking translation preserves the types and contexts of the terms.

Proposition 40 *Let $B_1, \ldots, B_n$ ($n \geq 0$) be types, Δ a context and a a λ-term in $\Lambda_{dB}(\mathcal{X})$ which is well typed in Δ. If a' is a sub-term of a such that $B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash a' : A$ then $B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash p(a', n) : A$.*

PROOF. The proof is by induction on the structure of a':

- If a' is a de Bruijn index or an application the result is straightforward.
- If $a' = \lambda_B.b$ is a term of type $B \to C$ then assume that $B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash \lambda_B.b : B \to C$, and hence $B \cdot B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash b : C$. By the induction hypothesis (IH) we have that $B \cdot B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash p(b, n+1) : C$. After one application of (*lambda*) we get that $B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash \lambda_B.p(b, n+1) : B \to C$

which is equivalent to $B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash p(\lambda_B.b, n) : B \to C$.

- If $a' = (X\ a_1 \ldots a_m)$, where X is a meta-variable of type $A_1 \to \ldots \to A_m \to A$ then assume that $B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash (X\ a_1 \ldots a_m) : A$. By IH we have that, for all $1 \leq i \leq m$: $B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash p(a_i, n) : A_i$. Let Y be a fresh meta-variable of type A and context $A_m \cdot \ldots \cdot A_1 \cdot \Delta$. Consider the derivation:

$$\dfrac{\boxdot \qquad \dfrac{}{A_m \cdot \ldots \cdot A_1 \cdot \Delta \vdash Y : A}\,(meta)}{B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash Y[p(a_m, n) \cdot \ldots \cdot p(a_1, n) \cdot \uparrow^n] : A}\,(clos)$$

where $\boxdot$ corresponds to:

$$\dfrac{\begin{matrix} & \dfrac{\dfrac{}{B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash p(a_1, n) : A_1}\,(IH) \quad \dfrac{}{B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash \uparrow^n \rhd \Delta}\,(shift)}{B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash p(a_1, n) \cdot \uparrow^n \rhd A_1 \cdot \Delta}\,(cons) \\ & \overline{\hspace{20em}}\,(cons) \\ \vdots & \vdots \end{matrix}}{B_n \cdot \ldots \cdot B_1 \cdot \Delta \vdash p(a_m, n) \cdot \ldots \cdot p(a_1, n) \cdot \uparrow^n \rhd A_m \cdot \ldots \cdot A_1 \cdot \Delta}\,(clos)$$

□

The following lemma formalises a relation between the pseudo-precooking and the precooking translation that will be important for the stepwise comparison presented afterwards.

Lemma 41 *Let P be a unification problem in the simply typed λ-calculus, P_F its precooking translation and $\overline{P}$ its pseudo-precooking translation. Then the normalisation of P_F w.r.t. the rules* **Exp-λ**, **Replace** *and* **Normalise** *results in $\overline{P}$ up to renaming of meta-variables. Conversely, the normalisation of $\overline{P}$ w.r.t. the rules* **Anti-Exp-λ**, **Replace** *and* **Normalise** *results in P_F up to renaming of meta-variables.*

PROOF. As usual we assume that the terms in P are in η-lnf. If P contains only meta-variables of atomic type then the result follows vacuously. Suppose X is a meta-variable of type $A_1 \to \ldots \to A_m \to A$ (A atomic and $m \geq 1$) that occurs in P. Since terms are assumed to be in η-lnf, we have that all occurrences of X in P are in sub-terms of the form:

$$(X\ a_1 \ldots a_m). \tag{9}$$

The precooking translation of the sub-term (9) is given by:

$$(X[\uparrow^n]\, f(a_1, n) \ldots f(a_m, n)) \tag{10}$$

for some $n \geq 0$ that represents the number of abstractors binding X.

After m applications of the strategy **Exp-λ** and **Replace** followed by an application of **Normalise** to P_F, the sub-term (10) assumes the form:

$$Y[f(a_m, n)' \cdot \ldots \cdot f(a_1, n)' \cdot \uparrow^n] \tag{11}$$

where Y is a fresh meta-variable with the type of the term (10) and the sub-terms $f(a_i, n)'$ $(1 \leq i \leq m)$ are recursively obtained from $f(a_i, n)$ by replacing all its sub-term of the form (10) by (11). Repeating this process for each meta-variable of functional type that occurs in P we get a new unification problem normalised w.r.t. the rules **Exp-λ**, **Replace** and **Normalise** and where every sub-term of the form (10) was replaced by a sub-term of the form (11); the resulting unification problem corresponds, from the definition of the pseudo-precooking translation, to $\overline{P}$ up to renaming of meta-variables.

Conversely, the pseudo-precooking translation of the sub-term (9) is given by:

$$Y[p(a_m, n) \cdot \ldots \cdot p(a_1, n) \cdot \uparrow^n] \tag{12}$$

where $n \geq 0$ and Y is a fresh meta-variable of type A and context $A_m \cdot \ldots \cdot A_1 \cdot \Delta$, for some Δ. After m applications of the strategy **Anti-Exp-λ** and **Replace** followed by an application of **Normalise** the sub-term (12) assumes the form:

$$Z[\uparrow^n]\, p(a_1, n)' \ldots p(a_m, n)' \tag{13}$$

where Z is a fresh meta-variable of type $A_1 \to \ldots A_m \to A$ and context Δ; the sub-terms $p(a_i, n)'$ $(1 \leq i \leq m)'$ are obtained from $p(a_i, n)$ by replacing all its sub-terms of the form (12) by (13). Repeating this process for each meta-variable of functional type that occurs in P we get a new unification problem normalised w.r.t. the rules **Anti-Exp-λ**, **Replace** and **Normalise** and where every sub-term of the form (12) is replaced by a sub-term of the form (13); the resulting unification problem corresponds, from the definition of the precooking translation, to P_F up to renaming of meta-variables. $\square$

The next lemma shows formally how unification problems in the λ-calculus are related to unification systems in the $\lambda\sigma$-calculus.

Lemma 42 *Let Γ be a context, P be a unification problem in $\Lambda_{dB}(\mathcal{X})$ which is well typed in Γ and $\mathcal{A}(P)$ a unification tree of P. For each derived problem P_α occurring in $\mathcal{A}(P)$, there exists a unification system P^* derived from P_F*

via **Unif** *such that, for each equation in P_α of the form:*

$$\lambda_{A_1} \dots \lambda_{A_n}.h_1\ e^1_1 \dots e^1_{p_1} =^? \lambda_{A_1} \dots \lambda_{A_n}.h_2\ e^2_1 \dots e^2_{p_2} \tag{14}$$

well typed in context Γ, where $n, p_1, p_2 \geq 0$ and h_1 and h_2 are either a de Bruijn index or a meta-variable, there is an equation in P^* of one of the following forms:

- *if h_1 is a meta-variable and h_2 is a de Bruijn index:*

$$Y[p(e^1_{p_1}, n) \cdot \dots \cdot p(e^1_1, n) \cdot \uparrow^n] =^?_{\lambda\sigma} h_2\ p(e^2_1, n) \dots p(e^2_{p_2}, n) \tag{15}$$

where Y is a fresh meta-variable of atomic type.

- *if h_1 and h_2 are de Bruijn indexes:*

$$h_1\ p(e^1_1, n) \dots p(e^1_{p_1}, n) =^?_{\lambda\sigma} h_2\ p(e^2_1, n) \dots p(e^2_{p_2}, n) \tag{16}$$

- *if h_1 and h_2 are meta-variables:*

$$Y[p(e^1_{p_1}, n) \cdot \dots \cdot p(e^1_1, n) \cdot \uparrow^n] =^?_{\lambda\sigma} Z[p(e^2_{p_2}, n) \cdot \dots \cdot p(e^2_1, n) \cdot \uparrow^n] \tag{17}$$

where Y and Z are meta-variables of atomic type.

The equations (16), (15) and (17) are well typed in context $A_n \cdot \dots \cdot A_1 \cdot \Gamma$.

In addition, all the equations introduced by applications of $\lambda\sigma$-unification rules become solved after an application of **Replace**.

PROOF. The proof is by induction on the length of the derivation that generates P_α. If $\alpha = \epsilon$ then for each equation eq in P, eq_F is in P_F and, if P_ϵ contains only flexible-flexible equations, we take P^* to be the unification system obtained from P_F by normalisation w.r.t. the rules **Exp-λ**, **Replace** and **Normalise**. If P_ϵ contains flexible-rigid or rigid-rigid equations then we consider each case separately:

• P_ϵ contains a flexible-rigid equation: In this case, P_ϵ contains equation (14) in which $h_1 = X$ and h_2 is a de Bruijn index. Then P_F contains the equation:

$$\lambda_{A_1} \dots \lambda_{A_n}.X[\uparrow^n]\ f(e^1_1, n) \dots f(e^1_{p_1}, n) =^?_{\lambda\sigma} \lambda_{A_1} \dots \lambda_{A_n}.h_2\ f(e^2_1, n) \dots f(e^2_{p_2}, n) \tag{18}$$

According to Lemma 41, after normalising the equation (18) w.r.t the rules **Exp-λ**, **Replace** and **Normalise** we get:

$$\lambda_{A_1} \dots \lambda_{A_n}.Y[p(e^1_{p_1}, n) \cdot \dots \cdot p(e^1_1, n) \cdot \uparrow^n] =^?_{\lambda\sigma} \lambda_{A_1} \dots \lambda_{A_n}.h_2\ p(e^2_1, n) \dots p(e^2_{p_2}, n)$$

and equation (15) is obtained after n applications of the rule **Dec-λ**.

• P_ϵ contains a rigid-rigid equation: In this case, P_ϵ contains equation (14) in which h_1 and h_2 are de Bruijn indexes. Then P_F contains the equation:

$$\lambda_{A_1} \ldots \lambda_{A_n}.h_1\ f(e^1_1, n) \ldots f(e^1_{p_1}, n) =^?_{\lambda\sigma} \lambda_{A_1} \ldots \lambda_{A_n}.h_2\ f(e^2_1, n) \ldots f(e^2_{p_2}, n) \tag{19}$$

From Lemma 41 the normalisation of equation (19) w.r.t. the rules **Exp-**λ, **Replace** and **Normalise** generates the following equation:

$$\lambda_{A_1} \ldots \lambda_{A_n}.h_1\ p(e^1_1, n) \ldots p(e^1_{p_1}, n) =^?_{\lambda\sigma} \lambda_{A_1} \ldots \lambda_{A_n}.h_2\ p(e^2_1, n) \ldots p(e^2_{p_2}, n)$$

and equation (16) is obtained after n applications of the rule **Dec-**λ. Applications of **Exp-**λ introduce new equations of the form $X =^?_{\lambda\sigma} \lambda_A.Y$. After an application of **Replace** every occurrence of X in the current unification system will be replaced by $\lambda_A.Y$ and the equation $X =^?_{\lambda\sigma} \lambda_A.Y$ becomes solved. In this way, all introduced equations get solved after a replacement.

For the induction step, let P_α be a unification problem in $\mathcal{A}(P)$ with $\alpha \neq \epsilon$ and let P^* be the unification system obtained from P_F according to this lemma; we need to find a unification system, say P^{**}, derived from P^* that satisfies this lemma for a problem derived from P_α in one step. We consider the two possible steps separately:

• *The unification problem derived from P_α is obtained after an application of SIMPL:*

In this case, the unification problem derived from P_α is $\overline{P_\alpha}$ according to the definition of unification trees. In order to apply the procedure SIMPL to P_α, P_α should contain (at least) one rigid-rigid equation of the form:

$$\lambda_{B_1} \ldots \lambda_{B_m}.\underline{k}\ f^1_1 \ldots f^1_p =^? \lambda_{B_1} \ldots \lambda_{B_m}.\underline{k}\ f^2_1 \ldots f^2_p \tag{20}$$

well typed in context Γ and where $m \geq 0$ and $p > 0$. After the decomposition, equation (20) is replaced by a conjunction of the form:

$$\lambda_{B_1} \ldots \lambda_{B_m}.f^1_1 =^? \lambda_{B_1} \ldots \lambda_{B_m}.f^2_1 \wedge \ldots \wedge \lambda_{B_1} \ldots \lambda_{B_m}.f^1_p =^? \lambda_{B_1} \ldots \lambda_{B_m}.f^2_p$$

whose equations are well typed in context Γ and all the other equations remain unchanged. If there exist rigid-rigid equations among the equations in this conjunction (or others that were already in P_α) the decomposition step is recursively applied to them. This way, this process finishes with a unification problem $\overline{P_\alpha}$ containing only flexible-rigid and/or flexible-flexible equations.

For the equations generated during the application of SIMPL the unification system P^{**} is built as follows:

- Suppose $\overline{P_\alpha}$ contains a flexible-flexible equation. If this flexible-flexible equation was already in P_α then we are done. If some new equation generated by the decomposition of equation (20) is rigid-rigid then the argument that follows can be applied recursively to these new equations. Therefore, without loss of generality we assume that the new flexible-flexible equation is given by:

$$\lambda_{B_1} \dots \lambda_{B_m}.f_1^1 =^? \lambda_{B_1} \dots \lambda_{B_m}.f_1^2$$

well typed in context Γ. By hypothesis, there exists a derivation of P_F that generates the problem P^* that contains the equation:

$$\underline{k}\ p(f_1^1, m) \dots p(f_p^1, m) =^?_{\lambda\sigma} \underline{k}\ p(f_1^2, m) \dots p(f_p^2, m) \tag{21}$$

which is well typed in context $B_m \cdot \dots \cdot B_1 \cdot \Gamma$. The decomposition of equation (21) is done by an application of the rule **Dec-App** that generates the unification system P^{**} which contains the flexible-flexible equation $p(f_1^1, m) =^?_{\lambda\sigma} p(f_1^2, m)$ that is well typed in context $B_m \cdot \dots \cdot B_1 \cdot \Gamma$.

- Suppose $\overline{P_\alpha}$ contains a flexible-rigid equation. If this flexible-rigid equation was already in P_α then we are done. If some equation generated by the decomposition of equation (20) is rigid-rigid then the argument that follows can be applied recursively to these equations. Therefore without loss of generality we suppose that the flexible-rigid equation in $\overline{P_\alpha}$ is given by:

$$\lambda_{B_1} \dots \lambda_{B_m}.f_p^1 =^? \lambda_{B_1} \dots \lambda_{B_m}.f_p^2 \tag{22}$$

which is well typed in context Γ. Moreover, we assume that:

$$\begin{cases} f_p^1 = \lambda_{C_1} \dots \lambda_{C_w}.H_1\ g_1^1 \dots g_{q_1}^1 \\ f_p^2 = \lambda_{C_1} \dots \lambda_{C_w}.\underline{h}\ g_1^2 \dots g_{q_2}^2 \end{cases} \tag{23}$$

where H_1 is a meta-variable and g_j^i $(1 \le i \le 2; 1 \le j \le q_i)$ are terms all well typed in context $B_m \cdot \dots \cdot B_1 \cdot \Gamma$.

The equation (22) can be written as:

$$\lambda_{A_1} \dots \lambda_{A_n}.H_1\ g_1^1 \dots g_{q_1}^1 =^? \lambda_{A_1} \dots \lambda_{A_n}.\underline{h}\ g_1^2 \dots g_{q_2}^2$$

where $A_i = \begin{cases} B_i & \text{, for } 1 \le i \le m; \\ C_{i-m} & \text{, for } m < i \le n (= m + w). \end{cases}$

By hypothesis, there exists a derivation from P_F that generates a unification system P^* containing the rigid-rigid equation (21), and after an application of the rule **Dec-App**, we get a unification system containing the equation:

$$p(f_p^1, m) =^?_{\lambda\sigma} p(f_p^2, m). \tag{24}$$

From the definition of the pseudo-precooking translation and from the assumption (23), we conclude that equation (24) has the form:
$\lambda_{C_1} \dots \lambda_{C_w}.Y[p(g_{q_1}^1, m+w) \cdot \dots \cdot p(g_1^1, m+w) \cdot \uparrow^{m+w}] =^?_{\lambda\sigma}$
$\lambda_{C_1} \dots \lambda_{C_w}.\underline{h}\ p(g_1^2, m+w) \dots p(g_{q_2}^2, m+w)$ and after w applications of **Dec-λ** we get the unification system P^{**} that contains the desired equation well

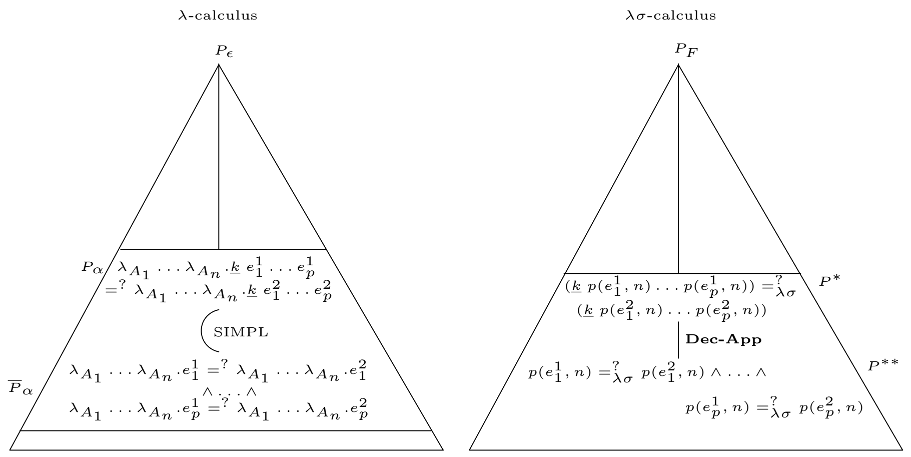

λ-calculus
λσ-calculus
P_ϵ
P_F
P_α
$\lambda_{A_1} \dots \lambda_{A_n} . \underline{k}\ e^1_1 \dots e^1_p =^? \lambda_{A_1} \dots \lambda_{A_n} . \underline{k}\ e^2_1 \dots e^2_p$
SIMPL
$\overline{P}_\alpha$
$\lambda_{A_1} \dots \lambda_{A_n} . e^1_1 =^? \lambda_{A_1} \dots \lambda_{A_n} . e^2_1 \wedge \dots \wedge \lambda_{A_1} \dots \lambda_{A_n} . e^1_p =^? \lambda_{A_1} \dots \lambda_{A_n} . e^2_p$
$(\underline{k}\ p(e^1_1, n) \dots p(e^1_p, n)) =^?_{\lambda\sigma} (\underline{k}\ p(e^2_1, n) \dots p(e^2_p, n))$
P^*
Dec-App
$p(e^1_1, n) =^?_{\lambda\sigma} p(e^2_1, n) \wedge \dots \wedge p(e^1_p, n) =^?_{\lambda\sigma} p(e^2_p, n)$
P^{**}

Figure 10. A Simplification Step

typed in context $C_w \cdot \ldots \cdot C_1 \cdot B_m \cdot \ldots \cdot B_1 \cdot \Gamma$ (see Fig. 10). Notice that during the simplification step in the $\lambda\sigma$-calculus no new equation is introduced.

• *The unification problem derived from P_α is obtained after an application of MATCH:*

Let $P_{\alpha r}$ ($r > 0$) be a unification problem generated after this application of MATCH. The problem $P_{\alpha r}$ must contain at least one equation of the form flexible-rigid or rigid-rigid (that may be trivial) because after an imitation step a rigid-rigid equation is generated and, after a projection the generated equation is either flexible-rigid or rigid-rigid.

Assume that P_α contains (at least) one flexible-rigid equation of the form:

$$\lambda_{A_1} \dots \lambda_{A_n} . X\ e^1_1 \dots e^1_{p_1} =^? \lambda_{A_1} \dots \lambda_{A_n} . \underline{h}\ e^2_1 \dots e^2_{p_2} \tag{25}$$

well typed in context Γ and where:

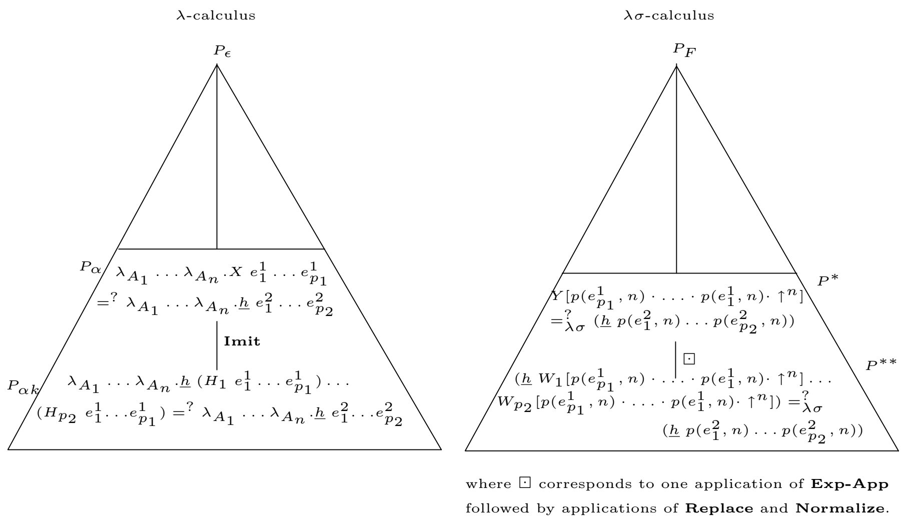
- $n, p_1, p_2 \geq 0$;
- X is a meta-variable with type $B_1 \to \dots \to B_{p_1} \to A$ with A atomic;
- $\underline{h}$ is a de Bruijn index with type $C_1 \to \dots \to C_{p_2} \to A$ with A atomic;
- If $p_1 > 0$ then e^1_i are λ-terms in η-lnf with type B_i for all $1 \leq i \leq p_1$;
- If $p_2 > 0$ then e^2_j are λ-terms in η-lnf with type C_j, for all $1 \leq j \leq p_2$.

We consider the imitation and projection substitutions separately:

(a) Imitation: An imitation is possible only if the head of the rigid term is a constant, i.e., when $h > n$. In this case, the imitation substitution is given by:

$$X / \lambda_{B_1} \dots \lambda_{B_{p_1}} . \underline{h - n + p_1}\ (H_1\ \underline{p_1} \dots \underline{1}) \dots (H_{p_2}\ \underline{p_1} \dots \underline{1})$$

where the H_i's are fresh meta-variables with types $B_1 \to \dots \to B_{p_1} \to C_i$ for all $1 \leq i \leq p_2$. After an application of this substitution to equation (25) we



λ-calculus
$\lambda\sigma$-calculus
P_ϵ
P_F
P_α
$\lambda_{A_1} \dots \lambda_{A_n}.X\ e^1_1 \dots e^1_{p_1}$
$=^? \lambda_{A_1} \dots \lambda_{A_n}.\underline{h}\ e^2_1 \dots e^2_{p_2}$
Imit
$P_{\alpha k}$
$\lambda_{A_1} \dots \lambda_{A_n}.\underline{h}\ (H_1\ e^1_1 \dots e^1_{p_1}) \dots$
$(H_{p_2}\ e^1_1 \dots e^1_{p_1}) =^? \lambda_{A_1} \dots \lambda_{A_n}.\underline{h}\ e^2_1 \dots e^2_{p_2}$
P^*
$Y[p(e^1_{p_1}, n) \cdot \dots \cdot p(e^1_1, n) \cdot \uparrow^n]$
$=^?_{\lambda\sigma} (\underline{h}\ p(e^2_1, n) \dots p(e^2_{p_2}, n))$
$\boxdot$
P^{**}
$(\underline{h}\ W_1[p(e^1_{p_1}, n) \cdot \dots \cdot p(e^1_1, n) \cdot \uparrow^n] \dots$
$W_{p_2}[p(e^1_{p_1}, n) \cdot \dots \cdot p(e^1_1, n) \cdot \uparrow^n]) =^?_{\lambda\sigma}$
$(\underline{h}\ p(e^2_1, n) \dots p(e^2_{p_2}, n))$
where $\boxdot$ corresponds to one application of **Exp-App**
followed by applications of **Replace** and **Normalize**.

Figure 11. The imitation step

get the unification problem $P_{\alpha r}$ that contains the following equation:

$$\lambda_{A_1} \dots \lambda_{A_n}.\underline{h}\ (H_1\ e^1_1 \dots e^1_{p_1}) \dots (H_{p_2}\ e^1_1 \dots e^1_{p_1}) =^? \lambda_{A_1} \dots \lambda_{A_n}.\underline{h}\ e^2_1 \dots e^2_{p_2}$$

well typed in context Γ. Notice that in $P_{\alpha r}$ all occurrences of X were replaced by $\lambda_{B_1} \dots \lambda_{B_{p_1}}.\underline{h - n + p_1}\ (H_1\ \underline{p_1} \dots \underline{1}) \dots (H_{p_2}\ \underline{p_1} \dots \underline{1})$.

By hypothesis, there exists a derivation from P_F that generates the system P^* that contains the equation:

$$Y[p(e^1_{p_1}, n) \cdot \dots \cdot p(e^1_1, n) \cdot \uparrow^n] =^?_{\lambda\sigma} \underline{h}\ p(e^2_1, n) \dots p(e^2_{p_2}, n) \qquad (26)$$

which is well typed in context $A_n \cdot \dots \cdot A_1 \cdot \Gamma$. Since $h > n$, after an application of **Exp-App** an equation of the following form is generated:

$$Y =^?_{\lambda\sigma} \underline{h - n + p_1}\ W_1 \dots W_{p_2}$$

where the W_j's are fresh meta-variables with type C_j, for all $1 \leq j \leq p_2$.

After an application of **Replace** and then **Normalise** to the current system, we get a new system containing the equation:
$\underline{h}\ W_1[p(e^1_{p_1}, n) \cdot \dots \cdot p(e^1_1, n) \cdot \uparrow^n] \dots W_{p_2}[p(e^1_{p_1}, n) \cdot \dots \cdot p(e^1_1, n) \cdot \uparrow^n] =^?_{\lambda\sigma}$
$\underline{h}\ p(e^2_1, n) \dots p(e^2_{p_2}, n)$ which is well typed in the context $A_n \cdot \dots \cdot A_1 \cdot \Gamma$. Notice that all occurrences of Y were replaced by the term $\underline{h - n + p_1}\ W_1 \dots W_{p_2}$ and hence the equation $Y =^?_{\lambda\sigma} \underline{h - n + p_1}\ W_1 \dots W_{p_2}$ introduced by the application of **Exp-App** is solved in the current system which we take to be P^{**}. The general scheme is shown in Fig. 11. There exists only one case when an equation is eliminated during this process: if the de Bruijn index $\underline{h}$ has an atomic type (because in this case no meta-variable is introduced and a trivial equation is generated). But if this is the case, then a trivial equation is also generated from equation (26) and the lemma holds.

(b) Projection: In this case, the head X of the flexible term is projected over each of its arguments whose target type is equal to the target type of X. Suppose, without loss of generality, that X is projected over its l-th argument $(1 \leq l \leq p_1)$, i.e., suppose that e^1_l has type of the form $D_1 \rightarrow \ldots \rightarrow D_q \rightarrow A$ $(q \geq 0)$. The projection of X over its l-th argument is given by:

$$X/\lambda_{B_1} \ldots \lambda_{B_{p_1}}.\underline{p_1 - l + 1}\ (H_1\ \underline{p_1} \ldots \underline{1}) \ldots (H_q\ \underline{p_1} \ldots \underline{1})$$

where the H_i's are fresh meta-variables of type $B_1 \rightarrow \ldots \rightarrow B_{p_1} \rightarrow D_i$ for all $1 \leq i \leq q$. After an application of this substitution, we get a problem that contains the following equation:

$$\lambda_{A_1} \ldots \lambda_{A_n}.e^1_l\ (H_1\ e^1_1 \ldots e^1_{p_1}) \ldots (H_q\ e^1_1 \ldots e^1_{p_1}) =^? \lambda_{A_1} \ldots \lambda_{A_n}.\underline{h}\ e^2_1 \ldots e^2_{p_2} \quad (27)$$

which is well typed in context Γ (see Fig. 12). We consider 2 sub-cases:
(b.1) The head of the term e^1_l is a de Bruijn index: Since e^1_l is in η-lnf, we may assume without loss of generality, that e^1_l is of the form $\lambda_{D_1} \ldots \lambda_{D_q}.\underline{k}\ f_1 \ldots f_s$ $(s \geq 0)$. After a normalisation step we get the unification problem $P_{\alpha r}$ that contains one of the following equations according to the value of k:

• $k > q$: $\quad \lambda_{A_1} \ldots \lambda_{A_n}.\underline{k - q}\ f^1_1 \ldots f^1_s =^? \lambda_{A_1} \ldots \lambda_{A_n}.\underline{h}\ e^2_1 \ldots e^2_{p_2}$

• $k \leq q$: $\quad \lambda_{A_1} \ldots \lambda_{A_n}.H_{q-k+1}\ e^1_1 \ldots e^1_{p_1}\ f^1_1 \ldots f^1_s =^? \lambda_{A_1} \ldots \lambda_{A_n}.\underline{h}\ e^2_1 \ldots e^2_{p_2}$
both equations well typed in context Γ and where, for all $1 \leq i \leq s$, f^1_i is obtained from f_i after replacing all free occurrences of $\underline{1}, \ldots, \underline{q}$, respectively by $(H_q\ e^1_1 \ldots e^1_{p_1})$, $\ldots$, $(H_1\ e^1_1 \ldots e^1_{p_1})$.

By hypothesis, there exists a derivation of P_F that generates a unification system P^* containing the equation (26). The precooking translation preserves types and, hence the target type of $p(e^1_l, n)$ coincides with the type of Y and, an application of **Exp-App** generates an equation of the form:

$$Y =^?_{\lambda\sigma} (\underline{p_1 - l + 1}\ W_1 \ldots W_q)$$

where the W_i's are fresh meta-variables of type D_i for all $1 \leq i \leq q$. After an application of **Replace** and **Normalise**, we get a system containing the equation:

$$p(e^1_l, n)\ W_1[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] \ldots W_q[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] =^?_{\lambda\sigma}$$
$$\underline{h}\ p(e^2_1, n) \ldots p(e^2_{p_2}, n) \quad (28)$$

well typed in context $A_n \cdot \ldots \cdot A_1 \cdot \Gamma$ and the introduced equation $Y =^?_{\lambda\sigma} (\underline{p_1 - l + 1}\ W_1 \ldots W_q)$ becomes solved. Since $p(e^1_l, n) = \lambda_{D_1} \ldots \lambda_{D_q}.\underline{k}\ p(f_1, n + q) \ldots p(f_s, n+q)$, the left-hand side of equation (28) reduces as follows according to the value of k:

• $k > q$:
$(\lambda_{D_1} \ldots \lambda_{D_q}.\underline{k}\ p(f_1, n + q) \ldots p(f_s, n + q))\ W_1[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] \ldots$

$W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \rightarrow^*_{\lambda\sigma} \underline{k-q}\, p(f_1,n+q)[W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot id] \ldots p(f_s,n+q)[W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot id]$.

The sub-terms $p(f_j,n+q)[W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot id]$ $(1 \leq j \leq s)$ are interpreted as follows: during the normalisation all the free occurrences of de Bruijn indexes $\underline{1}$, ..., $\underline{q}$ are respectively replaced by $W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n]$, ..., $W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n]$, and the meta-variables of f_j were initially in the scope of $n+q$ abstractors but now in the scope of only n of them because q of these abstractors were removed (by applications of $(Beta)$) to generate the substitution $[W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot id]$. This fact can be expressed through a simple example as follows: if X is a meta-variable that is in the scope of $n+q$ abstractors then it appears as $X[\uparrow^{n+q}]$ and when it is applied to a substitution containing q terms, i.e., a substitution of the form $[a_1 \cdot \ldots \cdot a_q \cdot id]$ then we get $X[\uparrow^{n+q}][a_1 \cdot \ldots \cdot a_q \cdot id] \rightarrow^*_\sigma X[\uparrow^n]$ which means that the q abstractors that were originally binding X were removed to generate the substitution $[a_1 \cdot \ldots \cdot a_q \cdot id]$ and hence X is now in the scope of only n abstractors. In this way the terms $p(f^1_j,n)$ $(1 \leq j \leq s)$ correspond to $p(f_j,n+q)[W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot id]$ and we get the desired equation:
$\underline{k-q}\, p(f^1_1,n) \ldots p(f^1_s,n) =^?_{\lambda\sigma} \underline{h}\, p(e^2_1,n) \ldots p(e^2_{p_2},n)$.

• $k \leq q$:

$(\lambda_{D_1} \ldots \lambda_{D_q}.\underline{k}\, p(f_1,n+q) \ldots p(f_s,n+q))\; W_1[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \ldots W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \rightarrow^*_{\lambda\sigma}$
$W_{q-k+1}[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n]\; p(f_1,n+q)[W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot id] \ldots p(f_s,n+q)[W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot id]$.

As in the previous case, for all $1 \leq j \leq s$, the sub-term:

$$p(f_j,n+q)[W_q[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n] \cdot id]$$

reduces to $p(f^1_j,n)$ and we get the equation:

$$W_{q-k+1}[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n]\; p(f^1_1,n) \ldots p(f^1_s,n) =^?_{\lambda\sigma} \underline{h}\, p(e^2_1,n) \ldots p(e^2_{p_2},n).$$

Without loss of generality suppose the type of the meta-variable W_{q-k+1} is given by $F_1 \rightarrow \ldots \rightarrow F_s \rightarrow A$. An application of the rule **Exp-**λ to the current unification system generates an equation of the form $W_{q-k+1} =^?_{\lambda\sigma} \lambda_{F_1}.X_1$, where X_1 is a fresh meta-variable of type $F_2 \rightarrow \ldots \rightarrow F_s \rightarrow A$. An application of the rule **Replace** generates the equation:

$$(\lambda_{F_1}.X_1)[p(e^1_{p_1},n) \cdot \ldots \cdot p(e^1_1,n)\cdot \uparrow^n]\; p(f^1_1,n) \ldots p(f^1_s,n) =^?_{\lambda\sigma} \underline{h}\, p(e^2_1,n) \ldots p(e^2_{p_2},n)$$

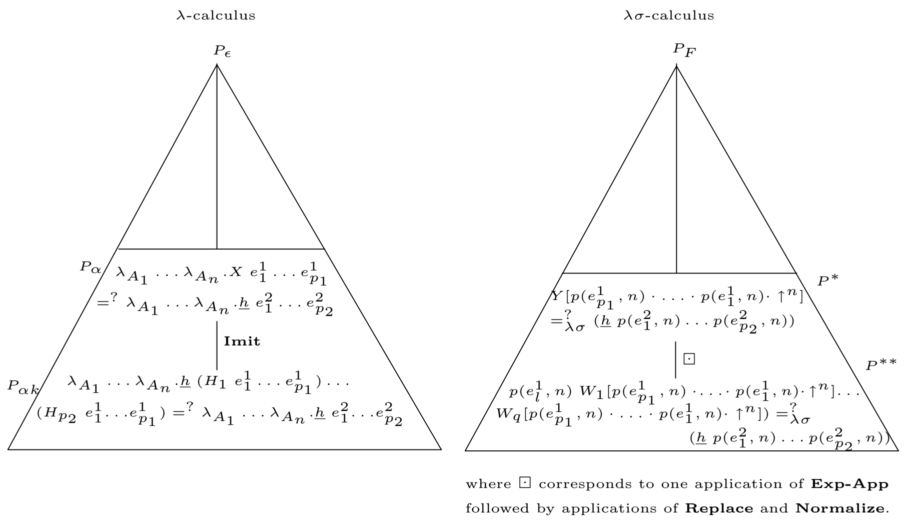

λ-calculus
$\lambda\sigma$-calculus
P_ϵ
P_F
P_α
$\lambda_{A_1} \ldots \lambda_{A_n}.X\ e^1_1 \ldots e^1_{p_1}$
$=^? \lambda_{A_1} \ldots \lambda_{A_n}.\underline{h}\ e^2_1 \ldots e^2_{p_2}$
Imit
$P_{\alpha k}$
$\lambda_{A_1} \ldots \lambda_{A_n}.\underline{h}\ (H_1\ e^1_1 \ldots e^1_{p_1}) \ldots$
$(H_{p_2}\ e^1_1 \ldots e^1_{p_1}) =^? \lambda_{A_1} \ldots \lambda_{A_n}.\underline{h}\ e^2_1 \ldots e^2_{p_2}$
P^*
$Y[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n]$
$=^?_{\lambda\sigma} (\underline{h}\ p(e^2_1, n) \ldots p(e^2_{p_2}, n))$
$\boxdot$
P^{**}
$p(e^1_l, n)\ W_1[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] \ldots$
$W_q[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n]) =^?_{\lambda\sigma}$
$(\underline{h}\ p(e^2_1, n) \ldots p(e^2_{p_2}, n))$
where $\boxdot$ corresponds to one application of **Exp-App**
followed by applications of **Replace** and **Normalize**.

Figure 12. The projection step

which can be normalised to

$X_1[p(f^1_1, n) \cdot p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n]\ p(f^1_2, n) \ldots p(f^1_s, n) =^?_{\lambda\sigma}$

$\underline{h}\ p(e^2_1, n) \ldots p(e^2_{p_2}, n)$.

Repeating the strategy **Exp-λ**, **Replace** and **Normalise** $p-1$ times, we get the desired equation:

$$W[p(f^1_s, n) \cdot \ldots \cdot p(f^1_1, n) \cdot p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] =^?_{\lambda\sigma} \underline{h}\ p(e^2_1, n) \ldots p(e^2_{p_2}, n)$$

where W is a fresh meta-variable of type A.

(b.2) The head of the term e^1_l is a meta-variable: In this case e^1_l is of the form $\lambda_{D_1} \ldots \lambda_{D_q}.Z\ f_1 \ldots f_s$ and normalising equation (27) we get the unification problem $P_{\alpha k}$ which contains the equation:

$$\lambda_{A_1} \ldots \lambda_{A_n}.Z\ f^1_1 \ldots f^1_s =^? \lambda_{A_1} \ldots \lambda_{A_n}.\underline{h}\ e^2_1 \ldots e^2_{p_2}$$

well typed in context Γ and, as in the previous case f^1_j is obtained from f_j by replacing all occurrences of $\underline{1}$, …, $\underline{q}$, respectively by the terms $(H_q\ e^1_1 \ldots e^1_{p_1})$, …, $(H_1\ e^1_1 \ldots e^1_{p_1})$. By hypothesis, there exists a unification system P^*, derived from P_F and containing equation (26) and after applications of **Exp-App**, **Replace** and **Normalise** we get a new system which contains equation (28). Since $p(e^1_l, n) = \lambda_{D_1} \ldots \lambda_{D_q}.Z[\uparrow^{n+q}]\ p(f_1, n+q) \ldots p(f_s, n+q)$, we have that the left hand side of the equation (28) assumes the form:

$(\lambda_{D_1} \ldots \lambda_{D_q}.Z[\uparrow^{n+q}]\ p(f_1, n+q) \ldots p(f_s, n+q))W_1[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] \ldots W_q[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] \rightarrow^*_{\lambda\sigma}$

$Z[\uparrow^n]\ p(f_1, n+q)[W_q[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] \cdot id] \ldots p(f_s, n+q)[W_q[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] \cdot id]$. As in case $(b.1)$, for all $1 \leq j \leq s$, the sub-term $p(f_j, n+q)[W_q[p(e^1_{p_1}, n) \cdot$

$\ldots \cdot p(e_1^1, n) \cdot \uparrow^n] \cdot \ldots \cdot W_1[p(e_{p_1}^1, n) \cdot \ldots \cdot p(e_1^1, n) \cdot \uparrow^n] \cdot id]$ reduces to $p(f_j^1, n)$ and we get the equation:

$$Z[\uparrow^n]\; p(f_1^1, n) \ldots p(f_s^1, n) =^?_{\lambda\sigma} \underline{h}\; p(e_1^2, n) \ldots p(e_{p_2}^2, n)$$

which after being normalised w.r.t. the rules **Exp-λ**, **Replace** and **Normalise** assumes the desired form:

$$U[p(f_s^1, n) \cdot \ldots \cdot p(f_1^1, n) \cdot \uparrow^n] =^?_{\lambda\sigma} \underline{h}\; p(e_1^2, n) \ldots p(e_{p_2}^2, n)$$

which is well typed in context $A_n \cdot \ldots \cdot A_1 \cdot \Gamma$ and, where U is a fresh meta-variable of type A. □

In Lemma 42, we established a relation between the structure of the equations of the subgoals generated during the unification process in the simply typed λ-calculus in de Bruijn's notation and the precooked translation of these subgoals (or derived problems) in the simply typed $\lambda\sigma$-calculus. This lemma is the key point for relating the solutions and the subtrees generated during the unification process. In fact, the next proposition shows that if $\mathcal{A}(P)$ is a unification tree of a given unification problem P, then for each sub-tree of $\mathcal{A}(P)$ there exists a sub-tree of the derivation tree of P_F with the same number of success and fail nodes. Moreover, the precooked version of derived problems of P can be obtained as derived problems of P_F.

Proposition 43 *Let P be a unification problem in $\Lambda_{dB}(\mathcal{X})$ which is well typed in a context Γ and, $\mathcal{A}(P)$ a unification tree of P. For each problem P_α in $\mathcal{A}(P)$, there exists a unification system P^*, derived from P_F using* **Unif**, *such that:*

(1) if P_α contains a branch that leads to a success node, then there exists a derivation of P^* that leads to a solved form;
(2) if P_α contains a branch that leads to a fail node, then there exists a derivation of P^* that leads to a fail node;
(3) if P_α is formed by the equations $eq_1, \ldots, eq_s$ that are well typed in context Γ, then there exists a unification system P_B^*, derived from P^* using the strategy **Back**, *which contains the equations $eq_{1_F}, \ldots, eq_{s_F}$ well typed in context Γ, up to renaming of meta-variables. Moreover, any other equation in P_B^* is either flexible-flexible or solved.*

PROOF.

(1) Suppose P_α contains a branch with a success node $P_{\alpha\gamma}$ (which contains only flexible-flexible equations). From Lemma 42 there exists a unification system P^* derived from P_F that contains only flexible-flexible and solved equations and hence P^* is a success node.

(2) Suppose that P_α contains a branch that leads to a fail node $P_{\alpha\gamma}$. There are two possible cases: either $P_{\alpha\gamma}$ contains a rigid-rigid equation with different heads (fail with SIMPL) or it contains a flexible-rigid equation in which no imitation or projection is possible (fail with MATCH). In the former case, from Lemma 42 there exists a unification system P^* derived from P_F that contains a rigid-rigid equation with different heads and hence P^* is a fail node. In the later case, there exists a unification system P^* derived from P_F that contains a flexible-rigid equation such that the application of **Exp-App** does not generate new equations because the pseudo-precooking preserves types (cf Proposition 40). Therefore, P^* is a fail node in this case as well.

(3) Suppose that $P_\alpha = eq_1 \wedge \ldots \wedge eq_s$ $(s > 0)$. It is enough to prove that, for an arbitrary equation eq_j $(1 \leq j \leq s)$ of P_α, we can obtain eq_{j_F} from the unification system P^* given by Lemma 42 via the strategy **Back**. In fact, the strategy **Back** does not propagate changes to other equations because it does not involve substitution. The proof is divided according to the structure of the equation eq_j:

- eq_j *is a flexible-rigid equation*: In this case, eq_j has the form:

$$\lambda_{A_1} \ldots \lambda_{A_n}.X\ e^1_1 \ldots e^1_{p_1} =^? \lambda_{A_1} \ldots \lambda_{A_n}.\underline{h}\ e^2_1 \ldots e^2_{p_2} \tag{29}$$

which is well typed in context Γ, where $n, p_1, p_2 \geq 0$, X is a meta-variable of type $B_1 \to \ldots \to B_{p_1} \to A$ (A is atomic). By Lemma 42, there exists a unification system P^* derived from P_F which contains an equation of the form:

$$Y[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] =^?_{\lambda\sigma} \underline{h}\ p(e^2_1, n) \ldots p(e^2_{p_2}, n) \tag{30}$$

which is well typed in context $A_n \cdot \ldots \cdot A_1 \cdot \Gamma$, where Y is a meta-variable of type A. The context of Y is given by $B_{p_1} \cdot \ldots \cdot B_1 \cdot \Gamma$. Applying Lemma 41 to the terms of equation (30) we get a new equation of the form:

$$W[\uparrow^n]\ f(e^1_1, n) \ldots f(e^1_{p_1}, n) =^?_{\lambda\sigma} \underline{h}\ f(e^2_1, n) \ldots f(e^2_{p_2}, n) \tag{31}$$

where W is a meta-variable of type $B_1 \to \ldots \to B_{p_1} \to A$ and context $\Gamma_{>n}$, by normalisation w.r.t. the rules **Anti-Exp-λ**, **Replace** and **Normalise**. The equation (31) is well typed in context $A_n \cdot \ldots \cdot A_1 \cdot \Gamma$ and hence, after n application of the rule **Anti-Dec-λ** we get eq_{j_F} up to renaming of meta-variables.

- eq_j *is a rigid-rigid equation*: In this case, eq_j has the form:

$$\lambda_{A_1} \ldots \lambda_{A_n}.\underline{k}\ e^1_1 \ldots e^1_{p_1} =^? \lambda_{A_1} \ldots \lambda_{A_n}.\underline{h}\ e^2_1 \ldots e^2_{p_2}$$

which is well typed in context Γ. By Lemma 42, there exists a unification system P^* derived from P_F which contains an equation of the form:

$$\underline{k}\ p(e^1_1, n) \ldots p(e^1_{p_1}, n) =^? \underline{h}\ p(e^2_1, n) \ldots p(e^2_{p_2}, n) \tag{32}$$

which is well typed in context $A_n \cdot \ldots \cdot A_1 \cdot \Gamma$. Applying Lemma 41 to the terms of equation (32) we get a unification system which contains an equation of the form:

$$\underline{k}\ f(e^1_1, n) \ldots f(e^1_{p_1}, n) =^? \underline{h}\ f(e^2_1, n) \ldots f(e^2_{p_2}, n)$$

which is well typed in context $A_n \cdot \ldots \cdot A_1 \cdot \Gamma$ and after n applications of the rule **Anti-Dec-λ** we get the desired equation eq_{j_F} up to renaming of meta-variables.

- *eq_j is a flexible-flexible equation*: In this case, eq_j has the form:

$$\lambda_{A_1} \ldots \lambda_{A_n}.X\ e^1_1 \ldots e^1_{p_1} =^? \lambda_{A_1} \ldots \lambda_{A_n}.Y\ e^2_1 \ldots e^2_{p_2}$$

which is well typed in context Γ, where X is a meta-variable of type $B_1 \to \ldots \to B_{p_1} \to A$ (A is atomic). According to Lemma 42, there exists a unification system P^* derived from P_F which contains the equation:

$$Z[p(e^1_{p_1}, n) \cdot \ldots \cdot p(e^1_1, n) \cdot \uparrow^n] =^?_{\lambda\sigma} W[p(e^2_{p_2}, n) \cdot \ldots \cdot p(e^2_1, n) \cdot \uparrow^n] \quad (33)$$

which is well typed in context $A_n \cdot \ldots A_1 \cdot \Gamma$, where Z and W are meta-variables of type A. Applying Lemma 41 to the terms of equation (33) we get a unification system which contains an equation of the form:

$$Z[\uparrow^n]\ f(e^1_1, n) \ldots f(e^1_{p_1}, n) =^?_{\lambda\sigma} W[\uparrow^n]\ f(e^2_1, n) \ldots f(e^2_{p_2}, n)$$

which is well typed in context $A_n \cdot \ldots \cdot A_1 \cdot \Gamma$. After n applications of the rule **Anti-Dec-λ** we get the desired equation eq_{j_F} up to renaming of meta-variables.

During the unification process in the $\lambda\sigma$-calculus, new equations are introduced after applications of **Exp-λ**, **Exp-App** or **Anti-Exp-λ**. These equations are of the form $X =^?_{\lambda\sigma} a$, where X is a meta-variable and a is a term without occurrences of X. After an application of **Replace** every occurrence of X in the unification system will be replaced by a and this equation becomes solved. It will remain solved during the whole process because no rule applies to X, although the term a can change. $\square$

The next example illustrates the contents of Proposition 43.

Example 44 *Let P be the unification problem given in Example 27 and a unification tree $\mathcal{A}(P)$ of P given in Fig. 4. Consider, for instance, the subgoal:*

$$\overline{P}_1 = \{\lambda_{B\to B}.X_1\ \underline{3} =^? \lambda_{B\to B}.\underline{4}\ \underline{3}\}$$

which is well typed in context $\Gamma = \{A \to B \cdot A \cdot A \to A \cdot nil\}$ and whose corresponding subtree contains one fail node and two success nodes. The proof *of Lemma 42 is constructive and leads us to the unification system:*

$$P^* = \{H_1[\underline{3} \cdot \uparrow] =^?_{\lambda\sigma} \underline{4}\ \underline{3} \wedge X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ H_1 \wedge Y =^?_{\lambda\sigma} \underline{2}\ H_1\}$$

whose corresponding subtree also contains one fail node and two success nodes (see Fig. 5). Applying the strategy **Back** *to P^* we get the unification system:*

$$P^*_B = \{\lambda_{B\to B}.N[\uparrow]\ \underline{3} =^?_{\lambda\sigma} \lambda_{B\to B}.\underline{4}\ \underline{3}\ \wedge\ X =^?_{\lambda\sigma} \lambda_A.\underline{2}\ (N[\uparrow]\ \underline{1}) \wedge \\ Y =^?_{\lambda\sigma} \underline{2}\ (N[\uparrow]\ \underline{1}) \wedge H_1 =^?_{\lambda\sigma} N[\uparrow]\ \underline{1}\}$$

where the equation:

$$\lambda_{B\to B}.N[\uparrow]\ \underline{3} =^?_{\lambda\sigma} \lambda_{B\to B}.\underline{4}\ \underline{3}$$

corresponds to the precooking translation of the equation:

$$\lambda_{B\to B}.X_1\ \underline{3} =^? \lambda_{B\to B}.\underline{4}\ \underline{3}$$

up to the renaming of meta-variables and all the other equations are solved. The solution σ of $\overline{P}_1$ is given by $\sigma = \{X_1/\lambda_A.\underline{4}\ \underline{1}, X_1/\lambda_A.\underline{4}\ \underline{3}\}$. It is easy to check that the grafting $\sigma_F = \{N \mapsto \lambda_A.\underline{4}\ \underline{1}, N \mapsto \lambda_A.\underline{4}\ \underline{3}\}$ obtained from σ, after renaming X_1 to N, is a solution to P^*_B.

Corollary 45 *Let P be a unification problem in the simply typed λ-calculus and $\mathcal{A}(P)$ a unification tree of P. For each unification problem P_α in $\mathcal{A}(P)$ with solution σ there exists a unification system P^*_B derived from P_F using the strategies* **Unif** *and* **Back** *that has σ_F as solution after an adequate renaming of meta-variables.*

PROOF. Let Γ be a context and P_α be a unification problem in $\mathcal{A}(P)$ which is well typed in context Γ. From Proposition 43, we know that there exists a derivation from P_F using **Unif** and **Back** that generates a unification system P^*_B that contains all the equations in P_{α_F} up to renaming of meta-variables. Moreover, all the other equations in P^*_B are solved and, hence the grafting σ_F, after an adequate renaming of meta-variables, is a solution to P^*_B according to Proposition 3.3 of (DHK00). Figure 13 shows the general scheme that relates unification in the λ-calculus and in the $\lambda\sigma$-calculus. □

By Corollary 45 one concludes that unification in the simply typed $\lambda\sigma$-calculus is a generalisation of Huet's algorithm since every solution for a unification system computed in the $\lambda\sigma$ is also computed by Huet's algorithm.

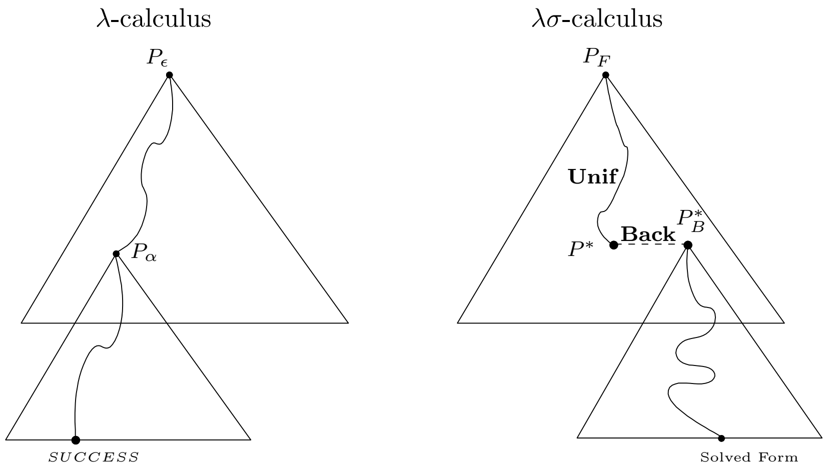


Figure 13. General Unification Scheme in the λ- and $\lambda\sigma$-calculus

5 Conclusions and Future Work

In a stepwise fashion, we compared two different styles of HOU: the classical HOU for the simply typed λ-calculus of Huet (Hue75) and HOU via the simply typed $\lambda\sigma$-calculus (DHK00). The contributions of this paper are:

- We enriched the *matching trees* of Huet's method by introducing a new structural notation called *unification tree*. This notation was essential to provide a precise presentation of the derivations of Huet's algorithm and, constituted an important tool for establishing the structural correspondence between HOU à la Huet and HOU via explicit substitutions.
- Although it is a straightforward translation of Huet's HOU algorithm, we explicitly introduced Huet's HOU algorithm for the simply typed λ-calculus in de Bruijn's notation. This was done in order to simplify the comparison between Huet's HOU algorithm and the $\lambda\sigma$-HOU method, since the latter uses de Bruijn's notation.
- Both the simply typed λ-calculus with names and in de Bruijn's notation include meta-variables. Although the use of meta-variables is not essential for the unification methods treated here, its use simplifies their presentation and allows us to keep a clear difference between substitutions generated by applications of β-reductions and substitutions generated by the unification rules. The difference between typing meta-variables in the λ-calculus and in the $\lambda\sigma$-calculus was emphasised through examples since the unification mechanism in the former uses (higher-order) substitution while the latter uses grafting (first-order substitution).
- Unification derivations in the simply typed $\lambda\sigma$-calculus were presented in a tree structure notation called *derivation tree* which jointly with the unification tree structure permits a better visualisation of the relations between unification derivations in both methods.
- By using these structures, we proved that Huet's HOU and the $\lambda\sigma$-HOU

preserve an important structural relation between (sub-)problems: For a given unification problem P in the simply typed λ-calculus, we have that for each (sub-)problem of P in a unification tree $\mathcal{A}(P)$ of P, there exists a counterpart in a derivation tree of P_F. This allows us to conclude that the $\lambda\sigma$-HOU is a generalisation of Huet's algorithm and that solutions computed by the latter are always computed by the former method.

We believe that this structural comparison is important to provide a better understanding of HOU methods based on explicit substitutions and to shed some light on questions related to practical and implementational issues as well as on the whole of explicit substitutions in higher-order unification.

Natural extensions of this work include considering an optimised implementation of the $\lambda\sigma$-HOU algorithm based on the ideas behind the notion of the pseudo-precooking that in fact combines the precooking with some unification rules. In addition, these ideas can be extended to other styles of explicit substitutions like the λs_e-calculus and the suspension calculus.

Acknowledgements

We would like to thank the careful reading, constructive corrections and detailed suggestions of one of the referees.

References

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

[ARK01] M. Ayala-Rincón and F. Kamareddine. Unification via the λs_e-Style of Explicit Substitution. *The Logical Journal of the Interest Group in Pure and Applied Logics*, 9(4):489–523, 2001.

[ARK03] M. Ayala-Rincón and F. Kamareddine. On Applying the λs_e-Style of Unification for Simply-Typed Higher Order Unification in the Pure lambda Calculus. *Matemática Contemporânea - WoLLIC 2001 selected papers*, 24:1–22, 2003.

[AMK05] M. Ayala-Rincón, F.L.C. de Moura and F. Kamareddine. Comparing and Implementing Calculi of Explicit Substitutions with Eta-Reduction *Annals of Pure and Applied Logic - WoLLIC 2002 selected papers*, 134(1):5–41, 2005.

[Bar84] H. P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics (revised edition).* North Holland, 1984.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and* All That. CUP, 1998.

[dB72] N.G. de Bruijn. Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indag. Mat.*, 34(5):381–392, 1972.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Inf. and Computation*, 157:183–235, 2000.

[Dow01] G. Dowek. Higher-Order Unification and Matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 16, pages 1009–1062. MIT P. & Elsevier, 2001.

[Gol81] W. Goldfarb. The Undecidability of the Second-Order Unification Problem. *TCS*, 13(2):225–230, 1981.

[Hin97] J. R. Hindley. *Basic Simple Type Theory.* Number 42 in Cambridge Tracts in Theoretical Computer Science. CUP, 1997.

[Hue75] G. Huet. A Unification Algorithm for Typed λ-Calculus. *TCS*, 1:27–57, 1975.

[Hue02] G. Huet. Higher Order Unification 30 Years Later. In V. A. Carreño, C. A. Muñoz and S. Tahar editors, *Theorem Proving in Higher Order Logics - TPHOLs 2002*, volume 2410 of *LNCS*, pages 3–12. Springer, 2002.

[LNQ04] C. Liang, G. Nadathur, and X. Qi. Choices in Representation and Reduction Strategies for Lambda Terms in Intesional Contexts. *Journal of Automated Reasoning*, 33(2):89–132, 2004.

[MAK06] F.L.C. de Moura, M. Ayala-Rincón and F. Kamareddine. SUBSEXPL: a Tool for Simulating and Comparing Explicit Substitutions Calculi. Special Issue on Implementation of Logics, *Journal of Applied Non-Classical Logics*, 16(1-2):119-150, 2006.

[Nip91] T. Nipkow. Higher-Order Critical Pairs. *Proc. 6th IEEE Symp. Logic in Computer Science*, 342–349, 1991.

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