International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 209-226
http://rewriting.loria.fr/rta/

AN EFFICIENT NOMINAL UNIFICATION ALGORITHM

JORDI LEVY [1] AND MATEU VILLARET [2]

[1] Artificial Intelligence Research Institute (IIIA),
Spanish Council for Scientific Research (CSIC), Barcelona, Spain.
E-mail address: levy@iiia.csic.es
URL: http://www.iiia.csic.es/~levy

[2] Departament d'Informàtica i Matemàtica Aplicada (IMA),
Universitat de Girona (UdG), Girona, Spain.
E-mail address: villaret@ima.udg.edu
URL: http://ima.udg.edu/~villaret

Abstract. Nominal Unification is an extension of first-order unification where terms can contain binders and unification is performed modulo α-equivalence. Here we prove that the existence of nominal unifiers can be decided in quadratic time. First, we linearly-reduce nominal unification problems to a sequence of freshness and equalities between atoms, modulo a permutation, using ideas as Paterson and Wegman for first-order unification. Second, we prove that solvability of these reduced problems may be checked in quadratic time. Finally, we point out how using ideas of Brown and Tarjan for unbalanced merging, we could solve these reduced problems more efficiently.

1. Introduction

Nominal techniques introduce mechanisms for renaming via name-swapping, for name-binding, and for freshness of names. They were introduced at the beginning of this decade by Gabbay and Pitts [Pit01, Gab01, Pit03]. These first works have inspired a sequel of papers where bindings and freshness are introduced in other topics, like nominal algebra [Gab06, Gab07, Gab09], equational logic [Clo07], rewriting [Fer05, Fer07], unification [Urb03, Urb04], and Prolog [Che04, Urb05].

In this paper we study the complexity of *Nominal Unification* [Urb03, Urb04], an extension of first-order unification where terms can contain binders and unification is performed modulo α-equivalence. Moreover, (first-order) variables (*unknowns*) are allowed to "capture" bound variables (*atoms*) contrarily to unification in λ-calculus. In [Urb03, Urb04] it is described a sound and complete, but inefficient (exponential), algorithm for nominal unification. Later this algorithm was extended to deal with the new-quantifier and locality in [Fer05]. In [Cal07] there is a description of a direct but exponential implementation in

Key words and phrases: Nominal Logic, Unification.

This research has been partially founded by the CICYT research projects Mulog2 (TIN2007-68005-C04) and SuRoS (TIN2008-04547).

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.209

Maude, and a polynomial implementation in OCAML based on termgraphs. In [Cal08], it is described a polynomial algorithm for nominal unification. In [Lev08] it is proved that the problem can be solved in quadratic time by quadratic reduction to Higher-Order Pattern Unification, that is claimed to be linear [Qia96]. Therefore, the present algorithm does not improve the complexity bounds already known. However, it has to be noticed that in this paper we describe a practical implementation, and that it is really difficult to obtain a practical algorithm from the proof described in [Qia96]. In [Cal10] there is a quadratic algorithm for nominal unification, independently found by Calvès, and also based on Paterson and Wegman's first-order unification algorithm. Other extensions of nominal unification have been studied in [Che05, Dow09, Dow10].

This paper proceeds as follows. In Section 2 we describe nominal logic and the nominal unification algorithm of [Urb03, Urb04]. In Section 3 we prove that freshness equations and suspensions are mere syntactic sugar. We can translate them in terms of basic nominal equations, with a linear increasing in the size of the problem. In Section 4 we describe the Paterson-Wegman linear algorithm for First-Order Unification [Pat78] and some preliminary ideas of how we plan to adapt this algorithm to nominal unification. In Section 5, we introduce *replacings* as $L = (a_1 \leftarrow b_1) \cdots (a_n \leftarrow b_n)$. We say that t and u are equivalent modulo L, written $t =_L u$, if $a_n. \cdots a_1.t \approx b_n. \cdots b_1.u$. In some cases we need to compute some kind of composition of replacings. This leads us to the introduction of *generalized replacings* in Section 6. The adaptation of Paterson-Wegman's algorithm is described in Section 7. It allows us to translate a nominal unification problem into a set of replacing equations in linear time. Section 8 is devoted to the verification of these replacing equations. There we prove that it can be done in quadratic time. Finally, in Section 9 we discuss on the possibility of improving this bound and do this verification in quasi-linear time.

2. Preliminaries

Nominal terms contain *variables* and *atoms*. Only variables may be instantiated, and only atoms may be bound. They roughly correspond to the higher-order notions of free and bound variables, respectively, but are considered as completely different entities. Therefore, contrarily to the higher-order perspective, in nominal terms it makes no sense the distinction between free and bound variables depending on the existence of a binder above them.

Nominal terms[1] (typically $t, u, \dots$) are given by the grammar:

$$t ::= \langle t_1, t_2 \rangle \mid f(t_1, \dots, t_n) \mid a \mid a.t \mid \pi\, X$$

where f is a function symbol, a is an atom, π is a permutation (finite list of swappings), and X is a variable.

A *swapping* $(a\, b)$ is a pair of atoms of the same sort. The effect of a swapping over an atom is defined by $(a\, b)\, a = b$ and $(a\, b)\, b = a$ and $(a\, b)\, c = c$, when $c \neq a, b$. For the rest of terms the extension is straightforward, in particular, $(a\, b)\, (c.t) = \big((a\, b)\, c\big).\big((a\, b)\, t\big)$. A *permutation* is a (possibly empty) sequence of swappings. *Suspensions* are uses of variables with a permutation of atoms waiting to be applied once the variable is instantiated.

Substitutions are sort-respecting functions and behave like in first-order logic, hence allowing atom capture, for instance $[X \mapsto a]a.X = a.a$.

[1] For simplicity, we do not consider the *unit value* nor the *pairing*. Instead of them we consider n-ary function symbols.

A *freshness environment* (typically ∇) is a list of *freshness constraints* $a \# X$ stating that the instantiation of X cannot contain free occurrences of a.

The notion of nominal term α*-equivalence*, noted $\approx$, is defined by means of the following theory:

$$\frac{\nabla \vdash t_1 \approx u_1 \cdots \nabla \vdash t_n \approx u_n}{\nabla \vdash f(t_1, \ldots, t_n) \approx f\,(u_1, \ldots, u_n)}\ (\approx\text{-function}) \qquad \frac{}{\nabla \vdash a \approx a}\ (\approx\text{-atom})$$

$$\frac{a \neq a' \quad \nabla \vdash t \approx (a\,a')\,t' \quad \nabla \vdash a\#t'}{\nabla \vdash a.t \approx a'.t'}\ (\approx\text{-abst-2}) \qquad \frac{\nabla \vdash t \approx t'}{\nabla \vdash a.t \approx a.t'}\ (\approx\text{-abst-1})$$

$$\frac{(a\#X) \in \nabla \text{ for all } a \text{ such that } \pi\, a \neq \pi'\, a}{\nabla \vdash \pi\, X \approx \pi'\, X}\ (\approx\text{-susp.})$$

where the *freshness* predicate $\#$ is defined by:

$$\frac{\nabla \vdash a\#t_1 \cdots \nabla \vdash a\#t_n}{\nabla \vdash a\#f(t_1, \ldots, t_n)}\ (\#\text{-function}) \qquad \frac{a \neq a'}{\nabla \vdash a\#a'}\ (\#\text{-atom})$$

$$\frac{}{\nabla \vdash a\#a.t}\ (\#\text{-abst-1}) \qquad \frac{a \neq a' \quad \nabla \vdash a\#t}{\nabla \vdash a\#a'.t}\ (\#\text{-abst-2}) \qquad \frac{(\pi^{-1}\, a\, \#X) \in \nabla}{\nabla \vdash a\#\pi\, X}\ (\#\text{-susp.})$$

Their intended meanings are: $\nabla \vdash a \# t$ holds if, for every substitution σ respecting the freshness environment ∇ (i.e. avoiding the atom captures forbidden by ∇), a is not free in $\sigma(t)$; $\nabla \vdash t \approx u$ holds if, for every substitution σ respecting the freshness environment ∇, t and u are α-convertible.

A *nominal unification problem* (typically P) is a set of equations of the form $t \overset{?}{\approx} u$ or $a \#^{?} t$, *equational problems* and *freshness problems* respectively. A *solution* of a nominal problem is given by a substitution σ and a freshness environment ∇. Formally, the pair $\langle \nabla, \sigma \rangle$ solves P if, $\nabla \vdash a \# \sigma(t)$, for freshness problems $a \#^{?} t \in P$, and $\nabla \vdash \sigma(t) \approx \sigma(u)$, for equational problems $t \overset{?}{\approx} u \in P$.

Example 2.1. The solutions of the equation $a.X \overset{?}{\approx} b.X$ can not instantiate X with terms containing free occurrences of the atoms a and b, for instance if we apply the substitution $[X \mapsto a]$ to both sides of the equation we get $[X \mapsto a]a.X = a.a$ for the left hand side and $[X \mapsto a]b.X = b.a$ for the right hand side, and obviously $a.a \not\approx b.a$.

The most general solution of this equation is $\langle \{a\#X, b\#X\}, [\,] \rangle$.

The first linear First-Order Unification algorithm was described by Paterson and Wegman [Pat78]. Here we describe it in terms of transformation rules as it is done by Martelli and Montanari in [Mar82].

Definition 2.2. The Paterson-Wegman can be described by the following two transformation rules.

Simplification:

$$\left.\begin{array}{l} \{X_1, X_1', \ldots\} = f(Y_1, \ldots, Y_m) \\ \{X_2, X_2', \ldots\} = f(Z_1, \ldots, Z_m) \\ X_1 = X_2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \{X_1, X_1', \ldots, X_2, X_2', \ldots\} = f(Y_1, \ldots, Y_m) \\ Y_1 = Z_1 \\ \ldots \\ Y_m = Z_m \end{array}\right.$$

Variable:

$$\left.\begin{array}{l}\{X_1, X_1', \dots\} = t\\ \{X_2, X_2', \dots\} = \emptyset\\ X_1 = X_2\end{array}\right\} \Longrightarrow \{X_1, X_1', \dots, X_2, X_2', \dots\} = t$$

At every transformation, the selected equation $X_1 = X_2$ has to be maximal in the sense that there is no other equation $X_1^m = X_2^m$ and a set of equations of the form $\{X_i^m, \dots\} = f_{m-1}(\dots, X_i^{m-1}, \dots),\ \dots\ ,\{X_i^1, \dots\} = f_1(\dots, X_i, \dots)$ for $i = 1$ or $i = 2$.

3. Three Initial Simplifications

In this section we show how we can simplify nominal unification problems getting rid of freshness equations, of suspensions, and flattening all applications and abstractions. We will show that these simplifications only increase the size of the problem linearly. Lemma 3.1 shows us how to encode a freshness equation as an equality equation, and Lemma 3.2, how to encode a suspension also as an equality. Therefore, we can conclude that freshness equations and suspensions are mere syntactic sugar in nominal unification.

Lemma 3.1. *Let $b \neq a$. Then, $P \cup \{a\#t\}$ and $P \cup \{a.b.t \overset{?}{\approx} b.b.t\}$ have the same solutions.*

Proof. We first prove that $\langle a\#t, Id\rangle$ is a solution of $\{a.b.t \overset{?}{\approx} b.b.t\}$ when $b \neq a$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ t \approx t\end{array} \quad \begin{array}{c}a\#t\\ \vdots\ (\text{lemma 2.7})\\ b\#(a\,b)\,t\end{array}}{b.t \approx a.(a\,b)\,t}\,(\approx\text{-abst-2}) \quad \dfrac{a\#t}{a\#b.t}\,(\#\text{-abst-2})}{a.b.t \approx b.b.t}\,(\approx\text{-abst-2})$$

In this proof we prove $t \approx t$ from an empty set of assumptions. We can prove that this is always possible, for any term t, by structural induction on t. We also prove $b\#(a\,b)\,t$ from $a\#t$, using Lemma 2.7 of [Urb04].

Now, since $\nabla' \vdash \sigma(\nabla)$ and $\nabla \vdash t \approx t'$ implies $\nabla' \vdash \sigma(t) \approx \sigma(t')$ (see Lemma 2.14 of [Urb04]), we have that, if $\langle \nabla, \sigma\rangle$ solves $a\#^? t$, then $\langle \nabla, \sigma\rangle$ solves $a.b.t \overset{?}{\approx} b.b.t$.

Second, analyzing the previous proof, we see that the inference rules applied in each situation were the only applicable rules. Therefore, any solution $\langle \nabla, \sigma\rangle$ solving $a.b.t \overset{?}{\approx} b.b.t$, also solves $a\#^? t$, because any proof of $\sigma(a.b.t) \approx \sigma(b.b.t)$ contains a proof of $a\#\sigma(t)$ as a sub-proof.

From, these two fact we conclude that $a \#^? t$ and $a.b.t \overset{?}{\approx} b.b.t$ have the same set of solutions, for any $b \neq a$. Therefore, $\{a \#^? t\} \cup P$ and $\{a.b.t \overset{?}{\approx} b.b.t\} \cup P$, also have the same set of solutions, for any nominal unification problem P. From this we conclude that $P \cup \{a\#t\}$ and $P \cup \{a.b.t \overset{?}{\approx} b.b.t\}$ have the same set of solutions. ∎

Lemma 3.2. *$P \cup \{t \overset{?}{\approx} (a\,b)u\}$ and $P \cup \{a.b.t \overset{?}{\approx} b.a.u\}$ have the same solutions.*

Proof. If $a = b$ the proof is obvious. If $a \neq b$, then the proof is similar to proof of Lemma 3.1. In this case, the proof of $a.b.t \overset{?}{\approx} b.a.u$ from $t \overset{?}{\approx} (a\,b)u$ is as follows:

$$\dfrac{\dfrac{t \approx (a\,b)u}{b.t \approx b.(a\,b)u}\,(\approx\text{-abst-1}) \quad \dfrac{}{a\#a.u}\,(\#\text{-abst-1})}{a.b.t \approx b.a.u}\,(\approx\text{-abst-2})$$

■

Lemma 3.3. *Let X be a fresh variable not occurring elsewhere. Then,*
$P \cup \{a.t \overset{?}{\approx} u\}$ *and* $P \cup \{a.X \overset{?}{\approx} u, X \overset{?}{\approx} t\}$ *are equivalent*
$P \cup \{f(t_1,\dots,t_n) \overset{?}{\approx} u\}$ *and* $P \cup \{f(t_1,\dots,t_{i-1},X,t_{i+1},t_n) \overset{?}{\approx} u, X \overset{?}{\approx} t_i\}$ *are equivalent,*
$P \cup \{(a\,b)\,t \overset{?}{\approx} u\}$ *and* $P \cup \{(a\,b)\,X \overset{?}{\approx} u, X \overset{?}{\approx} t\}$ *are equivalent,*
$P \cup \{t_1 \overset{?}{\approx} t_2\}$ *and* $P \cup \{X \overset{?}{\approx} t_1, X \overset{?}{\approx} t_2\}$ *are equivalent, and*
$P \cup \{Y_1 \overset{?}{\approx} Y_2\}$ *and* $[Y_1 \mapsto Y_2]P$ *are equivalent.*

Proof. Let us consider the first statement. If $\langle\nabla,\sigma\rangle$ solves $P \cup \{a.t \overset{?}{\approx} u\}$, then it is enough to extend σ with $X \mapsto \sigma(t)$ to get a solution of $P \cup \{a.X \overset{?}{\approx} u, X \overset{?}{\approx} t\}$. In the opposite direction, any solution of $P \cup \{a.X \overset{?}{\approx} u, X \overset{?}{\approx} t\}$ is a solution of $P \cup \{a.t \overset{?}{\approx} u\}$, because, for any three terms t_1, t_2 and t_3, if $a.t_2 \approx t_1$ and $t_2 \approx t_3$, then $a.t_3 \approx t_1$. ■

Notice that the previous lemma does not hold for unification in λ-calculus. For instance, $\{\lambda a.f(a) =^? \lambda b.f(b)\}$ is trivially solvable. However, $\{\lambda a.X =^? \lambda b.b, X =^? a\}$ is unsolvable because, in λ-calculus, we have to avoid variable-capture in substitutions. This fact prevented Qian [Qia96] to apply this simplification in his linear-time algorithm for higher-order pattern unification.

Theorem 3.4. *There exists a linear reduction from Nominal Unification to a simplified version of Nominal Unification where all equations are of the form* $X \overset{?}{\approx} a$, $X \overset{?}{\approx} f(Y_1,\dots,Y_n)$ *or* $X \overset{?}{\approx} a.Y$.

Proof. We apply four reductions. First, applying Lemma 3.1, we can remove all freshness equations. Second, applying the transformations of Lemma 3.3 widely, replacing the first set of equations by the second whenever t is not a variable (in the first and third rules), or t_i is not a variable (in the second rule), or t_1 and t_2 are not variables (in the forth rule), we can flat all equations. Now, all equations have a variable in one side and a term of the form a, $a.X$, $f(X_1,\dots,X_n)$, or $(a\,b)\,X$ in the other side. In particular, all suspensions will occur in equations of the form $X \overset{?}{\approx} (a\,b)Y$. Applying Lemma 3.2, we can remove all them, translating them into $a.b.X \overset{?}{\approx} b.a.Y$. Forth, all these equations can be translated into $Z_3 \overset{?}{\approx} a.Z_1$, $Z_3 \overset{?}{\approx} b.Z_2$, $Z_1 \overset{?}{\approx} b.X$, $Z_2 \overset{?}{\approx} a.Y$, where Z_1, Z_2 and Z_3 are fresh.

A simple analysis shows that all these transformations are linear. ■

4. A First (Naive) Idea

Considering the similarities between Nominal Unification and FO Unification, a natural way to address the implementation of an efficient nominal unification algorithm is to postpone as much as possible the test of freshness predicates and equality between atoms. We can adapt algorithm of Definition 4.1 as follows. Instead of equations between variables, we use equations between variables affected by a permutation: $X_1 = \pi X_2$. Moreover, these equations are coupled with a set of freshness restrictions with the form of an implication: $a \neq \pi_1 b_1 \wedge \dots \wedge a \neq \pi_n b_n \Rightarrow a\#\pi_0 X_2$. The application rule is quite similar to the one used in algorithm 4.1, but the abstraction rule involves the extension of the permutation, the addition of a new associated freshness restriction and of additional conditions to the rest of freshness restrictions.

Definition 4.1. Consider the following (sound but incomplete) nominal unification algorithm. Given a set of simplified equations, transform them into a set of multi-equations as follows. First, transform any equation $X \stackrel{?}{=} t$ into a multi-equation $\{X\} = t$, and second, transform any pair of multi-equations $\{X\} = t_1, \{X\} = t_2$ into $\{X\} = t_1, \{X'\} = t_2$, $X = X'$, and add a multi-equation $\{X\} = \emptyset$ for any variable not occurring in the left of any multi-equation, until all variables occur in the left of a multi-equation exactly once. Then, apply the following transformation rules wisely.

Application:

$$\left.\begin{array}{l}
\{X_1, S_1\} = f(Y_1, \dots, Y_m)\\
\{X_2, S_2\} = f(Z_1, \dots, Z_m)\\
X_1 = \pi X_2\\
P_1 \Rightarrow c_1 \# \pi_1 X_2\\
\cdots\\
P_n \Rightarrow c_n \# \pi_n X_2
\end{array}\right\} \Longrightarrow \left\{\begin{array}{l}
\{X_1, S_1, \pi X_2, \pi S_2\} = f(Y_1, \dots, Y_m)\\
\\
Y_1 = \pi Z_1, \cdots, Y_m = \pi Z_m\\
P_1 \Rightarrow c_1 \# \pi_1 Z_1, \dots, P_1 \Rightarrow c_1 \# \pi_1 Z_m\\
\cdots\\
P_n \Rightarrow c_n \# \pi_n Z_1, \dots, P_n \Rightarrow c_n \# \pi_n Z_m
\end{array}\right.$$

Abstraction:

$$\left.\begin{array}{l}
\{X_1, S_1\} = a.Y\\
\{X_2, S_2\} = b.Z\\
X_1 = \pi X_2\\
P_1 \Rightarrow c_1 \# \pi_1 X_2\\
\cdots\\
P_n \Rightarrow c_n \# \pi_n X_2\\
\\
\end{array}\right\} \Longrightarrow \left\{\begin{array}{l}
\{X_1, S_1, \pi X_2, \pi S_2\} = a.Y\\
\\
Y = (a\ \pi b)\pi Z\\
P_1 \wedge c_1 \neq \pi_1\, b \Rightarrow c_1 \# \pi_1 Z\\
\cdots\\
P_n \wedge c_n \neq \pi_1\, b \Rightarrow c_n \# \pi_n Z\\
a \neq \pi b \Rightarrow a \# \pi Z
\end{array}\right.$$

Atom:

$$\left.\begin{array}{l}
\{X_1, S_1\} = a\\
\{X_2, S_2\} = b\\
X_1 = \pi X_2\\
P_1 \Rightarrow c_1 \# \pi_1 X_2\\
\cdots\\
P_n \Rightarrow c_n \# \pi_n X_2
\end{array}\right\} \Longrightarrow \left\{\begin{array}{l}
\{X_1, S_1, \pi X_2, \pi S_2\} = a\\
\\
a = \pi b\\
P_1 \Rightarrow c_1 \neq \pi_1 b\\
\cdots\\
P_n \Rightarrow c_n \neq \pi_n b
\end{array}\right.$$

Notice that the algorithm previously described is incomplete. For instance, the variable X_1 in $\{X_1, S_1\} = f(Y_1, \dots, Y_m)$ could be already affected by a permutation, which makes the rule inapplicable. However, these rules allow us to solve the following example:

Example 4.2. The Nominal unification problem $a_3.a_2.a_1.f(c_1, c_2) \stackrel{?}{\approx} b_3.b_2.b_1.f(d_1, d_2)$ is transformed by the naive algorithm into the following set of conditional equalities and inequalities.

$$\begin{array}{l}
c_1 = (a_1\ (a_2\ (a_3 b_3) b_2)(a_3\ b_3) b_1)(a_2\ (a_3 b_3) b_2)(a_3\ b_3) d_1\\
c_2 = (a_1\ (a_2\ (a_3 b_3) b_2)(a_3\ b_3) b_1)(a_2\ (a_3 b_3) b_2)(a_3\ b_3) d_2\\
a_3 \neq b_3\ \wedge\ a_3 \neq b_2\ \wedge\ a_3 \neq b_1\ \Rightarrow\ a_3 \neq d_1\\
a_3 \neq b_3\ \wedge\ a_3 \neq b_2\ \wedge\ a_3 \neq b_1\ \Rightarrow\ a_3 \neq d_2\\
a_2 \neq (a_3\ b_3) b_2\ \wedge\ a_2 \neq (a_3\ b_3) b_1\ \Rightarrow\ a_2 \neq (a_3\ b_3) d_1\\
a_2 \neq (a_3\ b_3) b_2\ \wedge\ a_2 \neq (a_3\ b_3) b_1\ \Rightarrow\ a_2 \neq (a_3\ b_3) d_2\\
a_1 \neq (a_2\ (a_3 b_3) b_2)(a_3\ b_3) b_1\ \Rightarrow\ a_1 \neq (a_2\ (a_3 b_3) b_2)(a_3\ b_3) d_1\\
a_1 \neq (a_2\ (a_3 b_3) b_2)(a_3\ b_3) b_1\ \Rightarrow\ a_1 \neq (a_2\ (a_3 b_3) b_2)(a_3\ b_3) d_2
\end{array}$$

It is easy to see that a generalization of this simple problem to

$$a_n.\dots.a_1.f(c_1,\dots,c_m) \overset{?}{\approx} b_n.\dots.b_1.f(d_1,\dots,d_m)$$

would result in a set of inequalities of size $\mathcal{O}(n\,m)$. The number of comparisons of atoms that have to be checked in order to compute the result of applying the permutation and check the equalities is also $\mathcal{O}(n\,m)$.

5. Simple Replacings

In this section we introduce a new concept, similar to the idea of substitution and of swapping, but with some differences. Thus, we have preferred to call it with the new name *replacings.*

Definition 5.1. A *replacing* is a (possibly empty) list of pairs of atoms $L = (a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$.

Given two terms t and u and a replacing $L = (a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$, we say that t and u are equivalent modulo L, noted $t =_L u$, if $a_n.\cdots a_1.t \approx b_n.\cdots b_1.u$.

Any replacing may be associated with a permutation of atoms, defined as follows. This definition and the following lemma, helps us to see replacings as permutations, plus a set of associated freshness equations. The example bellow also shows that the associated permutation is not enough to characterize a replacing.

Definition 5.2. Given a replacing L, we define its associated permutation Π_L inductively as follows

(1) $\Pi_{[\,]} = [\,]$, being $[\,]$ the empty list, and empty sequence of swappings.
(2) $\Pi_{(a\leftarrow b)L} = (a\ \Pi_L b)\Pi_L$

Lemma 5.3. *Given a replacing $L = (a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$ and two terms t and u, $t =_L u$ holds, iff*

(1) *$t \approx \Pi_L u$, and*
(2) *for any $i = 1,\dots,n$, if $a_i \neq \Pi_{(a_{i+1}\leftarrow b_{i+1})\dots(a_n\leftarrow b_n)} b_j$ for all $j = i,\dots,1$, then $a_i \# \Pi_{(a_{i+1}\leftarrow b_{i+1})\dots(a_n\leftarrow b_n)} u$.*

Example 5.4. Notice that the permutation Π_L does not characterize the replacing L. For instance, we have

$$\Pi_{(a\leftarrow b)} = \Pi_{(b\leftarrow a)} = \Pi_{(b\leftarrow a)(a\leftarrow b)} = \Pi_{(a\leftarrow b)(b\leftarrow a)} = \Pi_{(a\leftarrow b)(a\leftarrow b)} = (a\ b) = (b\ a)$$

However, assuming $a \neq b$, we have

$$\begin{array}{rclcl} t =_{(a\leftarrow b)} u & \Leftrightarrow & t =_{(a\leftarrow b)(a\leftarrow b)} u & \Leftrightarrow & t = (a\ b)u \ \wedge\ a\#u \\ t =_{(b\leftarrow a)} u & \Leftrightarrow & t =_{(b\leftarrow a)(b\leftarrow a)} u & \Leftrightarrow & t = (a\ b)u \ \wedge\ b\#u \\ t =_{(b\leftarrow a)(a\leftarrow b)} u & \Leftrightarrow & t =_{(a\leftarrow b)(b\leftarrow a)} u & \Leftrightarrow & t = (a\ b)u \end{array}$$

If for any pair of term we have $t =_L u \Leftrightarrow t =_{L'} u$, then this will be also true for any pair of atoms, and we will have $\Pi_L = \Pi_{L'}$. This motivates the following definition.

Definition 5.5. We say that two replacings L and L' are equivalent if, for any pair of terms t and u, we have $t =_L u$ iff $t =_{L'} u$

Lemma 5.6. *$t =_{(a_1\leftarrow b_1)\cdots(a_n\leftarrow b_n)} u$ iff $u =_{(b_1\leftarrow a_1)\cdots(b_n\leftarrow a_n)} t$.*

The following lemma describes a method to check if $c =_L d$ in time $\mathcal{O}(|L|)$.

Lemma 5.7. *Given two atoms c and d and a replacing $(a \leftarrow b)L$:*

$$c =_{(a\leftarrow b)L} d \quad \textit{iff} \quad \begin{array}{l} c = a \textit{ and } b = d, \textit{ or} \\ c \neq a,\ b \neq d \textit{ and } c =_L d. \end{array}$$

Next, we will describe a normalization procedure of replacings. We say that a replacing $(a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$ is normalized if $a_1, \dots, a_n$ is a list of pairwise distinct atoms, and $b_1, \dots, b_n$ too. Lemma 5.8 states that, any normalized replacing may be characterized by a *set*, instead of a *list*), of pairs of atoms. Lemma 5.9 shows how we can remove duplicated pairs and normalized replacings, on the expenses of adding freshness equations.

When atoms are not repeated in a replacing, then they are basically[2] a permutation, as the following lemma states.

Lemma 5.8. *If $L = (a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$ is a normalized replacing, i.e. a replacing where $a_1, \dots, a_n$ is a list of pairwise distinct atoms, and $b_1, \dots, b_n$ too, then*

(1) *Π_L is a permutation satisfying $\Pi_L(b_i) = a_i$, for $i = 1, \dots, n$,*
(2) *$(a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$ and $(a_{\pi(1)} \leftarrow b_{\pi(1)})\cdots(a_{\pi(n)} \leftarrow b_{\pi(n)})$ are equivalent, for any permutation π.*
(3) *For any $a, b \in \mathbb{A}$, $a =_L b$ iff $\Pi_L(a) = b$.*

Proof. By induction on n. For any $i = 1, \dots, n$, we have

$$\Pi_{(a_1\leftarrow b_1)\cdots(a_n\leftarrow b_n)} b_i = (a_1\ \Pi_{(a_2\leftarrow b_2)\cdots(a_n\leftarrow b_n)} b_1)\cdots \underbrace{(a_i\ \Pi_{(a_{i+1}\leftarrow b_{i+1})\cdots(a_n\leftarrow b_n)} b_i)\ \overbrace{\quad\cdots\quad (a_n b_n)}^{=\Pi_{(a_{i+1}\leftarrow b_{i+1})\cdots(a_n\leftarrow b_n)}}\ b_i}_{=a_i}$$

Hence, the i-th swapping changes $\Pi_{(a_{i+1}\leftarrow b_{i+1})\cdots(a_n\leftarrow b_n)} b_i$ by a_i. Now we are going to prove that a_i is not affected by the swappings $(a_j\ \Pi_{(a_{j+1}\leftarrow b_{j+1})\cdots(a_n\leftarrow b_n)} b_j)$ where $j > i$. On one hand, by assumption, $a_j \neq a_i$ when $j > i$. On the other hand, $\Pi_{(a_{j+1}\leftarrow b_{j+1})\cdots(a_n\leftarrow b_n)} b_j \neq a_i$ because $(a_{j+1} \leftarrow b_{j+1})\cdots(a_n \leftarrow b_n)$ is a strictly shorter replacing, and $i \in \{j+1, \dots, n\}$, therefore by induction hypothesis $(\Pi_{(a_{j+1}\leftarrow b_{j+1})\cdots(a_n\leftarrow b_n)})^{-1}(a_i) = b_i \neq b_j$. ■

Lemma 5.9. *The replacing $L(a \leftarrow b)L'$ where a occurs on the left in L, and b occurs on the right in L, is equivalent to $L\,L'$. In other words, $L_1(a \leftarrow c)L_2(d \leftarrow b)L_3(a \leftarrow b)L_4$ and $L_1(a \leftarrow c)L_2(d \leftarrow b)L_3\,L_4$ are equivalent.*

If a occurs on the left in L, but b does not occur in the right in L, then, for any pair of terms t and u, $t =_{L(a\leftarrow b)L'} u$ iff $b \# u$ and $t =_{LL'} u$.

Similarly, if a does not occur on the left in L, but b occurs in the right in L, then, for any pair of terms t and u, $t =_{L(a\leftarrow b)L'} u$ iff $a \# t$ and $t =_{LL'} u$.

Proof. In nominal logic, and in λ-calculus we have the following implications:

$$\text{If } a\#t \text{ and } a.t \approx b.u, \text{ then } b\#u \text{ and } t \approx u \tag{5.1}$$

$$\text{If } t \approx u,\ a\#t \text{ and } b\#u, \text{ then } a.t \approx b.u \tag{5.2}$$

By definition of replacing, $t =_{(a_1\leftarrow b_1)\dots(a_n\leftarrow b_n)} u$ is equivalent to $a_n.\cdots.a_1.t \approx b_n.\cdots.b_1.u$.

For the first statement: For a given i, if $a_i \in \{a_{i-1}, \dots, a_1\}$, then $a_i \# a_{i-1}.\cdots.a_1.t$ and $t =_{(a_1\leftarrow b_1)\dots(a_n\leftarrow b_n)} u$ (using 5.1) imply $a_{i-1}.\cdots.a_1.t \approx b_{i-1}.\cdots.b_1.u$, hence $t =_{(a_1\leftarrow b_1)\dots(a_{i-1}\leftarrow b_{i-1})(a_{i+1}\leftarrow b_{i+1})\dots(a_n\leftarrow b_n)} u$. ■

[2] Notice that we still have to ensure the freshness conditions

Lemmas 5.8 and 5.9 describe a characterization of replacings in terms of a set of pairs of atoms (normalized replacing), and a set of freshness equations. In the following we make explicit this characterization in terms of a set of pairs, called *rewriting set*, and a set of *forbidden atoms.*

Definition 5.10. Given a replacing L, we define the sets of rewriting pairs and forbidden atoms, noted $\mathit{Rew}(L)$ and $\mathit{For}(L)$, as follows

$$\begin{aligned} \mathit{Rew}(L) &= \{(a \leftarrow b) \in \mathbb{A} \times \mathbb{A} \mid a \neq b \wedge a =_L b\} \\ \mathit{For}(L) &= \{a \in \mathbb{A} \mid \neg(a =_L a)\} \end{aligned}$$

Lemma 5.11. *Replacings L and L' are equivalent iff $\mathit{Rew}(L) = \mathit{Rew}(L')$ and $\mathit{For}(L) = \mathit{For}(L')$.*

Lemma 5.12. *For any replacing L we have*

$$\mathit{Rew}([\,]) = \emptyset$$

$$\mathit{Rew}(L(a \leftarrow b)) = \begin{cases} \mathit{Rew}(L) \cup \{a \leftarrow b\} & \textit{if } a \neq b \textit{ and } \forall c.a \leftarrow c \notin \mathit{Rew}(L) \textit{ and } \forall c.c \leftarrow b \notin \mathit{Rew}(L) \\ \mathit{Rew}(L) & \textit{otherwise} \end{cases}$$

$$\mathit{For}([\,]) = \emptyset$$

$$\mathit{For}(L(a \leftarrow b)) = \begin{cases} \mathit{For}(L) \cup \{b\} & \textit{if } \exists c.a \leftarrow c \in \mathit{Rew}(L) \textit{ and } \forall d.d \leftarrow b \notin \mathit{Rew}(L) \\ \mathit{For}(L) \cup \{a\} & \textit{if } \exists d.d \leftarrow b \in \mathit{Rew}(L) \textit{ and } \forall c.a \leftarrow c \notin \mathit{Rew}(L) \\ \mathit{For}(L) & \textit{otherwise} \end{cases}$$

Proof. Given a replacing, we can use Lemma 5.9 to remove pairs with a duplicated component wisely until we obtain a normalized replacing. By Lemma 5.8, this normalized replacing is the rewriting set, whereas the set of freshness equations define the set of forbidden atoms. Then we can check that the previous recursions hold. ∎

6. Generalized Replacings

Sometimes, simple replacings are not enough to represent the equations between atoms that we have to check. In some cases, we have to use a kind of *composition* of replacings. In this section we show how the notion of simple replacing may be generalized for this purpose, and how we can extend the definition of set of rewritings and set of forbidden atoms.

Definition 6.1. A generalized replacing is an expression generated by the grammar

$$L ::= \mathit{Id} \mid (a \leftarrow b) :: L \mid L_1 \circ L_2 \mid L^{-1}$$

with the following semantics

$t =_{\mathit{Id}} u$, if $t \approx u$,
$t =_{(a \leftarrow b)::L} u$, if $a.t =_L b.u$,
$t =_{L_1 \circ L_2} u$, if there exists a term v such that $t =_{L_1} v$ and $v =_{L_2} u$, and
$t =_{L^{-1}} u$, if $u =_L t$.

The sets $\mathit{Rew}(L)$ and $\mathit{For}(L)$ are defined for generalized replacings as for simple replacings.

Lemma 6.2. *Any generalized replacing is equivalent to a composition of simple replacings accordingly to the following equivalences between replacings*

$(L_1 \circ L_2) \circ L_3 = L_1 \circ (L_2 \circ L_3)$
$(a \leftarrow b) :: \big(L_1 \circ L_2\big) = \big((a \leftarrow b) :: L_1\big) \circ \big((a \leftarrow b) :: L_2\big)$
$(a_1 \leftarrow b_1) :: \cdots :: (a_n \leftarrow b_n) :: \mathit{Id} = (a_1 \leftarrow b_1) \cdots (a_n \leftarrow b_n)$

The following lemma shows us how we can recursively compute the set of rewritings and of forbidden atoms of a generalized replacing.

Lemma 6.3.

$$\begin{array}{l}
\mathit{Rew}(\mathit{Id}) = \mathit{For}(\mathit{Id}) = \emptyset \\
\mathit{Rew}\big((a \leftarrow b) :: L\big) = \mathit{Rew}(L) \setminus \{a \leftarrow c \mid \forall c \in \mathbb{A}\} \setminus \{c \leftarrow b \mid \forall c \in \mathbb{A}\} \cup \left\{ \begin{array}{ll} \{a \leftarrow b\} & \mathit{if}\ a \neq b \\ \emptyset & \mathit{if}\ a = b \end{array} \right. \\
\mathit{For}\big((a \leftarrow b) :: L\big) = \mathit{For}(L) \cup \{c \mid a \leftarrow c \in \mathit{Rew}(L) \vee c \leftarrow b \in \mathit{Rew}(L)\} \\
\begin{array}{rl}
\mathit{Rew}(L_1 \circ L_2) = & \{a \leftarrow c \mid \exists b \in \mathbb{A}\ a \leftarrow b \in \mathit{Rew}(L_1) \wedge b \leftarrow c \in \mathit{Rew}(L_2)\} \\
& \cup\{a \leftarrow b \mid a \leftarrow b \in \mathit{Rew}(L_1) \wedge b \notin \mathit{For}(L_2)\} \\
& \cup\{a \leftarrow b \mid a \leftarrow b \in \mathit{Rew}(L_2) \wedge a \notin \mathit{For}(L_1)\}
\end{array} \\
\mathit{For}(L_1 \circ L_2) = \mathit{For}(L_1) \cup \mathit{For}(L_2) \\
\mathit{Rew}(L^{-1}) = \{(b \leftarrow a) \mid (a \leftarrow b) \in \mathit{Rew}(L)\} \\
\mathit{For}(L^{-1}) = \mathit{For}(L)
\end{array}$$

7. A Paterson-Wegman Style Algorithm

In this section we describe our nominal unification algorithm in the style of Paterson and Wegman [Pat78], or, to be precise, in the style of the description that Martelli and Montanari [Mar82] makes of this algorithm.

First, w.l.o.g. we consider that we have a single nominal equation (we get rid of freshness equations, by Lemma 3.1, and reduce $\{t_1 \overset{?}{\approx} u_1, \dots, t_n \overset{?}{\approx} u_n\}$ to $f(t_1, \dots f(t_{n-1}, t_n) \dots) \overset{?}{\approx} f(u_1, \dots f(u_{n-1}, u_n) \dots)$, provided that there exists a binary constant f). Then, we flatten this equation, obtaining a set of equations of the form $X \overset{?}{\approx} f(Y_1, \dots, Y_n)$, $X \overset{?}{\approx} a.Y$, $X \overset{?}{\approx} a$ or $X \overset{?}{\approx} (a\,b)Y$, and a single equation $X_1 \overset{?}{\approx} X_2$, where X_1 and X_2 do not occur elsewhere bellow any other symbol. Finally, by Lemma 3.2, we can get rid of equations of the form $X \overset{?}{\approx} (a\,b)Y$. By Theorem 3.4, the resulting nominal unification problem has size $\mathcal{O}(|P|)$ on the size of the original problem.

Following the notation of [Mar82], equations of the form $X \overset{?}{\approx} f(Y_1, \dots, Y_n)$, $X \overset{?}{\approx} a.Y$, and $X \overset{?}{\approx} a$ are written in the form $\{X\} = f(Y_1, \dots, Y_n)$, $\{X\} = a.Y$, and $\{X\} = a$, respectively. The equation $X_1 \overset{?}{\approx} X_2$ is written as $X_1 =_{Id} X_2$, using the replacing Id. Then, we apply the following transformation rules wisely, where the equation $X_1 =_L X_2$ is in all cases a maximal equation, in the sense of Definition 2.2. Like in the classical Paterson-Wegman algorithm, there always exists an equation satisfying this condition, and we can find this equation intelligently, such that the total time consumed by this search is linearly bounded on the size of the original problem (see [Pat78] for more details).

Definition 7.1. Consider the following set of transformation rules:
Application:

$$\left.\begin{array}{l}
\{\Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots\} = f(Y_1, \dots, Y_m) \\
\{\Pi_{L_2} X_2, \Pi_{L'_2} X'_2, \dots\} = f(Z_1, \dots, Z_m) \\
X_1 =_L X_2
\end{array}\right\} \Longrightarrow \left\{\begin{array}{l}
\left\{\begin{array}{l} \Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots, \\ \Pi_{L_1 \circ L} X_2, \Pi_{L_1 \circ L \circ L_2^{-1} \circ L'_2} X'_2, \dots \end{array}\right\} = f(Y_1, \dots, Y_m) \\
Y_1 =_{L_1 \circ L \circ L_2^{-1}} Z_1 \\
\dots \\
Y_m =_{L_1 \circ L \circ L_2^{-1}} Z_m
\end{array}\right.$$

Abstraction:

$$\left.\begin{array}{l} \{\Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots\} = a.Y \\ \{\Pi_{L_2} X_2, \Pi_{L'_2} X'_2, \dots\} = b.Z \\ X_1 =_L X_2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \left\{\begin{array}{l} \Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots, \\ \Pi_{L_1 \circ L} X_2, \Pi_{L_1 \circ L \circ L_2^{-1} \circ L'_2} X'_2, \dots \end{array}\right\} = a.Y \\ Y =_{(a \leftarrow b)::(L_1 \circ L \circ L_2^{-1})} Z \end{array}\right.$$

Atom:

$$\left.\begin{array}{l} \{\Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots\} = a \\ \{\Pi_{L_2} X_2, \Pi_{L'_2} X'_2, \dots\} = b \\ X_1 =_L X_2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \left\{\begin{array}{l} \Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots, \\ \Pi_{L_1 \circ L} X_2, \Pi_{L_1 \circ L \circ L_2^{-1} \circ L'_2} X'_2, \dots \end{array}\right\} = a \\ a =_{L_1 \circ L \circ L_2^{-1}} b \end{array}\right.$$

Variable:

$$\left.\begin{array}{l} \{\Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots\} = t \\ \{\Pi_{L_2} X_2, \Pi_{L'_2} X'_2, \dots\} = \emptyset \\ X_1 =_L X_2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots \\ \Pi_{L_1 \circ L} X_2, \Pi_{L_1 \circ L \circ L_2^{-1} \circ L'_2} X'_2, \dots \end{array}\right\} = t$$

$$\left.\begin{array}{l} \{\Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots\} = \emptyset \\ \{\Pi_{L_2} X_2, \Pi_{L'_2} X'_2, \dots\} = t \\ X_1 =_L X_2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \Pi_{L_2 \circ L^{-1}} X_1, \Pi_{L_2 \circ L^{-1} \circ L_1^{-1} \circ L'_1} X'_1, \dots \\ \Pi_{L_2} X_2, \Pi_{L'_2} X'_2, \dots \end{array}\right\} = t$$

Theorem 7.2. *Given a simplified nominal unification problem P, P is solvable if, and only if, the rules of Definition 7.1 transform the problem into a set of equations of the form*

$$\left\{\begin{array}{c} \{\Pi_{L_1^1} X_1^1, \dots, \Pi_{L_1^{r_1}} X_1^{r_1}\} = t_1 \\ \dots \\ \{\Pi_{L_m^1} X_m^1, \dots, \Pi_{L_m^{r_m}} X_m^{r_m}\} = t_m \\ a_1 =_{L_1} b_1 \\ \dots \\ a_n =_{L_n} b_n \end{array}\right\}$$

where $\{a_1 =_{L_1} b_1, \dots, a_n =_{L_n} b_n\}$ holds, and the equations $t_i =_{L_i^j} X_i^j$, for $i = 1, \dots, m$ and $j = 1, \dots, r_i$, are solvable.

When P is solvable, then set of equations $t_i =_{L_i^j} X_i^j$ encode a solution.

Moreover, the size of the DAG representing the new set of equations is $\mathcal{O}(|P|)$, and it can be obtained in time $\mathcal{O}(|P|)$.

Proof. Soundness and completeness results from the rules ≈-abst-1, ≈-abst-2, and ≈-fun and ≈-atom of [Urb04], conveniently written in terms of replacings. The transformations resemble Paterson-Wegman transformations (Definition 2.2), and the termination proof is based on the same ideas. Notice that some transformations duplicate some L's. Therefore, the linear bound only applies representing equations as DAGs. ∎

Example 7.3. The equation $a.b.X \overset{?}{\approx} b.b.X$ can be simplified as:

$$\big\{\{X\} = \emptyset, \quad \{Y_1\} = a.Y_3, \quad \{Y_2\} = b.Y_4, \quad \{Y_3\} = b.X, \quad \{Y_4\} = b.X, \quad Y_1 =_{Id} Y_2\big\}$$

Applying twice the abstraction rule we obtain:

$$\big\{\{X\} = \emptyset, \quad \{Y_1, Y_2\} = a.Y_3, \quad \{Y_3, \Pi_{(a \leftarrow b)::Id} Y_4\} = b.X, \quad X =_{(b \leftarrow b)::(a \leftarrow b)::Id} X\big\}$$

One application of the variable rule gives us the simplified equations

$$\big\{\{Y_1, Y_2\} = a.Y_3, \quad \{Y_3, \Pi_{(a \leftarrow b)::Id} Y_4\} = b.X, \quad \{X, \Pi_{(b \leftarrow b)::(a \leftarrow b)::Id} X\} = \emptyset\big\}$$

Example 7.4. From $\underbrace{a.\underbrace{b.\underbrace{a}_{Y_4}}_{Y_3}}_{Y_1} \overset{?}{\approx} \underbrace{b.\underbrace{b.X}_{Y_5}}_{Y_2}$, we obtain
$$\begin{array}{l}\{Y_1, Y_2\} = a.Y_3 \\ \{Y_3, \Pi_{(a \leftarrow b)::Id} Y_5\} = b.Y_4 \\ \{Y_4, \Pi_{(b \leftarrow b)::(a \leftarrow b)::Id} X\} = a\end{array} \quad .$$

8. Efficient Checking of Replacings

Using the algorithm described in Definition 7.1, we obtain a set of replacing equations of the form $a =_L b$, a set of equations of the form $\{\Pi_{L^1} X^1, \dots, \Pi_{L^r} X^r\} = t$ that codify the solution, and a DAG that represents the generalized replacings L's. Now, we will describe how we can check the solvability of these equations in quadratic time.

The main idea is to compute, for every node of the DAG, the two sets $\mathit{Rew}(L)$ and $\mathit{For}(L)$, where L is the replacing represented by this node. We will use the values of these sets already computed for the descendants of the node. Therefore, we proceed from the leaves of the DAG to the roots. We assume that we have a total ordering on the atoms $\mathbb{A}$. For efficiency, we compute three lists for every node L: a list RL that contains the elements of $\mathit{Rew}(L)$ ordered by the first component, RR with the elements of $\mathit{Rew}(L)$ ordered by the second component, and an ordered list F with the elements of $\mathit{For}(L)$. Moreover, the lists RL and RR are doubly linked, such that knowing the position of an element $(a \leftarrow b)$ in RL, we can know its position in RR and vice versa. Lemma 6.3 describes how to compute these list. Just as an example, Figure 1 shows how to compute RL, RR and F for $L = L_1 \circ L_2$, being RL_i, RR_i and F_i, for $i = 1, 2$, the respective lists for L_i.

To check if a set of equations P of the form $\{\Pi_{L_1} X_1, \dots, \Pi_{L_r} X_r\} = t$ has solution, and what is this solution, we compute the set of atoms that cannot occur free in the instance of X, written $\mathit{For}(X)$. This computation aborts (using rule 5) if P is unsolvable.

Definition 8.1. Given a set of equations P, for every variable X, we compute $\mathit{For}(X)$ as the minimal set of atoms that satisfy all the following rules, or we abort.

(1) If P contains $\{\Pi_{L_1} X_1, \dots, \Pi_{L_r} X_r\} = t$,
then $\Pi_{L_j^{-1}}\big(\Pi_{L_i}\big(\mathit{For}(X_i)\big) \cup \mathit{For}(L_i)\big) \subseteq \mathit{For}(X_j)$, for $i \neq j = 1, \dots, r$.

(2) If P contains $\{\Pi_{L_1} X_1, \dots, \Pi_{L_r} X_r\} = f(Y_1, \dots, Y_m)$,
then $\Pi_{L_i}(\mathit{For}(X_i)) \cup \mathit{For}(L_i) \subseteq \mathit{For}(Y_j)$, for $i = 1, \dots, r$, and $j = 1, \dots, m$.

(3) If P contains $\{\Pi_{L_1} X_1, \dots, \Pi_{L_r} X_r\} = a.Y$,
then $\Pi_{L_i}(\mathit{For}(X_i)) \cup \mathit{For}(L_i) \setminus \{a\} \subseteq \mathit{For}(Y)$, for $i = 1, \dots, r$.

(4) If P contains $\{\Pi_L X, \Pi_{L'} X, \dots\} = t$ and $\Pi_L(a) \neq \Pi_{L'}(a)$, for some $a \in \mathbb{A}$, then $a \in \mathit{For}(X)$.

(5) If P contains $\{\Pi_{L_1} X_1, \dots, \Pi_{L_r} X_r\} = a$ and $a \in \Pi_{L_i}(\mathit{For}(X_i)) \cup \mathit{For}(L_i)$, for some $i = 1, \dots, r$, then P is unsolvable and abort.

Lemma 8.2. *Given a set of equations of the form $\{\Pi_{L_1} X_1, \dots, \Pi_{L_r} X_r\} = t$, we can compute $\mathit{For}(X)$, for every variable X, or abort, in quadratic time on the size of the DAG-representation of the equations.*

Moreover, the solution encoded by the equations is $\{a \# X \mid a \in \mathit{For}(X)\}$.

Proof. At every node of the DAG representing a generalized replacing L, we compute $\mathit{Rew}(L)$ and $\mathit{For}(L)$, using the values $\mathit{Rew}(L_i)$ and $\mathit{For}(L_i)$ previously computed for

```
Input: $RL_1, RR_1, F_1, RL_2, RR_2, F_2$
Output: $RL, RR, F$

$i_1 := 1$ ; $i_2 := 1$ ; $j_1 := 1$ ; $j_2 := 1$
while $i_1 \leq RR_1.size()$ and $i_2 \leq RL_2.size()$ do
    let $(a \leftarrow b) = RR_1[i_1]$ and $(b' \leftarrow c) = RL_1[i_2]$
    if $b = b'$ then
        following the double links, change $(a \leftarrow b)$ in $RL_1$ by $(a \leftarrow c)$
        following the double links, change $(b \leftarrow c)$ in $RR_2$ by $(a \leftarrow c)$
        remove $(a \leftarrow b)$ from $RR_1$ and $(b \leftarrow c)$ from $RL_2$
        $i_1 := i_1 + 1$
        $i_2 := i_2 + 1$
    else if $b < b'$ then
        while $j_2 \leq F_2.size()$ and $F_2[j_2] < b$ do  $j_2 := j_2 + 1$
        if $j_2 \leq F_2.size()$ and $F_2[j_2] = b$ then
            remove $(a \leftarrow b)$ from $RR_1$ and $RL_1$
        $i_1 := i_1 + 1$
    else while $j_1 \leq F_1.size()$ and $F_1[j_1] < b$ do $j_1 := j_1 + 1$
        if $j_1 \leq F_1.size()$ and $F_1[j_1] = b'$ then
            remove $(b' \leftarrow c)$ from $RR_2$ and $RL_2$
        $i_2 := i_2 + 1$
if $i_1 = RR_1.size()$ then
    while $i_2 \leq RL_2.size()$ do
        while $j_1 \leq F_1.size()$ and $F_1[j_1] < b'$ do $j_1 := j_1 + 1$
        if $j_1 \leq F_1.size()$ and $F_1[j_1] = b'$ then
            remove $(b' \leftarrow c)$ from $RR_2$ and $RL_2$
        $i_2 := i_2 + 1$
else while $i_1 \leq RR_1.size()$ do
    while $j_2 \leq F_2.size()$ and $F_2[j_2] < b$ do $j_2 := j_2 + 1$
    if $j_2 \leq F_2.size()$ and $F_2[j_2] = b$ then
        remove $(a \leftarrow b)$ from $RR_1$ and $RL_1$
    $i_1 := i_1 + 1$
$RL := merge(RL_1, RL_2)$
$RR := merge(RR_1, RR_2)$
$F := merge(F_1, F_2)$
return $RR, RL, F$
```

Figure 1: Computation of $Rew(L_1 \circ L_2)$ and $For(L_1 \circ L_2)$ in time $\mathcal{O}(|Rew(L_1)| + |Rew(L_2)| + |For(L_1)| + |For(L_2)|)$.

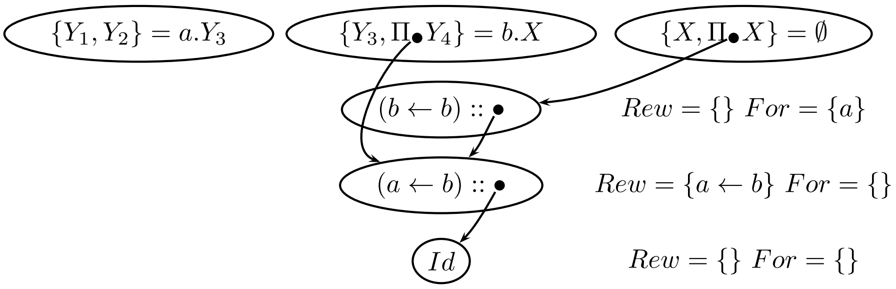
the descendants L_i of the node. This computation takes at worst linear time for every node, being the worst case the composition of replacings $L = L_1 \circ L_2$ with time $\mathcal{O}(|Rew(L_1)| + |Rew(L_2)| + |For(L_1)| + |For(L_2)|)$, described in Figure 1. Therefore, the overall computation takes quadratic time. Then, using the rules of Definition 8.1, in quadratic time we can check if all equations are solvable. ■

Theorem 8.3. *Nominal Unification can be decided in quadratic time.*

Proof. By Theorem 3.4 we can assume that nominal equations are simplified. Then, by Theorem 7.2, we can transform these equations into an equivalent set of equations of the form $a =_L b$ or $\{\Pi_{L_1} X_1, \dots, \Pi_{L_r} X_r\} = t$ represented as a DAG, in linear time on the size of the original equations. Equations $a =_L b$ are solvable if $(a \leftarrow b) \in \mathit{Rew}(L)$. By Lemma 8.2, we can compute $\mathit{For}(X)$ for every variable, checking the solvability of equations $\{\Pi_{L_1} X_1, \dots, \Pi_{L_r} X_r\} = t$. ∎

Example 8.4. Consider example 7.3, where we obtain
$\big\{\{Y_1, Y_2\} = a.Y_3, \quad \{Y_3, \Pi_{(a \leftarrow b)::\mathit{Id}} Y_4\} = b.X, \quad \{X, \Pi_{(b \leftarrow b)::(a \leftarrow b)::\mathit{Id}} X\} = \emptyset\big\}$.

The DAG representation with $\mathit{Rew}(L)$ and $\mathit{For}(L)$ of every node representing a generalized replacing is as follows.


$\{Y_1, Y_2\} = a.Y_3$
$\{Y_3, \Pi_\bullet Y_4\} = b.X$
$\{X, \Pi_\bullet X\} = \emptyset$
$(b \leftarrow b) :: \bullet$
$\mathit{Rew} = \{\}\ \mathit{For} = \{a\}$
$(a \leftarrow b) :: \bullet$
$\mathit{Rew} = \{a \leftarrow b\}\ \mathit{For} = \{\}$
Id
$\mathit{Rew} = \{\}\ \mathit{For} = \{\}$

Definition 8.1 computes $\mathit{For}(Y_3) = \mathit{For}(X) = \{a\}$, $\mathit{For}(Y_4) = \{b\}$, $\mathit{For}(Y_1) = \mathit{For}(Y_2) = \emptyset$. Now, considering only original variables, i.e. X, we obtain the solution $a\#X$.

In example 7.4, the equation $\{Y_4, \Pi_{(b \leftarrow b)::(a \leftarrow b)::\mathit{Id}} X\} = a$, using rule 5 of Definition 8.1, allows us to deduce that the problem is unsolvable.

9. Conclusions, can we do it better?

We have presented an efficient algorithm that computes nominal unifiers in quadratic time. This result does not improve the bound found by ourself by reduction to the problem of Higher-Order Pattern Unification [Lev08]. The natural question now is: can we still improve this bound?

A careful analysis of the algorithm of Figure 1 shows us that it is basically a merge function, and that the complete check of the whole DAG of replacings is not very distinct from a merge-sort algorithm. In fact, if we could ensure that, when $L = L_1 \circ L_2$, we have $|\mathit{Rew}(L)| + |\mathit{For}(L)| \geq |\mathit{Rew}(L_1)| + |\mathit{For}(L_1)| + |\mathit{Rew}(L_2)| + |\mathit{For}(L_2)|$ and $|\mathit{Rew}(L_1)| + |\mathit{For}(L_1)| \approx |\mathit{Rew}(L_2)| + |\mathit{For}(L_2)|$, then the cost of the algorithm would be dominated by $T(n) = 2\,T(n/2) + \mathcal{O}(n)$ that has solution $\mathcal{O}(n \log n)$. If we could ensure $|\mathit{Rew}(L)| + |\mathit{For}(L)| \geq |\mathit{Rew}(L_1)| + |\mathit{For}(L_1)| + |\mathit{Rew}(L_2)| + |\mathit{For}(L_2)|$, but not the balance between the data structures of L_1 and L_2, then we could implement the sorted lists using AVL, and apply the ideas of Brown and Tarjan [Bro79] for merging of unbalanced sorted lists. This unbalance merge of two lists of sizes n_1 and n_2 can be done in time $\mathcal{O}(n_1 \log \frac{n_2}{n_1})$. Therefore, the time of the complete checking would be dominated by $T(n) = T(n_1) + T(n_2) + \mathcal{O}(n_1 \log \frac{n_2}{n_1})$, where $n = n_1 + n_2$. In this case, the solution is also $\mathcal{O}(n \log n)$. Therefore, we can conclude that we can check a set of replacings in time $\mathcal{O}(n \log n)$ on the size of the *tree* (not the DAG) representing the replacing. This means that, when the DAG is a tree, for instance in example 4.2, we can check the replacings in quasi-linear time.

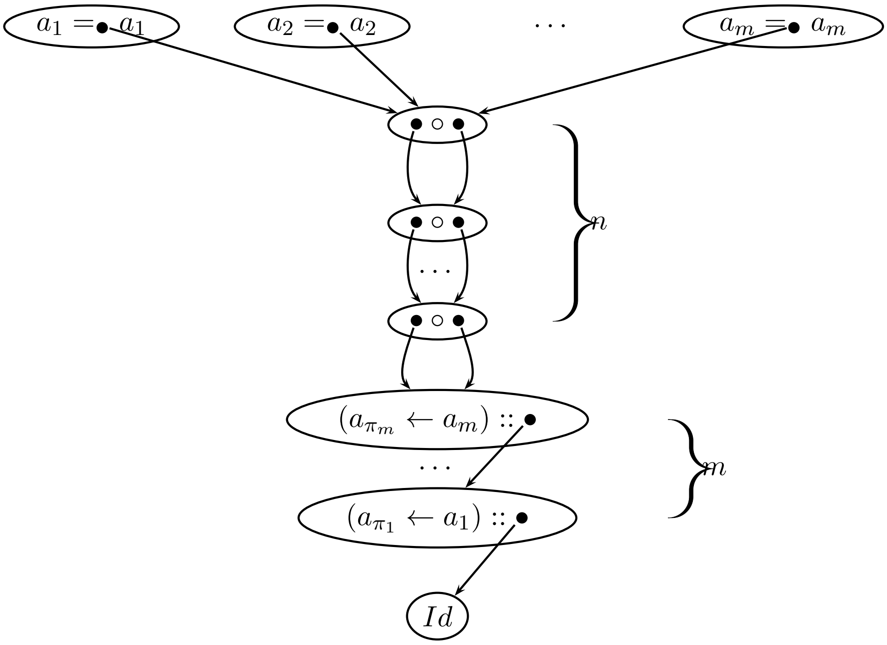
To conclude, consider the following example, that shows that the quadratic bound seems difficult to improve in the general case.

Example 9.1. Given a permutation π of $m = |\pi|$ elements, an a value n, we can construct the following two equation of size $\mathcal{O}(n+m)$

$$a_{\pi_1}.\cdots.a_{\pi_m}.f(f(\dots f(Y, X_n)\dots, X_2), X_1) \approx$$
$$a_1.\cdots.a_m.f(X_1, f(X_2, \dots f(X_n, Y)\dots))$$

$$Y \approx f(a_1, f(a_2, \dots f(a_{m-1}, a_m)\dots))$$

From these equations we get the following DAG. This problem is solvable, if we have $\pi^{2^n} = Id$. It seems difficult to answer this question in time faster than $\mathcal{O}(n\,m)$.



Appendix

Example 9.2 (Cont. of Example 4.2)**.** In fact, we would obtain the same result from the lazy application of the transformation rules of the nominal unification algorithm from [Urb04]. Those rules basically encode the inference rules presented in Section 2, namely we use here the ones for ≈-*abst-1* and ≈-*abst-2*. What we do is to delay the check for equality or difference between atoms of two abstractions because one of them can have a permutation applied on it, and we don't want to compute permutations until the end. By default we apply the rule for ≈-*abst-2* constraining the freshness predicate to the proviso of difference between abstractions.

$$
\begin{array}{l}
a_2.a_1.f(c_1,c_2) \overset{?}{\approx} (a_3 b_3)b_2.b_1.f(d_1,d_2) \\
a_3 \neq b_3 \implies a_3 \# b_2.b_1.f(d_1,d_2) \\
\hline \\
a_1.f(c_1,c_2) \overset{?}{\approx} (a_2(a_3 b_3)b_2)(a_3 b_3)b_1.f(d_1,d_2) \\
a_3 \neq b_3 \implies a_3 \# b_2.b_1.f(d_1,d_2) \\
a_2 \neq (a_3 b_3)b_2 \implies a_2 \# (a_3 b_3)b_1.f(d_1,d_2) \\
\hline \\
f(c_1,c_2) \overset{?}{\approx} (a_1(a_2(a_3 b_3)b_2)(a_3 b_3)b_1)(a_2(a_3 b_3)b_2)(a_3 b_3)f(d_1,d_2) \\
a_3 \neq b_3 \implies a_3 \# b_2.b_1.f(d_1,d_2) \\
a_2 \neq (a_3 b_3)b_2 \implies a_2 \# (a_3 b_3)b_1.f(d_1,d_2) \\
a_1 \neq (a_2(a_3 b_3)b_2)(a_3 b_3)b_1 \implies a_1 \# (a_2(a_3 b_3)b_2)(a_3 b_3)f(d_1,d_2) \\
\hline \\
c_1 \overset{?}{\approx} (a_1(a_2(a_3 b_3)b_2)(a_3 b_3)b_1)(a_2(a_3 b_3)b_2)(a_3 b_3)d_1 \\
c_2 \overset{?}{\approx} (a_1(a_2(a_3 b_3)b_2)(a_3 b_3)b_1)(a_2(a_3 b_3)b_2)(a_3 b_3)d_2 \\
a_3 \neq b_3 \implies a_3 \# b_2.b_1.f(d_1,d_2) \\
a_2 \neq (a_3 b_3)b_2 \implies a_2 \# (a_3 b_3)b_1.f(d_1,d_2) \\
a_1 \neq (a_2(a_3 b_3)b_2)(a_3 b_3)b_1 \implies a_1 \# (a_2(a_3 b_3)b_2)(a_3 b_3)f(d_1,d_2)
\end{array}
$$

Now, we get rid of the freshness constraints, translating them into disequalities by means of rules of the nominal unification algorithm from [Urb04] for the freshness predicates of Section 2.

$$
\begin{array}{l}
c_1 \overset{?}{\approx} (a_1(a_2(a_3 b_3)b_2)(a_3 b_3)b_1)(a_2(a_3 b_3)b_2)(a_3 b_3)d_1 \\
c_2 \overset{?}{\approx} (a_1(a_2(a_3 b_3)b_2)(a_3 b_3)b_1)(a_2(a_3 b_3)b_2)(a_3 b_3)d_2 \\
a_3 \neq b_3 \wedge a_3 \neq b_2 \wedge a_3 \neq b_1 \implies (a_3 \neq d_1 \wedge a_3 \neq d_2) \\
a_2 \neq (a_3 b_3)b_2 \wedge a_2 \neq (a_3 b_3)b_1 \implies (a_2 \neq (a_3 b_3)d_1 \wedge a_2 \neq (a_3 b_3)d_2) \\
a_1 \neq (a_2(a_3 b_3)b_2)(a_3 b_3)b_1 \implies (a_1 \neq (a_2(a_3 b_3)b_2)(a_3 b_3)d_1 \wedge a_1 \neq (a_2(a_3 b_3)b_2)(a_3 b_3)d_2)
\end{array}
$$

References

[Bro79] Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *J. of the ACM*, 26(2):211–226, 1979.

[Cal07] Christophe Calvès and Maribel Fernández. Implementing nominal unification. *ENTCS*, 176(1):25–37, 2007.

[Cal08] Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theoretical Computer Science*, 403(2-3):285–306, 2008.

[Cal10] Christophe Calvès. *Complexity and Implementation of Nominal Algorithms.* Ph.D. thesis, King's College London, 2010.

[Che04] James Cheney and Christian Urban. α-prolog: A logic programming language with names, binding and α-equivalence. In *Proc. of the 20th Int. Conf. on Logic Programming, ICLP'04*, *LNCS*, vol. 3132, pp. 269–283. 2004.

[Che05] James Cheney. Equivariant unification. In *Proc. of the 16th Int. Conf. on Term Rewriting and Applications, RTA'05*, *LNCS*, vol. 3467, pp. 74–89. 2005.

[Clo07] R. Clouston and A. Pitts. Nominal equational logic. *ENTCS*, 1496:223–257, 2007.

[Dow09] Gilles Dowek, Murdoch Gabbay, and Dominic Mulligan. Permissive nominal terms and their unification. In *Proc. of the 24th Convegno Italiano di Logica Computazionale, CILC'09.* 2009.

[Dow10] Gilles Dowek, Murdoch Gabbay, and Dominic Mulligan. Permissive nominal terms and their unification. *Logic Journal of the IGPL*, 2010.

[Fer05] Maribel Fernández and Murdoch Gabbay. Nominal rewriting with name generation: abstraction vs. locality. In *Proc. of the 7th Int. Conf. on Principles and Practice of Declarative Programming, PPDP'05*, pp. 47–58. 2005.

[Fer07] Maribel Fernández and Murdoch Gabbay. Nominal rewriting. *Information and Computation*, 205(6):917–965, 2007.

[Gab01] Murdoch Gabbay and A. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2001.

[Gab06] Murdoch Gabbay and Aad Mathijssen. Nominal algebra. In *Proc. of the 18^{t}h Nordic Workshop on Programming Theory, NWPT'06.* 2006.

[Gab07] Murdoch Gabbay and Aad Mathijssen. A formal calculus for informal equality with binding. In *Logic, Language, Information and Computation*, *LNCS*, vol. 4576, pp. 162–176. Springer, 2007.

[Gab09] Murdoch Gabbay and Aad Mathijssen. Nominal (universal) algebra: equational logic with names and binding. *Journal of Logic and Computation*, 19(6):1455–1508, 2009.

[Lev08] Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In *Proc. of the 19th Int. Conf on Rewriting Techniques and Applications, RTA'08*, *LNCS*, vol. 5117, pp. 246–260. 2008.

[Mar82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[Pat78] Mike Paterson and Mark N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.

[Pit01] Andrew Pitts. Nominal logic: A first order theory of names and binding. In *Proc. of the 4th Int. Symp. on Theoretical Aspects of Computer Software, TACS'01*, *LNCS*, vol. 2215, pp. 219–242. 2001.

[Pit03] A. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[Qia96] Zhenyu Qian. Unification of higher-order patterns in linear time and space. *J. of Logic and Computation*, 6(3):315–341, 1996.

[Urb03] C. Urban, A. Pitts, and M. Gabbay. Nominal unification. In *Proc. of the 17th Int. Work. on Computer Science Logic, CSL'03*, *LNCS*, vol. 2803, pp. 513–527. 2003.

[Urb04] C. Urban, A. Pitts, and M. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.

[Urb05] Christian Urban and James Cheney. Avoiding equivariance in alpha-prolog. In *Proc. of the Int. Conf. on Typed Lambda Calculus and Applications, TLCA'05*, *LNCS*, vol. 3461, pp. 401–416. 2005.