Available online at www.sciencedirect.com
ScienceDirect

Electronic Notes in Theoretical Computer Science 176 (2007) 25–37

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Implementing Nominal Unification

Christophe Calvès and Maribel Fernández[1] ,[2]

King's College London, Department of Computer Science
Strand, London WC2R 2LS, UK

Abstract

Nominal matching and unification underly the dynamics of nominal rewriting. Urban, Pitts and Gabbay gave a nominal unification algorithm which finds the most general solution to a nominal matching or unification problem, if one exists. Later the algorithm was extended by Fernández and Gabbay to deal with name generation and locality.
In this paper we describe first a direct implementation of the nominal unification algorithm, including the extensions, in Maude. This implementation is not efficient (it is exponential in time), but we will show that we can obtain a feasible implementation by using termgraphs.

Keywords: Unification, Nominal Syntax, Termgraphs

1 Introduction

Nominal terms [10] are trees with internal nodes labelled by term-constructors (also called function symbols), and leaves labelled by variables or constants, where:

- The set of variables is partitioned into a set of *atoms* $a, b, c, \ldots$; and a set of *unknowns* (or just *variables*) $X, Y, Z, \ldots$ which can be annotated with *swappings* $(a\ b)$ of atoms.
- There is a special term-constructor called *abstraction* and written $[a]t$ where a is an atom.

On nominal terms, we can define by induction a *freshness relation* $a\#t$ (read "the atom a is fresh for the term t") which roughly corresponds to the notion of a not occurring free in t. Using freshness and swappings we can inductively define a notion of α-equality of terms. Since t may contain variables, in order to deduce $a\#t$ we might need to use assumptions $a\#X$. For instance, we will see that it is

[1] This work has been partially funded by EPSRC grant EP/D501016/1 "Computational Applications of Nominal Sets".

[2] Email: Maribel.Fernandez,Christophe.Calves@kcl.ac.uk

doi:10.1016/j.entcs.2006.09.027

possible to deduce

$$a\#X, a\#Y \vdash a\#f(X, Y, [a]Z)$$
$$a\#X, b\#X \vdash [a]X \approx_\alpha [b]X$$

Nominal unification is the problem of deciding whether two nominal terms can be made α-equal by instantiating their variables. Urban, Gabbay and Pitts showed that this problem is decidable, and gave an algorithm which finds the most general unifier of two terms, when one exists [10]. The unification algorithm is specified in [10] as a set of transformation rules (or rewrite rules) on unification problems. Later, this algorithm was extended to deal with terms containing an additional operator for name generation (written $\mathsf{N}a.t$, and meaning that the name a is local to t), and with locality constraints $a@t$ (read "a is local in t") [4]. The extended nominal unification algorithm is also defined as a set of rewrite rules on problems.

It was conjectured in [10] that, although a direct implementation of the nominal unification algorithm is exponential, it would be possible to obtain a polynomial algorithm using directed acyclic graphs. In this paper we describe two implementations of nominal unification (we have actually implemented the extended nominal unification algorithm of [4]). The first one is a direct implementation of the rewrite rules in Maude [7]. Maude is a rewriting-logic based language, and since the nominal unification algorithm is specified as a set of rewrite rules, it is a natural choice for a first implementation. Also, since Maude is a high-level declarative language, programs are easy to understand and easy to extend (we first implemented the standard nominal unification algorithm, then extended it to deal with name generation).

This direct implementation is not an efficient one (as expected). It is well-known that first-order unification, which is a particular case of nominal unification, is exponential if subterms are not shared, but it is linear if terms are represented as graphs with maximal sharing. Our second implementation of the nominal unification algorithm, written in OCaml [8], is based on the use of termgraphs instead of terms. There is an additional complication with respect to first-order unification, in that to obtain a polynomial algorithm we also have to share subterms up to permutations of atoms. Because of the additional operations on permutations, our second implementation is not linear, but we can show that these additional computations are polynomial.

To summarise, the contributions of this paper are:

- Two implementations of (extended) nominal unification: the first one is simple but inefficient, the second is still high-level and polynomial. We remark that the actual complexity of nominal unification is not known; higher-order pattern unification, which is a closely related problem [1], is linear [9].
- The definition of a notion of nominal termgraph, which we use as our data structure for representation of nominal terms and rules in the second implementation of nominal unification. The theory of nominal termgraphs and nominal termgraph rewriting has not been developed yet, and will be the subject of future research.

Related Work

The nominal unification algorithm (or more precisely, nominal matching) was used by Fernández, Gabbay and Mackie to define a notion of rewriting on nominal terms in [5]. The implementation of unification and matching discussed in this paper is a first step towards obtaining an evaluator for nominal rewriting systems.

Nominal unification has practical applications in typing algorithms (see for instance [6]) and it is at the heart of α-Prolog [3], an extension of Prolog with binders using the nominal approach. α-Prolog has been implemented in OCaml, including an implementation of the nominal unification algorithm. This implementation uses trees to represent terms, and is exponential in time and quadratic in space [2]. Pottier has developed another OCaml implementation of nominal unification (private communication) without name generation and locality constraints, which appears to be polynomial. Our implementation does not rely on side-effects and does not use imperative features of Ocaml (except for name generation); instead, it rewrites the graphs used to represent unification problems.

2 Background

We recall the syntax of nominal terms, define nominal unification problems and give the rewriting rules described in [10,4] to solve them.

Let Σ be a denumerable set of **function symbols** f, g, ..., $\mathcal{X}$ be a denumerable set of **variables** $X, Y, \ldots$ (representing meta-level unknowns) and $\mathcal{A}$ be a denumerable set of **atoms** a, b, c, n (representing object-level variable symbols). We assume that Σ, $\mathcal{X}$ and $\mathcal{A}$ are pairwise disjoint. A **swapping** is a pair of atoms, which we write $(a\ b)$. **Permutations** π are lists of swappings, generated by the grammar

$$\pi ::= \mathtt{Id} \mid (a\ b)\cdot\pi.$$

We call `Id` the **identity permutation**. We call a pair of a permutation π and a variable X a **moderated variable** or a **suspension** and write it $\pi\cdot X$. We say that π is **suspended** on X. We write π^{-1} for the permutation obtained by reversing the list of swappings in π. We denote by $\pi \circ \pi'$ the permutation containing all the swappings in π followed by those in π'.

Nominal terms, or just **terms** for short, over $\Sigma, \mathcal{X}$ are generated by the grammar:

$$s, t ::= a \mid \pi\cdot X \mid (s_1, \ldots, s_n) \mid [a]s \mid (f\ t)$$

and are called respectively **atoms**, **moderated variables** (or just **variables** for short), **tuples**, **abstractions** and **function applications**. We refer the reader to [10,4] for more details and examples of nominal signatures and terms. Note that although X is not a term, $\mathtt{Id}\cdot X$ is, and we will abbreviate it as X when there is no ambiguity.

We can apply permutations and substitutions on terms, denoted $\pi\cdot t$ and $t[X\mapsto s]$ respectively. The action of a permutation on a term is defined by induction, with base cases: $\texttt{Id}{\cdot}t = t$ and

$$(a\ b){\cdot}a = b \quad (a\ b){\cdot}b = a \quad (a\ b){\cdot}c = c$$
$$(a\ b){\cdot}(\pi{\cdot}X) = ((a\ b) \circ \pi){\cdot}X \quad (a\ b){\cdot}(f\ t) = f(a\ b){\cdot}t \quad (a\ b){\cdot}[n]t = [(a\ b){\cdot}n](a\ b){\cdot}t$$
$$(a\ b){\cdot}(t_1, \ldots, t_n) = ((a\ b){\cdot}t_1, \ldots, (a\ b){\cdot}t_n)$$

where c is assumed to be different from a, b.

A **substitution** is generated by the grammar

$$\sigma ::= \texttt{Id} \mid [X{\mapsto}s]\sigma.$$

We write substitutions postfix and write $\circ$ for composition of substitutions: $t(\sigma \circ \sigma') = (t\sigma)\sigma'$. We define the instantiation of a term t by a substitution σ by induction as follows:

$$t\texttt{Id} = t \qquad t[X{\mapsto}s]\sigma = (t[X{\mapsto}s])\sigma$$

where

$$a[X{\mapsto}s] = a \quad (ft)[X{\mapsto}s] = f(t[X{\mapsto}s]) \quad ([a]t)[X{\mapsto}s] = [a](t[X{\mapsto}s])$$
$$(t_1, \ldots, t_n)[X{\mapsto}s] = (t_1[X{\mapsto}s], \ldots, t_n[X{\mapsto}s])$$
$$(\pi{\cdot}X)[X{\mapsto}s] = \pi{\cdot}s \quad (\pi{\cdot}Y)[X{\mapsto}s] = \pi{\cdot}Y$$

Note that permutations act top-down and accumulate on moderated variables whereas substitutions act on the variable symbols in the moderated variables.

The predicate # specifies a **freshness** relation between atoms and terms, and $\approx_\alpha$ denotes **alpha-equality**.

Constraints have the form: $a\#t$ or $s \approx_\alpha t$. A set Pr of constraints will be called a **problem**.

We give below an algorithm to *check* constraints, which is specified by a set of simplification rules acting on problems where a, b denote any pair of distinct atoms, $\pi{\cdot}X$ denotes a moderated variable, f a function symbol and ds denotes the difference set of two permutations, i.e., the set of atoms in which they differ:

$$ds(\pi, \pi') = \{n \mid \pi{\cdot}n \neq \pi'{\cdot}n\}$$

We write $ds(\pi, \pi')\#X$ as an abbreviation for $\{n\#X \mid n \in ds(\pi, \pi')\}$.

Simplification Rules on Problems:

$$
\begin{array}{rcll}
a\#b, Pr & \Longrightarrow & Pr & \\
a\#fs, Pr & \Longrightarrow & a\#s, Pr & \\
a\#(s_1,\ldots,s_n), Pr & \Longrightarrow & a\#s_1,\ldots,a\#s_n, Pr & \\
a\#[b]s, Pr & \Longrightarrow & a\#s, Pr & \\
a\#[a]s, Pr & \Longrightarrow & Pr & \\
a\#\pi\cdot X, Pr & \Longrightarrow & \pi^{-1}\cdot a\#X, Pr & \quad \pi \neq \mathtt{Id}
\end{array}
$$

$$
\begin{array}{rcl}
a \approx_\alpha a, Pr & \Longrightarrow & Pr \\
(l_1,\ldots,l_n) \approx_\alpha (s_1,\ldots,s_n), Pr & \Longrightarrow & l_1 \approx_\alpha s_1,\ldots,l_n \approx_\alpha s_n, Pr \\
fl \approx_\alpha fs, Pr & \Longrightarrow & l \approx_\alpha s, Pr \\
[a]l \approx_\alpha [a]s, Pr & \Longrightarrow & l \approx_\alpha s, Pr \\
[b]l \approx_\alpha [a]s, Pr & \Longrightarrow & (a\ b)\cdot l \approx_\alpha s, a\#l, Pr \\
\pi\cdot X \approx_\alpha \pi'\cdot X, Pr & \Longrightarrow & ds(\pi,\pi')\#X, Pr
\end{array}
$$

These rules define a reduction relation on problems: We write $Pr \Longrightarrow Pr'$ when Pr' is obtained from Pr by applying a simplification rule, and we write $\overset{*}{\Longrightarrow}$ for the transitive and reflexive closure of $\Longrightarrow$.

The algorithm to check constraints is defined as follows: Given a problem Pr, we apply the rules until we get an irreducible problem. If only a set Δ of constraints of the form $a\#X$ are left, then the original problem is valid in the context Δ (i.e., $\Delta \vdash Pr$), otherwise it is not valid. Note that a problem such as $X \approx_\alpha a$ is therefore not valid since it is irreducible. However, X can be made equal to a by *instantiation*; we say that this constraint can be *solved*.

A most general **solution** to a problem Pr is a *pair* (Γ, σ) obtained using an algorithm derived from the simplification rules above, enriched with **instantiating** rules, labelled with substitutions:

$$
\begin{array}{rcll}
\pi\cdot X \approx_\alpha u, Pr & \overset{X\mapsto\pi^{-1}\cdot u}{\Longrightarrow} & Pr[X\mapsto\pi^{-1}\cdot u] & \quad (X \notin V(u)) \\
u \approx_\alpha \pi\cdot X, Pr & \overset{X\mapsto\pi^{-1}u}{\Longrightarrow} & Pr[X\mapsto\pi^{-1}\cdot u] & \quad (X \notin V(u))
\end{array}
$$

The conditions in the instantiating rules are usually called **occurs check**.

We obtain in this way a correct and complete nominal unification algorithm. We refer to [10] for more details and examples.

The syntax of nominal terms was extended in [4] with an operator И to model name generation, and with a new kind of constraint to express locality (written $a@t$). Extended terms have the form $\text{И}A.t$ where A is a set of **local** names in t, and t is a nominal term which may contain И but not at the top level. We omit the unification rules dealing with И and locality constraints here, although we have implemented them. We refer the reader to [4] for the extension of the unification algorithm to terms with И.

3 Direct Implementation in Maude

Maude is a rewriting-logic programming language which supports both equational and rewriting specification and programming. As such, it is well-adapted to implementing algorithms specified as rewriting systems. A Maude program consists of a signature description (which specifies the syntax of terms and their sorts), and a set of equational and rewriting rules on terms. We refer the reader to [7] for more details on Maude.

Nominal terms can be easily encoded in Maude using the nominal signature plus some book-keeping equational rules to simulate their behaviour. We will show that a matching or unification problem is coded as easily. Below we describe our implementation of nominal unification in Maude.

3.1 Nominal Terms in Maude

Nominal terms are defined in a functional module called NOM-TERM, which defines the sorts `VarSusp`, `NomTermStruct`, `NomTermRed` and `NomTerm`:

```
sorts VarSusp   NomTermStruct   NomTermRed   NomTerm .
subsorts Var < VarSusp .
subsorts Atm VarSusp < NomTermStruct < NomTermRed < NomTerm
```

`NomTerm` is the sort of nominal terms in general and `NomTermRed` is the sort of terms simplified by some of the rules, which are also defined in NOM-TERM. The sorts of atoms (`Atm`) and suspensions (`VarSusp`)(i.e. variables with permutations) are defined in other modules, above we indicate that they are subsorts of `NomTermStruct` which is a subsort of `NomTermRed` which in turn is a subsort of `NomTerm`. The sort of variables (`Var`) is a subsort of suspensions (`VarSusp`). We omit the declaration of other sorts, such as `Set{AtmV}`, which is the sort of sets of atoms (used to build terms with И), `Fct`, which is the sort of function symbols, and `Perm` (permutations).

To mimic the syntax of nominal terms we declare the following operators, where `unit` is the empty product, _(_) is the function operator, _^_ is the permutation application operator, `abs` is the abstraction operator, `N` the И, _;_ and `tpl` are for tuples:

```
op unit : -> NomTerm [ctor] .
op _;_ : [NomTerm] [NomTerm] -> [NomTerm] [ctor assoc id: unit] .
op tpl : NomTerm -> NomTerm [ctor] .
op _(_) : Fct NomTerm -> NomTermStruct [ctor] .
op _^_ : Perm NomTerm -> NomTerm [ctor] .
op abs : Atm NomTerm -> NomTermStruct [ctor] .
op N : Set{AtmV} NomTerm -> NomTerm [ctor]
```

For example, if `a, b` are Maude terms of sort `Atm` (atoms); `f, g` are Maude terms of sort `Fct` (function name); `X, Y, Z` are variable names; `p, q` are Maude terms of sort `Perm` (permutations) and `A,B` are Maude terms of sort `Set{AtmV}` (set of atoms) then:

```
N(A , abs(a , f(a,X)))
N(a , g( abs(b,Y) ; Z))
p ^ f(N(B,g(a)); q ^ Z)
```

are Maude terms of sort `NomTerm`.

3.2 Simulating the behaviour of nominal terms

Terms of sort `NomTerm` do not correspond exactly to nominal terms. For example, $\text{И}A.\text{И}B.t$ is not a nominal term (only one И can occur at the root of an extended nominal term) and neither is $\pi{\cdot}(f\ t)$ (the permutation must be pushed to the variables in t), but Maude will consider such terms as valid terms of sort `NomTerm`. To obtain a direct correspondence between Maude terms and nominal terms, we have included in NOM-TERM rules that define equivalence classes of terms (Maude terms will be simplified by these rules). We give some examples below:

```
eq tpl(unit) = unit .

eq N(empty , t) = t .
eq N(A, (N(B , t))) = N( (A , B) , t ) .

eq id ^ t = t .
eq p ^ unit = unit
```

3.3 Unification Rules in Maude

Freshness, locality and α-equivalence predicates are defined as operators with the following syntax:

```
sorts    Fresh Local Alpha Contr .
subsorts Fresh Local Alpha < Contr .

op _#_ : Set{AtmV} NomTerm -> Fresh [ctor] .
op _@_ : Set{AtmV} NomTerm -> Local [ctor] .
op _~_ : NomTerm NomTerm -> Alpha [ctor] .
```

where `Fresh, Local` and `Alpha` are the sorts of freshness, locality and α-equivalence constraints respectively. `Contr` is the union of these sorts.

A unification problem is a set of terms of sort `Contr`. We give below some of the rewrite rules defined on those sets:

```
*** Some Freshness and Locality Rules
eq A # (t ; u ) =      (A # t) (A # u) .
eq A @ (t ; u ) =      (A @ t) (A @ u) .

eq A # tpl(t) =       A # t .
eq A @ tpl(t) =       A @ t .

eq A # (f ‘( t ‘) ) =       A # t .
eq A @ (f ‘( t ‘) ) =       A @ t .

eq A # abs(a , t ) = (A \ a) # t .
eq A @ abs(a , t ) = (A \ a) @ t .

eq A # N(B,t) =       A # t .
eq A @ N(B,t) = (A \ B) @ t .

eq A # (p ^ t) = (perm-inv(p) ^ A) # t .
eq A @ (p ^ v) = (perm-inv(p) ^ A) @ v .


*** Some Alpha-Equality rules

eq  t ~ t = empty .
eq (p ^ t) ~ u = t ~ (perm-inv(p) ^ u) .
eq N(A,(p ^ t)) ~ u = N((perm-inv(p) ^ A) , t) ~ (perm-inv(p)^u) .

eq  tpl(t) ~ tpl(u) =  t ~ u .
eq (t ; u) ~ (tp ; up) = (t ~ tp) (u ~ up) .
eq (f ‘( t ‘) ) ~ (f ‘( u ‘) ) =  t ~ u .
eq abs(a,t) ~ abs(b,u ) = (b # abs(a , t))(((a - b) ^ t) ~ u) .

eq v ~ (p ^ v) = perm-supp(p) # v .
```

Each of these Maude rules corresponds to a rule in the unification algorithm (see Section 2), therefore the correctness of the implementation is easy to prove. Note that we use sets of atoms in constraints for efficiency.

Finally, we have defined in Maude the functions `matching` and `unifying` which take a term representing a matching (resp. unification) problem and reduce it with the rules above.

To solve a nominal matching or a nominal unification problem we simply call `matching` or `unifying` in the Maude environment, as in the following examples (note that atoms, function symbols and variables are coded `a(s), f(s), v(s)` respectively, where `s` is the name of the atom, function or variable, i.e. a string):

```
(red matching( N(a("a") , abs(a("b") ,
 f("f")(a("b");v("X")))) ~ abs(a("c") ,
 f("f")(a("c");a("a"))) ) .)
```

produces the following result:

```
result Matching :
  matching(a("a")@ a("a"),v("X")-> a("a"))
```

meaning that the result is $a@a$ with the substitution of X by a. Since $a@a$ is false, this means that there is no solution for this problem.

To unify use:

```
(red unifying( N(a("a") , abs(a("b") ,
 f("f")(a("b");v("X")))) ~
 abs(a("c") , f("f")(a("c");a("a"))) ) .)
```

The code of `matching` is

```
 eq matching( C ) = matching( C , empty )

ceq matching( ((v ~ t) , C) , S ) =
       matching( subst(v , t , C) ,
         (subst(v , t , S) , (v -> t)))
   if not occurcheck(v , t)

ceq matching( (((p ^ v) ~ t) , C) , S ) =
       matching( subst(v , (perm-inv(p) ^ t) , C) ,
         (subst(v , (perm-inv(p) ^ t) , S) , (v -> t)))
   if not occurcheck(v , t) .
```

where `subst` is a substitution and `occurcheck` is the occurrence checking.

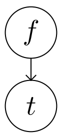
This implementation is simple, easy to understand and maintain, but is exponential in time (even for pure first-order unification problems). In the following section we discuss an alternative implementation, using graphs to represent terms.

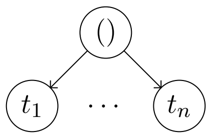
4 Using Directed Acyclic Graphs

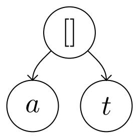
The naive implementation described above is simple but inefficient. To improve it, we have changed the data structure used to represent terms and unification problems: a whole nominal unification problem will be represented as a single directed acyclic graph. In this way we obtain a polynomial algorithm. This algorithm has been implemented in OCaml (a strongly typed, strict, functional programming language, with support for imperative features and object-oriented design; see [8] for more details). In the rest of the section we highlight the main difficulties encountered and the techniques used.

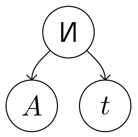
4.1 From Terms to Graphs

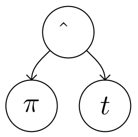
Nominal terms, and constraints, are inductively transformed into graphs as follows, where $\textcircled{t}$ represents the translation of the term t (i.e., we give an inductive definition).

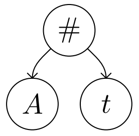
$f(t)$:


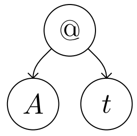
$(t_1, \ldots, t_n)$:


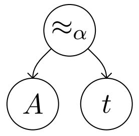
$[a].t$


$\text{И}A.t$:


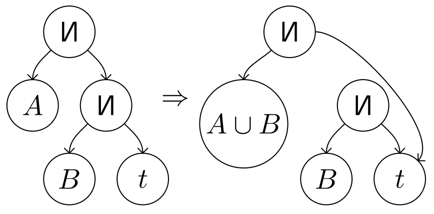
$\pi \,\hat{}\, t$:


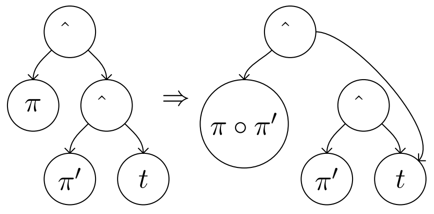
$A \# t$:


$A @ t$:


$t \approx_\alpha u$:


During the unification process, these graphs will be simplified and kept in normal form with respect to a correct, terminating and confluent set of rules (for which there is a strategy which computes normal forms in polynomial time). For example, two of these normalisation rules are:

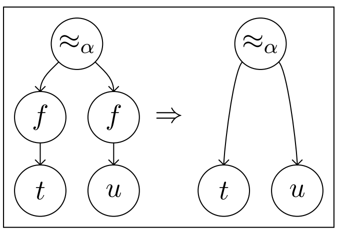




Rewriting rules on terms can be automatically transformed into graph rewriting rules. For example the unification rule

$$f(t) \approx_\alpha f(u) \to t \approx_\alpha u$$

is transformed into the graph rewriting rule:



We omit the other rules (the transformation is straightforward).

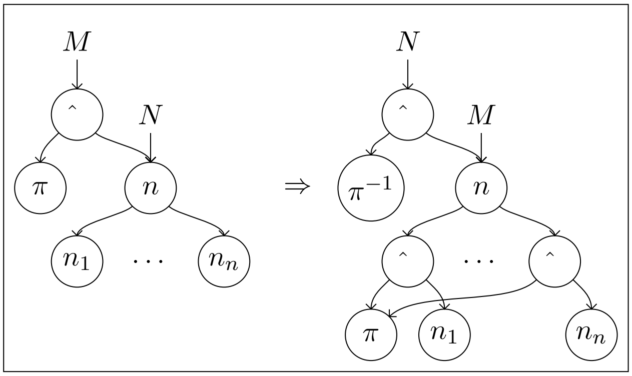
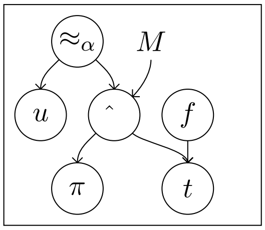


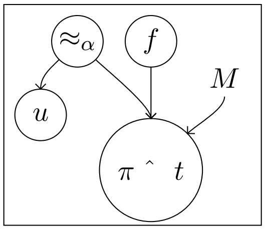
Fig. 1. Repositioning Permutations

4.2 Alpha-equivalence and Re-positioning of Permutations

The advantage of the graph representation is that it enables term sharing, but to obtain a good degree of sharing in nominal syntax, permutations have to be handled with care. In nominal terms, permutations are "eager" in the sense that they are automatically pushed to the leaves of the tree, they only suspend on variables. To get more sharing, permutations should be applied "lazily", but we must take into account the fact that we may be forced to apply permutations in order to be able to apply unification rules. The application of a permutation on a shared subterm is problematic: a naive application would be incorrect, or break sharing, as the following example shows. Consider the term:



Applying π on t gives :



and $f(t)$ becomes $f(\pi\ \hat{}\ \ t)$ which is incorrect.

To solve this problem without duplicating t (which is crucial to avoid the exponential explosion) we use a technique based on the *re-positioning* of permutations on terms, keeping maximal term sharing. The operation of re-positioning of permutations is described in Figure 1.

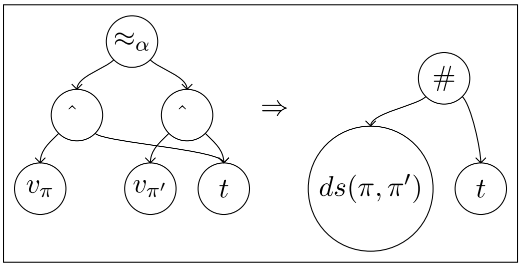
Note that this operation has to be carefully controlled: indeed, it is easy to see that it could be applied again in the right-hand side in Figure 1, leading to non-termination. We avoid this problem by allowing the application of re-positioning to a node n only when this enables the application of a unification rule R involving that node, and R is applied immediately after re-positioning. This technique may be generalised on bijective functions on terms.

4.3 Introducing Additional Rules on Graphs

In addition to the re-positioning technique mentioned above, we also need to add specific rules for α-equivalence on pointers to be efficient. For instance, we have an additional graph rewriting rule corresponding to:

$$\pi \ \hat{}\ \ t \approx_\alpha \pi' \ \hat{}\ \ t \rightarrow ds(\pi, \pi') \# t$$

depicted below.



Freshness and locality constraints can be solved in polynomial time by a set of graph rewriting rules which memorises the set of fresh atoms and the set of local atoms for each node, to avoid repeating computations.

4.4 Complexity of the Algorithm on Graphs

First we remark that the use of termgraphs with maximal sharing implies that we never duplicate subgraphs, hence the algorithm is linear in space.

The number of graph rewrites required to solve a unification problem is polynomial in the size of the graph (i.e. the size of the unification problem). However, in contrast with first-order unification where the algorithm based on termgraphs is linear, we could not achieve linear time because before applying a graph rewriting rule we might need to normalise permutations and И and apply re-positioning (but these are all polynomial operations).

5 Conclusions and Future Work

We have implemented in Maude a nominal unification algorithm for extended nominal terms (the first implementation of extended nominal unification), using a direct encoding of nominal terms into Maude terms, and unification rules into Maude rules. We have then discussed a better implementation using graphs to represent nominal terms and unification problems. This second implementation, which provides a polynomial nominal unification algorithm, is available from

www.dcs.kcl.ac.uk/staff/maribel/papers.html

The complexity of nominal unification is still an open problem. Unification of higher-order patterns, a closely related problem (see [1]), is linear [9] and there is therefore hope that nominal unification could also be linear. This is a challenging area for future work.

Acknowledgement

We thank James Cheney, Murdoch J. Gabbay, Ian Mackie, Andrew Pitts, François Pottier and Christian Urban for interesting discussions on the topics of this paper, and the anonymous referees for useful suggestions to improve the paper.

References

[1] J. Cheney. Relating nominal and higher-order pattern unification. Proceedings of UNIF 2005, p. 104–119.

[2] J. Cheney. Private communication, 2006.

[3] J. Cheney, C. Urban. AlphaProlog: A Logic Programming Language with Names, Binding and alpha-equivalence. ICLP 2004, p. 269–283.

[4] M. Fernández, M. Gabbay. Nominal rewriting with name generation: Abstraction vs. locality. Proceedings of the 7th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'05), Lisbon, Portugal, ACM Press, 2005.

[5] M. Fernández, M. Gabbay, I. Mackie. Nominal rewriting systems. Proceedings of the 6th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'04), Italy, ACM Press, 2004.

[6] N. Gauthier, F. Pottier. Numbering matters: first-order canonical forms for second-order recursive types. Proceedings of the 2004 ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'04), p. 150–161, 2004.

[7] Maude, http://maude.cs.uiuc.edu/

[8] OCAML, http://caml.inria.fr/

[9] Z. Qian. Unification of Higher-order Patterns in Linear Time and Space. Journal of Logic and Computation 6: 315-341, 1996.

[10] C. Urban, A. M. Pitts, M. J. Gabbay, Nominal unification, Theoretical Computer Science 323, 473 – 497, 2004.