A Certified Functional Nominal C-Unification Algorithm*

Mauricio Ayala-Rincón[1]**, Maribel Fernández[2], Gabriel Ferreira Silva[1]***, and Daniele Nantes-Sobrinho[1]

[1] Departments of Computer Science and Mathematics, Universidade de Brasília
[2] Department of Computer Science, King's College London

Abstract. The nominal approach allows us to extend first-order syntax and represent smoothly systems with variable bindings using, instead of variables, nominal atoms and dealing with renaming through permutations of atoms. Nominal unification is, therefore, the extension of first-order unification modulo α-equivalence by taking into account this nominal setting. In this work, we present a specification of a nominal C-unification algorithm (nominal unification with commutative operators) in PVS and discuss the proofs of soundness and completeness. Additionally, the algorithm has been implemented in Python. In relation to the only known specification of nominal C-unification, there are two novel features in this work: first, the formalization of a functional algorithm that can be directly executed (not just a set of non-deterministic inference rules); second, simpler proofs of termination, soundness and completeness, due to the reduction in the number of parameters of the lexicographic measure, from four parameters to only two.

Keywords: Nominal Terms, Nominal C-Unification, Verification of Functional Specifications

1 Introduction

The nominal approach allows us to extend first-order syntax and represent smoothly systems with bindings, which are frequent in computer science and mathematics. However, in order to represent bindings correctly, α-equivalence must be taken into account. For instance, despite their syntactical difference, the formulas $\exists x : x < 0$ and $\exists z : z < 0$ should be considered equivalent. The nominal theory allows us to deal with these bindings in a natural way, using atoms, atom permutations and freshness constraints, instead of using indices as in explicit substitutions *à la de Bruijn* (e.g. [20], [14]).

* Work supported by FAPDF grant 193001369/2016.
** Corresponding author, partially funded by CNPq research grant number 307672/2017-4.
*** Corresponding author, was funded by CNPq scholarship number 139271/2017-1. Work presentation funded by FAPDF.

Unification is an important problem in first-order theories, with applications to logic programming systems, type inference algorithms, theorem provers and so on (e.g. [12]). Since unification is essential for equational reasoning, the development of unification techniques for nominal logic has been an attractive area of research since the invention of the nominal approach.

The problem of nominal unification has been solved by Urban et al. ([23]), with further research devoted to algorithm improvements (e.g. [10], [17]). Extensions of nominal unification have also been studied (e.g [21], [16], [9]), among them nominal unification modulo equational theories (e.g. [2], [1], [5], [3]). Here, we consider nominal unification modulo commutative function symbols (nominal C-unification, for short).

Related Work. Nominal unification was first solved by Urban et al. in [23], by proposing a set of transformation rules and showing, using Isabelle/HOL, their correctness and completeness [22]. An alternative specification of nominal unification, as a function that maps (solvable) problems to solutions, was formalised in PVS and proved sound and complete [6]. This work brought two new perspectives to the problem. The first is the specification of a functional algorithm for nominal unification, not a set of inference rules, which made the specification closer to the implementation. The second is the separate treatment of freshness constraints and equational constraints. Both ideas are used in this paper.

Nominal C-unification extends nominal unification to handle commutative function symbols. In previous work, a set of non-deterministic transformation rules to solve nominal C-unification problems, in the style of Urban et al. [22], was shown correct and complete using the Coq proof assistant [2].

Contribution. In this paper, we present the first (to our knowledge) functional nominal C-unification algorithm and formalize its correctness and completeness using the proof assistant PVS. We emphasize the most interesting aspects of its formalization.

Although there is one other specification ([2]) for nominal C-unification, the approaches taken are different. In [2], a set of rules that gradually transforms the nominal C-unification problem into simpler ones is presented. Here, by contrast, we develop a recursive algorithm, specified and formalized in PVS and implemented in Python, not a set of inference rules. The advantage of this approach is that the algorithm can be executed, while the set of inference rules, specified through inductive definitions cannot, because it is non-deterministic.

As mentioned previously, [6] gave us a nominal unification algorithm and a new insight about the problem: the possibility to handle freshness constraints and equational constraints separately. We adapt a significant portion of the formalization in [6], adding and formalizing the necessary lemmas to obtain a sound and complete algorithm for nominal C-unification and keep the separate treatment of freshness constraints and equational constraints. This insight, along with a trick of separating fixed point equations (see Definition 9) from the unification problem, allowed us to reduce the lexicographic measure found in [2], from 4 parameters to only 2 parameters which made the proofs of termination, soundness and completeness simpler.

Finally, the formalization of soundness and completeness was done in PVS and is available at `http://www.github.com/gabriel951/c-unification`. PVS was chosen partly in order to reuse the definitions and lemmas previously used in [6] and partly because its specification language provides great support for the definition (and formalization) of functional recursive algorithms.

Possible Applications. As remarked before, nominal unification is used in logic programming. Therefore, the nominal C-unification algorithm could be used on a logic programming language that uses the nominal setting, such as α-Prolog [12]. Another application is related to matching (see [8], [7]). Matching two terms t and s can be seen as unification where one of the terms (suppose t, without loss of generality) is not affected by substitutions [2]. This can be accomplished by adding as an additional parameter to the algorithm a set of variables that are forbidden to be instantiated. Then, matching boils down to unifying, using as this additional parameter the set of variables in t [2]. The C-matching algorithm proposed could then, for instance, be used to extend the nominal rewriting relation introduced in [14] modulo commutativity. Also, nominal C-unification and matching are relevant to implement nominal narrowing introduced in [4] allowing commutative symbols.

Organization. The paper is organized as follows. First, in Section 2, we provide the necessary background. The nominal setting is explained and the problem of nominal C-unification is defined. In Section 3, we present and explain the pseudocode for the algorithm specified in PVS and implemented in Python. In Section 4, we discuss the main aspects of the formalization: the principal lemmas, the hardest cases, how introducing commutativity made the problem more complex and so on. Finally, in Section 6, we conclude the paper and offer possible paths of future work. This paper is an extended version of a paper originally submitted to LOPSTR 2019.

2 Background

In this section, we provide the necessary background in nominal theory.

2.1 Nominal Terms, Permutations and Substitutions

In nominal theory, we consider disjoint countable sets of atoms $\mathcal{A} = \{a, b, c, ...\}$ and of variables $\mathcal{X} = \{X, Y, Z, ...\}$. A permutation π is a bijection of the form $\pi : \mathcal{A} \to \mathcal{A}$ such that the domain of π (i.e., the set of atoms that are modified by π) is finite. Permutations are usually represented as a list of swappings, where the swapping $(a\ b)$ exchanges the atoms a and b and fixes the other atoms. Therefore, a permutation is represented as $\pi = (a_1\ b_1) :: ... :: (a_n\ b_n) :: nil$. π^{-1}, the inverse of this permutation, is computed by simply reversing the list.

Definition 1 (Nominal Terms). *Let Σ be a signature with function symbols and commutative function symbols. The set $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{X})$ of nominal terms is generated according to the grammar:*

$$s, t \quad ::= \quad \langle\rangle \ \mid \ a \ \mid \ \pi \cdot X \ \mid \ [a]t \ \mid \ \langle s, t\rangle \ \mid \ f\ t \ \mid \ f^C\langle s, t\rangle \qquad (1)$$

where $\langle\rangle$ is the unit, a is an atom term, $\pi \cdot X$ is a moderated variable or suspension (the permutation π is suspended on the variable X), $[a]t$ is an abstraction (a term with the atom a abstracted), $\langle s, t\rangle$ is a pair, $f\ t$ is a function application and $f^C\langle s, t\rangle$ is a commutative function application over a pair.

Remark 1. Pairs can be used to encode tuples with an arbitrary number of arguments. For instance, the tuple (t_1, t_2, t_3) could be constructed as $\langle t_1, \langle t_2, t_3\rangle\rangle$.

Remark 2. Following [2], we impose that commutative functions receive a pair as their argument. No generality is lost with this restriction and the analysis is simplified.

Remark 3. We use syntactic sugar and represent suspended variables of the form $nil \cdot X$ simply as X.

The specification of a term in PVS is given below. We define a subtype `pair`, in order to guarantee that the argument of a commutative function is a pair. However, when working with subtypes in PVS, every term must have a subtype. Therefore, a subtype `plain` was also defined and every term that is not a pair has subtype `plain`.

```
term[atom: TYPE+, perm: TYPE+, variable: TYPE+, symbol: TYPE+,
comm_symbol: TYPE+ ]: DATATYPE WITH SUBTYPES plain, pair
  BEGIN
      at (a: atom): atom? : plain
      * (p: perm, V: variable): susp? : plain
      unit: unit? : plain
      pair (term1: term, term2: term): pair? : pair
      abs (abstr: atom, body: term): abs? : plain
      app (sym: symbol, arg: term): app? : plain
      c_app (c_sym: comm_symbol, c_arg: pair): c_app? : plain
  END term
```

Definition 2 (Permutation Action). *The permutation action on atoms is defined recursively: $nil \cdot c = c$ and $((a\ b) :: \pi) \cdot c = a$, if $\pi \cdot c = b$; $((a\ b) :: \pi) \cdot c = b$, if $\pi \cdot c = a$; $((a\ b) :: \pi) \cdot c = \pi \cdot c$ otherwise. The action of permutations on terms is defined recursively:*

$$\pi \cdot \langle\rangle = \langle\rangle \qquad \pi \cdot (\pi' \cdot X) = (\pi :: \pi') \cdot X$$

$$\pi \cdot [a]t = [\pi \cdot a]\pi \cdot t \qquad \pi \cdot \langle s, t\rangle = \langle \pi \cdot s, \pi \cdot t\rangle$$

$$\pi \cdot f\ t = f\ \pi \cdot t \qquad \pi \cdot f^C\langle s, t\rangle = f^C\langle \pi \cdot s, \pi \cdot t\rangle$$

Remark 4. The same symbol $\cdot$ occurs in a suspended variable $\pi \cdot X$ and in a permutation π acting on a term t. This notation is used to convey the idea that the permutation in a suspended variable is "suspended" in the variable and is "waiting" for the variable to be instantiated and will then act on the value that X is instantiated to. This is one possible notation, and there are authors that distinguish between the symbol in the suspended variable and the symbol of a permutation acting on a term.

Remark 5. We follow Gabbay's permutative convention, which says that atoms differ in their names. Therefore, if we consider atoms a and b, it is redundant to say $a \neq b$.

Example 1. To illustrate the action of a permutation on a term, consider $\pi = (a\ b) :: (c\ d) :: nil$ and $t = f(a, c)$. Then, the result of the permutation action is $\pi \cdot t = f(b, d)$.

Definition 3 (Nuclear Substitution). *A nuclear substitution is a pair $[X \to t]$, where X is a variable and t is a term. Nuclear substitutions act over terms:*

$$\langle\rangle[X \to t] = \langle\rangle \qquad a[X \to t] = a$$

$$([a]s)[X \to t] = [a](s[X \to t]) \qquad \pi \cdot Y[X \to t] = \begin{cases} \pi \cdot Y & \text{if } X \neq Y \\ \pi \cdot t & \text{otherwise} \end{cases}$$

$$\langle s_1, s_2\rangle[X \to t] = \langle s_1[X \to t], s_2[X \to t]\rangle \qquad (f\ s)[X \to t] = f\ (s[X \to t])$$

$$(f^C\langle s_1, s_2\rangle)[X \to t] = f^C\langle s_1[X \to t], s_2[X \to t]\rangle$$

Definition 4 (Substitution Action on Terms). *A substitution σ is a list of nuclear substitutions, which are applied consecutively to a term:*

$$s\ id = s, \text{ where } id \text{ is the empty list} \qquad s(\sigma :: [X \to t]) = (s[X \to t])\sigma \tag{2}$$

Remark 6. The notion of substitution defined here differs from the more traditional view of a substitution as a simultaneous application of nuclear substitutions, although both are correct [6]. In our representation, the nuclear substitution applied first is the furthest from the term, i.e, the last one in the list of nuclear substitutions. The notion here presented is closer to the concept of triangular substitutions [15].

Example 2. Let $\sigma = [X \to a] :: [Y \to f(X, b)]$ and $t = [a]Y$. Then, $t\sigma = [a]f(a, b)$.

2.2 Freshness and α-equality

Two valuable notions in nominal theory are freshness and α-equality, which are represented, respectively, by the predicates $\#$ and $\approx_\alpha$.

- $a\#t$ means, intuitively, that if a occurs in t then it does so under an abstractor $[a]$.
- $s \approx_\alpha t$ means, intuitively, that s and t are α-equivalent, i.e, they are equal modulo the renaming of bound atoms.

These concepts are formally defined in Definitions 5 and 6.

Definition 5 (Freshness). *A freshness context ∇ is a set of constraints of the form $a\#X$. An atom a is said to be fresh on t under a context ∇, denoted by $\nabla \vdash a\#t$, if it is possible to build a proof using the rules:*

$$\frac{}{\nabla \vdash a\#\langle\rangle}\ (\#\langle\rangle) \qquad \frac{}{\nabla \vdash a\#b}\ (\#atom) \qquad \frac{(\pi^{-1}\cdot a\#X)\in\nabla}{\nabla\vdash a\#\pi\cdot X}\ (\#X)$$

$$\frac{}{\nabla\vdash a\#[a]t}\ (\#[a]a) \qquad \frac{\nabla\vdash a\#t}{\nabla\vdash a\#[b]t}\ (\#[a]b) \qquad \frac{\nabla\vdash a\#s \quad \nabla\vdash a\#t}{\nabla\vdash a\#\langle s,t\rangle}\ (\#pair)$$

$$\frac{\nabla\vdash a\#t}{\nabla\vdash a\#f\ t}\ (\#app) \qquad \frac{\nabla\vdash a\#s \quad \nabla\vdash a\#t}{\nabla\vdash a\#f^C\ \langle s,t\rangle}\ (\#c\text{-}app)$$

Remark 7. The rule $(\#X)$ says that in order to guarantee that $a\#\pi\cdot X$ under the context ∇, we must guarantee that $(\pi^{-1}\cdot a\#X)$ is in ∇. Notice that this rule correctly takes into account the effect of the atom permutation π that is part of the suspended variable $\pi\cdot X$ when deciding which atom must be fresh in the variable X. Consider, for instance, that we want to prove that $a\#\ (a\ b)\cdot X$. Since the permutation π swaps a and b, we must actually make sure that $\pi^{-1}\cdot a=(a\ b)\cdot a=b$ is fresh in X, as indicated in the rule $(\#X)$.

Example 3. Notice that $a\#X\vdash a\#\langle[a]\langle X,a\rangle,[b]h\langle X,b\rangle\rangle$, by application of rules $(\#pair)$, $(\#[a]a)$, $(\#[a]b)$, $(\#app)$, $(\#X)$ and $(\#atom)$.

Definition 6 (α-equality with commutative operators). *Two terms t and s are said to be α-equivalent under the freshness context Δ ($\Delta\vdash t\approx_\alpha s$) if it is possible to build a proof using the rules:*

$$\frac{\Delta\vdash s_0\approx_\alpha t_i,\ \ \Delta\vdash s_1\approx_\alpha t_{i+1(mod2)}}{\Delta\vdash f^C\langle s_0,s_1\rangle\approx_\alpha f^C\langle t_0,t_1\rangle}\ (\approx_\alpha C)_{\ i=0,1} \qquad \frac{}{\Delta\vdash a\approx_\alpha a}\ (\approx_\alpha atom)$$

$$\frac{\Delta\vdash s\approx_\alpha t}{\Delta\vdash f\ s\approx_\alpha f\ t}\ (\approx_\alpha app) \qquad \frac{\Delta\vdash s\approx_\alpha t}{\Delta\vdash [a]s\approx_\alpha [a]t}\ (\approx_\alpha [a]a)$$

$$\frac{\Delta\vdash s\approx_\alpha (a\ b)\cdot t,\ \ \Delta\vdash\ a\#t}{\Delta\vdash [a]s\approx_\alpha [b]t}\ (\approx_\alpha [a]b) \qquad \frac{ds(\pi,\pi')\#X\subseteq\Delta}{\Delta\vdash\pi\cdot X\approx_\alpha\pi'\cdot X}\ (\approx_\alpha var)$$

$$\frac{\Delta\vdash s_0\approx_\alpha t_0,\ \ \Delta\vdash s_1\approx_\alpha t_1}{\Delta\vdash\langle s_0,s_1\rangle\approx_\alpha\langle t_0,t_1\rangle}\ (\approx_\alpha pair) \qquad \frac{}{\Delta\vdash\langle\rangle\approx_\alpha\langle\rangle}\ (\approx_\alpha\langle\rangle)$$

Notation: *We define the difference set between two permutations π and π' as $ds(\pi,\pi')=\{a\in\mathcal{A}|\pi\cdot a\neq\pi'\cdot a\}$. By extension, $ds(\pi,\pi')\#X$ is the set containing every constraint of the form $a\#X$ for $a\in ds(\pi,\pi')$.*

Example 4. Notice that $[a]a\approx_\alpha[b]b$:

$$\frac{\dfrac{}{a\approx_\alpha(a\ b)\cdot b}\ (\approx_\alpha atom) \qquad \dfrac{}{a\#b}\ (\#atom)}{[a]a\approx_\alpha[b]b}\ (\approx_\alpha[a]b)$$

2.3 Nominal C-Unification

Definition 7 (Unification Problem). *A unification problem is a pair $\langle\nabla,P\rangle$ where ∇ is a freshness context and P is a finite set of equations and freshness constraints of the form $s\approx_?t$ and $a\#_?t$, respectively.*

Remark 8. Consider ∇ and ∇' freshness contexts and σ and σ' substitutions. We need the following notation to define a solution to a unification problem:

- $\nabla' \vdash \nabla\sigma$ denotes that $\nabla' \vdash a\#X\sigma$ holds for each $(a\#X) \in \nabla$.
- $\nabla \vdash \sigma \approx \sigma'$ denotes that $\nabla \vdash X\sigma \approx_\alpha X\sigma'$ for all X in $dom(\sigma) \cup dom(\sigma')$.

Definition 8 (Solution for a Triple or Problem). *Let δ be a substitution. A solution for a triple $\mathcal{P} = \langle \Delta, \delta, P\rangle$ is a pair $\langle \nabla, \sigma\rangle$ that fulfills the following four conditions:*

1. $\nabla \vdash \Delta\sigma$
2. $\nabla \vdash a\#t\sigma$, *if* $a\#_? t \in P$
3. $\nabla \vdash s\sigma \approx_\alpha t\sigma$, *if* $s \approx_? t \in P$
4. *There exists* λ *such that* $\nabla \vdash \delta\lambda \approx \sigma$

Then, a solution for a unification problem $\langle \Delta, P\rangle$ is a solution for the associated triple $\langle \Delta, id, P\rangle$.

Definition 9 (Fixed Point Equation). *An equation of the form $\pi \cdot X \approx_\alpha \pi' \cdot X$ is called a fixed point equation.*

Remark 9. In syntactic nominal unification, a fixed point equation of the form $\pi \cdot X \approx_\alpha \pi' \cdot X$ would be solved by requiring that $ds(\pi, \pi')\#\Delta$, where Δ is the context that composes the solution to the problem (see the rule for α-equivalence of suspended variables in Definition 6). This approach, while still correct in nominal C-unification, is not complete. Take, for instance the equation $(a\ b) \cdot X \approx_\alpha X$. A solution not captured by this traditional approach is $\langle \emptyset, [X \to a{+}b] :: id\rangle$, where $+$ is a commutative function symbol (here we are using infix notation). Moreover, there is an infinite number of solutions to fixed point equations. For instance, considering the equation above, among the infinite number of solutions we have: $\langle \emptyset, [X \to a + b] :: id\rangle$, $\langle \emptyset, [X \to (a + b) + (a + b)] :: id\rangle$ and so on. Consequently, fixed point equations are not solved in nominal C-Unification, instead, they are carried on as part of the solution to the unification problem [2].

Remark 10. One of the original features of this work is the separate treatment of fixed point equations from the set of equational and freshness constraints. There is a trivial extension of Definition 8 in order to consider this detachment. Let FP be a set of fixed point equations. $\langle \nabla, \sigma\rangle$ is a solution to the quadruple $\mathcal{P} = \langle \Delta, \delta, P, FP\rangle$ if all conditions of Definition 8 are satisfied and additionally:

- $\nabla \vdash \pi \cdot X\sigma \approx_\alpha \pi' \cdot X\sigma$, if $\pi \cdot X \approx_? \pi' \cdot X \in FP$

Remark 11. The problem of nominal C-unification, as the problem of first-order C-unification, is NP-complete (see [7] and [2]).

3 Specification

We developed a functional nominal C-unification algorithm for unifying the terms t and s. The algorithm is recursive and needs to keep track of the current context, the substitutions made so far, the remaining terms to unify and the current fixed point equations. Therefore, the algorithm receives as input a quadruple

$(\Delta, \sigma, PrbLst, FPEqLst)$, where Δ is the context we are working with, σ is a list of the substitutions already done, $PrbLst$ is a list of unification problems which we must still unify (each equational constraint $t \approx_? s$ is represented as a pair (t, s) in Algorithm 1) and $FPEqLst$ is a list of fixed point equations we have already computed.

The first call to the algorithm, in order to unify the terms t and s is simply: $\texttt{UNIFY}(\emptyset, id, [(t, s)], \emptyset)$. The algorithm eventually terminates, returning a list (possibly empty) of solutions, where each solution is of the form $(\Delta, \sigma, FPEqLst)$.

Although extensive, the algorithm is simple. It starts by analysing the list of terms it needs to unify. If $PrbLst$ is an empty list, then it has finished and can return the answer computed so far, which is the list: $[(\Delta, \sigma, FPEqLst)]$. If $PrbLst$ is not empty, then there are terms to unify, and the algorithm starts by trying to unify the terms t and s that are in the head of the list and only after that it goes to the tail of the list. The algorithm is recursive, calling itself on progressively simpler versions of the problem until it finishes.

3.1 Auxiliary Functions

Following the approach of [6], freshness constraints are treated separately from the main function. This has the advantage of making the main function `UNIFY` smaller, handling only equational constraints. To deal with the freshness constraints, the following auxiliary functions, which come from [6], were used:

- `fresh_subs?`(σ, Δ) returns the minimal context (Δ' in Algorithm 1) in which $a\#_? X\sigma$ holds, for every $a\#X$ in the context Δ.
- `fresh?`(a, t) computes and returns the minimal context (Δ' in Algorithm 1) in which a is fresh for t.

Both functions also return a boolean (*bool*1 in Algorithm 1), indicating if it was possible to find the mentioned context.

3.2 Main Algorithm

The pseudocode of the algorithm is presented in Algorithm 1.

Remark 12. When trying to unify $f^C\langle t_1, t_2\rangle$ with $f^C\langle s_1, s_2\rangle$ there are two possible paths to take: try to unify t_1 with s_1 and t_2 with s_2, or try to unify t_1 with s_2 and t_2 with s_1. This means that there are two branches that we must consider, and since each branch can generate a solution, we may have more than one solution. This is the reason why the algorithm here presented gives a list of solutions as output. In nominal unification, by contrast, only one most general unifier is given as solution.

Algorithm 1 - First Part - Functional Nominal C-Unification

```
1: procedure UNIFY(Δ, σ, PrbLst, FPEqLst)
2:     if nil?(PrbLst) then
3:         return list((Δ, σ, FPEqLst))
4:     else
5:         cons((t, s), PrbLst′) = PrbLst
6:         if (s matches π · X) and (X not in t) then
7:             σ′ = {X → π^{-1} · t}
8:             σ″ = σ′ ∘ σ
9:             (Δ′, bool1) = fresh_subs?(σ′, Δ)
10:            Δ″ = Δ ∪ Δ′
11:            PrbLst″ = append((PrbLst′)σ′, (FPEqLst)σ′)
12:            if bool1 then return UNIFY(Δ″, σ″, PrbLst″, nil)
13:            else return nil
14:            end if
15:        else
16:            if t matches a then
17:                if s matches a then
18:                    return UNIFY(Δ, σ, PrbLst′, FPEqLst)
19:                else
20:                    return nil
21:                end if
22:            else if t matches π · X then
23:                if (X not in s) then
24:                                          ▷ Similar to case above where s is a suspension
25:                else if (s matches π′ · X) then
26:                    FPEqLst′ = FPEqLst ∪ { π · X ≈_α π′ · X}
27:                    return UNIFY(Δ, σ, PrbLst′, FPEqLst′)
28:                else return nil
29:                end if
30:            else if t matches ⟨⟩ then
31:                if s matches ⟨⟩ then
32:                    return UNIFY(Δ, σ, PrbLst′, FPEqLst)
33:                else return nil
34:                end if
35:            else if t matches ⟨t1, t2⟩ then
36:                if s matches ⟨s1, s2⟩ then
37:                    PrbLst″ = cons((s1, t1), cons((s2, t2), PrbLst′))
38:                    return UNIFY(Δ, σ, PrbLst″, FPEqLst)
39:                else return nil
40:                end if
```

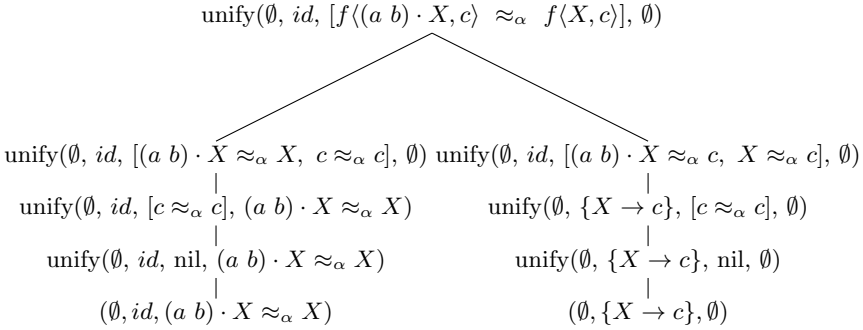
3.3 Examples

A simple example of the algorithm is given in Example 5. In this example, it is possible to see how commutativity introduces branches and how the algorithm calls itself with progressively simpler versions of the problem until it finishes. Example 6 is a slightly more complex example, which uses Example 5.

Example 5. Suppose f is a commutative function symbol. This example shows how the algorithm proceeds in order to unify $f\langle(a\ b)\cdot X, c\rangle$ with $f\langle X, c\rangle$.

Algorithm 1 - Second Part - Functional Nominal C-Unification

```
41:            else if t matches [a]t_1 then
42:                if s matches [a]s_1 then
43:                    PrbLst'' = cons((t_1, s_1), PrbLst')
44:                    return UNIFY(Δ, σ, PrbLst'', FPEqLst)
45:                else if s matches [b]s_1 then
46:                    (Δ', bool1) = fresh?(a, s_1)
47:                    Δ'' = Δ ∪ Δ'
48:                    PrbLst'' = cons((t_1, (a b) s_1), PrbLst')
49:                    if bool1 then
50:                        return UNIFY(Δ'', σ, PrbLst'', FPEqLst)
51:                    else return nil
52:                    end if
53:                else return nil
54:                end if
55:            else if t matches f t_1 then                    ▷ f is not commutative
56:                if s matches f s_1 then
57:                    PrbLst'' = cons((t_1, s_1), PrbLst')
58:                    return UNIFY(Δ, σ, PrbLst'', FPEqLst)
59:                else return nil
60:                end if
61:            else                                            ▷ t is of the form f^C(t_1, t_2)
62:                if s matches f^C(s_1, s_2) then
63:                    PrbLst_1 = cons((s_1, t_1), cons((s_2, t_2), PrbLst'))
64:                    sol_1 = UNIFY(Δ, σ, PrbLst_1, FPEqLst)
65:                    PrbLst_2 = cons((s_1, t_2), cons((s_2, t_1), PrbLst'))
66:                    sol_2 = UNIFY(Δ, σ, PrbLst_2, FPEqLst)
67:                    return APPEND(sol_1, sol_2)
68:                else return nil
69:                end if
70:            end if
71:        end if
72:    end if
73: end procedure
```



Example 6. Suppose f and g are commutative function symbols, and h is a non-commutative function symbol. This example shows how the algorithm would unify $g\langle h\ d, f\langle(a\ b)\cdot X, c\rangle\rangle$ with $g\langle f\langle X, c\rangle, h\ d\rangle$. Because g is commutative, the algorithm explores two branches:

- On the first branch, the algorithm tries to unify $h\ d$ with $f\langle X, c\rangle$ and $f\langle(a\ b)\cdot X, c\rangle$ with $h\ d$. However, since it is impossible to unify $h\ d$ with $f\langle X, c\rangle$ (different function symbols), the algorithm returns an empty list, indicating that no solution is possible for this branch.

– On the second branch, the algorithm tries to unify $h\ d$ with $h\ d$ and $f\langle(a\ b)\cdot X, c\rangle$ with $f\langle X, c\rangle$. First, $h\ d$ unifies with $h\ d$ without any alterations on the context Δ, the substitution σ or the list of fixed point equations $FPEqLst$. Finally, the unification of $f\langle(a\ b)\cdot X, c\rangle$ with $f\langle X, c\rangle$ was shown in Example 5, and gives two solutions, which are also the solutions to this example: $(\emptyset, id, (a\ b)\cdot X \approx_\alpha X)$ and $(\emptyset, \{X \to c\}, \emptyset)$.

4 Formalization

4.1 Termination

Termination of the algorithm was proved by proving the type-correctness conditions (TCCs) generated by PVS [19]. In order to do that, a lexicographic measure was defined:

$$lex2(|Vars(PrbLst) \cup Vars(FPEqLst)|, size(PrbLst)) \tag{3}$$

The first component in the lexicographic measure is the cardinality of the set of variables which occur in $PrbLst$ (the list of remaining unification problems) or in $FPEqLst$ (the list of fixed point equations). To compute the variables in a list, we consider the variables in all terms of the list. Finally, the variables in a term are computed recursively, as can be seen in Definition 10.

Definition 10 (Set of Variables). *The set of variables in a term is recursively defined as:*

$$\begin{array}{rr} Vars(a) = \emptyset & Vars(\langle\rangle) = \emptyset \\ Vars(\pi \cdot X) = \{X\} & Vars([a]t) = Vars(t) \\ Vars(\langle t_0, t_1\rangle) = Vars(t_0) \cup Vars(t_1) & Vars(f\ t) = Vars(t) \\ Vars(f^C\langle t_0, t_1\rangle) = Vars(t_0) \cup Vars(t_1) & \end{array}$$

The specification of the $Vars$ function in PVS is shown below.

```
Vars(t): RECURSIVE finite_set[variable] =
  CASES t OF
        at(a): emptyset,
        *(pm,v): singleton(v),
        unit: emptyset,
        pair(t1,t2): union(Vars(t1),Vars(t2)),
        abs(a,bd): Vars(bd),
        app(sl,ag): Vars(ag),
        c_app(sl, ag): Vars(ag)
  ENDCASES
  MEASURE t BY <<
```

The second component in the lexicographic measure is the sum of the size of every unification problem. To calculate the size of the unification problem (t, s), we only calculate the size of the first term t. This was an arbitrary choice, as the measure would still work if we had taken the size of s or even the size of t plus the size of s (in each recursive call, both the size of t and the size of s decrease). Finally, the size of t is computed recursively according to Definition 11.

Definition 11 (Size of Terms). *The size of terms is recursively computed as:*

$$\begin{array}{cc} size(a) = 1 & size(\langle\rangle) = 1 \\ size(\pi \cdot X) = 1 & size([a]t) = 1 + size(t) \\ size(\langle t_0, t_1\rangle) = 1 + size(t_0) + size(t_1) & size(f\ t) = 1 + size(t) \\ size(f^C\langle t_0, t_1\rangle) = 1 + size(t_0) + size(t_1) & \end{array}$$

The specification of the *size* function in PVS is shown below.

```
size(t): RECURSIVE nat =
  CASES t OF
        at(a): 1,
        *(pm,v): 1,
        unit: 1,
        pair(t1,t2): 1 + size(t1) + size(t2),
        abs(a,bd): 1 + size(bd),
        app(sl,ag): 1 + size(ag),
        c_app(sl, ag): 1 + size(ag)
  ENDCASES
  MEASURE t BY <<
```

The lexicographic measure decreases in each recursive call. The component that decreases depends on the type of the terms t and s that are in the head of the list of problems to unify. If one of them is a variable X, and we are not dealing with a fixed point equation, then the algorithm will instantiate this variable X, and the first component, $|Vars(PrbLst) \cup Vars(FPEqLst)|$, will decrease. In any other case, the second component, $size(PrbLst)$, will decrease.

Remark 13. It was possible to reduce the lexicographic measure used in [2], from 4 parameters to only 2 parameters. The measure adopted in [2] was:

$$|\mathcal{P}| = \langle |Var(P_{\approx})|, |P_{\approx}|, |P_{nfp}|, |P_{\#}| \rangle \tag{4}$$

where $P_{\approx}$ is the set of equation constraints in P, P_{nfp} is the set of non fixed point equations in P and $P_{\#}$ is the set of freshness constraints in P. Two ideas were used in order to accomplish this reduction. The first was to separate the treatment of freshness constraints from equational constraints, and treat the freshness constraints with the help of the auxiliary functions of 3.1. This idea comes from [6]. The second one is to separate the fixed point equations from the equational constraints. This way, when a fixed point equation is found in $PrbLst$, it is moved to $FPEqLst$, which makes $size(PrbLst)$ diminish.

4.2 Soundness and Completeness

To state the main theorems that allow us to prove soundness and completeness, we must first define the notion of a *valid* quadruple. A valid quadruple is an invariant of the `UNIFY` function in Algorithm 1 with useful properties.

Definition 12 (Valid Quadruple). *Let Δ be a freshness context, σ a substitution, P a list of unification problems and FP a list of fixed point equations. $\mathcal{P} = \langle \Delta, \sigma, P, FP \rangle$ is a valid quadruple if the following two conditions hold:*

$$- \ Vars(im(\sigma)) \cap dom(\sigma) = \emptyset \quad - \ dom(\sigma) \cap (Vars(P) \cup Vars(FP)) = \emptyset$$

where $im(\sigma)$ is the image of σ and $dom(\sigma)$ is the domain of σ.

Remark 14. A valid quadruple has two desirable properties: the substitution is idempotent (condition 1) and applying the substitution to P or FP produces no effect.

Soundness Corollary 1 states that `UNIFY` is sound. It follows directly by application of Theorem 1.

Theorem 1 (Main Theorem for Soundness of `UNIFY` Algorithm). *Suppose $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \texttt{UNIFY}(\Delta, \sigma, PrbLst,\ FPEqLst)$, (∇, δ) is a solution to $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$ and $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$ is a valid quadruple. Then (∇, δ) is a solution to $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$.*

Proof. The proof is by induction on the lexicographic measure, according to the form of the terms t and s that are in the head of $PrbLst$, the list of remaining unification problems. The hardest cases are the ones of suspended variables and abstractions (see Remark 17). Below we explain the case of commutative functions.

In the case of commutative function symbols $f\langle t_1, t_2 \rangle$ and $f\langle s_1, s_2 \rangle$, there are no changes in the context or the substitution from one recursive call to the next. Therefore, it is trivial to check that we remain with a valid quadruple and it is also trivial to check all but the third condition of Definition 8. For the third condition we have either $\nabla \vdash t_1\delta \approx_\alpha s_1\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_2\delta$ or $\nabla \vdash t_1\delta \approx_\alpha s_2\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_1\delta$. In any case, we are able to deduce $\nabla \vdash (f\langle t_1, t_2 \rangle)\delta \approx_\alpha (f\langle s_1, s_2 \rangle)\delta$ by noting that $(f\langle t_1, t_2 \rangle)\delta = f\langle t_1\delta, t_2\delta \rangle$, $(f\langle s_1, s_2 \rangle)\delta = f\langle s_1\delta, s_2\delta \rangle$ and then using rule $(\approx_\alpha \ \ c - app)$ for alpha equivalence of commutative function symbols.

Corollary 1 (Soundness of `UNIFY` Algorithm). *Suppose (∇, δ) is a solution to $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$, and $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \texttt{UNIFY}(\emptyset, id, [(t, s)], \emptyset)$. Then (∇, δ) is a solution to $\langle \emptyset, id, [(t, s)], \emptyset \rangle$.*

Proof. Notice that $\langle \emptyset, id, [(t, s)], \emptyset \rangle$ is a valid quadruple. Then, we apply Theorem 1 and prove the corollary.

Remark 15. An interpretation of Corollary 1 is that if (∇, δ) is a solution to one of the outputs of the algorithm `UNIFY`, then (∇, δ) is a solution to the original problem.

Completeness Corollary 2 states that UNIFY is complete. It follows directly by application of Theorem 2.

Theorem 2 (Main Theorem for Completeness of UNIFY). *Suppose* (∇, δ) *is a solution to* $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$ *and that* $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$ *is a valid quadruple. Then, there exists a computed output* $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \texttt{UNIFY}(\Delta, \sigma, PrbLst, FPEqLst)$ *such that the solution* (∇, δ) *is also a solution to* $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$.

Proof. The proof is by induction on the lexicographic measure, according to the form of the terms t and s that are in the head of $PrbLst$, the list of remaining unification problems. The hardest cases are again the ones of suspended variables and abstractions (see Remark 17). Below we explain the case of commutative functions.

In the case of commutative function symbols $f\langle t_1, t_2 \rangle$ and $f\langle s_1, s_2 \rangle$, there are no changes in the context or the substitution from one recursive call to the next. Therefore, it is trivial to check that we remain with a valid quadruple and it is also trivial to check all but the third condition of Definition 8. For the third condition we have $\nabla \vdash (f\langle t_1, t_2 \rangle)\delta \approx_\alpha (f\langle s_1, s_2 \rangle)\delta$ and must prove that either ($\nabla \vdash t_1\delta \approx_\alpha s_1\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_2\delta$) or ($\nabla \vdash t_1\delta \approx_\alpha s_2\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_1\delta$) happens. This again is solved by noting that $(f\langle t_1, t_2 \rangle)\delta = f\langle t_1\delta, t_2\delta \rangle$, $(f\langle s_1, s_2 \rangle)\delta = f\langle s_1\delta, s_2\delta \rangle$ and then using rule $(\approx_\alpha \ c - app)$ for alpha equivalence of commutative function symbols.

Corollary 2 (Completeness of UNIFY). *Suppose* (∇, δ) *is a solution to the input quadruple* $\langle \emptyset, id, [(t, s)], \emptyset \rangle$. *Then, there exists* $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \texttt{UNIFY}(\emptyset, id, [(t, s)], \emptyset)$ *such that* (∇, δ) *is a solution to* $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$.

Proof. Notice that $\langle \emptyset, id, [(t, s)], \emptyset \rangle$ is a valid quadruple. Then, we apply Theorem 2 and prove the corollary.

Remark 16. An interpretation of Corollary 2 is that if (∇, δ) is a solution to the initial problem, then (∇, δ) is also a solution to one of the outputs of UNIFY.

5 Interesting Points of Formalization and Implementation

We discuss interesting points of the formalization and implementation here.

Remark 17. To prove correctness and completeness of the algorithm, we work with the terms t and s that are in the head of $PrbLst$. We divide the proof by cases. The most interesting case is when t or s is a suspension $\pi \cdot X$ and X does not occur in the other term (see Algorithm 1).

In this case, the algorithm receives as arguments Δ, σ, $PrbLst$ and $FPEqLst$ and the next recursive call is made with four different parameters: Δ'', σ'', $PrbLst''$ and nil (see Algorithm 1). Therefore, all of the four conditions of the Definition 8 are not trivially satisfied. Moreover, since in the next recursive call we will be working with a new substitution σ'' we must prove that the quadruple

we are working with remains valid (this is proved by noting that when a variable X is added to the domain of the substitution, all occurrences of it in $PrbLst$ and in $FPEqLst$ are instantiated, maintaining the validity of the quadruple).

The case of unifying $t = [a]t_1$ with $s = [b]s_1$ is also interesting, since both the context and the list of problems to unify suffer modifications in the recursive call. In all the remaining cases, there are no changes in the context nor in the substitution, making them easier. In these remaining cases, only one condition (the third) of the four in Definition 8 is not trivially satisfied.

Remark 18. Introducing commutative function symbols to the nominal unification algorithm presented in [6] meant we had to:

- Unify terms rooted by commutative function symbols (for instance, $f(t_1, t_2)$ with $f(s_1, s_2)$). First, the algorithm tries to unify t_1 with s_1 and t_2 with s_2, generating a list of solutions sol_1 (see Algorithm 1). Then, the algorithm tries unifying t_1 with s_2 and t_2 with s_1, generating a list of solutions sol_2. The final result is then simply the concatenation of both lists.
- Handle fixed point equations. This was also straightforward. We keep a separate list of fixed point equation ($FPEqLst$), and when the algorithm recognizes a fixed point equation in $PrbLst$ it takes this equation out of $PrbLst$ and puts it on $FPEqLst$.
- Define an appropriate data structure for the problem and the solutions. This was not straightforward. As mentioned before, since commutativity introduced branches, the recursive calls of the algorithm can be seen as a tree (see Example 5). Therefore, initially, an approach using a tree data structure was planned (which would have complicated the analysis). However, since the algorithm simply solves one branch and then the other, we realized all that was needed was to do two recursive calls (one for each branch) and append the two lists of solutions generated. Therefore, we were able to avoid the tree data structure, working instead with lists, which simplified the specification.

Remark 19. Since PVS does not support automatic extraction of Python code, the translation of the PVS specification for the Python implementation was done manually. The code follows strictly the lines of the specification (Algorithm 1) with small adjustments, such as the inclusion of two parameters in the algorithm implementation, in order to support a verbose mode that prints the tree of recursive calls. This, and the representation of atoms, variables and terms in the implementation is discussed in Appendix A. An OCaml implementation of a nominal C-unification algorithm was previously developed [2], but in contrast to the current Python implementation, the OCaml implementation does not correspond in a direct way to the formalized non-deterministic inductive specification [2].

Remark 20. Python was chosen to implement the algorithm due to our previous knowledge, its expressivity and its support for the functional programming paradigm (Python is considered a multiparadigm programming language).

Remark 21. The "translation" from the PVS specification to the Python implementation was done manually and, as occurs in all manual translations of this kind, some errors may occur during this process. To diminish the probability of errors, the Python code was tested, as shown in the examples in Appendix A.

6 Conclusion and Future Work

In this paper, we explained the problem of nominal C-unification and presented a functional algorithm for doing this task. We observed how nominal C-unification has applications on logic programming languages and how the algorithm here presented could be straightforwardly converted to a matching algorithm, which in turn would have applications in nominal rewriting.

Our approach differs from the only other work in nominal C-unification ([2]) in two main points. First, we do not present a set of non-deterministic transformation rules, instead, we opt for a recursive specification, implemented in Python. Second, we follow the approach in [6] and deal with freshness contexts separately. This simplifies the main function and, along with the idea of using a different parameter to represent fixed point equations, allowed us to reduce the lexicographic measure used in [2] from four parameters to only two parameters, thus simplifying the formalizations of termination, soundness and completeness.

We are currently working in running executable code from PVS (using the PVSIO feature that lets you execute verified algorithms inside the PVS environment and provides input and output operators) and comparing this approach with the Python implementation. For further details, check the Appendix B.

Finally, a future study would be extending the formalization to handle the more general case of Mal'cev permutative theories, which include n-ary functions with permutative arguments [13]. Other possible path, as indicated in [11], is expanding the algorithm to handle other equational theories such as unification modulo associative and associative-commutative function symbols (A- and AC-unification).

References

1. Ayala-Rincón, M., de Carvalho-Segundo, W., Fernández, M., Nantes-Sobrinho, D.: On Solving Nominal Fixpoint Equations. In: 11th Int. Symposium on Frontiers of Combining Systems (FroCoS). LNCS, vol. 10483, pp. 209–226. Springer (2017)
2. Ayala-Rincón, M., de Carvalho-Segundo, W., Fernández, M., Nantes-Sobrinho, D.: Nominal C-unification. In: 27th Int. Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017), Revised Selected Papers. LNCS, vol. 10855, pp. 235–251. Springer (2018)
3. Ayala-Rincón, M., de Carvalho-Segundo, W., Fernández, M., Nantes-Sobrinho, D., Rocha-Oliveira, A.: A Formalisation of Nominal alpha-equivalence with A, C, and AC Function Symbols. Theoret. Comput. Sci. **781**, 3–23 (2019)
4. Ayala-Rincón, M., Fernández, M., Nantes-Sobrinho, D.: Nominal narrowing. In: 1st Int. Conference on Formal Structures for Computation and Deduction (FSCD). LIPIcs, vol. 52, pp. 11:1–11:17 (2016)

5. Ayala-Rincón, M., Fernández, M., Nantes-Sobrinho, D.: Fixed-point constraints for nominal equational unification. In: 3rd Int. Conference on Formal Structures for Computation and Deduction (FSCD). LIPIcs, vol. 108, pp. 7:1–7:16 (2018)
6. Ayala-Rincón, M., Fernández, M., Rocha-Oliveira, A.: Completeness in PVS of a nominal unification algorithm. Elect. Notes Theor. Comp. Sci. **323**, 57–74 (2016)
7. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge U.P. (1999)
8. Baader, F., Snyder, W.: Unification theory. In: Handbook of Automated Reasoning (in 2 volumes), pp. 445–532. Elsevier and MIT Press (2001)
9. Baumgartner, A., Kutsia, T., Levy, J., Villaret, M.: Nominal anti-unification. In: 26th Int. Conference on Rewriting Techniques and Applications (RTA). LIPIcs, vol. 36, pp. 57–73 (2015)
10. Calvès, C., Fernández, M.: A polynomial nominal unification algorithm. Theoret. Comput. Sci. **403**(2-3), 285–306 (2008)
11. de Carvalho Segundo, W.L.R.: Nominal Equational Problems Modulo Associativity, Commutativity and Associativity-Commutativity. Ph.D. thesis, Universidade de Brasilia (2019)
12. Cheney, J., Urban, C.: α-prolog: A logic programming language with names, binding and α-equivalence. In: 20th Int. Conference on Logic Programming (ICLP). LNCS, vol. 3132, pp. 269–283. Springer (2004)
13. Comon, H.: Complete Axiomatizations of Some Quotient Term Algebras. Theoret. Comput. Sci. **118**(2), 167–191 (1993)
14. Fernández, M., Gabbay, M.: Nominal rewriting. Information and Computation **205**(6), 917–965 (2007)
15. Kumar, R., Norrish, M.: (Nominal) Unification by Recursive Descent with Triangular Substitutions. In: First Int. Conference on Interactive Theorem Proving (ITP). LNCS, vol. 6172, pp. 51–66. Springer (2010)
16. Levy, J., Villaret, M.: Nominal unification from a higher-order perspective. In: 19th Int. Conference on Rewriting Techniques and Applications (RTA). LNCS, vol. 5117, pp. 246–260. Springer (2008)
17. Levy, J., Villaret, M.: An efficient nominal unification algorithm. In: Proceedings of the 21st Int. Conference on Rewriting Techniques and Applications (RTA). LIPIcs, vol. 6, pp. 209–226 (2010)
18. Muñoz, C.A., Butler, R.: Rapid prototyping in PVS. Tech. Rep. NASA/CR-2003-212418, NIA-2003-03, NASA Langley Research Center (NIA) (2003)
19. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS System Guide - Version 2.4 (2001), `http://pvs.csl.sri.com/documentation.shtml`
20. Pitts, A.: Nominal sets: Names and symmetry in computer science. Cambridge U.P. (2013)
21. Schmidt-Schauß, M., Kutsia, T., Levy, J., Villaret, M.: Nominal unification of higher order expressions with recursive let. In: 26th Int. Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016), Revised Selected Papers. LNCS, vol. 10184, pp. 328–344. Springer (2017)
22. Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning **40**(4), 327–356 (2008)
23. Urban, C., Pitts, A., Gabbay, M.: Nominal unification. Theoret. Comput. Sci. **323**(1-3), 473–497 (2004)

A Discussion on the Python Implementation

The implementation of Algorithm 1 may be found on file `unify.py`, while the examples that will be shown next can be run by executing file `tests.py`. Both of them are also available at `http://www.github.com/gabriel951/c-unification`. Specific instructions can be found on these files. Python 3 should be used (specifically, we used Python 3.5.2).

When implementing the algorithm, two optional parameters were added to function `unify`, in order to allow the user to run it choosing either printing or not the tree of recursive calls. The first parameter (`verb`) is a Boolean, which controls the verbosity of the algorithm: when set to true, the algorithm does print the tree of recursive calls and when set to false the algorithm does not. The second parameter (`indent_lvl`) is a string that regulates the indentation level. When the algorithm forks into two branches, this parameter is updated which in turn allows us to clearly distinguish when the algorithm enters a branch (see Examples 7 and 8).

As mentioned before, a term in the nominal setting can be an atom, a suspended variable, the unit, a pair, an abstraction, a function application or a commutative function application. Every one of these options is represented as a class by our algorithm implementation. An atom object a has one attribute, its name, which in turn is represented by the algorithm as a string. A suspended variable object contains two attributes: a permutation and a variable. The permutation is represented as a list, where each element of the list (corresponding to a swapping) is a pair consisting of two strings (to represent the two atoms being swapped). The variable is represented by the algorithm as a string. The unit is a class with no attributes. A pair object contains two attributes, to represent the two terms of a pair. An abstraction object contains two attributes: the first is an atom and the second a term. A function application and a commutative function application both have two attributes: the first is the function symbol, represented as a string and the second is their argument, which is another term. In the case of commutative function applications, an assertion verifies that the argument is indeed a pair, as commented in Remark 2.

Finally, the action of permutations and substitutions have no significant differences from what has been specified in PVS. The representation of the four parameters of the `unify` function in the Python implementation also follows closely the PVS specification.

The next two examples present two illustrative executions of the algorithm for Examples 5 and 6 given in the body of the paper.

Example 7. Let f be a commutative function symbol. Here we show how the Python algorithm would proceed to unify the terms in Example 5: $f\langle(a\ b)\cdot X, c\rangle$ and $f\langle X, c\rangle$. Note that each one of the two branches explored generates a solution.

Output 1. Algorithm running for the terms in Example 5. Trying to unify $f\langle(a\ b)\cdot X, c\rangle$ and $f\langle X, c\rangle$.

```
Trying to unify terms: f(<[(a b) ]*X, c>) and f(<[]*X, c>)


<[], id, [(f(<[(a b) ]*X, c>) = f(<[]*X, c>)), ], []>
|
   <[], id, [([(a b) ]*X = []*X), (c = c), ], []>
   |
   <[], id, [(c = c), ], [[('a', 'b')]*X = X ]>
   |
   <[], id, [], [[('a', 'b')]*X = X ]>
   |
   solution: <[], id, [[('a', 'b')]*X = X ]>


   <[], id, [([(a b) ]*X = c), (c = []*X), ], []>
   |
   <[], [X->c ], [(c = c), ], []>
   |
   <[], [X->c ], [], []>
   |
   solution: <[], [X->c ], []>

Finished.
```

Example 8. Let f and g be commutative function symbols, and let h be a non-commutative function symbol. We now show how the Python algorithm would proceed to unify the terms in Example 6: $g\langle h\ d, f\langle (a\ b)\cdot X, c\rangle\rangle$ and $g\langle f\langle X, c\rangle, h\ d\rangle$. Note that one branch does not generate any solution.

Output 2. Algorithm running for the terms in Example 6. Trying to unify the terms $g\langle h\ d, f\langle (a\ b)\cdot X, c\rangle\rangle$ and $g\langle f\langle X, c\rangle, h\ d\rangle$.

```
Trying to unify terms: g(<h(d), f(<[(a b) ]*X, c>)>) and g(<f(<[]*X, c>), h(d)>)


<[], id, [(g(<h(d), f(<[(a b) ]*X, c>)>) = g(<f(<[]*X, c>), h(d)>)), ], []>
|
   <[], id, [(h(d) = f(<[]*X, c>)), (f(<[(a b) ]*X, c>) = h(d)), ], []>
   |
   No solution

   <[], id, [(h(d) = h(d)), (f(<[(a b) ]*X, c>) = f(<[]*X, c>)), ], []>
   |
   <[], id, [(d = d), (f(<[(a b) ]*X, c>) = f(<[]*X, c>)), ], []>
   |
   <[], id, [(f(<[(a b) ]*X, c>) = f(<[]*X, c>)), ], []>
   |
      <[], id, [([(a b) ]*X = []*X), (c = c), ], []>
      |
      <[], id, [(c = c), ], [[('a', 'b')]*X = X ]>
      |
      <[], id, [], [[('a', 'b')]*X = X ]>
      |
      solution: <[], id, [[('a', 'b')]*X = X ]>

      <[], id, [([(a b) ]*X = c), (c = []*X), ], []>
      |
      <[], [X->c ], [(c = c), ], []>
      |
      <[], [X->c ], [], []>
      |
      solution: <[], [X->c ], []>

Finished.
```

B Experiments Comparing Implementations

PVSIO is a PVS package that extends the ground evaluator with a predefined library of imperative programming languages features, among them input and output operators [18]. For our purposes, this means that we can run the formalised PVS function that performs unification with the help of the ground evaluator, and use the input and output capabilities provided by PVSIO to perform meaningful experiments.

We are currently working on experiments to compare the Python 3 implementation with the executable code that can be run inside PVS via the PVSIO feature. In the next sections, we describe the methodology used, present and discuss the results obtained and tell about our next steps for more sophisticate investigations.

B.1 Methodology

To compare the Python and the PVS implementation, we generated a fixed number of terms t and s to be unified and ran the implementations, measuring the time. By printing the Python results in the same way as the PVS implementation prints, it was possible to check whether the algorithms match or not and how long each implementation took to unify. The results can be seen in Table 6.

We generate the term t randomly according to the probabilities presented in Table 3. The number of different atoms, variables, function applications and commutative function applications is shown in Table 4. Finally, to generate a permutation, we first define a probability p ($0 \leq p \leq 1$) of generating a new swapping. Then, we generate a random number n between 0 and 1. If $n > p$ we stop generating swappings and return the permutation (i.e., the list of swappings) constructed so far. If $n \leq p$ we generate a new swapping, add this swapping to our current list of swappings and go back to generating a new random number n. We repeat the procedure until we fall in the case $n > p$. We used $p = 0.5$ in our experiments.

Finally, we generate the term s as a "copy with modifications" of the term t. These modifications are:

- With probability p_{var} we substitute part of the term t by a random suspended variable.
- If we encounter a commutative function applicaton in t, with probability p_C we change the order of the two arguments.
- If we encounter an abstraction, with probability p_{abs} we correctly change the atom being abstracted (for instance, change a term $[a]t'$ to a term $[b](a\ b)\cdot t'$).
- If we encounter an atom, with probability p_{atom}, we change the atom.

The probabilities of doing these modifications are shown in Table 5.

Remark 22. Notice that if we encounter an atom in the term t and change it when constructing the term s this may result in non unifiable terms t and s. This is precisely what we hoped to accomplish, since we also want to test how the algorithm runs when the terms are not unifiable.

Table 3. Probability of Generating Each Type of Term.

Type of the term	Probability
Atom	0.1
Suspended Variable	0.2
Unit	0.1
Abstraction	0.2
Pair	0.1
Function Application	0.2
Commutative Function Application	0.1

Table 4. Number of Different Atoms, Variables, Function Applications and Commutative Function Applications.

Type of Term	Number of Different Terms in Our Domain
Atom	10
Variable	10
Function Application	5
Commutative Function Application	5

Table 5. Probability of Making Modifications in the Term s when Constructing It From the Term t.

Type of Modification	Probability
p_{var}	0.05
p_C	0.5
p_{abs}	0.5
p_{atom}	0.1

B.2 Results and Interpretation

First we checked if both implementations gave equal results. This was verified to hold.

Next, we measured the time it took for them, according to the number of terms we wanted to unify. The results are shown in Table 6.

Table 6. Time Each Implementation Took to Unify a Given Number of Terms.

Number of Unification Problems	Python - Time	PVS - Time
1000	< 1s	43s
2000	< 1s	1min24s
10000	3s	Error - Stack Overflow

The results show that the Python implementation is faster (which was expected since the PVS implementation is not LISP executing directly, but LISP executing under the PVS environment, which adds an overhead). What was not expected was the difference in the performance of the two implementations. Other surprising event was the PVS implementation giving an error when the number of unification problems grew to 10000.

B.3 Current Work Regarding the Experiments

We are currently investigating why the PVS performance was significantly slower than the Python implementation and if the PVS implementation systematically fails when dealing with a large number of unification problems. We also plan on refining our experiment, by generating a fixed amount of unification problems (in our case 1000, 2000 and 10000 unification problems) a multiple number of times (instead of only one) and calculating the average of time each implementation takes.

As future work, [2] presented a set of non-deterministic inference rules for the task of nominal C-unification, which were proved to be correct and complete. We plan on formalising a correct and complete algorithm based on this rules (this could be done, for instance, by giving an heuristic on how to apply the rules) and then use the Coq feature of code extraction to obtain executable code. Then, we could compare the Python implementation with this other implementation. With the Coq code extraction feature, we can obtain code in OCaml or Haskell that runs independently of the Coq environment. Therefore, we expect the code extracted this way to have a better performance than the PVS implementation and a competitive performance in relation to the Python implementation, although this needs to be verified with careful experiments.