Completeness in PVS of a Nominal Unification Algorithm

Mauricio Ayala-Rincón [a,2] Maribel Fernández [b,3]
Ana Cristina Rocha-Oliveira[a,1]

[a] *Departamentos de Matemática e Ciência da Computação*
Universidade de Brasília
Brasília D.F., Brasil

[b] *Department of Informatics*
King's College London
London, UK

Abstract

Nominal systems are an alternative approach for the treatment of variables in computational systems. In the nominal approach variable bindings are represented using techniques that are close to first-order logical techniques, instead of using a higher-order metalanguage. Functional nominal computation can be modelled through *nominal rewriting*, in which α-equivalence, nominal matching and nominal unification play an important role. Nominal unification was initially studied by Urban, Pitts and Gabbay and then formalised by Urban in the proof assistant Isabelle/HOL and by Kumar and Norrish in HOL4. In this work, we present a new specification of nominal unification in the language of PVS and a formalisation of its completeness. This formalisation is based on a natural notion of nominal α-equivalence, avoiding in this way the use of the intermediate auxiliary weak α-relation considered in previous formalisations. Also, in our specification, instead of applying simplification rules to unification and freshness constraints, we recursively build solutions for the original problem through a straightforward functional specification, obtaining a formalisation that is closer to algorithmic implementations. This is possible by the independence of freshness contexts guaranteed by a series of technical lemmas.

Keywords: Nominal terms, binders, α-equivalence, nominal unification, PVS.

1 Introduction

When one introduces variable binders in a language, one thing to be considered immediately is α-equivalence. For instance, it must be possible to derive the equivalence between the formulas $\exists x : x > 1$ and $\exists y : y > 1$, despite the syntactical differences. Nominal theories treat binders in a way that is closer to informal practice, using variable names and freshness constraints instead of using indices as in explicit substitutions *à la de Bruijn*. In nominal syntax, there are two kinds of variables: atoms, representing object-level variables, and meta-variables, or simply

[1] Email: anacrismarie@gmail.com. Author supported by a Ph.D. scholarship from CAPES Brazil.
[2] Email: ayala@unb.br. Work partially supported by grant CNPq UNIVERSAL 476952/2013-1.
[3] Email: maribel.fernandez@kcl.ac.uk. Work partially supported by grant CsF PVE CAPES 146/2012.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

variables. Atoms can be abstracted but not substituted, whereas variables cannot be abstracted but can be substituted. The notion of substitution is first-order in the sense that it allows capture, but freshness constraints are taken into account. Notions such as rewriting (cf. [FG07]) and unification (cf. [UPG04]) can be directly defined, without having to rely on involved notions such as β-reduction, as in the higher-order and explicit substitutions approaches (cf. [Hue75,DHK00,ARK01]).

Nominal unification problems can be solved (modulo α-equivalence) with first-order substitutions that act over meta-variables, i.e., simply filling the holes marked with meta-variables $(X, Y, Z, \dots)$ and allowing capture of variable names $(a, b, c, i, k, \dots)$. This can be illustrated by the expressions

$$\sum_{k=0}^{7}\sum_{i=0}^{5}(i - X)^i \text{ and } \sum_{i=0}^{7}\sum_{k=0}^{5}(X - Y)^k,$$

which admit a most general unifier according to the algorithm in [UPG04], with solution $[X \mapsto k][Y \mapsto i]$. Note that i and k are captured, because these names are bound or abstracted by the sum operator. In a higher-order unification approach, this solution would not be accepted because bound variable capture is forbidden.

On the other hand, the unification problem with the expressions

$$\sum_{i=0}^{5}(i - X)^i \text{ and } \sum_{k=0}^{5}(X - Y)^k$$

has no solution in the nominal setting. One could argue that a solution could be obtained instantiating $[X \mapsto i][Y \mapsto i]$ and renaming k as i. But this is not possible since i should be a "fresh" name in the scope of the second sum in order to proceed with this renaming, and the chosen substitution contradicts this condition. In other words, the meta-variable X should be instantiated uniformly throughout the problem. We can specify that a name is fresh for a term by writing a freshness constraint, for example, $i\#t$ states that the name i is fresh in the term t. In general, if two nominal terms are unifiable, the unifier is a pair consisting of a substitution and a set of freshness constraints.

Translations between nominal unification problems and higher-order pattern unification problems are given in [Che05,LV12].

Contribution

In this paper, we present a functional specification of a new nominal unification algorithm and formalise its correctness and completeness in the language of the higher-order proof assistant Prototype Verification System (PVS) [SORSC01]. PVS was chosen because it has a large library about term rewriting systems ([GAR10]) and our `nominal unification` theory extends this background about rewriting.

This paper is accompanied with the whole PVS development for nominal unification, which includes specifications of all notions and definitions as well as formalisations of the proofs of all lemmas and theorems given in this paper. The development is available for download in the PVS theory for term rewriting systems `trs.cic.unb.br`.

The style of our specification is close to the functional presentations of Robinson's first-order unification algorithm, and the formalisation avoids the use of intermediate equivalence relations, obtaining in a straightforward manner transitivity and symmetry of the nominal α-equivalence relation. Indeed, in [Urb10], a "weak equivalence" is used in order to simplify the proof of transitivity for the standard nominal α-equivalence. However, in this paper, we present an even simpler proof, avoiding formalisations of properties of this weak intermediate relation. This is obtained following the analytic scheme of proof shown in [FG07].

The nominal unification algorithm given in Isabelle/HOL in [Urb04] is essentially specified as the transformation rule system presented in [UPG04]. These rules transform unification problems with their associated freshness contexts into simpler ones. This approach is very elegant and allows a higher level of abstraction that simplifies the analysis of computational properties such as termination and uniqueness of solutions, but it is not so useful in implementations due to its inherent non-determinism (regarding the application of the transformation rules).

Here we present a new nominal unification algorithm that has only two nominal terms (but no freshness context) as inputs, as in [CF10,LV10]. However, the algorithms presented in [CF10,LV10] focus on efficiency, whereas our goal is to formalise the proof of correctness by specifying the algorithm in PVS as a recursive function "*unify*" working directly on terms and formalising separately properties of contexts. Although the function "*unify*" does not carry freshness contexts, it builds them at the end of the execution together with the substitution solution. The freshness problems generated during the recursive computation are solved separately due to the independence of solutions for freshness and without involving extra fresh variables as usual in a nominal setting. This differs from the treatment given in [LV10] where freshness constraints, as well as suspensions, are encoded as equations, that was proved equivalent to the treatment in [CF10] in [Cal13].

Related work

There are formalisations of nominal theories in other proof assistants. The most relevant formalisation has been implemented in Isabelle/HOL [Urb08], where α-equivalence between terms is effectively obtained by representing terms as "abstraction functions". Thus, Urban [Urb08] presents some basic conditions that are sufficient to guarantee the equivalence between two representations of terms. Then, an induction principle is presented, to obtain proofs by induction over abstracted terms in a more natural way. For instance, the Substitution Lemma (well-known in the context of λ-calculus) was formalised using these techniques.

A similar work was done in Coq [ABW07], but bound variables were encoded by using de Bruijn indices and the terms were defined as having the type of locally nameless terms. An induction principle was implemented in order to prove properties about well-formed terms without mentioning indices.

Another formalisation in Isabelle/HOL is available in [Urb04], to deal with nominal unification following [UPG04]. This formalisation is closer to ours in the sense that α-equivalence is defined under some side-conditions (namely, freshness conditions). The properties formalised in this system include the fact that the specified α-equivalence is indeed an equivalence relation, termination and soundness of the

unification algorithm and characterisation of the normal forms generated by the algorithm.

In [Urb10], Urban compares the proof of transitivity of the α-equivalence relation presented in [Urb04,FG07] and [KN10]. The proof shown in the last citation was then considered the best because it avoids a more complex inductive scheme on the size of terms. However, it requires the implementation of a "weak-equivalence" relation as a workaround. Here, we follow auxiliary lemmas developed in [FG07], but with a simpler proof of transitivity by induction on the structure of terms obtaining directly the necessary result that the specified α-equivalence relation is indeed an equivalence relation.

The specification of our algorithm, passing as parameters only pairs of terms to be unified, is closer to functional presentations in the style of Robinson's first-order unification that have been formalised in a variety of proof assistants (e.g., [Pau85,AGdMA14]).

Organisation

Section 2 presents the basic concepts and grammar used in the nominal context. Section 3 defines freshness and α-equivalence and makes explicit (subsection 3.1) the details about the proof of transitivity of α-equivalence used in previous formalisations in comparison with the ones strictly necessary in the current approach. Also, this section (subsection 3.2) presents a function that computes the minimal freshness context needed to derive a freshness constraint. This is crucial to obtain a unification algorithm that does not need to carry freshness contexts continuously. Section 4 presents the main contributions of this paper: the specification of a functional algorithm to solve nominal unification problems and the formalisation of its soundness and completeness.

2 Preliminaries

This section presents some basic definitions: permutations, terms and substitutions, which are needed to reason about a nominal unification algorithm.

Definition 2.1 **Atoms** or **names** are basic structures in the context of nominal theories. They represent object-level variables; the set $\mathcal{A}$ of all atoms is countably infinite. A **swapping** $(a\ b)$ is a bijection from $\mathcal{A}$ into $\mathcal{A}$ that exchanges a and b and that fixes any other atom. **Permutations** are also bijections of the form $\pi : \mathcal{A} \to \mathcal{A}$, which change a finite number of atoms and that are represented as lists of swappings. Then, the action of a permutation over atoms is recursively defined as:

$$id(c) = c \text{ , where } id \text{ is the null list;}$$

$$((a\ b) \circ \pi)(c) = \begin{cases} a \text{ , if } \pi(c) = b; & b \text{ , if } \pi(c) = a; & \pi(c) \text{ , otherwise.} \end{cases}$$

The inverse of π is the reverse list of swappings and it is denoted by π^{-1}.

Definition 2.2 Let Σ and $\mathcal{V}$ be a signature with function symbols and a countably infinite set of variables, respectively. Then, the set $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{V})$ of **nominal terms**

is generated by the following grammar:

$$t ::= \bar{a} \mid \pi \cdot X \mid () \mid (t_1, t_2) \mid [a]t \mid f\ t\ ,$$

where $\bar{a}$ is an **atom**, $\pi \cdot X$ is a **suspension** (a permutation π suspended in the variable $X \in \mathcal{V}$), $()$ is the **unit** or empty tuple, (t_1, t_2) is a **pair** of terms, $[a]t$ is an **abstraction** (a term with the atom a abstracted) and $f\ t$ is an **application** (a symbol $f \in \Sigma$ applied to a term).

Notice that to encode terms in PVS, we distinguish between the atom a and the term $\bar{a}$ that consists of the atom a (compare with the constructor `at` in the following code of the data structure of terms). Also, the function application works for symbols with arity one. To represent a greater arity, one can use pairs to encode tuples with any number of arguments. For instance, if the symbol f has arity 3, then we can describe the term $f(t_1, (t_2, t_3))$ using the present grammar. The next specification of terms in PVS allows us to have induction schemes generated automatically.

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

Definition 2.3 The **depth** of a term is computed by the following function:

$$depth(\bar{a}) = depth(\pi \cdot X) = depth(\,()\,) = 0 \qquad depth([a]t) = 1 + depth(t)$$
$$depth((t_1, t_2)) = 1 + max(depth(t_1), depth(t_2)) \qquad depth(f\ t) = 1 + depth(t)$$

The function *depth* is used as part of the measure provided to ensure termination of the nominal unification algorithm.

Actions of permutations can be homomorphically extended over terms. This means that permutations only change atoms and are accumulated into suspensions. A precise definition is given below.

Definition 2.4 The **action** of a permutation π over terms is defined as:

$$\pi \bullet \bar{a} = \overline{\pi(a)} \qquad \pi \bullet (\pi' \cdot X) = (\pi \circ \pi') \cdot X \qquad \pi \bullet () = ()$$
$$\pi \bullet (t_1, t_2) = (\pi \bullet t_1, \pi \bullet t_2) \qquad \pi \bullet [a]t = [\pi(a)]\pi \bullet t \qquad \pi \bullet f\ t = f\ \pi \bullet t$$

One important observation is that the variables in suspensions work as meta-variables, where a substitution that replaces variables by terms is a primitive notion. With this in mind, it is reasonable that nominal variables are not 'abstractable'. The denomination 'suspension' for $\pi \cdot X$ has to do with the fact that the permutation π cannot indeed apply to X until the instance of this variable is known; so it is suspended.

In PVS, permutations are specified as lists of pairs of atoms. The function `act` applies a permutation to an atom by the recursive action of the swappings

that represent the permutation. On the other hand, the function `ext` extends the action of permutations to terms homomorphically, i.e., it applies `act` to atoms and accumulates permutations in suspensions.

```
- perm: TYPE = list[[atom,atom]]
- act(pi:perm)(c): RECURSIVE atom =
   CASES pi OF
       null: c,
       cons((a,b),rest): LET d = act(rest)(c) IN
                         IF d = a THEN b
                         ELSIF d = b THEN a
                         ELSE d
                         ENDIF
    ENDCASES
   MEASURE pi BY <<

- ext(pi:perm)(t:term): RECURSIVE term =
      CASES t OF
            at(a): at(act(pi)(a)),
            *(pm, v): *(append(pi, pm), v),
            unit: unit,
            pair(t1,t2): pair(ext(pi)(t1),ext(pi)(t2)),
            abs(ab, bd): abs(act(pi)(ab), ext(pi)(bd)),
            app(sl, ag): app(sl, ext(pi)(ag))
     ENDCASES
   MEASURE t BY <<
```

Remark 2.5 The necessity of 'measure' functions in PVS recursive functions is for proving termination according to the operational semantics of termination of PVS. This measure on the parameters should decrease after each recursive call. In the previous functions the measure '`<<`' represents the standard measure on the data structures of permutations and terms; respectively, length of lists and the subterm relation. In some cases, as for these functions, the system can automatically verify the decrement of the measure provided.

Definition 2.6 A **nuclear substitution** is a pair of the form $[X \mapsto s]$, where X is a variable and s is a term, and its **action** over terms is defined as:

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

$$()[X \mapsto s] = () \qquad (t_1, t_2)[X \mapsto s] = (t_1[X \mapsto s], t_2[X \mapsto s])$$

$$[a]t[X \mapsto s] = [a](t[X \mapsto s]) \qquad (f\ t)[X \mapsto s] = f\ (t[X \mapsto s])$$

A **substitution** σ is a list of nuclear substitutions, which are applied one-by-one over terms, i.e:

$$t\, Id = t \text{ , where } Id \text{ is the empty list;} \qquad t\,(\sigma \circ [X \mapsto s]) = (t\,\sigma)[X \mapsto s].$$

Notation: If σ and γ are two substitutions, then $\sigma\gamma$ represents the composition of such substitutions, i.e., $\sigma \circ \gamma$.

Remark 2.7 This notion of substitution is different from the simultaneous application of nuclear substitutions. This approach is closer to triangular substitutions as explored in [KN10], with the view to be more space efficient.

Definition 2.8 The set of **variables** of a term is recursively computed by the

function $Vars$, as follows.

$$Vars(\bar{a}) = \varnothing \qquad Vars(\pi \cdot X) = \{X\} \qquad Vars(\,()\,) = \varnothing$$
$$Vars((t_1, t_2)) = Vars(t_1) \cup Vars(t_2) \qquad Vars([a]t) = Vars(t) \qquad Vars(f\ t) = Vars(t)$$

The next lemma states the invariance of alternating the application of a permutation and a substitution on a term.

Lemma 2.9 *For any term t, $\pi \bullet (t\sigma) = (\pi \bullet t)\sigma$.*

Proof. By induction on the structure of t. □

3 Freshness and α-equivalence

As mentioned earlier, [UPG04] presented an algorithm to decide α-equivalence of nominal terms, based on a notion of freshness of names in terms, without the necessity of generating new names.

Definition 3.1 (Freshness) A **freshness context** ∇ is a finite set of pairs of the form $a\#X$. We say that an atom a is **fresh** in t under ∇ (denoted by $\nabla \vdash a\#t$) if it is possible to build a proof of this judgement using the rules:

$$\frac{}{\nabla \vdash a\#\bar{b}}\ (\#\texttt{ab}) \qquad \frac{}{\nabla \vdash a\#(\,)}\ (\#\texttt{unit}) \qquad \frac{\pi^{-1}(a)\#X \in \nabla}{\nabla \vdash a\#\pi \cdot X}\ (\#\texttt{X})$$
$$\frac{\nabla \vdash a\#s_1 \qquad \nabla \vdash a\#s_2}{\nabla \vdash a\#(s_1, s_2)}\ (\#\texttt{pair}) \qquad \frac{}{\nabla \vdash a\#[a]s}\ (\#\texttt{absa})$$
$$\frac{\nabla \vdash a\#s}{\nabla \vdash a\#[b]s}\ (\#\texttt{absb}) \qquad \frac{\nabla \vdash a\#s}{\nabla \vdash a\#f\ s}\ (\#\texttt{f})$$

Notation: If ∇ and Δ are freshness contexts, then $\nabla \vdash \Delta$ means that $\Delta \subseteq \nabla$ and $\nabla\Delta$ denotes $\nabla \cup \Delta$.

The following two auxiliary lemmas express invariance of derivability in the previous calculus under the action of permutations and weakening of freshness contexts.

Lemma 3.2 $\nabla \vdash a\#t \Leftrightarrow \nabla \vdash \pi \bullet a\#\pi \bullet t$.

Lemma 3.3 *If $\nabla \vdash \Delta$ and $\Delta \vdash a\#t$, then $\nabla \vdash a\#t$.*

The proofs are by induction on the derivation of $\Delta \vdash a\#t$.

Now, with the notions of permutation and freshness, α-equivalence can be defined in a formal way.

Definition 3.4 (α-equivalence) The terms t and s are **α-equivalent** in the context ∇, denoted by $\nabla \vdash t \approx_\alpha s$, if there is a proof of this judgement using the

rules:

$$\frac{}{\nabla \vdash \bar{a} \approx_\alpha \bar{a}}\ (\approx_\alpha \mathtt{a}) \qquad \frac{ds(\pi,\pi')\#X \subseteq \nabla}{\nabla \vdash \pi \cdot X \approx_\alpha \pi' \cdot X}\ (\approx_\alpha \mathtt{X}) \qquad \frac{}{\nabla \vdash () \approx_\alpha ()}\ (\approx_\alpha \mathtt{()})$$

$$\frac{\nabla \vdash s_1 \approx_\alpha t_1 \quad \nabla \vdash s_2 \approx_\alpha t_2}{\nabla \vdash (s_1, s_2) \approx_\alpha (t_1, t_2)}\ (\approx_\alpha \mathtt{pair}) \qquad \frac{\nabla \vdash s \approx_\alpha t}{\nabla \vdash [a]s \approx_\alpha [a]t}\ (\approx_\alpha \mathtt{absa})$$

$$\frac{\nabla \vdash s \approx_\alpha (a\ b) \bullet t \quad \nabla \vdash a\#t}{\nabla \vdash [a]s \approx_\alpha [b]t}\ (\approx_\alpha \mathtt{absb}) \qquad \frac{\nabla \vdash s \approx_\alpha t}{\nabla \vdash f\ s \approx_\alpha f\ t}\ (\approx_\alpha \mathtt{f})$$

where $ds(\pi,\pi') = \{b \in \mathcal{A} \mid \ \pi(b) \neq \pi'(b)\}$ (namely, the difference set between two permutations) and $ds(\pi,\pi')\#X$ is the context formed by the pairs $b\#X$, for each $b \in ds(\pi,\pi')$.

3.1 A direct formalisation of transitivity of α-equivalence

The next four auxiliary lemmas relate α-equivalence, freshness and the action of permutations. The first one expresses preservation of freshness by α-equivalent terms; the second one, alternation of the action of a permutation and its inverse on α-equivalent terms; the third one, invariance of α-equivalence under the action of a permutation; and, the fourth one, preservation of α-equivalence of a term under the action of permutations whose difference set is fresh in the term.

Lemma 3.5 *$\nabla \vdash a\#s$ and $\nabla \vdash s \approx_\alpha t$ implies $\nabla \vdash a\#t$.*

Lemma 3.6 *$\nabla \vdash s \approx_\alpha \pi \bullet t \Rightarrow \nabla \vdash \pi^{-1} \bullet s \approx_\alpha t$.*

Lemma 3.7 *$\nabla \vdash s \approx_\alpha t \Leftrightarrow \nabla \vdash \pi \bullet s \approx_\alpha \pi \bullet t$.*

Lemma 3.8 *$\nabla \vdash ds(\pi_1,\pi_2)\#t$ implies $\nabla \vdash \pi_1 \bullet t \approx_\alpha \pi_2 \bullet t$.*

Lemmas 3.5-3.8 are proved by induction on s, applying Lemma 3.2. For Lemma 3.6, Lemma 3.5 is applied. The treatment is the same as in previous papers ([UPG04,FG07,Urb10]) and their complete formalisations are available in the accompanying PVS development.

The proof of the next lemma is shown in detail because, at this point, the formalisation differs from the one given in [Urb04] and reported in [Urb10].

Lemma 3.9 (Transitivity of α-equivalence) *The relation $\approx_\alpha$ is transitive under a given context ∇, i.e., $\nabla \vdash t_1 \approx_\alpha t_2$ and $\nabla \vdash t_2 \approx_\alpha t_3$ imply $\nabla \vdash t_1 \approx_\alpha t_3$.*

Proof. The proof is by induction on the structure of t_1.

- $t_1 = \bar{a}$: then by definition of $\approx_\alpha$, $t_2 = t_3 = \bar{a}$.
- $t_1 = \pi_1 \cdot X$: so $t_2 = \pi_2 \cdot X$ and $t_3 = \pi_3 \cdot X$. We need to prove that $ds(\pi_1,\pi_3)\#X \subseteq \nabla$. So, take c such that $\pi_1 \bullet c \neq \pi_3 \bullet c$. There are two cases: if $\pi_1 \bullet c = \pi_2 \bullet c$, then $\pi_2 \bullet c \neq \pi_3 \bullet c$ and $c\#X \in \nabla$ for $ds(\pi_2,\pi_3)\#X \subseteq \nabla$; if $\pi_1 \bullet c \neq \pi_2 \bullet c$, then $c\#X \in \nabla$ because $ds(\pi_1,\pi_2)\#X \subseteq \nabla$.
- $t_1 = ()$ implies $t_2 = ()$ and $t_3 = ()$.

- $t_1 = (s_1, s_2)$: then $t_2 = (u_1, u_2)$ and $t_3 = (w_1, w_2)$. By induction hypothesis (IH), $\nabla \vdash s_1 \approx_\alpha w_1$ and $\nabla \vdash s_2 \approx_\alpha w_2$.
- $t_1 = f\ s$: then $t_2 = f\ u$ and $t_3 = f\ w$. By IH, $\nabla \vdash s \approx_\alpha w$.
- $t_1 = [a]s$: then $t_2 = [b]u$ and $t_3 = [c]w$. It is necessary to compare the abstractors:
  - $a = b = c$: thus the result follows by IH trivially.
  - $a = b \neq c$: by definition, $\nabla \vdash s \approx_\alpha u$ and $\nabla \vdash u \approx_\alpha (b\ c) \bullet w$ and $\nabla \vdash b\#w$. By IH, $\nabla \vdash s \approx_\alpha (b\ c) \bullet w$. As $a = b$, then freshness condition is satisfied to a as well.
  - $a \neq b = c$: we have $\nabla \vdash a\#u$, $\nabla \vdash s \approx_\alpha (a\ c) \bullet u$ and $\nabla \vdash u \approx_\alpha w$. By Lemma 3.7, $\nabla \vdash (a\ c) \bullet u \approx_\alpha (a\ c) \bullet w$ and, by IH, $\nabla \vdash s \approx_\alpha (a\ c) \bullet w$. By Lemma 3.2, $\nabla \vdash c\#(a\ c) \bullet u$ and $\nabla \vdash c\#(a\ c) \bullet w$ by Lemma 3.5. Finally, again by Lemma 3.2, $\nabla \vdash a\#w$.
  - $b \neq a = c$: it is known that $\nabla \vdash s \approx_\alpha (b\ c) \bullet u$ and $\nabla \vdash u \approx_\alpha (b\ c) \bullet w$. Then, $\nabla \vdash (b\ c) \bullet u \approx_\alpha w$ by Lemma 3.6. By IH, $\nabla \vdash s \approx_\alpha w$.
  - $a \neq b \neq c \neq a$: it is necessary to prove that $\nabla \vdash s \approx_\alpha (a\ c) \bullet w$ and $\nabla \vdash a\#w$. Let us prove first freshness: by definition of $\approx_\alpha$, $\nabla \vdash a\#u$ and $\nabla \vdash u \approx_\alpha (b\ c) \bullet w$. By Lemma 3.5, $\nabla \vdash a\#(b\ c) \bullet w$ and, by Lemma 3.2($\Leftarrow$), $\nabla \vdash a\#w$. Now let us prove α-equivalence: by hypothesis, $\nabla \vdash s \approx_\alpha (a\ b) \bullet u$, $\nabla \vdash u \approx_\alpha (b\ c) \bullet w$ and $\nabla \vdash b\#w$. By Lemma 3.7, $\nabla \vdash (a\ b) \bullet u \approx_\alpha (a\ b)(b\ c) \bullet w$. As $ds((a\ b)(b\ c), (a\ c)) = \{a, b\}$ and both atoms are fresh in w, then $\nabla \vdash (a\ b)(b\ c) \bullet w \approx_\alpha (a\ c) \bullet w$ by Lemma 3.8. Now, applying IH twice, one obtains $\nabla \vdash s \approx_\alpha (a\ c) \bullet w$. □

Note that the critical point in this proof is the abstraction, particularly when all the abstractors differ. This is due to the asymmetry of rule ($\approx_\alpha$`absb`) in Definition 3.4. The previous lemma was also presented in [UPG04,FG07], but in [Urb10], a weak equivalence notion (Definition 3.10) is used as an intermediate relation to contour the problem with the abstraction case. However, auxiliary lemmas similar to the ones presented here were necessary in [Urb10], in addition to other technical results to deal specifically with this weak equivalence (some of those additional lemmas in [Urb10] are particular cases of transitivity). In the current formalisation, weak equivalence was not needed and the abstractions were treated as given in the five cases in the proof of Lemma 3.9.

Definition 3.10 (Weak-equivalence) Given two terms s, t, they are said to be **weak equivalent** (notation: $s \sim t$) whenever there exists a derivation of $s \sim t$ using the following rules:

$$\frac{}{\bar{a} \sim \bar{a}}\ (\sim\texttt{a}) \qquad \frac{ds(\pi, \pi') = \varnothing}{\pi \cdot X \sim \pi' \cdot X}\ (\sim\texttt{X}) \qquad \frac{}{() \sim ()}\ (\sim\texttt{()})$$

$$\frac{s_1 \sim t_1 \quad s_2 \sim t_2}{(s_1, s_2) \sim (t_1, t_2)}\ (\sim\texttt{pair}) \qquad \frac{s \sim t}{[a]s \sim [a]t}\ (\sim\texttt{absa}) \qquad \frac{s \sim t}{f\ s \sim f\ t}\ (\sim\texttt{f})$$

In the previous definition, observe that when $s \sim t$, then s and t differ only in possible representations of permutations π and π' in suspended variables. Even so, the action of those permutations must be equal. Thus, the relation $\sim$ actually

is closer to syntactic equality than to α-equivalence. To obtain transitivity of $\approx_\alpha$ using this definition, several auxiliary steps are necessary, among others, proving that $\sim$ is invariant under the action of permutations, preservation of freshness by weak-equivalent terms, etc. These lemmas are similar to the previously mentioned for $\approx_\alpha$. In addition, it is necessary to prove that, under a freshness context Δ, $(\approx_\alpha \circ \sim) \subseteq \approx_\alpha$, which is the key property for concluding transitivity of $\approx_\alpha$. All this work is unnecessary in our approach.

Lemma 3.11 (Equivalence) *$\approx_\alpha$ is an equivalence relation under any context ∇.*

Proof. Transitivity is guaranteed by Lemma 3.9. Reflexivity ($\nabla \vdash t \approx_\alpha t$) and symmetry ($\nabla \vdash t \approx_\alpha s$ implies $\nabla \vdash s \approx_\alpha t$) are easy to verify through an inductive proof on the structure of t. The interesting case is the proof of symmetry for abstractions with different abstractors. In this case, $\nabla \vdash [a]t' \approx_\alpha [b]s'$ means $\nabla \vdash t' \approx_\alpha (a\,b) \bullet s'$ and $\nabla \vdash a\#s'$. Applying $(a\,b)$ to the freshness, we obtain $\nabla \vdash b\#(a\,b) \bullet s'$ and, by Lemma 3.5, $\nabla \vdash b\#t'$. Now, by IH, $\nabla \vdash (a\,b) \bullet s' \approx_\alpha t'$ and, by Lemma 3.6, $\nabla \vdash s' \approx_\alpha (a\,b) \bullet t'$. This proves $\nabla \vdash [b]s' \approx_\alpha [a]t'$. □

Notice that, unlike the proofs given in [UPG04,Urb10], this formalised proof of symmetry does not use transitivity. Thus, these two properties are independent from each other.

3.2 Minimal Freshness Contexts

A solution for a unification problem is a pair (∇, σ) of a freshness context and a substitution (see Section 4). A nominal unification algorithm should generate "most general solutions" with respect to an ordering "$\leq$" as in the first-order case (see Definition 4.12). In the current formalisation, a function was specified that can compute a minimal freshness context ∇ which derives a freshness problem $a\#t$ when possible, i.e., $\nabla \vdash a\#t$ and ∇ is a subset of any other context Δ such that $\Delta \vdash a\#t$.

In the next function, the measure "<<" denotes the proper subterm relation that is generated by PVS when the abstract data structure specified for terms is type-checked. As for the example in Remark 2.5, termination with respect to this measure can be automatically verified.

Definition 3.12 Let a be an atom and t be a term. Define the function $\langle \_\#\_ \rangle_{sol}$ that takes as input the pair (a, t) and outputs a freshness context and a Boolean, as follows:

$$
\begin{array}{lll}
\langle a\#t \rangle_{sol} := & CASES\ OF\ t: & \\
 & \bar{b}: & (\varnothing, a \neq b), \\
 & \pi \cdot X: & (\{\pi^{-1} \bullet a\#X\}, True), \\
 & (): & (\varnothing, True), \\
 & (t_1, t_2): & LET\ (\Delta_1, \mathtt{b}_1) = \langle a\#t_1 \rangle_{sol},\ (\Delta_2, \mathtt{b}_2) = \langle a\#t_2 \rangle_{sol} \\
 & & \quad IN\ IF\ \mathtt{b}_1 = \mathtt{b}_2 = True\ THEN\ (\Delta_1\Delta_2, True) \\
 & & \quad\quad ELSE\ (\varnothing, False), \\
 & [b]\hat{t}: & IF\ a = b\ THEN\ (\varnothing, True)\ ELSE\ \langle a\#\hat{t} \rangle_{sol}, \\
 & f\ \hat{t}: & \langle a\#\hat{t} \rangle_{sol} \\
MEASURE << & &
\end{array}
$$

The function above was taken from the transformation rules related to the unification algorithm in [UPG04]. The difference is that here the freshness solutions

are obtained separately from the substitutions which solve the equational problems in the unification algorithm. In this way, it is clear that the freshness constraints can restrict the validity of a unification problem, but they cannot modify the substitution that solves the problem.

The following lemma formalises the correctness of the previous definition.

Lemma 3.13 (Correctness of $\langle \_\#\_ \rangle_{sol}$) *Take* $(\Delta, \mathtt{b}) = \langle a\#t \rangle_{sol}$. *Then,*

(i) $\mathtt{b} = True \Rightarrow \Delta \vdash a\#t$, *and*

(ii) for any ∇, $\nabla \vdash a\#t \Rightarrow \mathtt{b} = True$ *and* $\nabla \vdash \Delta$.

Proof. The proof is by induction on the structure of t. The interesting case is when $t = (t_1, t_2)$, because to use rule (#pair), we need to have the same context in the derivations $\nabla \vdash a\#t_1$ and $\nabla \vdash a\#t_2$. However, the function $\langle \_\#\_ \rangle_{sol}$ returns minimal contexts Δ_1 and Δ_2 to t_1 and t_2, respectively. For this reason, Δ_1 and Δ_2 have to be joined when computing $\langle \_\#\_ \rangle_{sol}$. Then, using Lemma 3.3, it is possible to enlarge the contexts into the derivations $\Delta_1\Delta_2 \vdash a\#t_1$ and $\Delta_1\Delta_2 \vdash a\#t_2$ in order to be able to use the mentioned rule. □

This function is crucial to build independently a freshness context for a whole nominal unification problem from its partial solutions, and it is used in the recursive treatment for the case of abstractions and pairs as will be explained in the next section.

Notation: The function $\langle \cdot \rangle_{sol}$ can be generalised to sets of freshness constraints. In particular, $\langle \nabla\sigma \rangle_{sol} = (\Delta, True)$, where Δ is the union of all the freshness contexts computed by $\langle a\#(id \cdot X)\sigma \rangle_{sol}$, for each $a\#X \in \nabla$, if every subproblem is consistent, and $\langle \nabla\sigma \rangle_{sol} = (\varnothing, False)$ otherwise.

The notation $\Delta \vdash \nabla\sigma$ states that $\Delta \vdash a\#(id \cdot X)\sigma$ is derivable for all $a\#X \in \nabla$.

4 Nominal unification algorithm

In order to construct a nominal unification algorithm as a recursive function in the specification language of PVS, it is necessary to provide a recognisable answer in cases of failure, because PVS does not allow partial functions. To deal with failure, our algorithm will return triplets of the form $(\nabla, \sigma, \mathtt{b})$, which are a freshness context, a substitution and a Boolean, respectively, instead of pairs of the form (∇, σ). The triplet of the form $(\varnothing, Id, False)$ identifies failure cases and triplets of the form $(\nabla, \sigma, True)$ successful cases with solutions of the form (∇, σ). For the sake of efficiency, in failure cases, the freshness context and the substitution are cleared into $\varnothing$ and Id respectively. If any branch fails, then it is not worth to carry partial solutions throughout recursive calls.

Definition 4.1 (Unifiable terms and unifiers) Two terms t, s are said to be **unifiable** if there exists a context ∇ and a substitution σ such that $\nabla \vdash t\sigma \approx_\alpha s\sigma$. Under these conditions, the pair (∇, σ) is called a **unifier** of t and s.

Definition 4.2 (Nominal Unification Function) Let t, s be two nominal terms. Then, we define the function

$$
\begin{array}{rl}
\mathit{unify}(t,s) := & \mathit{IF}\ s = \pi_s \cdot X_s\ \mathit{AND}\ X_s \notin \mathit{Vars}(t)\ \mathit{THEN}\ (\varnothing, [X_s \mapsto \pi_s^{-1} \bullet t], \mathit{True}) \\
& \mathit{ELSE} \\
& \quad \mathit{CASES\ OF}\ (t,s): \\
& \qquad (\pi_t \cdot X,\ \pi_s \cdot X): (ds(\pi_t,\pi_s)\#X,\ \mathit{Id},\ \mathit{True}), \\
& \qquad (\pi_t \cdot X_t,\ s): \mathit{IF}\ X_t \notin \mathit{Vars}(s)\ \mathit{THEN}\ (\varnothing, [X_t \mapsto \pi_t^{-1} \bullet s], \mathit{True}), \\
& \qquad (\bar{a}, \bar{a}): (\varnothing,\ \mathit{Id},\ \mathit{True}), \\
& \qquad ((),()): (\varnothing,\ \mathit{Id},\ \mathit{True}), \\
& \qquad ((t_1,t_2),(s_1,s_2)): \mathit{LET}\ (\nabla_1,\ \sigma_1,\ \mathtt{b}_1) = \mathit{unify}(t_1,s_1), \\
& \qquad\qquad\qquad\qquad (\nabla_2,\ \sigma_2,\ \mathtt{b}_2) = \mathit{unify}(t_2\sigma_1,\ s_2\sigma_1), \\
& \qquad\qquad\qquad\qquad (\nabla_3,\ \mathtt{b}_3) = \langle \nabla_1\sigma_2 \rangle_{sol} \\
& \qquad\qquad\qquad\quad \mathit{IN}\ (\nabla_2\nabla_3,\ \sigma_1\sigma_2,\ \mathtt{b}_1 \wedge \mathtt{b}_2 \wedge \mathtt{b}_3), \\
& \qquad ([a]\hat{t}, [b]\hat{s}): \mathit{IF}\ a = b\ \mathit{THEN}\ \mathit{unify}(\hat{t},\ \hat{s}) \\
& \qquad\qquad\qquad \mathit{ELSE\ LET}\ (\nabla_1,\ \sigma,\ \mathtt{b}_1) = \mathit{unify}(\hat{t},\ (a\,b) \bullet \hat{s}), \\
& \qquad\qquad\qquad\qquad\qquad (\nabla_2,\ \mathtt{b}_2) = \langle a\#\hat{s}\sigma \rangle_{sol} \\
& \qquad\qquad\qquad\qquad\quad \mathit{IN}\ (\nabla_1\nabla_2,\ \sigma,\ \mathtt{b}_1 \wedge \mathtt{b}_2), \\
& \qquad (f\ \hat{t}, f\ \hat{s}): \mathit{unify}(\hat{t},\ \hat{s}), \\
& \qquad \mathit{ELSE}: (\varnothing,\ \mathit{Id},\ \mathit{False}) \\
& \mathit{MEASURE}\ \mathit{lex}(|\mathit{Vars}(t,s)|,\ \mathit{depth}(t))
\end{array}
$$

The measure function provided (see Remark 2.5) is lexicographic, with first component the number of variables in the unification problem and second component the *depth* of the first term of the unification problem.

The next remarks explain how the function $\langle \_\#\_ \rangle_{sol}$ correctly builds the necessary contexts for the abstraction and pair cases avoiding passing as parameter the freshness contexts, as done in unification mechanisms based on transformation rules (cf. [Urb04]). In these remarks, unifiable terms are considered.

Remark 4.3 In case of pairs, $(\nabla_2\nabla_3, \sigma_1\sigma_2)$ has to be a unifier for (t_1,t_2) and (s_1,s_2), i.e., $\nabla_2\nabla_3 \vdash t_1\sigma_1\sigma_2 \approx_\alpha s_1\sigma_1\sigma_2$ and $\nabla_2\nabla_3 \vdash t_2\sigma_1\sigma_2 \approx_\alpha s_2\sigma_1\sigma_2$. Initially, *unify* builds the unifier (∇_1, σ_1) for t_1 and s_1. Afterwards, (∇_2, σ_2) is computed as a unifier for $t_2\sigma_1$ and $s_2\sigma_1$. If $\langle \nabla_1\sigma_2 \rangle_{sol} = (\nabla_3, \mathit{True})$, then $\nabla_1 \vdash t_1\sigma_1 \approx_\alpha s_1\sigma_1$ implies $\nabla_3 \vdash t_1\sigma_1\sigma_2 \approx_\alpha s_1\sigma_1\sigma_2$. Finally, since $\nabla_2 \vdash t_2\sigma_1\sigma_2 \approx_\alpha s_2\sigma_1\sigma_2$, weakening the contexts we obtain the desired unifier.

Remark 4.4 When unifying two abstractions with different abstractors, the answer $(\nabla_1\nabla_2, \sigma)$ has to be a unifier for $[a]t$ and $[b]s$. Indeed, initially the recursive call $\mathit{unify}(t, (a\,b) \bullet s)$ provides a unifier (∇_1, σ) for this problem, if it is possible. Hence, $\nabla_1 \vdash t\sigma \approx_\alpha (a\,b) \bullet s\sigma$, but not necessarily ∇_1 would be able to derive $a\#s\sigma$. Then, $\langle \_\#\_ \rangle_{sol}$ computes the minimal context ∇_2 which derives $a\#s\sigma$ separately. Joining both contexts, the derivation $\nabla_1\nabla_2 \vdash [a]t\sigma \approx_\alpha [b]s\sigma$ can be completed.

Example 4.5 Take the problem of unifying (X, X) and $((a\,b)\cdot X, a)$. First, one unifies X and $(a\,b)\cdot X$. The result is the substitution Id and the context $\{a\#X, b\#X\}$. Then, to unify $X\,Id$ and $a\,Id$, we need the substitution $[X \mapsto a]$ and the empty context $\varnothing$. Then, $\{a\#X, b\#X\}$ is updated with $[X \mapsto a]$, and $\langle a\#a \rangle_{sol}$ returns failure.

Formalisation of termination of the function *unify* is not obtained automatically and requires human intervention to show that $\mathit{lex}(|\mathit{Vars}(t,s)|,\ \mathit{depth}(t))$ decreases

in each recursive call. Observe that there are recursive calls in the cases of pairs, abstractions and applications. In the last two cases one advances on the structure of the first (and second) terms calling recursively a problem with the same number of variables, but smaller *depth*. The same happens for the first recursive call in the case of pairs. For the second recursive call of the case of pairs, when $unify(t_2\sigma_1,\ s_2\sigma_1)$ is computed, if $\sigma_1 \neq Id$, the number of variables in the problem decreases for the nature of the nuclear substitutions generated in suspensions. So it is necessary to prove that the substitutions generated by *unify* have a special characterisation, as explained in the next lemma.

Definition 4.6 (Type $Subs(s)$ substitutions) The substitution $[X_1 \mapsto t_1]\dots[X_n \mapsto t_n]$ is said to be of type $Subs(s)$ if

$$\bigcup_{i=1}^{n} Vars((X_i, t_i)) \subseteq Vars(s) \text{ and } X_i \notin Vars(t_i), \forall i = 1, \dots, n.$$

Lemma 4.7 (Decrement of variables for substitutions of type $Subs(s)$) *Let σ be a substitution of type $Subs(s)$.*

(i) $Vars(t\sigma) \subseteq Vars((t, s))$.

(ii) $\sigma \neq Id$ *implies that* $|Vars(t\sigma)| < |Vars((t, s))|$.

Proof. By induction on the length of σ.

(i) If $\sigma = Id$, then obviously $Vars(t) \subseteq Vars((t, s))$. If $\sigma = \sigma'[X \mapsto u]$, then $t\sigma = (t\sigma')[X \mapsto u]$. By IH, $Vars(t\sigma') \subseteq Vars((t, s))$. As $X \notin Vars(u)$, it is known that $Vars(t\sigma) = Vars(t\sigma'[X \mapsto u]) = Vars((t\sigma', u)) \setminus \{X\} \subseteq Vars((t, s)) \setminus \{X\} \subseteq Vars((t, s))$.

(ii) From (i), $Vars(t\sigma'[X \mapsto u]) \subseteq Vars((t, s)) \setminus \{X\}$. Since $X \in Vars(s)$, the cardinality indeed decreases, i.e., $|Vars((t, s)) \setminus \{X\}| = |Vars((t, s))| - 1$.

□

Lemma 4.8 (Type of subsitutions built by *unify*) *If $unify(t, s) = (\nabla, \sigma, \mathtt{b})$, then the substitution σ is of type $Subs((t, s))$.*

Proof. This is easily checked observing the nuclear substitutions generated in the cases of suspended variables. Note that, one condition to build $[X \mapsto \pi^{-1} \bullet u]$, for instance, is $X \notin Vars(u)$. □

The last two lemmas ensure termination for the function $unify$:

Corollary 4.9 (Termination of *unify*) *The function unify is total.*

Notation: It is said that $\Delta \vdash \sigma \approx_\alpha \gamma$ if, for any Y, $\Delta \vdash (id \cdot Y)\sigma \approx_\alpha (id \cdot Y)\gamma$.

An auxiliary lemma regarding the action of α-equivalent substitutions over a term is necessary for the formalisation of the completeness of the unification algorithm and it is presented below.

Lemma 4.10 $\Delta \vdash \sigma \approx_\alpha \gamma$ *implies* $\Delta \vdash t\sigma \approx_\alpha t\gamma$, *for all term* t.

Proof. By induction on the structure of t. □

The next results are the most difficult part of the formalisation (fully available at trs.cic.unb.br). Soundness and completeness formalisations follow the same inductive proof technique and the analysis of cases are also analogous. Thus, we focus only on completeness.

Lemma 4.11 (Soundness) *Let* $(\nabla, \sigma, \mathtt{b})$ *be the solution for unify*(t, s)*. If* $\mathtt{b} =$ *True, then* (∇, σ) *is a unifier of* t *and* s*.*

Proof. The proof is by induction on $lex(|\,Vars((t,s))|, depth(t))$. □

The previous lemma alone is not enough in the sense that, if the algorithm returns always *False*, then no unifier is provided, even to unifiable terms. The next theorem guarantees that the algorithm actually gives a unifier whenever the terms are unifiable and that the answer is the most general unifier.

Definition 4.12 (More general solutions) Let ∇, Δ be two contexts and γ, σ two substitutions. Then $(\nabla, \gamma) \leq (\Delta, \sigma)$ if there exists θ such that

$$\Delta \vdash \nabla\theta \text{ and } \Delta \vdash \gamma\theta \approx_\alpha \sigma.$$

If (∇, γ) is the least unifier for a unification problem according to "$\leq$", then it is a **most general unifier** (mgu).

Theorem 4.13 (Completeness) *Let* $(\nabla, \gamma, \mathtt{b})$ *be the solution for unify*(t, s)*. If there exists any other solution* (Δ, σ) *for the unification problem, i.e.,* $\Delta \vdash t\sigma \approx_\alpha s\sigma$*, then* $\mathtt{b} = True$ *and* $(\nabla, \gamma) \leq (\Delta, \sigma)$*.*

Proof. The proof is by induction on $lex(|\,Vars(t,s)|,\ depth(t))$. There are some cases to consider: either t or s are suspensions or both have the same structure, that is, t and s are units or abstractions, for instance. That is due to the α-equivalence between $t\sigma$ and $s\sigma$ and the fact that σ cannot change the structure of a term, unless when acting over suspended variables. The proof follows distinguishing cases according to the form (t, s). Below, we present the cases where s is a suspension, both are pairs, and both are abstractions; these are the most interesting cases.

- $(t, \pi \cdot X)$ and $X \notin Vars(t)$: so $\Delta \vdash t\sigma \approx_\alpha (\pi \cdot X)\sigma = \pi \bullet (X\sigma)$ by Lemma 2.9. We need to prove $(\varnothing, [X \mapsto \pi^{-1} \bullet t]) \leq (\Delta, \sigma)$. By definition of $\leq$, it is necessary to provide θ such that $\forall Y :\ \Delta \vdash Y[X \mapsto \pi^{-1} \bullet t]\theta \approx_\alpha Y\sigma$. Instantiate it with σ.
  · $Y \neq X$ implies $\Delta \vdash Y[X \mapsto \pi^{-1} \bullet t]\sigma = Y\sigma \approx_\alpha Y\sigma$.
  · $Y = X$: $\Delta \vdash t\sigma \approx_\alpha \pi \bullet (X\sigma)$ implies $\Delta \vdash \pi^{-1} \bullet (t\sigma) \approx_\alpha X\sigma$, by Lemma 3.6. As $X[X \mapsto \pi^{-1} \bullet t]\sigma = \pi^{-1} \bullet t\sigma$, the α-equivalence is derivable.
- $((t_1, t_2), (s_1, s_2))$: by hypothesis, $\Delta \vdash t_1\sigma \approx_\alpha s_1\sigma$ and $\Delta \vdash t_2\sigma \approx_\alpha s_2\sigma$.
  By IH, $unify(t_1, s_1) = (\nabla_1, \gamma_1, True)$ and $(\nabla_1, \gamma_1) \leq (\Delta, \sigma)$, i.e.,

  $$\text{there exists } \theta \text{ such that } \Delta \vdash \nabla_1\theta \text{ and } \Delta \vdash \gamma_1\theta \approx_\alpha \sigma.$$

  By Lemma 4.10, transitivity and symmetry, $\Delta \vdash t_2\gamma_1\theta \approx_\alpha s_2\gamma_1\theta$, that is, (Δ, θ) is a unifier for $t_2\gamma_1$ and $s_2\gamma_1$.
  Using IH again, with $unify(t_2\gamma_1, s_2\gamma_1) = (\nabla_2, \gamma_2, True)$, we obtain $\Delta \vdash \nabla_2\tilde{\theta}$ and $\Delta \vdash \gamma_2\tilde{\theta} \approx_\alpha \theta$ for some $\tilde{\theta}$.

As $\mathit{unify}((t_1, t_2), (s_1, s_2)) = (\nabla_1\gamma_2\nabla_2, \gamma_1\gamma_2, \mathtt{b})$, all we need to prove is that $\Delta \vdash \gamma_1\gamma_2\tilde{\theta} \approx_\alpha \sigma$ and $\Delta \vdash \nabla_1\gamma_2\tilde{\theta}$ (because $\Delta \vdash \nabla_2\tilde{\theta}$ follows by IH).

By Lemma 4.10, for any variable Y, it is possible to derive

$$\Delta \vdash (id \cdot Y\gamma_1)\gamma_2\tilde{\theta} \approx_\alpha (id \cdot Y\gamma_1)\theta \approx_\alpha id \cdot Y\sigma.$$

So, by transitivity, $\Delta \vdash \gamma_1\gamma_2\tilde{\theta} \approx_\alpha \sigma$ holds.

Finally, as $\Delta \vdash \gamma_2\tilde{\theta} \approx_\alpha \theta$ and $\Delta \vdash \nabla_1\theta$, then $\Delta \vdash \nabla_1\gamma_2\tilde{\theta}$ by Lemmas 3.5 and 4.10.

- $([a]\hat{t}, [b]\hat{s})$: by premisse, $\Delta \vdash a\#\hat{s}\sigma$ and $\Delta \vdash \hat{t}\sigma \approx_\alpha (a\ b) \bullet (\hat{s}\sigma)$; by Lemma 2.9, the latter term is equal to $((a\ b) \bullet \hat{s})\sigma$.

  By IH, $\mathit{unify}(\hat{t}, (a\ b) \bullet \hat{s}) = (\nabla_1, \gamma, True)$ and $(\nabla_1, \gamma) \leq (\Delta, \sigma)$, i.e.,

  $$\text{there is } \theta \text{ such that } \Delta \vdash \nabla_1\theta \text{ and } \Delta \vdash \gamma\theta \approx_\alpha \sigma.$$

  By Lemma 3.5, $\Delta \vdash a\#\hat{s}\sigma$ implies $\Delta \vdash a\#\hat{s}\gamma\theta$. As θ cannot eliminate any inconsistency in "$a\#\hat{s}\gamma$", then $\Delta \vdash a\#\hat{s}\gamma$.

  By Lemma 3.13, as $\langle \_\#\_ \rangle_{sol}$ is complete, so $\langle a\#\hat{s}\gamma \rangle_{sol} = (\nabla_2, True)$.

  Thus, the algorithm computes $\mathit{unify}([a]\hat{t}, [b]\hat{s}) = (\nabla_1\nabla_2, \gamma, True)$. To show that $(\nabla_1\nabla_2, \gamma) \leq (\Delta, \sigma)$, we only need to see that $\Delta \vdash \nabla_2\theta$. Finally, since $(\nabla_2, True) = \langle a\#\hat{s}\gamma \rangle_{sol}$ and $\Delta \vdash a\#\hat{s}\gamma\theta$, then the result follows by Lemma 3.13. □

Example 4.14 The notions of β and η-reduction for the λ-calculus can be defined using a nominal rewriting system [FG07]. In this example, the signature contains term-formers λ of arity 1, and *app* and *subst* of arity 2. Below, application is denoted by juxtaposition and $\mathit{subst}([a]X, Y)$ is written $X[a \mapsto Y]$ as usual (syntactic sugar). Freshness contexts are used in rewrite rules to express conditions on the matching substitutions used to generate the rewrite relation.

$$\begin{array}{lllll}
(\mathsf{Beta}) & & \vdash (\lambda[a]X)Y & \to X[a \mapsto Y] \\
(\mathsf{Eta}) & b\#Z & \vdash \lambda[b](Z\,b) & \to Z \\
(\sigma_{app}) & & \vdash (XX')[a \mapsto Y] & \to X[a \mapsto Y]X'[a \mapsto Y] \\
(\sigma_{var}) & & \vdash a[a \mapsto X] & \to X \\
(\sigma_{lam}) & b\#Y & \vdash (\lambda[b]X)[a \mapsto Y] & \to \lambda[b](X[a \mapsto Y]) \\
(\sigma_{\epsilon}) & a\#X & \vdash X[a \mapsto Y] & \to X
\end{array}$$

To analise one of the overlaps between (Beta) and (Eta), we can compute $\mathit{unify}((\lambda[a]X)Y, Z\,b) = (\varnothing, [Y \mapsto b][Z \mapsto \lambda[a]X], True)$ and apply the resulting substitution to the freshness context $\{b\#Z\}$, obtainig $(\{b\#X\}, True)$. In the case that the version $a\#Z \vdash \lambda[a](Z\,a) \to Z$ of (Eta) is chosen, then the solution of $\mathit{unify}((\lambda[a]X)Y, Z\,a)$ is $(\varnothing, [Y \mapsto a][Z \mapsto \lambda[a]X], True)$ and $\langle \{a\#Z\}[Y \mapsto a][Z \mapsto \lambda[a]X] \rangle_{sol} = (\varnothing, True)$.

5 Conclusions and future work

In this work, a nominal unification algorithm that only takes terms as parameters was presented. Unlike other approaches, which use transformation rules and take the corresponding freshness problems as part of the unification problem, here we have designed a function that can compute the freshness contexts separately. Our nominal unification algorithm is more straightforward and closer to the ones that implement first-order unification.

Additionally, we formalised transitivity for $\approx_\alpha$ in a direct manner without using a weak intermediate relation as in [Urb10]. Here, the proof was based on elementary lemmas about permutations, freshness and α-equivalence; such lemmas are well-known in the context of nominal unification. In [Urb10], the same auxiliary lemmas to demonstrate transitivity were proved, including some extra lemmas to deal with this weak-equivalence. We believe that the current formalisation of transitivity of $\approx_\alpha$ is simpler in the sense that it only uses the essential notions and results. Symmetry of $\approx_\alpha$ is also formalised independently from transitivity, diverging from [UPG04,Urb10].

The style of proof formalised here could have been formalised in any higher-order proof assistant; PVS was chosen with the goal of enriching the libraries for term rewriting systems, as mentioned in the introduction. Important features of PVS such as dependent types can be replaced by other mechanisms in Isabelle/HOL, for instance. For example, the substitution generated in the computation of $\mathit{unify}(\mathtt{t},\mathtt{s})$ must be of type `Subs_unif(t,s)` (this is the PVS specification for the type $\mathit{Subs}((\mathtt{t},\mathtt{s}))$ in Definition 4.6) in order to prove termination. In Isabelle/HOL, this is overcome by defining substitutions in a slightly different way. PVS also allows to use type variables when defining a theory; those variables can be parameterised when such theory is imported by another one. In Isabelle/HOL, parameterising theories is not straightforward, but functions can be defined polymorphically, which provides different feasible solutions for the same kind of formalisation. Of course, a formalisation in Isabelle/HOL will bring out the possibility of a direct comparison regarding the previous formalisations of unification in [Urb04], but it should be emphasised that the advantages of the current formalisation arise from the differences in the theoretical proofs.

Future work: Although nominal approaches have several advantages in the treatment of bound variables, there is still work to be done regarding the study of relevant computational properties. At a first glance, a subsequent study to be done is applying nominal unification for the construction of a nominal completion algorithm *à la* Knuth-Bendix as part of a PVS development for nominal rewriting. A completion algorithm for closed nominal rewriting systems is provided in [FR12].

Another possible application of this formalisation of the nominal unification algorithm is in the verification of nominal resolution approaches (as done, for instance, in the propositional case in [CM09]).

References

[ABW07] B. Aydemir, A. Bohannon, and S. Weihrich. Nominal Reasoning Techniques in Coq (Extended Abstract). *Electronic Notes in Theoretical Computer Science*, 174(5):69–77, 2007.

[AGdMA14] A. B. Avelar, A. L. Galdino, F. L. C. de Moura, and M. Ayala-Rincón. First-order unification in the PVS proof assistant. *Logic Journal of the IGPL*, 22(5):758–789, 2014.

[ARK01] M. Ayala-Rincón and F. Kamareddine. Unification via the λs_e-Style of Explicit Substitution. *Logic Journal of the IGPL*, 9(4):489–523, 2001.

[Cal13] C. Calvès. Unifying Nominal Unification. In *24^{th} International Conference on Rewriting Techniques and Applications, RTA 2013*, volume 21 of *LIPIcs*, pages 143–157, 2013.

[CF10] C. Calvès and M. Fernández. The first-order nominal link. In *Logic-Based Program Synthesis and Transformation - 20^{th} International Symposium, LOPSTR 2010, Revised Selected Papers*, volume 6564 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2010.

[Che05] J. Cheney. Relating nominal and higher-order pattern unification. In *Proceedings of the 19^{th} International Workshop on Unification (UNIF 2005)*, pages 104–119, 2005.

[CM09] R. Constable and W. Moczydlowski. Extracting the resolution algorithm from a completeness proof for the propositional calculus. *Annals of Pure and Applied Logic*, 161(3):337–348, 2009.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order Unification via Explicit Substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

[FG07] M. Fernández and M. J. Gabbay. Nominal Rewriting. *Information and Computation*, 205(6):917–965, June 2007.

[FR12] M. Fernández and A. Rubio. Nominal Completion for Rewrite Systems with Binders. In *Automata, Languages, and Programming - 39^{th} International Colloquium, ICALP 2012, Part II*, volume 7392 of *Lecture Notes in Computer Science*, pages 201–213, 2012.

[GAR10] A. L. Galdino and M. Ayala-Rincón. A Formalization of the Knuth-Bendix(-Huet) Critical Pair Theorem. *Journal of Automated Reasoning*, 45(3):301–325, 2010.

[Hue75] G. P. Huet. A Unification Algorithm for Typed $\boldsymbol{\lambda}$-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[KN10] R. Kumar and M. Norrish. (Nominal) Unification by Recursive Descent with Triangular Substitutions. In *Interactive Theorem Proving, First International Conference, ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2010.

[LV10] J. Levy and M. Villaret. An efficient nominal unification algorithm. In *Proceedings of the 21^{st} International Conference on Rewriting Techniques and Applications, RTA 2010*, volume 6 of *LIPIcs*, pages 209–226, 2010.

[LV12] J. Levy and M. Villaret. Nominal unification from a higher-order perspective. *ACM Transactions on Computational Logic*, 13(2):10, 2012.

[Pau85] L.C. Paulson. Verifying the Unification Algorithm in LCF. *Science of Computer Programming*, 5(2):143–169, 1985.

[SORSC01] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide. Technical report, SRI International, 2001. http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf.

[UPG04] C. Urban, A. M. Pitts, and M. Gabbay. Nominal Unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.

[Urb04] C. Urban. Formalisation of Nominal Unification in Isabelle/HOL. Technical report, TU Munich, 2004. http://www4.in.tum.de/∼urbanc/Unification, last visited April 2015.

[Urb08] C. Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[Urb10] C. Urban. Nominal Unification Revisited. In *Proceedings 24^{th} International Workshop on Unification, UNIF 2010*, volume 42 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–11, 2010.