Formalising Nominal AC-Unification

Mauricio Ayala-Rincón[1,3]*, Maribel Fernández[2], and Gabriel Ferreira Silva[3]†

[1] Department of Computer Science, Universidade de Brasília
ayala@unb.br
[2] Department of Informatics, Kings College London
maribel.fernandez@kcl.ac.uk
[3] Department of Mathematics, Universidade de Brasília
gabrielfsilva1995@gmail.com

Abstract

The nominal setting extends first-order syntax and represents smoothly systems with variable bindings, using instead of variables, nominal atoms and atom permutations for renaming them. Nominal unification adapts first-order unification modulo α-equivalence by taking into account this nominal approach. Nominal AC-unification is then simply nominal unification with associative-commutative function symbols. In this paper, we present a functional specification of a nominal AC-unification algorithm and discuss relevant aspects of current work on formalising its soundness and completeness. The algorithm explores the combinatory of the problem without taking into consideration efficiency, simplifying, in this manner, the formalisation.

1 Introduction

The nominal setting allows us to extend first-order syntax and represent smoothly systems with bindings, which are frequent in mathematics and computer science. Nevertheless, to represent these bindings correctly, α-equivalence must be taken into account. For instance, despite their syntactical difference, the formulas $\exists y : y > 0$ and $\exists z : z > 0$ should be considered equivalent. The nominal theory allows us to deal with these bindings in a natural way, instead of using indices as in explicit substitutions *à la de Bruijn* (e.g. [8], [7]).

1.1 Related Work

Nominal Unification was originally solved by Urban, Pitts and Gabbay [10], who proposed a set of inference rules to compute the most general unifier of a (solvable) nominal unification problem. The rules were formalised and proved to be correct and complete with the help of the proof assistant Isabelle/HOL [9]. A functional nominal unification algorithm was later formalised in PVS and proved correct and complete [4]. Nominal unification was extended to take into account commutative axioms [1]: a nominal C-unification algorithm given as a set of inference rules was proposed and formalised sound and complete in Coq. Based on the previous two papers, a functional algorithm for nominal C-unification was specified and verified in PVS [2]. In the standard first-order syntax, the first formalisation of AC-matching, introduced in [6], opens the way for formalisations of AC-unification, which is (to the best of our knowledge) yet to come.

*Author partially funded by CNPq research grant number 307672/2017-4.
†Author partially funded by CNPq scholarship scholarship number 139271/2017-1.

1.2 Contributions and Possible Applications

In this extended abstract, we discuss a functional specification of a nominal AC-unification algorithm pointing out interesting aspects of its formalisation in PVS. Although the formalisation is not yet complete, the specification of the algorithm is finished and fully available at: www.github.com/gabriel951/ac-unification. The work here presented, when completed, would not only give the first formalisation of nominal AC-unification but also, as far as we know, the first formalisation of first-order AC-unification, since the nominal theory encompasses first-order theory.

Since unification has applications in logic programming systems, type inference algorithms, theorem provers and so on, a nominal AC-unification algorithm has interesting potential uses. It could, for instance, be used in a logic programming language that employs the nominal setting, such as α-Prolog (see [5]). Another possibility is to translate the algorithm into an AC-matching algorithm. Since 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 a substitution, the translation to AC-matching would be performed by marking all variables that occur in t as "protected" variables at the beginning of the matching process and adapting the algorithm to prohibit the instantiation of "protected" variables. These AC algorithms could then be used to extend nominal rewriting [7] or nominal narrowing [3].

2 Preliminaries

Only the part of nominal theory relevant to unifying AC function symbols is explained. For a complete account, one can check, for instance, [4] or [3].

2.1 Nominal Terms, Permutations and Substitutions

In nominal theory, we have a countable set of atoms $\mathcal{A} = \{a, b, c, ...\}$ and a countable set of variables $\mathcal{X} = \{X, Y, Z, ...\}$, which are disjoint. A permutation π is a bijection of the form $\pi : \mathcal{A} \to \mathcal{A}$ such that the set of atoms that are modified by π is finite.

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

$$s, t \quad ::= \quad \langle\rangle \;\mid\; \overline{a} \;\mid\; \pi \cdot X \;\mid\; [a]t \;\mid\; \langle s, t\rangle \;\mid\; f\ t \;\mid\; f^{AC}\ t$$

where $\langle\rangle$ is the unit, $\overline{a}$ is an atom term, $\pi \cdot X$ is a suspended variable (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 $f^{AC}\ t$ is an AC function application.

Remark. Although the function application has arity one, this is not a limitation, for we can use the pair to encode tuples with an arbitrary number of arguments. For instance, the tuple (t_1, t_2, t_3) could be constructed as $\langle t_1, \langle t_2, t_3\rangle\rangle$.

Finally, a substitution σ in the nominal setting is analogous to the concept in first-order theory: a mapping that sends a finite amount of variables in $\mathcal{X}$ to terms in $\mathcal{T}$.

2.2 Freshness and α-Equality

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

- $a\#t$ means that if a occurs in t then it does so under an abstractor [a].
- $s \approx_\alpha t$ means that s and t are α-equivalent.

After explaining the ideas behind these two concepts, we define them formally for AC function symbols.

Definition 2 (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 ∇ (which we denote by $\nabla \vdash a\#t$) if it is possible to build a proof using the rules for freshness, in accordance with the type of t (see [7]). The rule for freshness of an AC function application is the same for a function application.*

Definition 3 (α-equality with AC operators)**.** *Two terms t and s are said to be α-equivalent under the context Δ, written as $\Delta \vdash t \approx_\alpha s$, if it is possible to build a proof using the rules for α-equivalence, in accordance with the type of t (see [7]). The rule for an AC function symbol is:*

$$\frac{\Delta \vdash S_1(fs) \approx_\alpha S_i(ft) \qquad \Delta \vdash D_1(fs) \approx_\alpha D_i(ft)}{\Delta \vdash fs \approx_\alpha ft} \ (\approx_\alpha \texttt{ac-app})$$

for some i. Here f is an AC function symbol, $S_n(f*)$ is an operator that selects the nth argument of the flattened subterm $f*$ and $D_n(f*)$ is an operator that deletes the nth argument of the flattened subterm $f*$.

Remark. If the flattened subterm $f*$ contains only two arguments, then $D_n(f*)$ will contain only one argument. Also, if the flattened subterm $f*$ contains only one argument, then $D_n(f*)$ returns the unit. For instance, $D_1(f\langle a, b\rangle) = fb$ and $D_1(fb) = \langle\rangle$.

Example 1. *Let f be an AC-function symbol. In the above definition,*
$S_2(f\langle f\langle a, b\rangle, f\langle [a]X, \pi \cdot Y\rangle\rangle)$ is b, and $D_2(f\langle f\langle a, b\rangle, f\langle [a]X, \pi \cdot Y\rangle\rangle\rangle)$ is $f\langle fa, f\langle [a]X, \pi \cdot Y\rangle\rangle\rangle$.

2.3 Nominal AC-Unification

Definition 4 (Unification Problem)**.** *A unification problem is a pair $\langle \Delta, P\rangle$ where Δ 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, with s and t terms and a an atom.*

Example 2. *An example of a unification problem with an empty context and one equation constraint: $\langle \Delta, P\rangle = \langle \emptyset,\ f^{AC}\langle f^{AC}\langle X, Y\rangle, c\rangle\rangle \ \approx_?\ f^{AC}\langle c, f^{AC}\langle a, b\rangle\rangle\rangle$*

Remark. Let ∇ and ∇' freshness contexts and σ and σ' substitutions. In order to define a solution to a unification problem, we need the following notation:

- $\nabla' \vdash \nabla\sigma$ 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 5 (Solution for a Triple or Problem)**.** *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:*

- $\nabla \vdash \Delta\sigma$
- $\nabla \vdash s\sigma \approx_\alpha t\sigma$, *if* $s \approx_? t \in P$
- $\nabla \vdash a\#t\sigma$, *if* $a\#_? t \in P$
- *There exist 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 $\langle \Delta, id, P\rangle$.

Remark. As in C-Unification, in AC-unification equations of the form $\pi \cdot X \approx_\alpha \pi' \cdot X$, called fixed point equations, are not solved because there is an infinite number of solutions to them. Instead, they are carried on, as part of the solution to the unification problem [1].

Example 3. *Consider the unification problem of Example 2:* $\langle \Delta, id, P\rangle = \langle \emptyset, id,\ f^{AC}\langle f^{AC}\langle X, Y\rangle, c\rangle \approx_?\ f^{AC}\langle c, f^{AC}\langle a, b\rangle\rangle\rangle$. *A possible solution is* $\langle \emptyset, \{X \to a, Y \to b\}\rangle$

3 Specification

We specified a functional nominal AC-unification algorithm for unifying two terms t and s. The algorithm is recursive, calling itself on progressively simpler versions of the problem until it has finished. It is an extension of the algorithm in [2], in order to deal with the case of t or s being rooted by an AC function symbol. See also Appendix A.

When one of t or s is rooted by an AC function symbol and the other term is a suspended variable, the algorithm instantiates the suspended variable term appropriately and solves the problem. Alternatively, when one of the terms is rooted by an AC function symbol and the other is not a suspended variable or rooted by the same AC function symbol, the algorithm recognises it is a situation where no solution is possible. The interesting case is, therefore, when both t and s are rooted by the same AC function symbol.

In this situation, the algorithm first extracts all arguments of t and then generates all *pairings* of those arguments, in any order. After that, the algorithm extracts all arguments of s and then generates all pairings of these arguments, again in any order. Finally, the algorithm tries to unify every pairing of arguments of t with every pairing of arguments of s.

Example 4. *Suppose we are trying to unify the terms in Example 2.*

- *The two pairings generated for the term* $f^{AC}\langle f^{AC}\langle X, Y\rangle, c\rangle$ *in the order* (X, Y, c) *are:* $\langle X, \langle Y, c\rangle\rangle$ *and* $\langle\langle X, Y\rangle, c\rangle$. *Two pairings would be generated for every order and the possible orders are:* (X, Y, c), (X, c, Y), (Y, X, c), (Y, c, X), (c, X, Y) *and* (c, Y, X).
- *The twelve pairings generated for the term* $f^{AC}\langle c, f^{AC}\langle a, b\rangle\rangle$ *include, for instance:* $\langle c, \langle a, b\rangle\rangle$, $\langle\langle c, a\rangle, b\rangle$, $\langle\langle b, c\rangle, a\rangle$ *and* $\langle a, \langle b, c\rangle\rangle$.

Remark. Our first idea to deal with the pairings of the arguments of t and s was to generate all pairings of the arguments of t, preserving the order and all pairings of the arguments of s, in any order. This approach, however, is not complete.

To see that, consider f an AC function symbol, $t = f\langle a, \langle b, c\rangle\rangle$ and $s = f\langle X, b\rangle$. The substitution $\sigma = \{X \to \langle a, c\rangle\}$ would not be found if we had generated only the pairings of the arguments in t preserving the order, but it can be found by our approach. That is because the substitution σ is found when trying to unify $\langle\langle a, c\rangle, b\rangle$ with $\langle X, b\rangle$ and the arguments of the pairing $\langle\langle a, c\rangle, b\rangle$ are not in the same order that they are in t.

4 Formalisation

Theorems 1 and 2 formalise soundness and completeness of unifying AC functions. The function `gen_unif_prb`(ft, fs), for f AC, generates all pairings of ft (in any order) and all pairings of fs (in any order) and then combines them to generate a list of unification problems.

Theorem 1 (Soundness of Unifying AC functions)**.** *Let* ft *and* fs *be AC function applications. Suppose that* $(t_1, s_1) \in$ `gen_unif_prb`(ft, fs) *and that* $\nabla \vdash t_1\sigma \approx_\alpha s_1\sigma$. *Then,* $\nabla \vdash (ft)\sigma \approx_\alpha (fs)\sigma$.

Theorem 2 (Completeness of Unifying AC functions)**.** *Let ft and fs be AC function applications. Suppose that $\nabla \vdash (ft)\sigma \approx_\alpha (fs)\sigma$. Then, there exists $(t_1, s_1) \in \texttt{gen\_unif\_prb}(ft, fs)$ such that $\nabla \vdash t_1\sigma \approx_\alpha s_1\sigma$.*

The natural way of proving the theorem of soundness would be by induction on the size of the term. If we had decided to prove the theorem directly, we would find the i that makes $\nabla \vdash S_1((ft)\sigma) \approx_\alpha S_i((fs)\sigma)$ and then try to use the induction hypothesis to prove that $\nabla \vdash D_1((ft)\sigma) \approx_\alpha D_i((fs)\sigma)$. We would not, however, be able to apply the induction hypothesis, since the term being deleted of $t\sigma$ could be the first term of t but it could have also being introduced by the substitution σ. A similar problem could happen for s. To get around this problem, we must first eliminate the substitutions from our problem, and then solve a version of the problem without substitutions by induction. As explained next, Lemmas 1 and 2, together with a convenient renaming of variables, are used to eliminate the substitution, while Lemma 3 solves a simplified version of the problem.

For taking substitutions out of the equation, we will need the operator F_{AO}, which generates all possible flattened versions of a term, in any order. Therefore, after applying this operator, we get a term that is not an AC function application.

Lemma 1. *Let ft and fs be AC applications. Suppose that $(t_1, s_1) \in \texttt{gen\_unif\_prb}(ft, fs)$. Then, $\forall t'_1 \in F_{AO}(t_1\sigma),\ s'_1 \in F_{AO}(s_1\sigma)$: $(t'_1, s'_1) \in \texttt{gen\_unif\_prb}(ft\sigma, fs\sigma)$.*

Remark. The operator F_{AO} is needed in the Lemma 1 because, although we have the guarantee that t_1 and s_1 are pairings of ft and fs, the substitution σ may reintroduce the AC function symbol f into the terms $t_1\sigma$ and $s_1\sigma$. Since an output of $\texttt{gen\_unif\_prb}((ft)\sigma, (fs)\sigma)$ cannot contain the AC function symbol, we must apply the flattener operator F_{AO} to $t_1\sigma$ and $s_1\sigma$. A case where this reintroduction of the AC function symbol occurs is illustrated in Example 5.

Example 5. *Let f be an AC function symbol. Consider $ft = fX$, $fs = fY$ and $\sigma = \{X \to f\langle a, b\rangle, Y \to f\langle b, a\rangle\}$. Then, $t_1 = X$ and $s_1 = Y$ do not indeed contain the AC function symbol f. However, the substitution σ given reintroduces this AC function symbol and we have $t_1\sigma = f\langle a, b\rangle$ and $s_1\sigma = f\langle b, a\rangle$.*

Lemma 2. *Let ft and fs be AC applications. Suppose that $(t_1, s_1) \in \texttt{gen\_unif\_prb}(ft, fs)$ and that $\nabla \vdash t_1\sigma \approx_\alpha s_1\sigma$. Then, $\exists t'_1 \in F_{AO}(t_1\sigma), s'_1 \in F_{AO}(s_1\sigma) : \nabla \vdash t'_1 \approx_\alpha s'_1$.*

Using Lemmas 1 and 2 we can obtain, from our original hypothesis of soundness, the existence of t'_1 and s'_1 such that $(t'_1, s'_1) \in \texttt{gen\_unif\_prb}((ft)\sigma, (fs)\sigma)$ and $\nabla \vdash t'_1 \approx_\alpha s'_1$ and we must prove that $\nabla \vdash (ft)\sigma \approx_\alpha (fs)\sigma$. Then, with a convenient renaming of variables, this is equivalent to proving Lemma 3, which can be done by induction on the size of the term t.

Lemma 3. *Let ft and fs be AC function applications, with the same function symbol. Suppose that $(t_1, s_1) \in \texttt{gen\_unif\_prb}(ft, fs)$ and that $\nabla \vdash t_1 \approx_\alpha s_1$. Then, $\nabla \vdash ft \approx_\alpha fs$.*

A similar analysis could be applied to prove the lemma of completeness.

Remark. We have not yet fully formalised any of the stated lemmas or theorems.

5 Conclusion

We commented on the specification of a functional algorithm that has been developed for doing nominal AC-unification and highlighted important points of the formalisation we are working on. We opted for a simple and inefficient algorithm in order to simplify the formalisation. Since

nominal AC-unification extends first-order AC-unification, completing this work is significant also to provide a formalisation of first-order AC-unification, which to the best of our knowledge does not yet exist.

References

[1] Mauricio Ayala-Rincón, Washington de Carvalho-Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal C-unification. In *27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017), Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 235–251. Springer, 2018.

[2] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. Soundness and completeness in PVS of a functional nominal C-unification algorithm. Available at http://ayala.mat.unb.br/publications.html, 2019.

[3] Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal narrowing. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD)*, volume 52 of *LIPIcs*, pages 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[4] Mauricio Ayala-Rincón, Maribel Fernández, and Ana Rocha-Oliveira. Completeness in PVS of a nominal unification algorithm. *Electronic Notes in Theoretical Computer Science*, 323:57–74, 2016.

[5] James Cheney and Christian Urban. α-prolog: A logic programming language with names, binding and α-equivalence. In *20th International Conference on Logic Programming (ICLP)*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004.

[6] Evelyne Contejean. A certified AC matching algorithm. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

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

[8] Andrew Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.

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

[10] Christian Urban, Andrew Pitts, and Murdoch Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.

A Nominal AC-unification algorithm

The algorithm is specified in the functional PVS language and is available at `www.github.com/gabriel951/ac-unification`, as mentioned in the introduction. For brevity, we present here a partial description emphasising the elements related with AC functions. The treatment given to the other nominal elements can be checked in [2], where the case of nominal C-unification is treated.

Algorithm 1 is recursive and keeps track of the current context (Δ), the substitutions made so far (σ), the remaining terms to unify ($PrbLst$) and the current fixed point equations ($FPEqLst$). Therefore, the algorithm receives as input the quadruple $(\Delta, \sigma, PrbLst, FPEqLst)$.

Although extense, the algorithm is straightforward. It starts by analysing $PrbLst$, the list of terms it needs to unify. If it 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 proceeds by trying to unify the terms t and s that are in the head of the list. Only after that it goes to the tail of the list.

When the algorithm is unifying two AC function symbols t and s, the first step is to generate all pairings of t and s. As mentioned before, this is done by function `gen_unif_prb`. The result is a list of unification problems (each of these unification problems corresponds to an attempt of unifying a pairing for t and a pairing for s), stored in variable $NewPrbLst$. The algorithm will call the function UNIFY recursively, with the same Δ, σ and $FPEqLst$, but with a new list of unification problems, consisting of one unification problem from $NewPrbLst$ concatenated with the list of remaining problems $PrbList'$. The algorithm will do this for every unification problem in $NewPrbLst$. The algorithm does that in two steps. In the first step, the function `get_lst_quad` is called, constructing a list of quadruples, which are stored in variable $LstQuad$. Each quadruple in $LstQuad$ is of the form $(\Delta, \sigma, PrbLst, FPEqLst)$. Next, the `map` function applies recursively the function UNIFY to every quadruple in $LstQuad$ and since every application of UNIFY generates a list of solutions, the returned value of the `map` function is a list of lists of solutions, stored in variable $LstLstSol$. The algorithm then returns a flattened version of this list, via the use of the FLATTEN function.

Algorithm 1 Functional Nominal AC-Unification

procedure UNIFY($\Delta, \sigma, PrbLst, FPEqLst$)
  if null($PrbLst$) **then**
    return list($(\Delta, \sigma, FPEqLst)$)
  else
    $(t, s) + PrbLst' = PrbLst$
    if ($s == \pi \cdot X$) and (X not in t) **then**
      check [2]
    else
      if $t == a$ **then**
        check [2]
      else if $t == \pi \cdot X$ **then**
        check [2]
      else if $t == \langle\rangle$ **then**
        check [2]
      else if $t == \langle t_1, t_2 \rangle$ **then**
        check [2]
      else if $t == [a]t_1$ **then**
        check [2]
      else if $t == f\ t_1$ **then**
        check [2]
      else if $t == f^C\langle t_1, t_2 \rangle$ **then**
        check [2]
      else ▷ t is of the form $f^{AC}t'$
        if s != $f^{AC}s'$ **then return** null
        else
          $NewPrbLst =$ `gen_unif_prb`(t, s)
          $LstQuad =$ `get_lst_quad`$(NewPrb, \Delta, \sigma, PrbList', FPEqLst)$
          $LstLstSol = map($UNIFY$, LstQuad)$
          return FLATTEN($LstLstSol$)
        end if
      end if
    end if
  end if
end procedure