King's Research Portal

Document Version
Peer reviewed version

[Link to publication record in King's Research Portal](#)

Download date: 27. Nov. 2023

Nominal AC-Matching

Mauricio Ayala-Rincón[1][0000−0003−0089−3905], Maribel
Fernández[2][0000−0002−1959−8730], Gabriel Ferreira Silva[1][0000−0003−1679−3597],
Temur Kutsia[3][0000−0003−4084−7380], and Daniele
Nantes-Sobrinho[1,4][000−0002−1959−8730]

[1] University of Brasília, Brazil
ayala@unb.br, and gabrielfsilva1995@gmail.com
[2] King's College London, U.K. maribel.fernandez@kcl.ac.uk
[3] Johannes Kepler University Linz, Austria kutsia@risc.jku.at
[4] Imperial College London, U.K dnantess@ic.ac.uk

Abstract. The nominal syntax is an extension of the first-order syntax that smoothly represents languages with variable bindings. Nominal matching is first-order matching modulo alpha-equivalence. This work extends a certified first-order AC-unification algorithm to solve nominal AC-matching problems. To our knowledge, this is the first mechanically-verified nominal AC-matching algorithm. Its soundness and completeness were verified using the proof assistant PVS. The formalisation enriches the first-order AC-unification algorithm providing structures and mechanisms to deal with the combinatorial aspects of nominal atoms, permutations and abstractions. Furthermore, by adding a parameter for "protected variables" that cannot be instantiated during the execution, it enables nominal matching. Such a general treatment of protected variables also gives rise to a verified nominal AC-equality checker as a byproduct.

Keywords: Nominal Matching · Nominal AC-Matching · Formal Methods · PVS

1 Introduction

The nominal approach to the specification of systems with binders [20,25] extends first-order syntax with notions of name and binding that allow us to represent systems with binders smoothly. Such systems frequently appear in the formalisation of mathematics and when reasoning about the properties of programming languages. Taking into account α-equivalence is essential to represent bindings correctly. For example, the formulas $\forall x : x + 1 > 0$ and $\forall y : y + 1 > 0$ should be considered equivalent despite being syntactically different. From the user point of view it is easier to use systems with variable names than systems with indices. Hence, instead of using indices to represent bound variables, as in explicit substitution calculi à la de Bruijn, the nominal theory uses atoms, atom permutations and freshness constraints to represent binders more naturally [25,19].

Given terms t and s, syntactic unification is the problem of finding a substitution σ such that $\sigma t = \sigma s$ and syntactic matching is the problem of finding a substitution σ such that $\sigma t = s$. Algorithms to solve matching problems are an essential component of functional languages and equational theorem provers: matching is used to decide if an equation can be applied to a term. The problem of syntactic matching can be generalised to consider an equational theory E. In this case, called E-matching, we must find a substitution σ such that σt and s are equal modulo E, which we denote $\sigma t \approx_E s$. For example, if the system includes associative and commutative (AC) operators, such as $+$ in the example above, then the matching algorithm should consider the AC axioms. Furthermore, equational programming languages, such as Maude, require efficient implementations of AC-matching to deal with AC-theories (see [16]).

If the system under study includes binders and AC operators, then α-equivalence should also be considered: for example, $\forall x : x + 1 > 0$ should be considered equivalent to $\forall y : 1 + y > 0$. This paper focuses on the matching problem for languages that include binders and AC operators.

Nominal matching is the extension of first-order matching to the nominal syntax, replacing the notion of syntactic equality by α-equivalence. It has applications in rewriting, functional programming, and metaprogramming. For instance, various versions of matching modulo α-equivalence are used in functional programming languages that provide constructs for manipulating abstract syntax trees involving binders (e.g. [29,26]). In this work, we specify a nominal matching algorithm modulo AC function symbols (nominal AC-matching, for short) and prove its correctness and completeness using the proof assistant PVS.

Related Work. Nominal syntactic (i.e. modulo α-equivalence) equality-check, matching and unification were solved since the beginning of the development of the nominal approach; more than twenty years ago, Urban et al. [34] developed the first rule-based algorithm for nominal syntactic unification and further, Urban mechanised its correctness and completeness in Isabelle/HOL as part of the formalisation of the nominal approach in this proof assistant [32,33]. Furthermore, different approaches were designed to deal with nominal syntactic unification efficiently. Calvès and Fernández [12,11] and Levy and Villaret [22,23] developed efficient nominal syntactic unification algorithms to solve nominal unification problems. Furthermore, Ayala-Rincón et al. [6] developed a nominal syntactic unification algorithm specified as a functional program and verified it in the proof assistant PVS. Enriching the nominal equational analysis with equational theories started with developing rule-based techniques for commutative operators. Such developments were initially checked in the proof assistant Coq and further in PVS [1,4]. Remarkable differences between nominal unification and nominal C-unification were discovered, such as the fact that when expressing solutions as pairs consisting of a freshness context and substitutions, nominal unification is unitary whereas nominal C-unification is not finitary [2,3].

Avoiding freshness constraints through a fixed-point approach was also studied as a mechanism to obtain finite complete sets of solutions [5]. Such fixed-point

equations also appear in nominal techniques designed to deal with higher-order recursive let operators [27,28].

First-order AC-unification algorithms were proposed almost half a century ago, when Stickel [30,31] showed the connection between solving this problem and computing solutions to linear Diophantine equations until a certain bound. Almost a decade later, Fages [17,18] fixed a mistake in Stickel's proof of termination. Since then, ideas to obtain more efficient AC-unification algorithms have been proposed, either by using a smaller bound when computing the solutions to the linear Diophantine equation [14], or by solving those equations more efficiently [14], or even by solving whole systems of linear Diophantine equations and using suitable data structures to represent the problem [10,8]. First-order AC-unification algorithms were not formalised until recently when a version of Fages' AC-unification algorithm was proved correct and complete using the proof assistant PVS [7]. This mechanisation applies the linear-Diophantine AC unification method discovered and fixed in works by Stickel and Fages [30,31,17,18], and can easily be adapted to deal with AC-equality and AC-matching problems as well. It is important to stress that such mechanisation was not a routine-formalisation effort; before this formalisation, only a formalisation of AC-matching (which has simpler combinatorics) was reported in the proof assistant Coq [15].

Contributions. Adapting first-order syntactic AC unification to the nominal setting is challenging since the new variables included in the Diophantine systems (used to generate new possible AC combinations) give rise to new AC-unification problems of the same complexity as the input problems. This paper shows that such cyclicity is not possible when only nominal AC-matching problems are considered. We present a novel nominal AC-matching algorithm adapted from the Stickel-Fages linear-Diophantine approach and prove its termination, correctness and completeness in the proof assistant PVS.

Organisation. Section 2 recalls the main concepts and notations needed in the paper. In Section 3, we present and explain the pseudocode for the algorithm specified in PVS. Section 4 discusses the main features of the formalisation, while Section 5 discusses the challenges in adapting our approach to nominal AC-unification. Finally, in Section 6, we conclude the paper and suggest possible paths for future work. We assume familiarity with PVS (see [24]) and include hyperlinks (with the ⤴ icon) to specific points of interest of the PVS formalisation. An extended version of this paper is available at https://www.mat.unb.br/ayala/publications.html.

2   Background

2.1   Nominal Terms, Permutations and Substitutions

Assume disjoint countable sets of atoms $\mathbb{A} = \{a, b, c, \ldots\}$ and of variables $\mathbb{X} = \{X, Y, Z, \ldots\}$, and a signature Σ of function symbols which contains associative-commutative function symbols. A permutation π is a bijection of the form $\pi :$

$\mathbb{A} \to \mathbb{A}$ such that the domain of π (i.e., the set of atoms modified by π) is finite. Permutations are usually represented as a list of swappings, where the swapping $(a\ b)$ exchanges atoms a and b and fixes all the other atoms. Therefore, a permutation is represented as $\pi = (a_1\ b_1) :: ... :: (a_n\ b_n) :: nil$. The inverse of this permutation, denoted by π^{-1}, can be computed simply by reversing the list. The identity permutation is denoted by id.

Definition 1 (Nominal Terms $\boxed{\nearrow}$). *The set $\mathcal{T}(\Sigma, \mathbb{A}, \mathbb{X})$ of nominal terms is generated according to the grammar:*

$$s, t \quad ::= \quad a \mid \pi \cdot X \mid \langle\rangle \mid [a]t \mid \langle s, t \rangle \mid f\ t \mid f^{AC}\ t \tag{1}$$

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 $f^{AC}\ t$ is an associative-commutative function application.

Remark 1. We represent moderated variables of the form $id{\cdot}X$ simply as X. We follow Gabbay's name convention, which says that atoms differ in their names. Therefore, if we consider atoms a and b, it is redundant to say $a \neq b$.

Definition 2 (Well-formed Terms $\boxed{\nearrow}$). *We say that a term t is* well-formed *if t is not a pair and every AC-function application that is a subterm of t has at least two arguments.*

As was done in [7], we have restricted the terms that our algorithm receives to well-formed terms to ease our formalisation (more details in the extended version). Excluding pairs is a natural decision since they are used to encode a list of arguments to a function.

Definition 3 (Permutation Action). *The action of permutations on atoms $\boxed{\nearrow}$ 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 $\boxed{\nearrow}$ is defined recursively:*

$$\pi \cdot \langle\rangle = \langle\rangle \qquad\qquad \pi \cdot (\pi' \cdot X) = (\pi :: \pi') \cdot X \qquad \pi \cdot [a]t = [\pi \cdot a]\pi \cdot t$$
$$\pi \cdot \langle s, t \rangle = \langle \pi \cdot s, \pi \cdot t \rangle \qquad \pi \cdot f\ t = f\ \pi \cdot t \qquad\qquad \pi \cdot f^{AC}t = f^{AC}\pi \cdot t$$

Notation 1 *When convenient, we may mention that a function symbol f is an AC-function symbol, omit the superscript and write simply f instead of f^{AC}.*

A substitution σ is a function from variables to terms, such that $\sigma X \neq id{\cdot}X$ only for a finite set of variables, called the domain of σ and denoted as $dom(\sigma)$. The image of σ is then defined as $im(\sigma) = \{\sigma X \mid X \in dom(\sigma)\}$. We denote the identity substitution by id. From now on, when composing substitution σ with δ we may omit the composition symbol and write $\sigma\delta$ instead of $\sigma \circ \delta$.

A well-formed substitution $\boxed{\nearrow}$ only instantiates variables to well-formed terms. In the proofs of soundness and completeness of the algorithm, we restrict ourselves to well-formed substitutions. Let V be a set of variables. If $dom(\sigma) \subseteq V$

and $Vars(im(\sigma)) \subseteq V$ we write $\sigma \subseteq V$. In our PVS code, substitutions are represented by a list, where each entry of the list is called a *nuclear substitution* and is of the form $\{X \to t\}$.

Definition 4 (Nuclear substitution action on terms ⬀). *A nuclear substitution $\{X \to t\}$ acts over a term by induction as shown below:*

$$\{X \to t\}\langle\rangle = \langle\rangle \qquad\qquad \{X \to t\}\langle s_1, s_2\rangle =$$
$$\{X \to t\}([a]s) = [a](\{X \to t\}s) \qquad\qquad \langle\{X \to t\}s_1, \{X \to t\}s_2\rangle$$
$$\{X \to t\}(f \ s) = f \ (\{X \to t\}s) \qquad \{X \to t\}\pi \cdot Y = \begin{cases} \pi \cdot Y \ \text{if } X \neq Y \\ \pi \cdot t \ \ otherwise \end{cases}$$
$$\{X \to t\}a = a \qquad\qquad \{X \to t\}(f^{AC} \ s) = f^{AC} \ (\{X \to t\}s)$$

Definition 5 (Substitution acting on terms ⬀). *Since a substitution σ is a list of nuclear substitutions, the action of a substitution is defined as:*

- NIL $t = t$, *where* NIL *is the null list, used to represent the identity substitution.*
- CONS($\{X \to s\}$, σ) $t = \{X \to s\}(\sigma t)$.

Remark 2. The notion of substitution used here differs from the more traditional view of a substitution as a simultaneous application of nuclear substitutions, although both are correct. The way we defined substitution here is closer to triangular substitutions [21]. In the definition of action of substitutions the nuclear substitution in the head of the list is applied last. This lets us, given substitutions σ and δ, obtain the substitution $\sigma \circ \delta$ in our code simply as APPEND(σ, δ).

2.2   Freshness and α-equality

Freshness and α-equality are two valuable notions in nominal theory and are represented by the predicates # and $\approx_\alpha$. Intuitively, $a\#t$ means that if a occurs in t then it does so under an abstractor $[a]$, and $s \approx_\alpha t$ means that s and t are α-equivalent, that is, they are equal modulo the renaming of bound atoms. These concepts are given in Definitions 6 and 7.

Definition 6 (Freshness ⬀). *A freshness context ∇ is a set of constraints of the form $a\#X$. We denote contexts by letters $\Delta, \Gamma, \nabla, \dots$ 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:*

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

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

$$\frac{\nabla \vdash a\#t}{\nabla \vdash a\#f \ t} \ (\#app) \qquad \frac{\nabla \vdash a\#t}{\nabla \vdash a\#f^{AC} \ t} \ (\#AC)$$

Definition 7 (α-equality with AC operators [↗]). *Let f be an AC function symbol, $S_n(f\,t)$ be an operator that selects the nth argument of $f\,t$ (considering the flattened form) and $D_n(f\,t)$ be an operator that deletes the nth argument of $f\,t$ (considering the flattened form). If there exist i and j such that $\Delta \vdash S_i(f^{AC}s) \approx_\alpha S_j(f^{AC}t)$ and $\Delta \vdash D_i(f^{AC}s) \approx_\alpha D_j(f^{AC}t)$, then $\Delta \vdash f^{AC}s \approx_\alpha f^{AC}t$. In other words, the rule of α-equality for an AC-function application is:*

$$\frac{\Delta \vdash S_i(f^{AC}s) \approx_\alpha S_j(f^{AC}t) \quad \Delta \vdash D_i(f^{AC}s) \approx_\alpha D_j(f^{AC}t)}{\Delta \vdash f^{AC}s \approx_\alpha f^{AC}t} \ (\approx_\alpha AC)$$

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 rule ($\approx_\alpha AC$) and the rules:

$$\frac{}{\Delta \vdash \langle\rangle \approx_\alpha \langle\rangle} \ (\approx_\alpha \langle\rangle) \qquad\qquad \frac{}{\Delta \vdash a \approx_\alpha a} \ (\approx_\alpha atom)$$

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

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

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

Notation 2 *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 $a \in ds(\pi,\pi')$.*

2.3   Solution to Quintuples and Additional Notation

For the proofs of soundness and completeness of the algorithm, we need the notion of a solution to a quintuple (Definition 8). This definition depends on a parameter $\mathcal{X}$, a set of "protected variables", i.e., variables that cannot be instantiated.

Let P be a finite set of equational constraints. We denote the left-hand side of P by $lhs(P)$ [↗] and the right-hand side of P by $rhs(P)$ [↗]. The set of variables in $t \approx_? s$ is denoted as $Vars(t,s)$ [↗]. Finally, if Γ is a context then we denote by $Vars(\Gamma)$ [↗] the set $\{X \mid a\#X \in \Gamma, \text{for some atom } a\}$.

Notation 3 *Let ∇ and ∇' be freshness contexts and σ and σ' substitutions. We need the following notation to define a solution to a quintuple:*

 - *$\nabla' \vdash \sigma\nabla$ denotes that $\nabla' \vdash a\#\sigma X$ holds for each $(a\#X) \in \nabla$.*
 - *$\nabla \vdash \sigma \approx_V \sigma'$ denotes that $\nabla \vdash \sigma X \approx_\alpha \sigma' X$ for all X in V. When V is the set of all variables $\mathbb{X}$, we write $\nabla \vdash \sigma \approx \sigma'$.*

Definition 8 (Solution for a Quintuple [↗]). *Suppose that Γ is a context, P is a set of freshness constraints (of the form $a\#_? t$) and equational constraints (of the form $t \approx_? s$), σ is a substitution, V is a set of variables and $\mathcal{X}$ is a*

set of protected variables that cannot be instantiated. A solution for a quintuple $(\Gamma, P, \sigma, V, \mathcal{X})$ is a pair (Δ, δ), where the following conditions are satisfied:

1. $\Delta \vdash \delta\Gamma$.

2. if $a\#_? t \in P$ then $\Delta \vdash a\#\delta t$.

3. if $t \approx_? s \in P$ then $\Delta \vdash \delta t \approx_\alpha \delta s$.

4. there exists λ such that $\Delta \vdash \lambda\sigma \approx_V \delta$.

5. $dom(\delta) \cap \mathcal{X} = \emptyset$.

Remark 3. Note that if (Δ, δ) is a solution of $(\Gamma, \text{NIL}, \sigma, \mathbb{X}, \mathcal{X})$ this corresponds to the notion of (Δ, δ) being an instance of (Γ, σ) that does not instantiate variables in $\mathcal{X}$.

Definition 9 (Solution for an AC-unification/matching/equality problem). *A solution for an AC-unification problem with protected variables $(\Gamma, P, \mathcal{X})$ is a solution for the associated quintuple $(\Gamma, P, id, Vars(P), \mathcal{X})$. When $\mathcal{X} = Vars(rhs(P))$, we have the definition for an AC-matching problem and when $\mathcal{X} = Vars(P)$ we have the definition of solution to an AC-equality checking problem.*

3 Algorithm

We present the algorithm's pseudocode instead of the actual PVS code for readability. We developed a nominal algorithm (Algorithm 1 ⬀) for matching terms t and s. The algorithm is recursive and needs to keep track of the current context Γ, the equational constraints P that we have to unify, the substitution σ computed so far, the set of variables V that are/were in the problem and the set of protected variables $\mathcal{X}$. Hence, its input is a quintuple $(\Gamma, P, \sigma, V, \mathcal{X})$. The output is a list of solutions, each of the form (Γ_1, σ_1). The freshness constraints are treated by auxiliary functions (see Section 3.1), and the equational constraints P are represented as a list in our PVS code, where each element of the list is a pair (t_i, s_i) that represents an equation $t_i \approx_? s_i$. The first call to the algorithm, in order to match t to s, is done with $P = \{t \approx_? s\}$; $\Gamma = \emptyset$ and $\sigma = id$ (because we have not computed any freshness constraint or substitution yet); $V = Vars(t, s)$ and $\mathcal{X} = Vars(s)$.

Although extensive, Algorithm 1 is simple. It starts by analysing the list P of terms to match. If it is empty (line 2), it has finished and can return the answer computed so far, a list with a unique element: (Γ, σ). Otherwise, the algorithm calls the auxiliary function CHOOSEEQ (line 4), which returns a pair (t, s) and a list of equational constraints P_1 such that $P = \{t \approx_? s\} \cup P_1$. Then, P is updated by simplifying $\{t \approx_? s\}$ and it does so by seeing the form of t (an atom, a moderated variable, a unit, and so on).

3.1 Functions CHOOSEEQ and DECOMPOSE

The function CHOOSEEQ(P) ⬀ selects an equational constraint $t \approx_? s$ in P, picking the equation with the biggest size. This heuristic aims to aid us in the proof of termination (see Section 4.2).

Algorithm 1 Nominal AC-Matching Algorithm 1 🔗

1: **procedure** ACMATCH$(\Gamma, P, \sigma, V, \mathcal{X})$
2:      **if** nil?(P) **then** $cons((\Gamma, \sigma), \text{NIL})$
3:      **else**
4:          **let** $((t, s), P_1) = \text{CHOOSEEQ}(P)$ **in**
5:          **if** t matches a and s matches a **then** ACMATCH$(\Gamma, P_1, \sigma, V, \mathcal{X})$
6:          **else if** t matches $\pi \cdot X$ and $X \notin Vars(s)$ and $X \notin \mathcal{X}$ **then**
7:              **let** $\sigma_1 = \{X \mapsto \pi^{-1}s\},$
8:                  $(\Gamma_1, \mathit{flag}) = \text{FRESHSUBS?}(\sigma_1, \Gamma)$ **in**
9:              **if** flag **then** ACMATCH$(\Gamma_1 \cup \Gamma, \sigma_1 P_1, \sigma_1 \sigma, V, \mathcal{X})$
10:             **else** NIL
11:         **else if** t matches $\pi \cdot X$ and s matches $\pi' \cdot X$ **then**
12:             **let** $\Gamma_1 = ds(\pi, \pi')\#X \cup \Gamma$ **in** ACMATCH$(\Gamma_1, P_1, \sigma, V, \mathcal{X})$
13:         **else if** t matches $\langle\rangle$ and s matches $\langle\rangle$ **then** ACMATCH$(\Gamma, P_1, \sigma, V, \mathcal{X})$
14:         **else if** t matches $f\ t_1$ and s matches $f\ s_1$ **then**
15:             **let** $(P_2, \mathit{flag}) = \text{DECOMPOSE}(t_1, s_1)$ **in**
16:             **if** flag **then** ACMATCH$(\Gamma, P_2 \cup P_1, \sigma, V, \mathcal{X})$
17:             **else** NIL
18:         **else if** t matches $[a]\ t_1$ and $s = [a]\ s_1$ **then**
19:             **let** $(P_2, \mathit{flag}) = \text{DECOMPOSE}(t_1, s_1)$ **in**
20:             **if** flag **then** ACMATCH$(\Gamma, P_2 \cup P_1, \sigma, V, \mathcal{X})$
21:             **else** NIL
22:         **else if** t matches $[a]\ t_1$ and $s = [b]s_1$ **then**
23:             **let** $(\Gamma_1, \mathit{flag1}) = \text{FRESH?}(a, s_1),$
24:                 $(P_2, \mathit{flag2}) = \text{DECOMPOSE}(t_1, (a\ b) \cdot s_1)$ **in**
25:             **if** $\mathit{flag1}$ and $\mathit{flag2}$ **then** ACMATCH$(\Gamma \cup \Gamma_1, P_2 \cup P_1, \sigma, V, \mathcal{X})$
26:             **else** NIL
27:         **else if** t matches $f^{AC}\ t_1$ and s matches $f^{AC}\ s_1$ **then**
28:             **let** $InputLst = \text{APPLYACSTEP}\ (\Gamma, cons((t, s), P_1), \sigma, V, \mathcal{X}),$
29:                 $LstResults = \text{MAP}(\text{ACMATCH}, InputLst)$ **in** FLATTEN$(LstResults)$
30:         **else** NIL

The function DECOMPOSE 🔗 (lines 15, 19 and 24) receives two terms t and s, and if they are both pairs, it recursively tries to decompose them, returning a tuple (P, flag), where P is a list of equational constraints and flag is a boolean that is $True$ if the decomposition was successful. This function guarantees that only well-formed terms are in the matching problem.

Example 1. Examples of the function DECOMPOSE are given below.

- DECOMPOSE$(\langle a, \langle b, c\rangle\rangle,\ \langle c, \langle X, Y\rangle\rangle) = (\{a \approx_? c,\ b \approx_? X,\ c \approx_? Y\},\ True)$.
- DECOMPOSE$(a, Y) = (\{a \approx_? Y\},\ True)$.
- DECOMPOSE$(X, \langle c, d\rangle) = (\text{NIL},\ False)$.

3.2   Handling Freshness Constraints - Functions FRESHSUBS? and FRESH?

Following the approach of [6], freshness constraints are handled separately by the auxiliary functions FRESH? 🔗 and FRESHSUBS? 🔗. These functions were already

implemented in [6], and extending them to handle AC-functions is straightforward. FRESHSUBS?(σ, Γ) returns the minimal context (Γ_1 in Algorithm 1) in which $a\#_?\sigma X$ holds, for every $a\#X$ in the context Γ. FRESH?(a, t) computes and returns the minimal context (Γ_1 in Algorithm 1) in which a is fresh for t. Both functions also return a boolean (*flag* in Algorithm 1), indicating if it was possible to find the aimed context.

3.3   The Function APPLYACSTEP

The function APPLYACSTEP ☑ was adapted from the formalisation of first-order AC-unification (see [7]). It handles equations $t \approx_? s$, where t and s are rooted by the same AC function symbol. This function returns a list (*InputLst* in line 28 of Algorithm 1) with each entry in this list corresponding to a branch ACMATCH will explore. ACMATCH explores every branch generated by calling itself recursively on every input in *InputLst* (line 29 of the algorithm). The algorithm's output is a list of solutions of the form (Γ, σ), where Γ is a context and σ is a substitution. In addition, the result of calling MAP(ACMATCH, *InputLst*), *LstResults* in line 29 of Algorithm 1, is a list of lists of solutions. Hence, *LstResults* is flattened and then returned.

*Remark 4 (*SOLVEAC *and* INSTANTIATESTEP*).* APPLYACSTEP relies on two functions: SOLVEAC ☑ and INSTANTIATESTEP ☑, which are fully described in [7]. In synthesis, the function SOLVEAC finds the linear Diophantine equational system associated with the AC-matching equational constraint, generates the basis of solutions, and uses these solutions to generate the new AC-matching equational constraints. The function INSTANTIATESTEP instantiates the moderated variables that it can.

3.4   An Example of First-order AC-Unification and How We Adapted It to the Nominal Setting

We give a very high-level example (taken from [31] and more detailed in the extended version) of how we would solve the first-order AC-unification problem $\{f(X, X, Y, a, b, c) \approx_? f(b, b, b, c, Z)\}$. The first step is to eliminate common arguments. Next we associate our unification problem with a linear Diophantine equation ($2U_1 + U_2 + U_3 = 2V_1 + V_2$ in our case) and generate a basis of solutions to this equation, associating a new variable ($Z_1, Z_2, \ldots, Z_7$ in our case) to each solution. The algorithm may branch into (possibly) many unification problems and these new variables will be the building blocks for these unification problems. Finally, before proceeding to unify the new unification problems, we can drop the cases where a variable term is paired with an AC-function application. In the end, the solutions computed are:

$$\sigma_1 = \{Y \mapsto f(b, b), Z \mapsto f(a, X, X)\} \quad \sigma_2 = \{Y \mapsto f(Z_2, b, b), Z \mapsto f(a, Z_2, X, X)\}$$
$$\sigma_3 = \{X \mapsto b, Z \mapsto f(a, Y)\} \qquad\qquad \sigma_4 = \{X \mapsto f(Z_6, b), Z \mapsto f(a, Y, Z_6, Z_6)\}$$

With this example in mind, there are four main modifications (more details in the extended version) when moving from first-order AC-unification to nominal AC-matching. When eliminating common arguments we do not eliminate arguments t_i and s_j of t and s if they are equal modulo AC, we eliminate them if they are α-equivalent (modulo AC) under the context Γ that we are working with. Regarding the new variables introduced: the permutation suspended on them is always the identity. Additionally, we drop the cases where a moderated variable $\pi \cdot X$, with $X \in \mathcal{X}$, is paired with an AC-function application. Finally, we must guarantee that the new variables Z_is introduced by the algorithm can be instantiated, i.e. $Z_i \notin \mathcal{X}$.

4   Formalisation

As is done in [7], to help us in the proofs of termination (Section 4.2), soundness (Section 4.3) and completeness (Section 4.4) we define the notion of a *nice input* (Section 4.1).

4.1   Nice Inputs

Nice inputs are invariant under the action of the ACMATCH function with valuable properties. Notice that Item 7 of Definition 10 would need to be removed for the proofs of termination, soundness, and completeness to be used in unification.

Definition 10 (Nice input ⬀). *An input $(\Gamma, P, \sigma, V, \mathcal{X})$ is said to be nice if:*
1. *σ is idempotent.*
2. *$Vars(P) \cap dom(\sigma) = \emptyset$.*
3. *$\sigma \subseteq V$.*
4. *$Vars(P) \subseteq V$.*
5. *$Vars(\Gamma) \subseteq V$.*
6. *$\mathcal{X} \subseteq V$.*
7. *$Vars(rhs(P)) \subseteq \mathcal{X}$.*

4.2   Termination

For the lexicographic measure used in the proof of termination, we need the definition of the size of an equational constraint $t \approx_? s$ (Definition 11).

Definition 11 (Size of an Equational Constraint ⬀). *The size of an equational constraint $t \approx_? s$ is $size(t) + size(s)$, where the size of a term t ⬀ is recursively defined as follows:*
- *$size(a) = 1$.*
- *$size(\pi \cdot X) = 1$.*
- *$size(\langle\rangle) = 1$.*
- *$size(\langle t_1, t_2 \rangle) = 1 + size(t_1) + size(t_2)$.*
- *$size(f\ t_1) = 1 + size(t_1)$.*
- *$size(f^{AC}\ t_1) = 1 + size(t_1)$.*
- *$size([a]t_1) = 1 + size(t_1)$*

Although the nominal AC-matching algorithm is based on the first-order AC-unification algorithm ([7]), the proof of termination was much easier for nominal AC-matching than for first-order AC-unification. Instead of the intricate lexicographic measure used in [7] (which came from the work of [17]), it was

possible to prove that for the particular case of matching (unlike unification) all the new moderated variables introduced by SOLVEAC are instantiated by INSTANTIATESTEP.

Hence, the lexicographic measure used has as its first component the number of variables in the equational constraints P and as a second component the multiset order of the size of each equation $t \approx_? s \in P$. Although PVS does not directly implement multiset orders, this part can be emulated easily by analysing the maximum size n of all equations $t \approx_? s$ in P and the number of equations $t \approx_? s$ in P with maximal size (in this order). The algorithm selects an equation with maximal size to simplify (the heuristic selection is enforced by the function CHOOSEEQ).

4.3   Soundness

As mentioned, to match terms t and s we first call the Algorithm 1 with parameters $\Gamma = \emptyset$, $P = \{t \approx_? s\}$, $\sigma = id$, $V = Vars(t,s)$ and $\mathcal{X} = Vars(s)$. However, since the parameters of ACMATCH change after recursive calls, the proof of soundness (Corollary 1) cannot be done directly by induction, and we must instead prove first the Theorem 1 with generic parameters Γ, P, σ, V and $\mathcal{X}$. Once the Theorem 1 is proved, it is also immediate to adapt the algorithm to solve nominal AC-equality checking and to prove its soundness (Corollary 2).

Theorem 1 (Soundness for Nice Inputs ☑). *Let the pair (Γ_1, σ_1) an output of* $\mathrm{ACMATCH}(\Gamma, P, \sigma, V, \mathcal{X})$ *and suppose that $(\Gamma, P, \sigma, V, \mathcal{X})$ is a nice input. If (Δ, δ) is a solution to $(\Gamma_1, \mathrm{NIL}, \sigma_1, \mathbb{X}, \mathcal{X})$ then (Δ, δ) is a solution to $(\Gamma, P, \sigma, \mathbb{X}, \mathcal{X})$.*

Corollary 1 (Soundness for AC-Matching ☑). *Let the pair (Γ_1, σ_1) an output of* $\mathrm{ACMATCH}(\emptyset, \{t \approx_? s\}, id, Vars(t,s), Vars(s))$. *If (Δ, δ) is an instance of (Γ_1, σ_1) that does not instantiate the variables in s, then (Δ, δ) is a solution to $(\emptyset, \{t \approx_? s\}, id, \mathbb{X}, Vars(s))$.*

Corollary 2 (Soundness for AC-Equality Checking ☑). *Let (Γ_1, σ_1) be an output of* $\mathrm{ACMATCH}(\emptyset, \{t \approx_? s\}, id, Vars(t,s), Vars(t,s))$. *If (Δ, δ) is an instance of (Γ_1, σ_1) that does not instantiate the variables in t or s, then (Δ, δ) is a solution to $(\emptyset, \{t \approx_? s\}, id, \mathbb{X}, Vars(t,s))$.*

Remark 5. An interpretation of Corollary 1 is that if (Δ, δ) is an AC-matching instance to one of the outputs of ACMATCH, then (Δ, δ) is an AC-matching solution to the original problem. Corollary 2 has a similar interpretation, replacing AC-matching with AC-equality checking.

4.4   Completeness

Completeness of Algorithm 1 is given by the Corollary 3 and similarly to the soundness proof, it is derived easily after proving the Theorem 2.

Theorem 2 (Completeness for Nice Inputs $\boxed{C}$ **).** *Let* $(\Gamma, P, \sigma, V, \mathcal{X})$ *be a nice input. Suppose that* (Δ, δ) *is a solution to* $(\Gamma, P, \sigma, \mathbb{X}, \mathcal{X})$, *that* $\delta \subseteq V$ *and that* $Vars(\Delta) \subseteq V$. *Then, there exists* $(\Gamma_1, \sigma_1) \in \mathrm{ACMATCH}(\Gamma, P, \sigma, V, \mathcal{X})$ *such that* (Δ, δ) *is an instance (restricted to the variables of* V*) of* (Γ_1, σ_1) *that does not instantiate the variables in* $\mathcal{X}$.

Corollary 3 (Completeness for AC-Matching $\boxed{C}$ **).** *Suppose that* (Δ, δ) *is a solution to* $(\emptyset, \{t \approx_? s\}, id, \mathbb{X}, Vars(s))$, *that* $\delta \subseteq V$ *and that* $Vars(\Delta) \subseteq V$. *Then, there exists* $(\Gamma_1, \sigma_1) \in \mathrm{ACMATCH}(\emptyset, \{t \approx_? s\}, id, V, Vars(s))$ *such that* (Δ, δ) *is an instance (restricted to the variables of* V*) of* (Γ_1, σ_1) *that does not instantiate the variables of* s.

Corollary 4 (Completeness for AC-equality Checking $\boxed{C}$ **).** *Suppose* (Δ, δ) *is a solution to* $(\emptyset, \{t \approx_? s\}, id, \mathbb{X}, Vars(t, s))$ *satisfying* $\delta \subseteq V$ *and* $Vars(\Delta) \subseteq V$. *Then, there exists* $(\Gamma_1, \sigma_1) \in \mathrm{ACMATCH}(\emptyset, \{t \approx_? s\}, id, V, Vars(t, s))$ *such that* (Δ, δ) *is an instance (restricted to the variables of* V*) of* (Γ_1, σ_1) *that does not instantiate the variables of* t *or* s.

Remark 6. An interpretation of Corollary 3 is that if (Δ, δ) is an AC-matching solution to the initial problem, then (Δ, δ) is an AC-matching instance of one of the outputs of ACMATCH. Corollary 4 has a similar interpretation, replacing AC-matching with AC-equality checking.

As was the case for first-order AC-unification (see [7]), the hypothesis $\delta \subseteq V$ in the proof of completeness is merely a technicality that was put in order to guarantee the new variables introduced by the algorithm in the AC-part do not clash with the variables in $dom(\delta)$ or in the terms in $im(\delta)$. This mechanism could be replaced by a different one that assures that the variables introduced by the AC-part of ACMATCH are indeed new. When going from the first-order setting to the nominal setting, we go from having a unifier δ to a pair (Δ, δ) and hence we must add the hypothesis $Vars(\Delta) \subseteq V$.

Remark 7 (High-level description of how to remove hypotheses $\delta \subseteq V$ *and* $Vars(\Delta) \subseteq V$*).* The critical step to prove a variant of Corollary 3 with $V = Vars(t, s)$ and without the hypothesis $\delta \subseteq V$ and $Vars(\Delta) \subseteq V$ is to prove that the outputs computed when we call ACMATCH with input $(\Gamma, P, \sigma, V, \mathcal{X})$ "differ only by the name of the new variables" from the outputs computed when we call ACMATCH with input $(\Gamma, P, \sigma, V', \mathcal{X})$. However, this cannot be proved directly by induction because if V and V' differ and ACMATCH enters in the AC-part, the new variables introduced for each input may "differ only by a renaming" and once we instantiate those variables, it may happen that the substitutions computed so far (the third component in the input quintuple) will also "differ only by the name of the new variables". Similar to what was done in first-order AC-unification, the solution is to prove the more general statement that if the inputs $(\Gamma, P, \sigma, V, \mathcal{X})$ and $(\Gamma, P, \sigma', V', \mathcal{X}')$ "differ only by the name of the new variables", then the output of ACMATCH with the first input "differ only by the name of the new variables" from the output of ACMATCH with the second input.

5   Towards a Nominal AC-Unification Algorithm

Stickel's AC-unification algorithm relies on solving Diophantine equations where new variables are used to represent arguments of AC operators. Using the same approach to solve nominal AC-unification problems leads to non-termination in cases where the same variable occurs as an argument of an AC operator multiple times with *different* suspended permutations.

As an example, suppose that we are working under an empty context (i.e. $\Gamma = \emptyset$) and want to solve the equational constraint $f(X, W) \approx_? f(\pi \cdot X, \pi \cdot Y)$, with $\mathcal{X} = \emptyset$. Additionally, assume that we apply Stickel's AC-unification algorithm to this equational constraints and let Z_1, W_1, Y_1, X_1 be the name of the new variables introduced (we choose these names deliberately to make the loop in nominal AC-unification clearer). Then, 7 branches (more details in the extended version) are generated and one of them is:

$$\{X \approx_? Y_1 + X_1, W \approx_? Z_1 + W_1, \pi \cdot X \approx_? W_1 + X_1, \pi \cdot Y \approx_? Z_1 + Y_1\}$$

After instantiating the variables we obtain

$$\sigma = \{X \mapsto f(Y_1, X_1), \ W \mapsto f(Z_1, W_1), Y \mapsto f(\pi^{-1} \cdot Z_1, \pi^{-1} \cdot Y_1)\}$$

and one equational constraint remain: $f(X_1, W_1) \approx_? f(\pi \cdot X_1, \pi \cdot Y_1)$. Notice that our final problem is essentially a renaming of our initial problem:

$$f(X, W) \approx_? f(\pi \cdot X, \pi \cdot Y)$$
$$f(X_1, W_1) \approx_? f(\pi \cdot X_1, \pi \cdot Y_1)$$

This problem does not arise in first-order AC-unification because, in the corresponding first-order problem, we would not have two different permutations (*id* and π in this case) suspended on the same variable (X in this case). Instead, we would have the same variable X as an argument to both terms and eliminate it. Finally, this problem also does not arise in nominal AC-matching because X would be a protected variable. Hence, we would not compute the substitution $\sigma = \{X \mapsto f(Y_1, X_1), W \mapsto W_1, Y \mapsto \pi^{-1} \cdot Y_1\}$, we would instead discard this branch. In future work, we will consider the alternative approach to AC-unification proposed by Boudet, Contejean and Devie [10,8], which was used to define AC higher-order pattern unification [9]. To our knowledge, this AC unification approach has not been formalised yet. However, it has the advantage of generating simpler Diophantine systems, which could simplify the task of nominal AC-unification.

6   Conclusion and Future Work

We propose the first (to the best of our knowledge) nominal AC-matching algorithm, together with proofs of its termination, soundness and completeness. All proofs were formalised in the proof assistant PVS. As a byproduct, we also

obtained a formalised nominal AC-equality checking algorithm. Nominal AC-matching has applications for nominal AC-rewriting, being the first step towards a nominal AC-unification algorithm.

Our formalisation extends the formalisation of first-order AC-unification by Ayala-Rincón et al. [7] to nominal terms and uses the functions that deal with freshness constraints from [6], extending them to deal with AC-function symbols. Furthermore, by adding a parameter $\mathcal{X}$ for protected variables, it enables both AC-matching and AC-equality checking, according to whether $\mathcal{X}$ is the set of variables in the right-hand side of the problem or the set of variables in the problem. The `.pvs` files have a combined size of 290 KB and contain the specification of functions and the statements of the theorems. The `.prf` files contain the proofs of the theorems and have a combined size of 22 MB.

Future work will explore ways to define a nominal AC-unification algorithm, avoiding the loop described in Section 5. We will consider alternative AC-unification algorithms as a starting point [10,9] and explore the connection between higher-order pattern unification and nominal unification (e.g., [13,23]).

A nominal AC-unification algorithm would have applications in logic programming languages that employ the nominal paradigm, such as α-Prolog. A second possible future work path is to use this formalisation to formalise a more efficient nominal AC-matching algorithm. Finally, a third future work path would be formalising matching/unification algorithms for different equational theories and a fourth path would be investigating if/how nominal unification algorithms can be used for term indexing.

References

1. Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., Nantes-Sobrinho, D.: Nominal C-Unification. In: Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR, Revised Selected Papers. LNCS, vol. 10855, pp. 235–251. Springer (2017). https://doi.org/10.1007/978-3-319-94460-9_14

2. Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., Nantes-Sobrinho, D.: On Solving Nominal Fixpoint Equations. In: Frontiers of Combining Systems - 11th International Symposium, FroCoS. LNCS, vol. 10483, pp. 209–226. Springer (2017). https://doi.org/10.1007/978-3-319-66167-4_12

3. Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., Nantes-Sobrinho, D., Oliveira, A.C.R.: A formalisation of nominal α-equivalence with A, C, and AC function symbols. Theor. Comput. Sci. **781**, 3–23 (2019). https://doi.org/10.1016/j.tcs.2019.02.020

4. Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., Silva, G.F., Nantes-Sobrinho, D.: Formalising Nominal C-Unification Generalised with Protected Variables. Math. Struct. Comput. Sci. **31**(3), 286–311 (2021). https://doi.org/10.1017/S0960129521000050

5. Ayala-Rincón, M., Fernández, M., Nantes-Sobrinho, D.: On Nominal Syntax and Permutation Fixed Points. Log. Methods Comput. Sci. **16**(1) (2020). https://doi.org/10.23638/LMCS-16(1:19)2020

6. Ayala-Rincón, M., Fernández, M., Oliveira, A.C.R.: Completeness in PVS of a Nominal Unification Algorithm. In: Proc. of the 10th Workshop on Logical and Semantic Frameworks, with Applications, LSFA. ENTCS, vol. 323, pp. 57–74. Elsevier (2015). https://doi.org/10.1016/j.entcs.2016.06.005

7. Ayala-Rincón, M., Fernández, M., Silva, G.F., Sobrinho, D.N.: A Certified Algorithm for AC-Unification. In: 7th International Conference on Formal Structures for Computation and Deduction, FSCD. LIPIcs, vol. 228, pp. 8:1–8:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.8

8. Boudet, A.: Competing for the AC-Unification Race. J. of Autom. Reasoning **11**(2), 185–212 (1993). https://doi.org/10.1007/BF00881905

9. Boudet, A., Contejean, E.: AC-Unification of Higher-Order Patterns. In: Third International Conference on Principles and Practice of Constraint Programming CP97. LNCS, vol. 1330, pp. 267–281. Springer (1997). https://doi.org/10.1007/BFb0017445

10. Boudet, A., Contejean, E., Devie, H.: A New AC Unification Algorithm with an Algorithm for Solving Systems of Diophantine Equations. In: Proc. of the 5th Annual Symposium on Logic in Computer Science, LICS. pp. 289–299. IEEE Computer Society (1990). https://doi.org/10.1109/LICS.1990.113755

11. Calvès, C.F., Fernández, M.: Matching and Alpha-Equivalence Check for Nominal Terms. J. of Computer and System Sciences **76**(5), 283–301 (2010). https://doi.org/http://dx.doi.org/10.1016/j.jcss.2009.10.003

12. Calvès, C., Fernández, M.: A polynomial nominal unification algorithm. Theor. Comput. Sci. **403**(2-3), 285–306 (2008). https://doi.org/10.1016/j.tcs.2008.05.012

13. Cheney, J.: Relating nominal and higher-order pattern unification. In: Proc. of the 19th international workshop on Unification, UNIF. pp. 104–119 (2005)

14. Clausen, M., Fortenbacher, A.: Efficient Solution of Linear Diophantine Equations. J. of Sym. Computation **8**(1-2), 201–216 (1989). https://doi.org/10.1016/S0747-7171(89)80025-2

15. Contejean, E.: A Certified AC Matching Algorithm. In: Proc. of the 15th International Conference on Rewriting Techniques and Applications, RTA. LNCS, vol. 3091, pp. 70–84. Springer (2004). https://doi.org/10.1007/978-3-540-25979-4_5

16. Eker, S.: Associative-Commutative Rewriting on Large Terms. In: Proc. of the 14th International Conference on Rewriting Techniques and Applications, RTA. LNCS, vol. 2706, pp. 14–29. Springer (2003). https://doi.org/10.1007/3-540-44881-0_3

17. Fages, F.: Associative-Commutative Unification. In: 7th International Conference on Automated Deduction CADE. LNCS, vol. 170, pp. 194–208. Springer (1984). https://doi.org/10.1007/978-0-387-34768-4_12

18. Fages, F.: Associative-Commutative Unification. J. of Sym. Computation **3**(3), 257–275 (1987). https://doi.org/10.1016/S0747-7171(87)80004-4

19. Fernández, M., Gabbay, M.J.: Nominal rewriting. Information and Computation **205**(6), 917–965 (2007). https://doi.org/10.1016/j.ic.2006.12.002

20. Gabbay, M.J., Pitts, A.M.: A New Approach to Abstract Syntax with Variable Binding. Formal Aspects of Computing **13**(3), 341–363 (Jul 2002). https://doi.org/10.1007/s001650200016
21. Kumar, R., Norrish, M.: (nominal) unification by recursive descent with triangular substitutions. In: Interactive Theorem Proving, ITP 2010, Edinburgh, UK, 2010. Lecture Notes in Computer Science, vol. 6172, pp. 51–66. Springer (2010). https://doi.org/10.1007/978-3-642-14052-5_6
22. Levy, J., Villaret, M.: An Efficient Nominal Unification Algorithm. In: Proc. of the 21st International Conference on Rewriting Techniques and Applications, RTA. LIPIcs, vol. 6, pp. 209–226. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2010). https://doi.org/10.4230/LIPIcs.RTA.2010.209
23. Levy, J., Villaret, M.: Nominal Unification from a Higher-Order Perspective. ACM Trans. Comput. Log. **13**(2), 10:1–10:31 (2012). https://doi.org/10.1145/2159531.2159532
24. Owre, S., Shankar, N.: The Formal Semantics of PVS. Tech. Rep. 97-2R, SRI International Computer Science Laboratory, Menlo Park CA 94025 USA (1997, revised 1999)
25. Pitts, A.M.: Nominal Sets: Names and Symmetry in Computer Science. Cambridge University Press (2013)
26. Pottier, F.: An Overview of CαML. In: Benton, N., Leroy, X. (eds.) Proc. of the ACM-SIGPLAN Workshop on ML, ML. Electronic Notes in Theoretical Computer Science, vol. 148, pp. 27–52. Elsevier (2005). https://doi.org/10.1016/j.entcs.2005.11.039
27. Schmidt-Schauß, M., Kutsia, T., Levy, J., Villaret, M.: Nominal Unification of Higher Order Expressions with Recursive Let. In: Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR, Revised Selected Papers. LNCS, vol. 10184, pp. 328–344. Springer (2016). https://doi.org/10.1007/978-3-319-63139-4_19
28. Schmidt-Schauß, M., Kutsia, T., Levy, J., Villaret, M., Kutz, Y.D.K.: Nominal Unification and Matching of Higher Order Expressions with Recursive Let. Fundam. Informaticae **185**(3), 247–283 (2022). https://doi.org/10.3233/FI-222110
29. Shinwell, M.R., Pitts, A.M., Gabbay, M.: FreshML: programming with binders made simple. In: Proc. of the 8th ACM SIGPLAN International Conference on Functional Programming, ICFP. pp. 263–274. ACM (2003). https://doi.org/10.1145/944705.944729
30. Stickel, M.E.: A Complete Unification Algorithm for Associative-Commutative Functions. In: Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, IJCAI. pp. 71–76 (1975), http://ijcai.org/Proceedings/75/Papers/011.pdf
31. Stickel, M.E.: A Unification Algorithm for Associative-Commutative Functions. J. of the ACM **28**(3), 423–434 (1981). https://doi.org/10.1145/322261.322262
32. Urban, C.: Nominal Techniques in Isabelle/HOL. J. Autom. Reason. **40**(4), 327–356 (2008). https://doi.org/10.1007/s10817-008-9097-2
33. Urban, C.: Nominal Unification Revisited. In: Proc. of the 24th International Workshop on Unification, UNIF. EPTCS, vol. 42, pp. 1–11 (2010). https://doi.org/10.4204/EPTCS.42.1
34. Urban, C., Pitts, A.M., Gabbay, M.: Nominal unification. Theor. Comput. Sci. **323**(1-3), 473–497 (2004). https://doi.org/10.1016/j.tcs.2004.06.016