Nominal C-Unification

Mauricio Ayala-Rincón[1(✉)], Washington de Carvalho-Segundo[1(✉)], Maribel Fernández[2(✉)], and Daniele Nantes-Sobrinho[1(✉)]

[1] Depts. de Matemática e Ciência da Computação, Universidade de Brasília, Brasília, Brazil
{ayala,dnantes}@unb.br, wtonribeiro@gmail.com
[2] Department of Informatics, King's College London, London, UK
maribel.fernandez@kcl.ac.uk

Abstract. Nominal unification is an extension of first-order unification that takes into account the α-equivalence relation generated by binding operators, following the nominal approach. We propose a sound and complete procedure for nominal unification with commutative operators, or nominal C-unification for short, which has been formalised in Coq. The procedure transforms nominal C-unification problems into simpler (finite families) of *fixed point* constraints, whose solutions can be generated by algebraic techniques on combinatorics of permutations.

1 Introduction

Unification, where the goal is to solve equations between first-order terms, is a key notion in logic programming systems, type inference algorithms, protocol analysis tools, theorem provers, etc. Solutions to unification problems are represented by substitutions that map variables $(X, Y, \dots)$ to terms.

When terms include binding operators, a more general notion of unification is needed: unification modulo α-equivalence. In this paper, we follow the nominal approach to the specification of binding operators [20,26,30], where the syntax of terms includes, in addition to variables, also *atoms* $(a, b, \dots)$, which can be abstracted, and α-equivalence is axiomatised by means of a *freshness relation* $a\#t$ and *name-swappings* $(a\,b)$. For example, the first-order logic formula $\forall a.a \geq 0$ can be written as a nominal term $\forall([a]geq(a, 0))$, using function symbols $\forall$ and geq and an abstracted atom a. Nominal unification [30] is the problem of solving equations between nominal terms modulo α-equivalence; it is a decidable problem and efficient nominal unification algorithms are available [9,11,24], that compute solutions consisting of *freshness contexts* (containing freshness constraints of the form $a\#X$) and substitutions.

In many applications, operators obey equational axioms. Nominal reasoning and unification have been extended to deal with equational theories presented by rewrite rules (see, e.g., [5,17,18]) or defined by equational axioms (see, e.g., [14,

Work supported by the Brazilian agencies FAPDF (DE 193.001.369/2016), CAPES (Proc. 88881.132034/2016-01, 2nd author) and CNPq (PQ 307009/2013, 1st author).

F. Fioravanti and J. P. Gallagher (Eds.): LOPSTR 2017, LNCS 10855, pp. 235–251, 2018.
https://doi.org/10.1007/978-3-319-94460-9_14

19]). The case of associative and commutative nominal theories was considered in [3], where a parametric $\{\alpha, AC\}$-equivalence relation was formalised in Coq. However, only equational deduction was considered (not unification). In this paper, we study nominal C-unification.

Contributions: We present a nominal C-unification algorithm, based on a set of *simplification rules*. The algorithm transforms a given *nominal C-unification problem* $\langle \Delta, Q \rangle$, where Δ is a freshness context and Q a set of freshness constraints and equations, respectively of the form $a\#_? s$ and $s \approx_? t$, into a finite set of triples of the form $\langle \nabla, \sigma, P \rangle$, consisting of a freshness context ∇, a substitution σ and a set of fixed point equations (for short, FP equations) P of the form $\pi.X \approx_? X$. The simplifications are based on a set of deduction rules for freshness and α-C-equivalence (denoted as $\approx_{\{\alpha,C\}}$).

The role of FP equations in nominal C-unification is tricky: while in standard nominal unification [30], solving a FP equation of the form $(a\ b).X \approx_? X$ reduces to checking whether the constraints $a\#X, b\#X$ (a and b fresh in X) are satisfied, and in this case the solution is the *identity* substitution, in nominal C-unification, for $*$ and $+$ commutative operators, one can have additional combinatory solutions of the form $\{X/a{+}b\}$, $\{X/(a{+}b){*}\ldots{*}(a{+}b)\}$, $\{X/f(a){+}f(b)\}$, etc. We show that in general there is no finitary representation of solutions using only freshness contexts and substitutions, hence a nominal C-unification problem may have a potentially infinite set of independent most general unifiers (unlike standard C-unification, which is well-known to be finitary).

We adapt the proof of NP-completeness of syntactic C-unification to show that nominal C-unification is NP-complete as well. Soundness and completeness of the simplification rules were formalised in Coq. The formalisation, an extended version of the paper with all proof details and an OCaml implementation are available at http://ayala.mat.unb.br/publications.html.

Related work: To generate the set of combinatorial solutions for FP equations we can use an enumeration procedure given in [4], which is based on the combinatorics of permutations. By combining the simplification and enumeration methods, we obtain a nominal C-unification procedure in two phases: a *simplification phase*, described in this paper, which outputs a finite set of most general solutions that may include FP constraints, and a *generation phase*, which eliminates the FP constraints according to [4].

Several extensions of the nominal unification algorithm have been defined, in addition to the equational extensions already mentioned.

An algorithm for nominal unification of higher-order expressions with recursive *let* was proposed in [23]; as in the case of nominal C-unification, FP equations are obtained in the process. Using the techniques in [4], it is possible to proceed further and generate the combinatorial solutions of FP equations.

Recently, Aoto and Kikuchi [1] proposed a rule-based procedure for nominal equivariant unification [13], an extension of nominal unification that is useful in confluence analysis of nominal rewriting systems [2,16].

Furthermore, several formalisations and implementations of the nominal unification algorithm are available. For example, formalisations of its soundness and completeness were developed by Urban et al [29,30], Ayala-Rincón et al [6], and Kumar and Norrish [22] using, respectively, the proof assistants Isabelle/HOL, PVS and HOL4. An implementation in Maude using term graphs [10] is also available. Urban and Cheney used a nominal unification algorithm to develop a Prolog-like language called α-Prolog [12]. Our formalisation of nominal C-unification is based on the formalisation of equivalence modulo $\{\alpha, AC\}$ presented in [3]. The representations of permutations and terms are similar, but here we deal also with substitutions and unification rules, and prove soundness and completeness of the unification algorithm.

Reasoning modulo equational theories (but without considering the nominal approach to deal with α-equivalence) has been subject of formalisations. For instance, Nipkow [25] presented a set of Isabelle/HOL tactics for reasoning modulo A, C and AC; Braibant and Pous [8] designed a plugin for Coq, with an underlying AC-matching algorithm, that extends the system tactic `rewrite` to deal with AC function symbols; also, Contejean [15] formalised in Coq the correction of an AC-matching algorithm implemented in C*i*ME.

Syntactic unification with commutative operators is an NP-complete problem and its solutions can be finitely generated [21,28]. Since C-unification problems are a particular case of nominal C-unification problems, our simplification algorithm, checked in Coq, is also a formalisation of the C-unification algorithm.

Organisation: Section 2 presents basic concepts and notations. Section 3 introduces the formalised equational and freshness inference rules for nominal C-unification, and briefly discusses NP-completeness; Sect. 4 shows that a single FP equation can have infinite independent solutions; Sect. 5 shortly discusses the formalisation in Coq and Sect. 6 concludes and proposes future work.

2 Background

Consider countable disjoint sets of variables $\mathcal{X} := \{X, Y, Z, \cdots\}$ and atoms $\mathcal{A} := \{a, b, c, \cdots\}$. A *permutation* π is a bijection on $\mathcal{A}$ with a finite *domain*, where the domain (i.e., the *support*) of π is the set $dom(\pi) := \{a \in \mathcal{A} \mid \pi \cdot a \neq a\}$. The inverse of π is denoted by π^{-1}. Permutations can be represented by lists of *swappings*, which are pairs of different atoms $(a\,b)$; hence a *permutation* π is a finite list of the form $(a_1\,b_1) :: \ldots :: (a_n\,b_n) :: nil$, where the empty list *nil* corresponds to the identity permutation; concatenation is denoted by $\oplus$ and, when no confusion may arise, :: and *nil* are omitted. We follow Gabbay's permutative convention: Atoms differ on their names, so for atoms a and b the expression $a \neq b$ is redundant. Also, $(a\,b)$ and $(b\,a)$ have identical *action*: they exchange a and b; thus, they represent the same swapping.

We will assume as in [3] countable sets of function symbols with different equational properties such as associativity, commutativity, idempotence, etc. Function symbols have superscripts that indicate their equational properties;

thus, f_k^C will denote the k^{th} function symbol that is commutative and $f_j^{\emptyset}$ the j^{th} function symbol without any equational property.

Nominal terms are generated by the following grammar:

$$s, t := \langle\rangle \mid \bar{a} \mid [a]t \mid \langle s, t\rangle \mid f_k^E\, t \mid \pi.X$$

$\langle\rangle$ denotes the *unit* (that is the empty tuple), $\bar{a}$ denotes an *atom term*, $[a]t$ denotes an *abstraction* of the atom a over the term t, $\langle s, t\rangle$ denotes a *pair*, $f_k^E\, t$ the *application* of f_k^E to t and, $\pi.X$ a *moderated variable* or *suspension*. Suspensions of the form $nil.X$ will be represented just by X.

The set of variables occurring in a term t will be denoted as $Var(t)$. This notation extends to a set S of terms in the natural way: $Var(S) = \bigcup_{t\in S} Var(t)$. As usual, $|\_|$ will be used to denote the cardinality of sets as well as to denote the size or number of symbols occurring in a given term.

Definition 1 (Permutation action). *The action of a permutation on atoms is defined as:* $nil \cdot a := a$; $(b\,c) :: \pi \cdot a := \pi \cdot a$; *and,* $(b\,c) :: \pi \cdot b := \pi \cdot c$. *The action of a permutation on terms is defined recursively as:*

$$\begin{array}{llll} \pi \cdot \langle\rangle := \langle\rangle & \pi \cdot \langle u, v\rangle := \langle \pi \cdot u, \pi \cdot v\rangle & \pi \cdot f_k^E\, t := f_k^E\, (\pi \cdot t) \\ \pi \cdot \bar{a} := \overline{\pi \cdot a} & \pi \cdot ([a]t) := [\pi \cdot a](\pi \cdot t) & \pi \cdot (\pi' \,.\, X) := (\pi' \oplus \pi)\,.\,X \end{array}$$

Notice that according to the definition of the action of a permutation over atoms, the composition of permutations π and π', usually denoted as $\pi \circ \pi'$, corresponds to the append $\pi' \oplus \pi$. Also notice that $\pi' \oplus \pi \cdot t = \pi \cdot (\pi' \cdot t)$. The *difference set* between two permutations π and π' is the set of atoms where the action of π and π' differs: $ds(\pi, \pi') := \{a \in \mathcal{A} \mid \pi \cdot a \neq \pi' \cdot a\}$.

A *substitution* σ is a mapping from variables to terms such that its *domain*, $dom(\sigma) := \{X \mid X \neq X\sigma\}$, is finite. For $X \in dom(\sigma)$, $X\sigma$ is called the *image* of X. Define the *image* of σ as $im(\sigma) := \{X\sigma \mid X \in dom(\sigma)\}$. Let $dom(\sigma) = \{X_1, \cdots, X_n\}$, then σ can be represented as a set of *bindings* in the form $\{X_1/t_1, \cdots, X_n/t_n\}$, where $X_i\sigma = t_i$, for $1 \leq i \leq n$.

Definition 2 (Substitution action). *The* action of a substitution σ *on a term* t, *denoted* $t\sigma$, *is defined recursively as follows:*

$$\begin{array}{lll} \langle\rangle\sigma := \langle\rangle & \bar{a}\sigma := \bar{a} & (f_k^E\, t)\sigma := f_k^E\, t\sigma \\ \langle s, t\rangle\sigma := \langle s\sigma, t\sigma\rangle & ([a]t)\sigma := [a]t\sigma & (\pi.X)\sigma := \pi \cdot X\sigma \end{array}$$

The following result can be proved by induction on the structure of terms.

Lemma 1 (Substitutions and Permutations Commute). $(\pi \cdot t)\sigma = \pi \cdot (t\sigma)$

The inference rules defining freshness and α-equivalence are given in Figs. 1 and 2. The symbols ∇ and Δ are used to denote *freshness contexts* that are sets of constraints of the form $a\#X$, meaning that the atom a is fresh in X. The domain of a freshness context $dom(\nabla)$ is the set of atoms appearing in it; $\nabla|_X$ denotes the restriction of ∇ to the freshness constraints on X: $\{a\#X \mid a\#X \in \nabla\}$. The

$$\frac{}{\nabla \vdash a \# \langle\rangle}\ (\#\langle\rangle) \qquad \frac{}{\nabla \vdash a \# \overline{b}}\ (\#\ \mathbf{atom}) \qquad \frac{\nabla \vdash a \# t}{\nabla \vdash a \# f_k^E\ t}\ (\#\ \mathbf{app}) \qquad \frac{}{\nabla \vdash a \# [a]t}\ (\#\ \mathbf{a[a]})$$

$$\frac{\nabla \vdash a \# t}{\nabla \vdash a \# [b]t}\ (\#\ \mathbf{a[b]}) \qquad \frac{(\pi^{-1} \cdot a \# X) \in \nabla}{\nabla \vdash a \# \pi.X}\ (\#\ \mathbf{var}) \qquad \frac{\nabla \vdash a \# s \quad \nabla \vdash a \# t}{\nabla \vdash a \# \langle s, t\rangle}\ (\#\ \mathbf{pair})$$

Fig. 1. Rules for the freshness relation

$$\frac{}{\nabla \vdash \langle\rangle \approx_\alpha \langle\rangle}\ (\approx_\alpha \langle\rangle) \qquad \frac{}{\nabla \vdash \overline{a} \approx_\alpha \overline{a}}\ (\approx_\alpha\ \mathbf{atom}) \qquad \frac{\nabla \vdash s \approx_\alpha t}{\nabla \vdash f_k^E\ s \approx_\alpha f_k^E\ t}\ (\approx_\alpha\ \mathbf{app})$$

$$\frac{\nabla \vdash s \approx_\alpha t}{\nabla \vdash [a]s \approx_\alpha [a]t}\ (\approx_\alpha\ [\mathbf{aa}]) \qquad \frac{\nabla \vdash s \approx_\alpha (a\,b) \cdot t \quad \nabla \vdash a \# t}{\nabla \vdash [a]s \approx_\alpha [b]t}\ (\approx_\alpha\ [\mathbf{ab}])$$

$$\frac{ds(\pi, \pi')\#X \subseteq \nabla}{\nabla \vdash \pi.X \approx_\alpha \pi'.X}\ (\approx_\alpha\ \mathbf{var}) \qquad \frac{\nabla \vdash s_0 \approx_\alpha t_0 \quad \nabla \vdash s_1 \approx_\alpha t_1}{\nabla \vdash \langle s_0, s_1\rangle \approx_\alpha \langle t_0, t_1\rangle}\ (\approx_\alpha\ \mathbf{pair})$$

Fig. 2. Rules for the relation $\approx_\alpha$

rules in Fig. 1 are used to check if an atom a is fresh in a nominal term t under a freshness context ∇, also denoted as $\nabla \vdash a\#t$. The rules in Fig. 2 are used to check if two nominal terms s and t are α-equivalent under some freshness context ∇, written as $\nabla \vdash s \approx_\alpha t$. These rules use the inference system for freshness constraints: specifically freshness constraints are used in rule $(\approx_\alpha\ [\mathbf{ab}])$.

Example 1. Let $\sigma = \{X/[a]a\}$. Verify that $\langle (a\ b).X, f(e)\rangle\sigma \approx_\alpha \langle X, f(e)\rangle\sigma$.

By $dom(\pi)\#X$ and $ds(\pi, \pi')\#X$ we abbreviate the sets $\{a\#X \mid a \in dom(\pi)\}$ and $\{a\#X \mid a \in ds(\pi, \pi')\}$, respectively.

Key properties of the nominal freshness and α-equivalence relations have been extensively explored in previous works [3,6,29,30].

2.1 The Relation $\approx_{\{\alpha,C\}}$ as an Extension of $\approx_\alpha$

In [3], the relation $\approx_\alpha$ was extended to deal with associative and commutative theories. Here we will consider α-equivalence modulo commutativity, denoted as $\approx_{\{\alpha,C\}}$. This means that some function symbols in our syntax are commutative, and therefore the rule for function application $(\approx_\alpha\ \mathbf{app})$ in Fig. 2 should be replaced by the rules in Fig. 3.

The following properties for $\approx_{\{\alpha,C\}}$ were formalised as simple adaptations of the formalisations given in [3] for $\approx_\alpha$.

Lemma 2 (Inversion). *The inference rules of* $\approx_{\{\alpha,C\}}$ *are* invertible.

This means, for instance, that for rules $(\approx_\alpha\ [\mathbf{ab}])$ one has $\nabla \vdash [a]s \approx_{\{\alpha,C\}} [b]t$ implies $\nabla \vdash s \approx_{\{\alpha,C\}} (a\,b) \cdot t$ and $\nabla \vdash a \# t$; and for $(\approx_{\{\alpha,\mathbf{C}\}}\ \mathbf{app})$,

$$\frac{\nabla \vdash s \approx_{\{\alpha,C\}} t}{\nabla \vdash f_k^E\, s \approx_{\{\alpha,C\}} f_k^E\, t}\,,\quad E \neq C \text{ or both } s \text{ and } t \text{ are not pairs}\quad (\approx_{\{\alpha,\mathbf{C}\}} \mathbf{app})$$

$$\frac{\nabla \vdash s_0 \approx_{\{\alpha,C\}} t_i,\quad \nabla \vdash s_1 \approx_{\{\alpha,C\}} t_{(i+1)\bmod 2}}{\nabla \vdash f_k^C\, \langle s_0, s_1\rangle \approx_{\{\alpha,C\}} f_k^C\, \langle t_0, t_1\rangle}\,,\quad i = 0,1\quad (\approx_{\{\alpha,\mathbf{C}\}} \mathbf{C})$$

Fig. 3. Additional rules for $\{\alpha, C\}$-equivalence

$\nabla \vdash f_k^C\, \langle s_0, s_1\rangle \approx_{\{\alpha,C\}} f_k^C\, \langle t_0, t_1\rangle$ implies $\nabla \vdash s_0 \approx_{\{\alpha,C\}} t_0$ and $\nabla \vdash s_1 \approx_{\{\alpha,C\}} t_1$, or $\nabla \vdash s_0 \approx_{\{\alpha,C\}} t_1$ and $\nabla \vdash s_1 \approx_{\{\alpha,C\}} t_0$.

Lemma 3 (Freshness preservation). *If* $\nabla \vdash a \# s$ *and* $\nabla \vdash s \approx_{\{\alpha,C\}} t$ *then* $\nabla \vdash a \# t$.

Lemma 4 (Intermediate transitivity for $\approx_{\{\alpha,C\}}$ with $\approx_\alpha$). *If* $\nabla \vdash s \approx_{\{\alpha,C\}} t$ *and* $\nabla \vdash t \approx_\alpha u$ *then* $\nabla \vdash s \approx_{\{\alpha,C\}} u$.

Lemma 5 (Equivariance). $\nabla \vdash \pi \cdot s \approx_{\{\alpha,C\}} \pi \cdot t$ *whenever* $\nabla \vdash s \approx_{\{\alpha,C\}} t$.

Lemma 6 (Equivalence). $\_ \vdash \_ \approx_{\{\alpha,C\}} \_$ *is an equivalence relation.*

Remark 1. According to the grammar for nominal terms, function symbols have no fixed arity: any function symbol can apply to any term. Despite this, in the syntax of our Coq formalisation commutative symbols apply only to tuples.

3 A Nominal C-Unification Algorithm

Inference rules are given that transform a nominal C-unification problem into a finite family of problems that consist exclusively of FP equations of the form $\pi.X \approx_? X$, together with a substitution and a set of freshness constraints.

Definition 3 (Unification problem). *A* unification problem *is a pair* $\langle \nabla, P\rangle$, *where* ∇ *is a* freshness context *and* P *is a finite set of* equations *and* freshness constraints *of the form* $s \approx_? t$ *and* $a \#_? s$, *respectively, where* $\approx_?$ *is symmetric,* s *and* t *are terms and* a *is an atom. Nominal terms in the equations preserve the syntactic restriction that commutative symbols are only applied to tuples.*

Given $\langle \nabla, P\rangle$, by $P_\approx, P_\#, P_{\mathrm{fp}_\approx}$ and $P_{\mathrm{nfp}_\approx}$ we will resp. denote the sets of equations, freshness constraints, FP and non FP equations in the set P.

Example 2. Given the nominal unification problem $\mathcal{P} = \langle \emptyset, \{[a][b]X \approx_? [b][a]X\}\rangle$, the standard unification algorithm [30] reduces it to $\langle \emptyset, \{X \approx_? (a\,b).X\}\rangle$, which gives the solution $\langle \{a\#X, b\#X\}, id\rangle$. However, we will see that infinite independent solutions are feasible when there is at least a commutative operator.

We design a nominal C-unification algorithm using one set of transformation rules to deal with equations (Fig. 4) and another set of rules to deal with freshness constraints and contexts (Fig. 5). These rules act over triples of the form $\langle \nabla, \sigma, P\rangle$, where σ is a substitution. The triple that will be associated by default with a unification problem $\langle \nabla, P\rangle$ is $\langle \nabla, id, P\rangle$. We will use calligraphic uppercase letters (e.g., $\mathcal{P}, \mathcal{Q}, \mathcal{R}$, etc) to denote triples.

Remark 2. Let ∇ and ∇' be freshness contexts and σ and σ' be substitutions.

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

Definition 4 (Solution for a triple or problem). *A* solution *for a triple* $\mathcal{P} = \langle \Delta, \delta, P\rangle$ *is a pair* $\langle \nabla, \sigma\rangle$*, where the following conditions are satisfied:*

1. $\nabla \vdash \Delta\sigma$;
2. $\nabla \vdash a \# t\sigma$, *if* $a\#_? t \in P$;
3. $\nabla \vdash s\sigma \approx_{\{\alpha,C\}} t\sigma$, *if* $s \approx_? t \in P$;
4. *there is a substitution* λ *such that* $\nabla \vdash \delta\lambda \approx \sigma$.

A solution for a unification problem $\langle \Delta, P\rangle$ *is a solution for the associated triple* $\langle \Delta, id, P\rangle$*. The* solution set *for a problem or triple* $\mathcal{P}$ *is denoted by* $\mathcal{U}_C(\mathcal{P})$.

Definition 5 (More general solution and complete set of solutions). *For* $\langle \nabla, \sigma\rangle$ *and* $\langle \nabla', \sigma'\rangle$ *in* $\mathcal{U}_C(\mathcal{P})$*, we say that* $\langle \nabla, \sigma\rangle$ *is* more general *than* $\langle \nabla', \sigma'\rangle$*, denoted* $\langle \nabla, \sigma\rangle \preccurlyeq \langle \nabla', \sigma'\rangle$*, if there exists a substitution* λ *satisfying* $\nabla' \vdash \sigma\lambda \approx \sigma'$ *and* $\nabla' \vdash \nabla\lambda$*. A subset* $\mathcal{V}$ *of* $\mathcal{U}_C(\mathcal{P})$ *is said to be a* complete set of solutions *of* $\mathcal{P}$ *if for all* $\langle \nabla', \sigma'\rangle \in \mathcal{U}_C(\mathcal{P})$*, there exists* $\langle \nabla, \sigma\rangle$ *in* $\mathcal{V}$ *such that* $\langle \nabla, \sigma\rangle \preccurlyeq \langle \nabla', \sigma'\rangle$.

We will denote the set of variables occurring in the set P of a problem $\langle \Delta, P\rangle$ or triple $\mathcal{P} = \langle \nabla, \sigma, P\rangle$ as $Var(P)$. We also will write $Var(\mathcal{P})$ to denote this set.

The unification algorithm proceeds by simplification. Derivation with rules of Figs. 4 and 5 is respectively denoted by $\Rightarrow_{\approx}$ and $\Rightarrow_{\#}$. Thus, $\langle \nabla, \sigma, P\rangle \Rightarrow_{\approx} \langle \nabla, \sigma', P'\rangle$ means that the second triple is obtained from the first one by application of one rule. We will use the standard rewriting nomenclature, e.g., we will say that $\mathcal{P}$ is a *normal form* or *irreducible* by $\Rightarrow_{\approx}$, denoted by $\Rightarrow_{\approx}$-*nf*, whenever there is no $\mathcal{Q}$ such that $\mathcal{P} \Rightarrow_{\approx} \mathcal{Q}$; $\Rightarrow_{\approx}^{*}$ and $\Rightarrow_{\approx}^{+}$ denote respectively derivations in zero or more and one or more applications of the rules in Fig. 4.

The only rule that can generate branches is ($\approx_?$ **C**), which is an abbreviation for two rules providing the different forms in which one can relate the arguments s and t in an equation $f_k^C\, s \approx_?\, f_k^C\, t$ for a commutative function symbol (s, t are tuples, by the syntactic restriction in Definition 3): either $\langle s_0, s_1\rangle \approx_? \langle t_0, t_1\rangle$ or $\langle s_0, s_1\rangle \approx_? \langle t_1, t_0\rangle$.

The syntactic restriction on arguments of commutative symbols being only tuples, is not crucial since any equation of the form $f_k^C \pi.X \approx_?\, t$ can be translated into an equation of form $f_k^C\langle \pi.X_1, \pi.X_2\rangle \approx_?\, t$, where X_1 and X_2 are new variables and ∇ is extended to ∇' in such a way that both X_1 and X_2 inherit all freshness constraints of X in ∇: $\nabla' = \nabla \cup \{a\#X_i \mid i = 1, 2, \text{ and } a\#X \in \nabla\}$.

$$\frac{\langle \nabla, \sigma, P \uplus \{s \approx_? s\}\rangle}{\langle \nabla, \sigma, P\rangle}\,(\approx_? \textbf{refl}) \qquad \frac{\langle \nabla, \sigma, P \uplus \{\langle s_1, t_1\rangle \approx_? \langle s_2, t_2\rangle\}\rangle}{\langle \nabla, \sigma, P \cup \{s_1 \approx_? s_2, t_1 \approx_? t_2\}\rangle}\,(\approx_? \textbf{pair})$$

$$\frac{\langle \nabla, \sigma, P \uplus \{f_k^E\, s \approx_? f_k^E\, t\}\rangle}{\langle \nabla, \sigma, P \cup \{s \approx_? t\}\rangle}\,,\ \text{if } E \neq C\ (\approx_? \textbf{app})$$

$$\frac{\langle \nabla, \sigma, P \uplus \{f_k^C\, s \approx_? f_k^C\, t\}\rangle}{\langle \nabla, \sigma, P \cup \{s \approx_? v\}\rangle}\,,\ \left\{\begin{array}{l}\text{where } s = \langle s_0, s_1\rangle \text{ and } t = \langle t_0, t_1\rangle\\ v = \langle t_i, t_{(i+1)\,mod\,2}\rangle, i = 0, 1\end{array}\right\}\,(\approx_? \textbf{C})$$

$$\frac{\langle \nabla, \sigma, P \uplus \{[a]s \approx_? [a]t\}\rangle}{\langle \nabla, \sigma, P \cup \{s \approx_? t\}\rangle}\,(\approx_? [\textbf{aa}]) \qquad \frac{\langle \nabla, \sigma, P \uplus \{[a]s \approx_? [b]t\}\rangle}{\langle \nabla, \sigma, P \cup \{s \approx_? (a\,b)\,t, a\#_? t\}\rangle}\,(\approx_? [\textbf{ab}])$$

$$\frac{\langle \nabla, \sigma, P \uplus \{\pi.X \approx_? t\}\rangle \ \text{ let } \sigma' := \sigma\{X/\pi^{-1} \cdot t\}}{\left\langle \nabla, \sigma', P\{X/\pi^{-1} \cdot t\} \ \cup \bigcup_{\substack{Y \in dom(\sigma'),\\ a\#Y \in \nabla}} \{a\#_? Y\sigma'\} \right\rangle}\,,\ \text{if } X \notin Var(t)\ (\approx_? \textbf{inst})$$

$$\frac{\langle \nabla, \sigma, P \uplus \{\pi.X \approx_? \pi'.X\}\rangle}{\langle \nabla, \sigma, P \cup \{\pi \oplus (\pi')^{-1}.X \approx_? X\}\rangle}\,,\ \text{if } \pi' \neq nil\ (\approx_? \textbf{inv})$$

Fig. 4. Reduction rules for equational problems

In the rule ($\approx_?$ **inst**) the inclusion of new constraints in the problem, given in $\bigcup_{\substack{Y \in dom(\sigma'),\\ a\#Y \in \nabla}} \{a\#_? Y\sigma'\}$ is necessary to guarantee that the new substitution σ' is *compatible* with the freshness context ∇.

Examples 3, 4 and 5 are running examples of the C-unification procedure. A graphic representation of the derivation tree for these examples, generated using the OCaml implementation, is depicted in the extended version of this paper.

Example 3. Let $*$[1] be a commutative function symbol. Below, we show how the problem $\mathcal{P} = \langle \emptyset, \{[e](a\,b).X * Y \approx_? [f](a\,c)(c\,d).X * Y\}\rangle$ reduces (via rules

$$\frac{\langle \nabla, \sigma, P \uplus \{a\#_? \langle\rangle\}\rangle}{\langle \nabla, \sigma, P\rangle}\,(\#_? \langle\rangle) \qquad \frac{\langle \nabla, \sigma, P \uplus \{a\#_? \bar{b}\}\rangle}{\langle \nabla, \sigma, P\rangle}\,(\#_? \textbf{a}\bar{\textbf{b}})$$

$$\frac{\langle \nabla, \sigma, P \uplus \{a\#_? f\,t\}\rangle}{\langle \nabla, \sigma, P \cup \{a\#_? t\}\rangle}\,(\#_? \textbf{app}) \qquad \frac{\langle \nabla, \sigma, P \uplus \{a\#_? [a]t\}\rangle}{\langle \nabla, \sigma, P\rangle}\,(\#_? \textbf{a}[\textbf{a}])$$

$$\frac{\langle \nabla, \sigma, P \uplus \{a\#_? [b]t\}\rangle}{\langle \nabla, \sigma, P \cup \{a\#_? t\}\rangle}\,(\#_? \textbf{a}[\textbf{b}]) \qquad \frac{\langle \nabla, \sigma, P \uplus \{a\#_? \pi.X\}\rangle}{\langle \{(\pi^{-1} \cdot a)\#X\} \cup \nabla, \sigma, P\rangle}\,(\#_? \textbf{var})$$

$$\frac{\langle \nabla, \sigma, P \uplus \{a\#_? \langle s, t\rangle\}\rangle}{\langle \nabla, \sigma, P \cup \{a\#_? s, a\#_? t\}\rangle}\,(\#_? \textbf{pair})$$

Fig. 5. Reduction rules for freshness problems

[1] Infix notation is adopted for commutative symbols: $s * t$ abbreviates $*\langle s, t\rangle$.

in Figs. 4 and 5). Application of rule ($\approx_?$ **C**) gives two branches that reduce into two FP problems: $\mathcal{Q}_1$ and $\mathcal{Q}_2$. Highlighted terms show where the rules are applied. For brevity, let $\pi_1 = (a\,c)(c\,d)(e\,f)$, $\pi_2 = (a\,b)(e\,f)(c\,d)(a\,c)$, $\pi_3 = (a\,c)(c\,d)(e\,f)(a\,b)$ and $\sigma = \{X/(e\,f)(a\,b).Y\}$.

$\langle \emptyset, id, \{ [e](a\,b).X * Y \approx_? [f](a\,c)(c\,d).X * Y \}\rangle \Rightarrow_{(\approx_?[\mathbf{ab}])}$

$\langle \emptyset, id, \{ (a\,b).X * Y \approx_? \pi_1.X * (e\,f).Y,\ e\#_?(a\,c)(c\,d).X * Y\}\rangle \Rightarrow_{(\approx_?\mathbf{C})}$

branch 1: $\langle \emptyset, id, \{ (a\,b).X \approx_? \pi_1.X,\ Y \approx_? (e\,f).Y,\ e\#_?(a\,c)(c\,d).X * Y\}\rangle$

$\Rightarrow_{(\approx_?\mathbf{inv})(2\times)} \langle \emptyset, id, \{(a\,b)[\pi_1]^{-1}.X \approx_? X,\ [(e\,f)]^{-1}.Y \approx_? Y,\ e\#_?(a\,c)(c\,d).X * Y\}\rangle$

$\Rightarrow_{(\#_?\mathbf{app}),(\#_?\mathbf{pair})} \langle \emptyset, id, \{\pi_2.X \approx_? X, (e\,f).Y \approx_? Y,\ e\#_?(a\,c)(c\,d).X,\ e\#_?Y\}\rangle$

$\Rightarrow_{(\#_?\mathbf{var})(2\times)} \langle \{e\#X, e\#Y\}, id, \{\pi_2.X \approx_? X, (e\,f).Y \approx_? Y\}\rangle = \mathcal{Q}_1$

branch 2: $\langle \emptyset, id, \{ (a\,b).X \approx_? (e\,f).Y,\ Y \approx_? \pi_1.X,\ e\#_?(a\,c)(c\,d).X * Y\}\rangle$

$\Rightarrow_{(\approx_?\mathbf{inst})} \langle \emptyset, \sigma, \{ Y \approx_? (a\,c)(c\,d)(e\,f)(e\,f)[(a\,b)]^{-1}.Y, e\#_?\pi_1[(a\,b)]^{-1}.Y * Y\}\rangle$

$\Rightarrow_{(\approx_?\mathbf{inv})} \langle \emptyset, \sigma, \{[(a\,c)(c\,d)(a\,b)]^{-1}.Y \approx_? Y,\ e\#_?\pi_3.Y * Y\}\rangle$

$\Rightarrow_{(\#_?\mathbf{app}),(\#_?\mathbf{pair})} \langle \emptyset, \sigma, \{(a\,b)(c\,d)(a\,c).Y \approx_? Y,\ e\#_?\pi_3.Y,\ e\#_?Y\}\rangle$

$\Rightarrow_{(\#_?\mathbf{var})(2\times)} \langle \{e\#Y, f\#Y\}, \sigma, \{(a\,b)(c\,d)(a\,c).Y \approx_? Y\}\rangle = \mathcal{Q}_2$

Definition 6 (Set of $\Rightarrow_\approx$ and $\Rightarrow_\#$-normal forms). *We denote by $\mathcal{P}_{\Rightarrow_\approx}$ (resp. $\mathcal{P}_{\Rightarrow_\#}$) the set of normal forms of $\mathcal{P}$ with respect to $\Rightarrow_\approx$ (resp. $\Rightarrow_\#$).*

Definition 7 (Fail and success for $\Rightarrow_\approx$). *Let $\mathcal{P}$ be a triple, such that the rules in Fig. 4 give rise to a normal form $\langle \nabla, \sigma, P\rangle$. The rules in Fig. 4 are said to* fail *if P contains non FP equations. Otherwise $\langle \nabla, \sigma, P\rangle$ is called a* successful *triple regarding $\Rightarrow_\approx$ (i.e., in a successful triple, P consists only of FP equations and, possibly, freshness constraints).*

The rules in Fig. 5 will only be applied to successful triples regarding $\Rightarrow_\approx$.

Definition 8 (Fail and success for $\Rightarrow_\#$). *Let $\mathcal{Q} = \langle \nabla, \sigma, Q\rangle$ be a successful triple regarding $\Rightarrow_\approx$, and $\mathcal{Q}' = \langle \nabla', \sigma, Q'\rangle$ its normal form via rules in Fig. 5, that is $\mathcal{Q} \Rightarrow_\#^* \mathcal{Q}'$ and $\mathcal{Q}'$ is in $\mathcal{Q}_{\Rightarrow_\#}$. If Q' contains freshness constraints it is said that $\Rightarrow_\#$* fails *for $\mathcal{Q}$; otherwise, $\mathcal{Q}'$ will be called a* successful *triple for $\Rightarrow_\#$.*

Remark 3. Since in a successful triple regarding $\Rightarrow_\approx$, $\mathcal{Q}$, one has only FP equations and $\Rightarrow_\#$ acts only over freshness constraints, Q' in the definition above contains only FP equations and freshness constraints. Also, by a simple case analysis on t one can check that any triple with freshness constraints $a\#_?t$ is reducible by $\Rightarrow_\#$, except when $t \equiv \bar{a}$. Hence the freshness constraints in Q' would be only of the form $a\#_?\bar{a}$.

The relation $\Rightarrow_\approx$, starts from a triple with the identity substitution and always maintains a triple $\langle \nabla, \sigma', P'\rangle$ in which the substitution σ' does not affect the current problem P'. The same happens for $\Rightarrow_\#$ since the substitution does not change with this relation. This motivates the next definition and lemma.

Definition 9 (Valid triple). $\mathcal{P} = \langle \nabla, \sigma, P \rangle$ is valid *if* $im(\sigma) \cap dom(\sigma) = \emptyset$ *and* $dom(\sigma) \cap Var(P) = \emptyset$.

Remark 4. A substitution σ in a valid triple $\mathcal{P}$ is *idempotent*, that is, $\sigma\sigma = \sigma$.

Lemma 7 is proved by case analysis on the rules used by $\Rightarrow_{\approx}$ and $\Rightarrow_{\#}$.

Lemma 7 (Preservation of valid triples). *If* $\mathcal{P} = \langle \nabla, \sigma, P \rangle$ *is valid and* $\mathcal{P} \Rightarrow_{\approx} \cup \Rightarrow_{\#} \mathcal{P}' = \langle \nabla', \sigma', P' \rangle$, *then* $\mathcal{P}'$ *is also valid.*

From now on, we consider only valid triples.

Lemma 8 (Termination of $\Rightarrow_{\approx}$ and $\Rightarrow_{\#}$). *There is no infinite chain of reductions* $\Rightarrow_{\approx}$ *(or* $\Rightarrow_{\#}$*) starting from an arbitrary triple* $\mathcal{P} = \langle \nabla, \sigma, P \rangle$.

Proof.
- The proof for $\Rightarrow_{\approx}$ is by well-founded induction on $\mathcal{P}$ using the measure $\|\mathcal{P}\| = \langle |Var(P_{\approx})|, \|P\|, |P_{\text{nfp}_{\approx}}| \rangle$ with a lexicographic ordering, where $\|P\| = \sum_{s \approx_? t \in P_{\approx}} |s| + |t| + \sum_{a \#_? u \in P_{\#}} |u|$. Note that this measure decreases after each step $\langle \nabla, \sigma, P \rangle \Rightarrow_{\approx} \langle \nabla, \sigma', P' \rangle$: for ($\approx_?$ **inst**), $|Var(P_{\approx})| > |Var(P'_{\approx})|$; for ($\approx_?$ **refl**), ($\approx_?$ **pair**), ($\approx_?$ **app**), ($\approx_?$ [**aa**]), ($\approx_?$ [**ab**]) and ($\approx_?$ **C**), $|Var(P_{\approx})| \geq |Var(P'_{\approx})|$, but $\|P\| > \|P'\|$; and, for ($\approx_?$ **inv**), both $|Var(P_{\approx})| = |Var(P' \approx)|$ and $\|P\| = \|P'\|$, but $|P_{\text{nfp}_{\approx}}| > |P'_{\text{nfp}_{\approx}}|$.
- The proof for $\Rightarrow_{\#}$ is by induction on $\mathcal{P}$ using as measure $\|P_{\#}\|$. It can be checked that this measure decreases after each step: $\langle \nabla, \sigma, P \rangle \Rightarrow_{\#} \langle \nabla, \sigma', P' \rangle$.

To solve a unification problem, $\langle \nabla, P \rangle$, one builds the derivation tree for $\Rightarrow_{\approx}$, labelling the root node with $\langle \nabla, id, P \rangle$. This tree has leaves labelled with $\Rightarrow_{\approx}$-nf's that are either failing or successful triples. Then, the tree is extended by building $\Rightarrow_{\#}$-derivations starting from all successful leaves. The extended tree will include failing leaves and successful leaves. The successful leaves will be labelled by triples $\mathcal{P}'$ in which the problem P' consists only of FP equations. Since $\Rightarrow_{\approx}$ and $\Rightarrow_{\#}$ are both terminating (Lemma 8), the process described above must be also terminating.

Definition 10 (Derivation tree for $\langle \Delta, P \rangle$). *A* derivation tree *for the unification problem* $\langle \Delta, P \rangle$, *denoted as* $\mathcal{T}_{\langle \Delta, P \rangle}$, *is a tree with root label* $\mathcal{P} = \langle \Delta, id, P \rangle$ *built in two stages:*

- *Initially, a tree is built, whose branches end in leaf nodes labelled with the triples in* $\mathcal{P}_{\Rightarrow_{\approx}}$. *The labels in each path from the root to a leaf correspond to a* $\Rightarrow_{\approx}$*-derivation.*
- *Further, for each leaf labelled with a successful triple* $\mathcal{Q}$ *in* $\mathcal{P}_{\Rightarrow_{\approx}}$, *the tree is extended with a path to a new leaf that is labelled with a* $\bar{\mathcal{Q}} \in \mathcal{Q}_{\Rightarrow_{\#}}$. *The labels in the extended path from the node with label* $\mathcal{Q}$ *to the new leaf correspond to a* $\Rightarrow_{\#}$*-derivation.*

Remark 5. For $\langle \Delta, P \rangle$, all labels in the nodes of $\mathcal{T}_{\langle \Delta, P \rangle}$ are *valid* by Lemma 7.

The next lemma is proved by case analysis on elements of $\mathcal{P}_{\Rightarrow_{\approx}}$ and $\mathcal{P}_{\Rightarrow_{\#}}$.

Lemma 9 (Characterisation of leaves of $\mathcal{T}_{\langle\Delta,P\rangle}$). *Let $\langle\Delta, P\rangle$ be a unification problem. If $\mathcal{P}' = \langle\nabla, \sigma', P'\rangle$ is the label of a leaf in $\mathcal{T}_{\langle\Delta,P\rangle}$, then P' can be partitioned as follows: $P' = P'' \cup P_\perp$, where P'' is the set of all FP equations in P' and $P_\perp = P' - P''$. If $P_\perp \neq \emptyset$ then $\mathcal{U}_C(\mathcal{P}') = \emptyset$.*

The next definition is motivated by the previous characterisation of the labels of leaves in derivation trees.

Definition 11 (Successful leaves). *Let $\langle\Delta, P\rangle$ be a unification problem. A leaf in $\mathcal{T}_{\langle\Delta,P\rangle}$ that is labelled with a triple of the form $\mathcal{Q} = \langle\nabla, \sigma, Q\rangle$, where Q consists only of FP equations, is called a* successful leaf *of $\mathcal{T}_{\langle\Delta,P\rangle}$. In this case $\mathcal{Q}$ is called a* successful triple *of $\mathcal{T}_{\langle\Delta,P\rangle}$. The sets of successful leaves and triples of $\mathcal{T}_{\langle\Delta,P\rangle}$ are denoted respectively by $SL(\mathcal{T}_{\langle\Delta,P\rangle})$ and $ST(\mathcal{T}_{\langle\Delta,P\rangle})$.*

The soundness theorem states that successful leaves of $\mathcal{T}_{\langle\Delta,P\rangle}$ produce *correct* solutions. The proof is by induction on the number of steps of $\Rightarrow_{\approx}$ and $\Rightarrow_{\#}$ and uses Lemma 9 and auxiliary results on the *preservation of solutions* by $\Rightarrow_{\approx}$ and $\Rightarrow_{\#}$. Proving preservation of solutions for rules ($\approx_?$ [**ab**]) and ($\approx_?$ **inst**) is not straightforward and uses Lemmas 1, 2, 3 and 5 to check that the four conditions of Definition 4 are valid before, if one supposes their validity after the rule application.

Theorem 1 (Soundness of $\mathcal{T}_{\langle\Delta,P\rangle}$). *$\mathcal{T}_{\langle\Delta,P\rangle}$ is correct, i.e., if $\mathcal{P}' = \langle\nabla, \sigma, P'\rangle$ is the label of a leaf in $\mathcal{T}_{\langle\Delta,P\rangle}$, then 1. $\mathcal{U}_C(\mathcal{P}') \subseteq \mathcal{U}_C(\langle\Delta, id, P\rangle)$, and 2. if P' contains non FP equations or freshness constraints then $\mathcal{U}_C(\mathcal{P}') = \emptyset$.*

The completeness theorem guarantees that the set of successful triples provides a complete set of solutions. Its proof uses case analysis on the rules of the relations $\Rightarrow_{\approx}$ and $\Rightarrow_{\#}$ by an argumentation similar to the one used for Theorem 1. For $\Rightarrow_{\#}$ one has indeed equivalence: $\mathcal{P} \Rightarrow_{\#} \mathcal{P}'$, implies $\mathcal{U}_C(\mathcal{P}) = \mathcal{U}_C(\mathcal{P}')$. The same is true for all rules of the relation $\Rightarrow_{\approx}$ except the branching rule ($\approx_?$ **C**), for which it is necessary to prove that all solutions of a triple reduced by ($\approx_?$ **C**) must belong to the set of solutions of one of its children triples.

Theorem 2 (Completeness of $\mathcal{T}_{\langle\Delta,P\rangle}$). *Let $\langle\Delta, P\rangle$ and $\mathcal{T}_{\langle\Delta,P\rangle}$ be a unification problem and its derivation tree. Then $\mathcal{U}_C(\langle\Delta, id, P\rangle) = \bigcup_{\mathcal{Q}\in ST(\mathcal{T}_{\langle\Delta,P\rangle})} \mathcal{U}_C(\mathcal{Q})$.*

Corollary 1 (Generality of successful triples). *Let $\mathcal{P} = \langle\Delta, P\rangle$ be a unification problem and $\langle\nabla'', \sigma'\rangle \in \mathcal{U}_C(\mathcal{P})$. Then there exists a successful triple $\mathcal{Q} \in ST(\mathcal{T}_{\langle\Delta,P\rangle})$ where $\mathcal{Q} = \langle\nabla, \sigma, Q\rangle$ such that $\langle\nabla'', \sigma'\rangle \in \mathcal{U}_C(\mathcal{Q})$, and hence, $\nabla'' \vdash \nabla\sigma'$ and there exists λ such that $\nabla'' \vdash \sigma\lambda \approx \sigma'$.*

Proof. By Theorem 2, $\mathcal{U}_C(\mathcal{P}) = \bigcup_{\mathcal{P}'\in ST(\mathcal{T}_{\langle\Delta,P\rangle})} \mathcal{U}_C(\mathcal{P}')$. Then there exists $\mathcal{Q} \in ST(\mathcal{T}_{\langle\Delta,P\rangle})$ such that $\langle\nabla'', \sigma'\rangle \in \mathcal{U}_C(\mathcal{Q})$. Suppose $\mathcal{Q} = \langle\nabla, \sigma, Q\rangle$. Then by the first and fourth conditions of the definition of solution (Definition 4) we have that $\nabla'' \vdash \nabla\sigma'$ and there exists λ such that $\nabla'' \vdash \sigma\lambda \approx \sigma'$.

Remark 6. The nominal C-unification problem is to decide, for a given $\mathcal{P}$, if $\mathcal{U}_C(\mathcal{P})$ is non empty; that is, whether $\mathcal{P}$ has nominal C-unifiers. To prove that this problem is in NP, a non-deterministic procedure using the reduction rules in the same order as in Definition 10 is designed. In this procedure, whenever rule $(\approx_? \mathbf{C})$ applies, only one of the two possible branches is guessed. In this manner, if the derivation tree has a successful leaf, this procedure will guess a path to the successful leaf, answering positively to the decision problem. According to the measures used in the proof of termination (Lemma 8), reduction with both the relations $\Rightarrow_{\approx}$ and $\Rightarrow_{\#}$ is polynomially bound, which implies that this non-deterministic procedure is polynomially bound.

To prove NP-completeness, one can polynomially reduce the well-known NP-complete positive 1-in-3-SAT problem into nominal C-unification, as done in [7] for the C-unification problem. An instance of the positive 1-in-3-SAT problem consists of a set of clauses $\mathcal{C} = \{\mathcal{C}_i | 1 \leq i \leq n\}$, where each $\mathcal{C}_i$ is a disjunction of three propositional variables, say $\mathcal{C}_i = p_i \vee q_i \vee r_i$. A solution of $\mathcal{C}$ is a valuation with exactly one variable true in each clause. The proposed reduction of $\mathcal{C}$ into a nominal C-unification problem would require just a commutative function symbol, say $\oplus$, two atoms, say a and b, a variable for each clause $\mathcal{C}_i$, say Y_i, and a variable for each propositional variable p in $\mathcal{C}$, say X_p. Instantiating X_p as $\overline{a}$ or $\overline{b}$, would be interpreted as evaluating p as true or false, respectively. Each clause $\mathcal{C}_i = p_i \vee q_i \vee r_i$ in $\mathcal{C}$ is translated into an equation E_i of the form $((X_{p_i} \oplus X_{q_i}) \oplus X_{r_i}) \oplus Y_i \approx_? ((\overline{b} \oplus \overline{b}) \oplus \overline{a}) \oplus ((\overline{b} \oplus \overline{a}) \oplus \overline{b})$. The nominal C-unification problem for $\mathcal{C}$ is given by $\mathcal{P}_{\mathcal{C}} = \langle \emptyset, \{E_i | 1 \leq i \leq n\} \rangle$. Simplifying $\mathcal{P}_{\mathcal{C}}$ would not introduce freshness constraints since the problem does not include abstractions. Thus, to conclude it is only necessary to check that $\langle \emptyset, \sigma \rangle$ is a solution for $\mathcal{P}_{\mathcal{C}}$ if and only if σ instantiates exactly one of the variables X_{p_i}, X_{q_i} and X_{r_i} in each equation with $\overline{a}$ and the other two with $\overline{b}$, which means that $\mathcal{C}$ has a solution.

4 Generation of Solutions for Successful Leaves of $\mathcal{T}_{\langle \mathbf{\Delta}, P \rangle}$

To build solutions for a successful leaf $\mathcal{P} = \langle \nabla, \sigma, P \rangle$ in the derivation tree of a given unification problem, we will select and combine solutions generated for FP equations $\pi.X \approx_? X$, for each $X \in Var(P)$. We introduce the notion of *pseudo-cycle of a permutation*, in order to provide precise conditions to build terms t by combining the atoms in $dom(\pi)$, such that $\pi \cdot t \approx_{\{\alpha, C\}} t$. For convenience, we use the algebraic cycle representation of permutations. Thus, instead of sequences of swappings, permutations in nominal terms will be read as products of disjoint cycles [27].

Example 4. (Continuing Example 3) The permutations $(a\,b) :: (e\,f) :: (c\,d) :: (a\,c) :: nil$ and $(a\,b) :: (c\,d) :: (a\,c) :: nil$ are respectively represented as the product of permutation cycles $(a\,b\,c\,d)(e\,f)$ and $(a\,b\,c\,d)(e)(f)$.

Permutation cycles of length one are omitted. In general the cyclic representation of a permutation consists of the product of all its cycles.

Let π be a permutation with $dom(\pi) = n$. Given $a \in dom(\pi)$ the elements of the sequence $a, \pi(a), \pi^2(a), \ldots$ cannot be all distinct. Taking the first $k \leq n$, such that $\pi^k(a) = a$, we have the k-cycle $(a\ \pi(a)\ \ldots \pi^{k-1}(a))$, where $\pi^{j+1}(a)$ is the *successor* of $\pi^j(a)$. For the 4-cycle in the permutation $(a\,b\,c\,d)\,(e\,f)$, the 4-cycles generated by a, b, c and d are the same: $(a\,b\,c\,d) = (b\,c\,d\,a) = (c\,d\,a\,b) = (d\,a\,b\,c)$.

Definition 12 establishes the notion of a *pseudo-cycle w.r.t. a k-cycle* κ. Intuitively, given a k-cycle κ and a commutative function symbol $*$, a pseudo-cycle w.r.t κ, $(A_0 \ldots A_l)$, is a cycle whose elements are either atom terms built from the atoms in κ or terms of the form $A'_i * A'_j$, for A'_i, A'_j elements of a pseudo-cycle w.r.t κ.

Definition 12 (Pseudo-cycle). *Let $\kappa = (a_0\ a_1\ \ldots\ a_{k-1})$ be a k-cycle of a permutation π. A* pseudo-cycle *w.r.t. κ is inductively defined as follows:*

1. $\overline{\kappa} = (\overline{a_0} \cdots \overline{a_{k-1}})$ *is a* pseudo-cycle *w.r.t. κ, called* trivial pseudo-cycle *of κ.*
2. $\kappa' = (A_0 \ldots A_{k'-1})$ *is a* pseudo-cycle *w.r.t. κ, if the following conditions are simultaneously satisfied:*
   (a) *each element of κ' is of the form $B_i * B_j$, where $*$ is a commutative function symbol in the signature, and B_i, B_j are different elements of κ'', a* pseudo-cycle *w.r.t. κ. κ' will be called a* first-instance pseudo-cycle *of κ'' w.r.t. κ.*
   (b) *$A_i \not\approx_{\alpha,C} A_j$ for $i \neq j$, $0 \leq i, j \leq k' - 1$;*
   (c) *for each $0 \leq i < k' - 1$, $\kappa \cdot A_i \approx_{\{\alpha,C\}} A_{(i+1) mod\, k'}$.*

The *length* of the pseudo-cycle κ, denoted by $|\kappa|$, consists of the number of elements in κ. A pseudo-cycle of length one will be called *unitary*.

Example 5. A (Continuing Example 2) The unitary pseudo-cycles of $\kappa = (a\,b)$ are of the form $(\overline{a} * \overline{b})$ for $*$ any commutative symbol in the signature. These pseudo-cycles are the basis for a more elaborated construction used to build infinite independent solutions for the leaf $\langle \emptyset, id, \{X \approx_? (a\,b).X\}\rangle$. Examples of these solutions are: $\langle \emptyset, \{X/\overline{a}*\overline{b}\}\rangle$, $\langle \emptyset, \{X/(\overline{a}*\overline{a})*(\overline{b}*\overline{b})\}\rangle$, $\langle \emptyset, \{X/(\overline{a}*\overline{b})*(\overline{a}*\overline{b})\}\rangle$, $\langle \emptyset, \{X/((\overline{a} * \overline{a}) * \overline{a}) * ((\overline{b} * \overline{b}) * \overline{b})\}\rangle$, $\langle \emptyset, \{X/(\overline{a} * (\overline{a} * \overline{a})) * (\overline{b} * (\overline{b} * \overline{b}))\}\rangle$, etc.

B (Continuing Examples 3 and 4) In $\mathcal{Q}_1$ and $\mathcal{Q}_2$ we have the occurrences of the 4-cycle $\kappa = (a\,b\,c\,d)$. Suppose $*, \oplus, +$ are commutative operators in the signature. The following are pseudo-cycles w.r.t. κ: $\overline{\kappa} = (\overline{a}\ \overline{b}\ \overline{c}\ \overline{d})$; $\kappa_1 = ((\overline{a}*\overline{b})\ (\overline{b}*\overline{c})\ (\overline{c}*\overline{d})\ (\overline{d}*\overline{a}))$; $\kappa_2 = ((\overline{a}\oplus\overline{c})\ (\overline{b}\oplus\overline{d}))$; $\kappa_{11} = (((\overline{a}*\overline{b})+(\overline{b}*\overline{c}))\ ((\overline{b}*\overline{c})+(\overline{c}*\overline{d}))((\overline{c}*\overline{d})+(\overline{d}*\overline{a}))\ ((\overline{d}*\overline{a})+(\overline{a}*\overline{b})))$; $\kappa_{12} = (((\overline{a}*\overline{b})*(\overline{c}*\overline{d}))\ ((\overline{b}*\overline{c})*(\overline{d}*\overline{a})))$; $\kappa_{21} = (((\overline{a}\oplus\overline{c}) * (\overline{b}\oplus\overline{d})))$; $\kappa_{121} = (((\overline{a}*\overline{b}) * (\overline{c}*\overline{d})) * ((\overline{b}*\overline{c}) * (\overline{d}*\overline{a})))$. κ_1 and κ_2 are first-instance pseudo-cycles of $\overline{\kappa}$, and κ_{11} and κ_{12} of κ_1 and κ_{21} of κ_2. Notice that, $|\overline{\kappa}| = |\kappa_1| = |\kappa_{11}| = 4$, $|\kappa_{12}| = 2$, and $|\kappa_{21}| = |\kappa_{121}| = 1$. Also, κ_1 corresponds to $((\overline{a} * \overline{d})\ (\overline{b} * \overline{a})\ (\overline{c} * \overline{b})\ (\overline{d} * \overline{c}))$, a first-instance pseudo-cycle of $\overline{\kappa}$. Finally, observe that for the elements of the unitary pseudo-cycles κ_{21} and κ_{121}, say $s = (\overline{a} \oplus \overline{c}) * (\overline{b} \oplus \overline{d})$ and $t = ((\overline{a} * \overline{b}) * (\overline{c} * \overline{d})) * ((\overline{b} * \overline{c}) * (\overline{d} * \overline{a}))$, $\{X/s\}$ and $\{X/t\}$ (resp. $\{Y/s\}$ and $\{Y/t\}$) are solutions of the FP equation $(a\ b\ c\ d)(e\ f).X \approx_? X$ (resp. $(a\ b\ c\ d).Y \approx_? Y$).

Let κ be a pseudo-cycle. Notice that only item 2 of Definition 12 may build a first-instance pseudo-cycle κ' w.r.t. κ with fewer elements. If $|\kappa'| < |\kappa|$ then, due to algebraic properties of cycles and commutativity of the operator applied $(*)$, one must have that $|\kappa'| = |\kappa|/2$. Thus, unitary pseudo-cycles can only be generated from cycles of length a power of two. This is the intuition behind the next theorem, proved by induction on the size of the cycle κ.

Theorem 3. *A pseudo-cycle κ generates unitary pseudo-cycles iff $|\kappa|$ is a power of two.*

Notice that, according to item 2.c of Definition 12, if $\kappa' = (A_0 \dots A_{k'-1})$ is a pseudo-cycle w.r.t. π then $\pi \cdot A_{k'-1} \approx_{\{\alpha,C\}} A_0$; particularly, if $k' = 1$ then $\pi \cdot A_0 \approx_{\{\alpha,C\}} A_0$. Below, given $\mathcal{P} = \langle \emptyset, \{\pi.X \approx_? X\}\rangle$ a FP equational problem, we call a *combinatory solution* of $\mathcal{P}$, a substitution $\{X/t\}$, such that $\pi \cdot t \approx_C t$, and t contains only atoms from π and commutative function symbols, built as unary pseudo-cycles w.r.t. κ a cycle in π.

The next theorem is proved by contradiction, supposing that κ has an odd factor and using Theorem 3.

Theorem 4. *Let $\mathcal{P} = \langle \emptyset, \{\pi.X \approx_? X\}\rangle$ be a FP problem. $\mathcal{P}$ has a combinatory solution iff there exists a unitary pseudo-cycle κ w.r.t. π.*

Remark 7. Since one can generate infinitely many unitary pseudo-cycles from a given 2^n-cycle κ in π, $n \in \mathbb{N}$, there exist infinite independent solutions for the FP problem $\langle \emptyset, \{\pi.X \approx_? X\}\rangle$.

General solutions for FP problems. To compute the set of solutions for a FP equation, we use a method described in [4], which is based on the computation of unitary *extended pseudo-cycles* (`epc`). We refer to [4] for the definition of extended pseudo-cycles and an algorithm to enumerate all the solutions of a successful leaf in the derivation tree.

Pseudo-cycles are built just from atom terms in $dom(\pi)$ and commutative function symbols, while `epc`'s consider all nominal syntactic elements including new variables, and also non commutative function symbols. The soundness and completeness of the generator of solutions described in [4] relies on the properties of pseudo-cycles described above, in particular the fact that only unitary pseudo-cycles generate solutions.

5 Formal Proofs

In the Coq formalisation, nominal terms are specified inductively, which permits to use induction to formalise properties of terms (to check nominal α-equality modulo C we use the rules given in [3]; see Fig. 3). The relations $\Rightarrow_{\#}$ and $\Rightarrow_{\approx}$ are inductivelty specified, as propositions from problems to problems, resp. as `fresh_sys` and `equ_sys`, and normal forms and their reflexive-transitive closures are specified using abstract relations as shown below.

```
Definition NF (T:Type) (R:T→T→Prop) (s:T) := ∀ t, ¬ R s t.

Inductive tr_clos (T:Type) (R:T→T→Prop) : T→T→Prop :=
 | tr_rf : ∀ s, tr_clos T R s s
 | tr_os : ∀ s t, R s t → tr_clos T R s t
 | tr_ms : ∀ s t u, R s t → tr_clos T R t u → tr_clos T R s u
```

A unification step, unif_step, is a reduction step either with the relation equ_sys or with the relation fresh_sys, the latter restricted to FP problems; and a leaf is a normal form for this relation.

```
Inductive unif_step : Triple → Triple → Prop :=
 | equ_unif_step : ∀ T T', equ_sys T T' → unif_step T T'
 | fresh_unif_step : ∀ T T', fixpoint_Problem (equ_proj (snd T)) →
                             fresh_sys T T' → unif_step T T' .

Definition leaf (T : Triple) := NF _ unif_step T .
```

Unification paths are derivations with the relation unif_step to a leaf:

```
Definition unif_path (T T' : Triple) := tr_clos _ unif_step T T' ∧ leaf T'.
```

Soundness is specified as the Theorem below, which reads: for any unification problem T that reduces into a problem T' with the relation unif_path, and such that Sl is a solution of T', Sl is also a solution of T.

```
Theorem c_unif_path_soundness : ∀ T T' Sl,
        valid_triple T → unif_path T T' → sol_c Sl T' → sol_c Sl T.
```

The formalisation of soundness is given in a theory that consists of 902 lines or 35 KB. This theory also includes lemmas that characterise successful leaves and their solutions. The theorem uses three auxiliary lemmas, also proved by induction. A lemma expresses preservation of the set of solutions of unification problems under reduction by the relation $\Rightarrow_{\#}$:

```
Lemma fresh_sys_compl : ∀ T T' Sl, fresh_sys T T' → (sol_c Sl T ↔ sol_c Sl T') .
```

Another lemma, the longer one, states that the solutions of a unification problem obtained from a given problem through application of the relation $\Rightarrow_{\approx}$ are solutions of the given problem:

```
Lemma equ_sol_preserv : ∀ T T' Sl, valid_triple T →
                        equ_sys T T' → sol_c Sl T' → sol_c Sl T .
```

Finally, the last auxiliary lemma applied to prove soundness states that solutions are preserved in each unification step:

```
Lemma unif_step_preserv : ∀ T T' Sl,
        valid_triple T → unif_step T T' → sol_c Sl T' → sol_c Sl T.
```

Since except $(\approx_{\{\alpha,\mathbf{C}\}}\ \mathbf{C})$ unification rules are invertible, the formalisation of the proof of completeness is shorter, consisting only of 351 lines or 13 KB. The additional element to be considered is the nondeterminism of $(\approx_{\{\alpha,\mathbf{C}\}}\ \mathbf{C})$, indeed implemented as two rules. The key theorem states that Sl is a solution for T iff there exists a unification path form T to some T' with solution Sl.

```
Theorem unif_path_compl : ∀ T Sl,
        valid_triple T → (sol_c Sl T ↔ ∃ T', unif_path T T' ∧ sol_c Sl T').
```

Excluding formalisation of nominal terms and E-equivalence, subject of [3], the whole theory consists of theories Completeness, Soundness, Termination, C-Unif, Substs, Problems and C-Equiv, which consist of 5474 lines or 204 KB.

6 Conclusions and Future Work

A Coq formalisation of a sound and complete nominal C-unification algorithm was obtained by combining $\Rightarrow_{\approx}$- and $\Rightarrow_{\#}$-reduction. The algorithm builds finite derivation trees, such that the leaves, which may contain FP equations, represent a complete set of unifiers. We have shown that nominal C-unification is infinitary and NP-complete. An OCaml implementation of the simplification phase has been developed, which outputs derivation trees. Extensions to deal with different equational theories will be considered in future work.

References

1. Aoto, T., Kikuchi, K.: A rule-based procedure for equivariant nominal unification. In: Pre-proceeding of Higher-Order Rewriting (HOR), pp. 1–5 (2016)
2. Aoto, T., Kikuchi, K.: Nominal confluence tool. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 173–182. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_12
3. Ayala-Rincón, M., Carvalho-Segundo, W., Fernández, M., Nantes-Sobrinho, D.: A formalisation of nominal equivalence with associative-commutative function symbols. ENTCS **332**, 21–38 (2017)
4. Ayala-Rincón, M., de Carvalho-Segundo, W., Fernández, M., Nantes-Sobrinho, D.: On solving nominal fixpoint equations. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 209–226. Springer, Cham (2017)
5. Ayala-Rincón, M., Fernández, M., Nantes-Sobrinho, D.: Nominal narrowing. In: Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction (FSCD). LIPIcs, vol. 52, pp. 11:1–11:17 (2016)
6. Ayala-Rincón, M., Fernández, M., Rocha-oliveira, A.C.: Completeness in PVS of a nominal unification algorithm. ENTCS **323**, 57–74 (2016)
7. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge UP, New York (1998)
8. Braibant, T., Pous, D.: Tactics for reasoning modulo AC in Coq. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 167–182. Springer, Heidelberg (2011)
9. Calvès, C.F.: Complexity and implementation of nominal algorithms. Ph.D Thesis, King's College London (2010)
10. Calvès, C.F., Fernández, M.: Implementing nominal unification. ENTCS **176**(1), 25–37 (2007)
11. Calvès, C., Fernández, M.: The first-order nominal link. In: Alpuente, M. (ed.) LOPSTR 2010. LNCS, vol. 6564, pp. 234–248. Springer, Heidelberg (2011)
12. Cheney, J.: αProlog Users Guide & Language Reference Version 0.3 DRAFT (2003)
13. Cheney, J.: Equivariant unification. J. Autom. Reasoning **45**(3), 267–300 (2010)

14. Clouston, R.A., Pitts, A.M.: Nominal equational logic. ENTCS **172**, 223–257 (2007)
15. Contejean, E.: A certified AC matching algorithm. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 70–84. Springer, Heidelberg (2004)
16. Fernández, M., Gabbay, M.J.: Nominal rewriting. Inf. Comput. **205**(6), 917–965 (2007)
17. Fernández, M., Gabbay, M.J.: Closed nominal rewriting and efficiently computable nominal algebra equality. In: Proceedings of the 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP). EPTCS, vol. 34, pp. 37–51 (2010)
18. Fernández, M., Gabbay, M.J., Mackie, I.: Nominal rewriting systems. In: Proceedings of the 6th International Conference on Principles and Practice of Declarative Programming (PPDP), pp. 108–119. ACM Press (2004)
19. Gabbay, M.J., Mathijssen, A.: Nominal (Universal) algebra: equational logic with names and binding. J. Logic Comput. **19**(6), 1455–1508 (2009)
20. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects Comput. **13**(3–5), 341–363 (2002)
21. Kapur, D., Narendran, P.: Matching unification and complexity. SIGSAM Bull. **21**(4), 6–9 (1987)
22. Kumar, R., Norrish, M.: (Nominal) Unification by recursive descent with triangular substitutions. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 51–66. Springer, Heidelberg (2010)
23. Schmidt-Schauß, M., Kutsia, T., Levy, J., Villaret, M.: Nominal unification of higher order expressions with recursive let. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 328–344. Springer, Cham (2017)
24. Levy, J., Villaret, M.: An efficient nominal unification algorithm. In: Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA). LIPIcs, vol. 6, pp. 209–226 (2010)
25. Nipkow, T.: Equational reasoning in Isabelle. Sci. Comput. Program. **12**(2), 123–149 (1989)
26. Pitts, A.M.: Nominal Sets: Names and Symmetry in Computer Science. Cambridge UP, Cambridge (2013)
27. Sagan, B.E.: The Symmetric Group: Representations, Combinatorial Algorithms, and Symmetric Functions, 2nd edn. Springer, New York (2001)
28. Siekmann, J.: Unification of commutative terms. In: Ng, E.W. (ed.) Symbolic and Algebraic Computation. LNCS, vol. 72, pp. 22–29. Springer, Heidelberg (1979). https://doi.org/10.1007/3-540-09519-5_53
29. Urban, C.: Nominal unification revisited. In: Proceedings of the 24th International Workshop on Unification (UNIF). EPTCS, vol. 42, pp. 1–11 (2010)
30. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theor. Comput. Sci. **323**(1–3), 473–497 (2004)