



Universidade de Brasília

**Nominal Unification With Commutative
and Associative-Commutative Function
Symbols**

Gabriel Ferreira Silva

Advisor: Mauricio Ayala-Rincón

Department of Mathematics
University of Brasilia

Dissertation submitted in partial fulfillment of the requirements for the degree
of
Master in Mathematics

August 2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Matemática

Unificação nominal com símbolos comutativos e
associativos comutativos

por

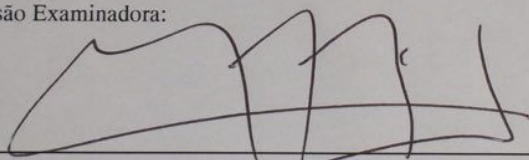
Gabriel Ferreira Silva

*Dissertação apresentada ao Departamento de Matemática da Universidade
de Brasília, como parte dos requisitos para obtenção do grau de*

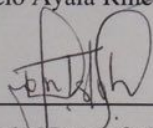
MESTRE EM MATEMÁTICA

Brasília, 26 de julho de 2019.

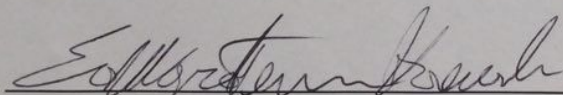
Comissão Examinadora:



Prof. Dr. Mauricio Ayala Rincón - MAT/UnB (Orientador)



Profa. Dra. Daniele Nantes Sobrinho – MAT/UnB (Membro)



Prof. Dr. Edward Hermann Haeusler – PUC/RJ (Membro)

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

Fu Ferreira Silva, Gabriel
Unificação Nominal com Símbolos Comutativos e Associativos
Comutativos / Gabriel Ferreira Silva; orientador Mauricio
Ayala-Rincón. -- Brasília, 2019.
64 p.

Dissertação (Mestrado - Doutorado em Matemática) --
Universidade de Brasília, 2019.

1. Unificação . 2. Sintaxe Nominal. 3. C-Unificação . 4.
AC-Unificação . 5. PVS. I. Ayala-Rincón, Mauricio , orient.
II. Título.

Dedicated to people who contribute to projects like StackExchange and Wikipedia.

Acknowledgements

I thank God for giving me a great family, health and disposition for work.

I thank my family for always being there for me, specially my mom, my dad, my sister and my grandparents. I thank my mates Alexander, Manno and Fugi for our long friendship and also for all the fun we always have while playing videogames. I thank the friends I made while studying in room 408/10 (also known as “Sala Top”) specially Maximal (Deivid) and Edna. I thank Santiago, Thiago and Pavel for the fun dinners we constantly have. I thank all the friends I made during these two years. I thank my friends from soccer, I could not ask for better teammates. In short, I thank my family and my friends, for the constant support and for the happy moments I have: you make life worth living!

I thank Mauricio Ayala-Rincón for supervising me and for accepting doing so even after I failed the analysis exam. I thank Daniele Nantes Sobrinho and Maribel Fernández for working with me and Ayala in the papers we have published. I thank the committee for the helpful feedback given.

I thank the employees, students and professors of the departments of computer science and mathematics, for the cordial environment I have encountered while doing my studies.

I thank the CNPq for the scholarship I received.

Abstract

The nominal approach allows us to extend first-order syntax and represent smoothly systems with variable bindings. Hence, nominal unification is simply the extension of first-order unification modulo α -equivalence by taking into account this nominal setting. This work is about nominal unification with commutative function symbols (nominal C-unification, for short) and nominal unification with associative-commutative function symbols (nominal AC-unification, for short). Regarding nominal C-unification, we specified a functional algorithm for this task and proved the algorithm is indeed correct and complete with the use of the proof assistant PVS. In relation to the only known specification of nominal C-unification, there are two novelties in our work. The first is the formalisation of a functional algorithm that can be directly executed (not just a set of non-deterministic inference rules). The second is simpler proofs of termination, soundness and completeness, due to the reduction in the number of parameters of the lexicographic measure, from four parameters to only two. Additionally, we programmed and tested the algorithm in Python. Regarding nominal AC-unification, we discuss our work in progress in extending the algorithm of nominal C-unification to also handle associative-commutative function symbols. The specification we currently have is presented, the strategy for the formalisation is detailed, the current problem we are facing is explained and interesting facts found so far are given.

Resumo

O paradigma nominal permite estender a sintaxe de primeira ordem, representando convenientemente o conceito de variáveis ligadas. Assim sendo, unificação nominal é simplesmente uma extensão da unificação de primeira ordem, considerando a configuração nominal e igualdade módulo α -equivalência. Este trabalho é acerca de unificação nominal na presença de símbolos de função comutativos (C-unificação nominal) e associativo-comutativos (AC-unificação nominal). Em relação à C-unificação nominal, especificamos um algoritmo funcional e provamos que o algoritmo é correto e completo com o uso do assistente de provas PVS. Comparado com a única outra especificação de C-unificação nominal, há duas inovações neste trabalho. A primeira inovação é a formalização de um algoritmo funcional que pode ser diretamente executado (não apenas um conjunto de regras de inferência não-determinísticas). A segunda inovação são provas mais simples de terminação, corretude e completude, graças à redução no número de parâmetros da medida lexicográfica, de quatro parâmetros para apenas dois. Adicionalmente, programamos e implementamos o algoritmo em Python. Em relação à AC-unificação nominal, discutimos o nosso trabalho em progresso visando estender o algoritmo de C-unificação nominal para também tratar o caso de símbolos de funções associativo-comutativos. A especificação atual é apresentada, a estratégia para a demonstração é detalhada, o problema atual que estamos enfrentando é explicado e descobertas interessantes feitas até aqui são mostradas.

Table of contents

1	Introduction	1
1.1	Related Work	2
1.2	Contribution	3
1.3	Organisation	4
2	Preliminaries	5
2.1	The Proof Assistant PVS	5
2.1.1	Specifications in PVS	6
2.1.2	Formalisations in PVS	7
2.2	First-Order Syntax and Unification	8
2.2.1	First-Order Syntax	8
2.2.2	Syntactic First-Order Unification	10
2.2.3	First-Order Unification Modulo C and AC	11
2.3	The Nominal Setting	14
2.3.1	Nominal Terms, Permutations and Substitutions	14
2.3.2	Freshness and α -equality	16
2.4	Extending the Nominal Setting to Handle Commutative and Associative-Commutative Function Symbols	18
2.4.1	Extending Nominal Terms, Permutations and Substitutions	18
2.4.2	Extending Freshness and α -equality	19
2.4.3	Nominal C and AC-Unification	20
3	A Correct and Complete Algorithm for Functional Nominal C-Unification	23
3.1	The PVS Theory Organisation and Where to Find Further Information	23
3.2	Specification	25
3.2.1	Auxiliary Functions	28
3.3	Examples	28
3.4	Formalisation	30

3.4.1	The Lexicographic Measure and Termination of the Algorithm	30
3.4.2	Soundness and Completeness	31
3.4.3	Interesting Points	33
3.5	Implementation	35
3.6	Possible Applications	38
4	Formalising Nominal AC-Unification	41
4.1	Hierarchy and Organisation of the PVS theory	41
4.2	Current Specification	43
4.3	Our Work in Progress on the Formalisation	44
4.3.1	Main Theorems That Must Be Proved	44
4.3.2	A Plan to the Proof	46
4.3.3	A Loose End	47
4.3.4	Interesting Discoveries	48
4.4	Possible Applications	48
5	Conclusion and Future Work	49
5.1	Conclusion	49
5.2	Future Work	49
	References	51
	Appendix A The PVS Theory for Nominal C-Unification	55
A.1	Fragments of the Specification	55
A.1.1	Defining a Nominal Term in PVS	55
A.1.2	Specification of the Main Functions in the Lexicographic Measure	56
A.2	A Fragment of the Formalisation	57
	Appendix B Fragments of The Python Code For Nominal C-Unification	61
B.1	The Classes	61
B.2	Fragments of the Unify Function	62

Chapter 1

Introduction

The concept of binding is crucial in both mathematics and computer science. We employ this concept, for instance, when specifying parameters in the definition of a function: in $f : x \mapsto x^2$ the variable x is said to be bound. There is also binding when using quantifiers: in $\forall x : P(x, y)$, where P is some property of interest, the variable x is bound, while the variable y is not bound (alternatively, one can say that y is a free variable).

Since first-order syntax does not handle binding, extensions of it that consider bound and free variables are attractive areas of work. This extension is not trivial, since it is possible to have syntactically different expressions with the same semantics. For example, the functions $f : x \mapsto x^2$ and $f : y \mapsto y^2$ are semantically equal, but not syntactically equal. They can, however, be made syntactically equal by simply renaming y to x , for instance. The expressions $\exists x : x > 0$ and $\exists z : z > 0$ are another example of semantically equal expressions that can be made syntactically equal by renaming: simply substitute the z in the second expression by x . The notion of semantically equivalent expressions that become syntactically equal after a suitable renaming of bound variables is known as α -equivalence.

The nominal setting takes α -equivalence into account and represent smoothly systems with bindings. This has advantages over using indices as in explicit substitutions *à la de Bruijn*, for the bindings are represented in a more human-friendly way - see Pitts ([Pit13]) or Fernández and Gabbay ([FG07]).

On the other hand, the problem of unification is concerned with finding a “way” to make two terms equal. For instance, the terms $f(X, b)$ and $f(a, Y)$ can be made equal by “sending” X to a and Y to b , since both terms then become $f(a, b)$. This “way” of making terms equal consists of substituting certain expressions in a term (known as variables) with other expressions. In the example given, the variables of the problem were X and Y , and they were substituted respectively by a and b . In a problem of unification, it is clear what expressions are function symbols, which ones are variables and so on.

We now give a more practical, although more intricate, example. Imagine we are trying to calculate $\int \ln(x)dx$ and we remember the formula of integration by parts: $\int u dv = uv - \int v du$. In order to use this strategy, we must unify $\int \ln(x)dx$ with $\int u dv$. This is done by substituting the variable u by $\ln(x)$ and v by x . Then, we get:

$$\int \ln(x) dx = x \ln(x) - \int x * \frac{1}{x} dx = x \ln(x) - x + C \quad (1.1)$$

It should be noted, however, that in this preceding case only one of the terms contained variables that were affected by the substitution. This is a particular case of unification known as matching.

Much work has been done in unification (e.g. [For87], [KKN85], [Baa89]). Unification is an important topic in first-order theories, since it has applications in logic programming systems, theorem provers, type inference algorithms and so on (see Baader and Nipkow [BN99]). Given this importance, the development of unification techniques for the nominal paradigm has been an attractive area of research since the invention of the nominal approach.

Nominal unification was first solved by Urban et al. in [Urb08]. Then, research continued in the direction of making algorithm improvements to solve this problem (see the work [CF08] of Calves and Fernández who obtained an $O(n^4 \log^2 n)$ bound and the work [LV10] of Levy and Villaret, who obtained an $O(n^2)$ bound) and in the direction of extending nominal unification (see the work of Levy and Villaret [LV08], Schmidt-Schauß et al. [SSKLV17], or Baumgartner et al. [BKL15]), for instance by considering nominal unification modulo equational theories (e.g. [ARdCSFNS18], [ARdCSFNS17b], [ARFNS18], [ARdCSFNS17a]). This work is in the area of nominal unification modulo equational theories, since we discuss nominal unification with commutative function symbols (nominal C-unification, for short) and nominal unification with associative-commutative function symbols (nominal AC-unification, for short).

1.1 Related Work

Nominal unification was originally solved by Urban et al. in [Urb08], by proposing a set of inference rules and showing, with the help of proof assistant Isabelle/HOL, that they were correct and complete.

In [ARFRO16], Ayala-Rincón et al. came back to the problem of nominal unification and, using the proof assistant PVS, proposed a functional algorithm for the task and formalised its soundness and completeness. Moreover, two novel approaches were given. The first is the specification of a functional algorithm for performing nominal unification, not a set of inference rules, which makes the specification closer to the implementation. The second is the possibility

of separating the treatment of freshness constraints from equational constraints (these concepts will be defined in Chapter 2). This research uses both approaches.

After that, Ayala-Rincón et al. (see [ARdCSFNS18]) extended nominal unification to deal with commutative functions. As in [Urb08], a set of inference rules specified through an inductive definition was used. These rules were shown correct and complete with the use of the Coq proof assistant. An implementation in OCaml was made, but it diverges from the non-deterministic specification proved to be correct and complete.

In relation to AC-unification, there are no formalisations (to the best of our knowledge) available, neither in the nominal setting nor in first-order theory. The closest that has been done is the formalisation of AC-Matching in standard first-order syntax (see the work of Contejean [Con04]).

1.2 Contribution

This work presents contributions to both nominal C-unification and nominal AC-unification.

In relation to nominal C-unification, we present the first (to our knowledge) functional nominal C-unification algorithm and, using PVS, formalise its soundness and completeness. Even though there is one other formalisation for nominal C-unification, the approaches taken are different. In [ARdCSFNS18], a set of rules that progressively transforms the nominal C-unification problem into a simpler one is presented, while here a recursive algorithm is indeed developed. The set of rules approach is elegant and allows a higher level of abstraction, thus simplifying the analysis of the computational properties we are interested. However, due to its non-determinism, the set of rules cannot be directly executed, while the recursive algorithm here given can. This MSc thesis also presents a simplification on the proofs of termination, soundness and completeness from the ones found in [ARdCSFNS18], since we were able to reduce the number of parameters in the lexicographic measure, from four parameters to only two (see Section 3.4). Additionally, the described algorithm has been implemented in Python.

The formalisation is fully available at <http://www.github.com/gabriel951/c-unification>. PVS was chosen partly in order to reuse a significant portion of the definitions and lemmas from [ARFRO16] and partly because it provides great support for formalising functionally recursive algorithms.

In relation to AC-unification, we discuss our work in progress on extending the algorithm to deal with associative-commutative function symbols. We present the specification and point interesting aspects of the formalisation, which is not yet completed. If completed, it would be (to the best of our knowledge) the first formalisation of nominal unification and

of first-order unification. The work in progress is fully available at <http://www.github.com/gabriel951/ac-unification>.

1.3 Organisation

The chapters of this MSc thesis are organised as follows.

- First, in Chapter 2, we provide the necessary background for understanding the research. The mathematical foundations of the proof assistant PVS are discussed, first-order theory and nominal theory are explained and the problem of nominal unification with commutative and associative-commutative function symbols is defined.
- Then, in Chapter 3, we begin by presenting the hierarchy of the PVS files for nominal C-unification. Next, we give the specification for the algorithm that performs nominal C-unification. We comment on the most interesting points of the formalisation: how introducing commutativity made the problem harder, the principal lemmas and the hardest cases. Finally, we discuss our Python implementation of the algorithm and present possible applications.
- In Chapter 4, we comment on what was added in the specification for the algorithm to handle associative-commutative function symbols. Then, we discuss on interesting points of the partial formalisation that we have.
- Finally, in Chapter 5, we conclude the MSc thesis and offer some possible suggestions of future work.

This MSc thesis also contains two appendices:

- Appendix A presents fragments of the specification and the formalisation and discusses them.
- Appendix B comments on fragments of the Python code for nominal C-unification.

Chapter 2

Preliminaries

In this chapter we delve into the preliminaries for understanding the research. We comment on the proof assistant PVS that was used. Next, we explain first-order theory. Then, we introduce the nominal setting. Finally, we comment on how to extend this nominal setting to take into account commutative and associative-commutative function symbols and define the problem of nominal C and AC-Unification.

2.1 The Proof Assistant PVS

A proof assistant, also called interactive theorem prover, is a software used to help humans with the development of formal proofs. A verified proof has every step of it accepted as correct by the proof assistant, thus diminishing the probability of wrong proofs. However, wrong proofs can still occur, for instance if a given mathematical theory is wrongly defined in the proof assistant. Examples of interactive theorem provers are: Coq, Isabelle/Hol, PVS, Lean and so on (see [BBC⁺97], [NPW02], [OSRSC01], [dMKA⁺15]).

PVS is a system for specifying and verifying properties of software systems [OS99]. Specifications are kept in `.pvs` files while the corresponding formalisations are found in `.prf` files.

As remarked before, there were a number of different proof assistants we could have chosen, and we opted for PVS for two different reasons. The first was to reuse a great portion of the definitions and lemmas from [ARFRO16], instead of proving them from scratch (it should come as no surprise that nominal unification modulo commutativity and modulo associativity-commutativity has a lot in common with syntactic nominal unification). The second is the support offered by PVS to specify functional algorithms.

2.1.1 Specifications in PVS

PVS provides a functional language to specify the properties we want to verify. Examples 2.1 and 2.2 illustrate this.

Example 2.1 (Adapted from [ARDM17]). The specification of function `gcd` (which computes the greatest common divisor of two number m and n) is given below:

```
gcd(m, n): RECURSIVE nat =
  IF n = 0
    THEN m
  ELSIF m > n
    THEN gcd(n, m)
  ELSE
    gcd(m, n - m)
  ENDIF
MEASURE lex2(m, n)
```

In this particular specification, the variable m was previously defined to be a positive natural number via the fragment (not shown in the figure) `m:VAR posnat` while the variable n was previously defined as a natural number via fragment (not shown in the figure) `n:VAR nat`.

Since variables in PVS have a type (for instance, in Example 2.1, the variable n has type natural number), one of the steps of using PVS is type checking an specification. Type checking generates Type Correctness Conditions (for short TCCs) which must be proved in order for a function to be considered well-defined. TCCs can be related with successfully preserving a type or with the termination of a function, among others. In Example 2.1, TCCs would be generated forcing us to guarantee that, in every recursive call of `gcd`, the first parameter of the function is always a positive natural number and the second parameter is always a natural number. This can be easily verified.

A termination TCC would also be generated requiring us to prove that the function `gcd` really terminates. This is done by defining a measure and proving that in every recursive call to `gcd` the defined measure decreases. In the specification above, `MEASURE lex2(m, n)` defines the measure that will be used and `lex2` is a built-in function in PVS that implements a lexicographic order with 2 parameters. Therefore, for $lex2(x, y) < lex2(w, z)$ we must have $x < w$ or $x = w$ and $y < z$. This, again, can be easily verified to hold in the given example.

Example 2.2 (Adapted from [SORSC01]). The function `sum` receives a function f and a natural number n . The value of $sum(f, n)$ is computed recursively and corresponds to $f(1) + \dots + f(n-1)$.

```

sum: THEORY
  BEGIN

  n: VAR nat
  f, g: VAR [nat -> nat]

  sum(f, n): RECURSIVE nat =
  IF n = 0
    THEN 0
  ELSE f(n-1) + sum(f, n-1)
  ENDIF
  MEASURE n

  sum_plus: LEMMA
    sum((lambda n: f(n) + g(n)), n)
  = sum(f, n) + sum(g, n)

END sum

```

Furthermore, the `sum` theory given specifies a lemma, `sum_plus`, which says that $sum(f + g, n) = sum(f, n) + sum(g, n)$. We will see how to prove this lemma in Section 2.1.2.

2.1.2 Formalisations in PVS

PVS uses a sequent calculus with every sequent of the form $\Gamma \vdash \Delta$, where Γ is the set of antecedent formulas and Δ is the set of consequent formulas. The meaning of this sequent is that by using all formulas in Γ we can derive at least one of the formulas in Δ . Thus, the deductive system used by PVS is a Gentzen systems [TS00]. Inference rules are of the form:

$$\frac{\text{Premises}}{\text{Conclusion}} \text{ (Rule Name)}$$

Example 2.3 (Adapted from [ARDM17]). Suppose we have $\Gamma \vdash \Delta$, where $\Gamma = \{\Gamma_1, \dots, \Gamma_n\}$ and $\Delta = \{\Delta_1, \dots, \Delta_n\}$. The premises would be separated by the conclusion by the symbols $|--$. Premises and conclusions are respectively labeled with negative and positive numbers, as shown below.

$$\begin{array}{l}
 [-1] \Gamma_1 \\
 \cdot \\
 \cdot \\
 \cdot \\
 [-n] \Gamma_n \\
 | \text{---} \\
 [1] \Delta_1 \\
 \cdot \\
 \cdot \\
 \cdot \\
 [-n] \Delta_n
 \end{array}$$

Example 2.4 (Adapted from [SORSC01]). In Example 2.2 the sum function was specified, along with a lemma. By using the command `induct-and-rewrite!` with parameter "n", PVS applies induction in n and then completes the proof by expanding the functions used in the theorem and applying convenient instantiation when necessary. Figure 2.1 shows the result of using this command.

2.2 First-Order Syntax and Unification

The definitions from this section come from [BN99] and [dCS19].

2.2.1 First-Order Syntax

In order to define first-order terms (Definition 2.2), first we must define a signature (Definition 2.1).

Definition 2.1. A signature Σ is a countable set of function symbols f with a given arity. Each function symbol may have associated to it equational properties, and a function symbol that has no equational property is said to be syntactic. A function symbol of arity 0 is called a constant.

Remark 2.1. In this work, the equational properties we are interested in are commutativity (from now on, also denoted with the symbol C) and associativity-commutativity (from now on, also denoted with the symbol AC).

Definition 2.2 (First-Order Term). Let Σ be a signature and \mathcal{V} be a countable set of variables. The set of first-order terms $\mathcal{T}(\Sigma, \mathcal{V})$ has its terms generated according to the grammar:

$$t := c \mid X \mid f(t_1, \dots, t_n)$$

Fig. 2.1 Example of a Proof in PVS

```

sum_plus :
  |-----
  {1} (FORALL (f: [nat -> nat], g: [nat -> nat], n: nat):
      sum((LAMBDA (n: nat): f(n) + g(n)), n) = sum(f, n) + sum(g, n))

Rule? (induct-and-rewrite! "n")
sum rewrites sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 0)
  to 0
sum rewrites sum(f!1, 0)
  to 0
sum rewrites sum(g!1, 0)
  to 0
sum rewrites sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 1 + j!1)
  to f!1(j!1) + g!1(j!1) + sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1)
sum rewrites sum(f!1, 1 + j!1)
  to f!1(j!1) + sum(f!1, j!1)
sum rewrites sum(g!1, 1 + j!1)
  to g!1(j!1) + sum(g!1, j!1)
By induction on n and rewriting,
Q.E.D.

Run time = 0.85 secs.
Real time = 6.47 secs.

```

where $c \in \Sigma$ is a constant, $X \in \mathcal{V}$ is a variable and $f \in \Sigma$ is a function symbol with arity $n > 0$.

Example 2.5. Examples of terms are c , X , $f(X, f(a, c))$ and so on.

Relevant concepts of the first-order syntax are substitutions (Definition 2.3), the size of a term (Definition 3.2) and the variable set of a term (Definition 2.5).

Definition 2.3 (Substitution). A substitution σ is a mapping from variables to terms such that $\{X | X \neq X\sigma\}$ is finite.

- The set $\{X | X \neq X\sigma\}$ is called the domain of the substitution, and is represented by $dom(\sigma)$.
- If $X \in dom(\sigma)$, the image of X by σ is the term $X\sigma$.

Remark 2.2. If $dom(\sigma) = \{X_1, \dots, X_n\}$, then σ can be represented as $\{X_1/t_1, \dots, X_n/t_n\}$ or as $\{X_1 \rightarrow t_1, \dots, X_n \rightarrow t_n\}$, where $X_i\sigma = t_i$, for $1 \leq i \leq n$.

Definition 2.4 (Size of a Term). The size of a term t , represented as $|t|$, is defined recursively:

- $|c| = 1$
- $|X| = 1$
- $f(t_1, \dots, t_n) = 1 + \sum_{1 \leq i \leq n} |t_i|$

Definition 2.5 (Variables of a Term). The set of variables of a term t , represented as $Vars(t)$, is defined recursively:

- $Vars(c) = \emptyset$
- $Vars(X) = \{X\}$
- $Vars(f(t_1, \dots, t_n)) = \bigcup_{1 \leq i \leq n} Vars(t_i)$

2.2.2 Syntactic First-Order Unification

As remarked before, unification is about finding a substitution σ that makes two terms t and s equal. In syntactic first-order unification, this equality is seen as syntactical equality. The problem of syntactic first-order unification is defined in Definition 2.6, the solution for a given problem is defined in Definition 2.7 and an example is provided in Example 2.7.

Definition 2.6. A first-order problem P is a finite set of equations of the form $\{s_1 \approx_{\gamma} t_1, \dots, s_n \approx_{\gamma} t_n\}$ where $s_i, t_i \in \mathcal{T}(\Sigma, \mathcal{V})$, for $i = 1, \dots, n$. A syntactic first-order problem is a first-order problem such that all terms from the set of equations are syntactic first-order terms.

Definition 2.7. Let P be a first-order unification problem. A substitution σ is a first-order unification solution for P if for each equation $s \approx_{\gamma} t$ in P it is true that $s\sigma \approx t\sigma$.

Remark 2.3. In syntactic unification, the meaning of $s \approx t$ is that s and t are syntactically equal. In unification modulo equational theories, the meaning of $s \approx t$ is that s and t are semantically equal according to the equational theory we are considering.

Definition 2.8. A substitution σ is more general than a substitution δ if there is a substitution θ such that $\delta = \theta\sigma$. In this case, δ is said to be an instance of σ .

Example 2.6. The substitution $\sigma = \{Y \rightarrow f(X)\}$ is more general than $\delta = \{Y \rightarrow f(a), X \rightarrow a\}$, since with $\theta = \{X \rightarrow a\}$ we have $\delta = \theta\sigma$.

Very close to the notion of unification is the concept of matching, given in Definition 2.9.

Definition 2.9. A substitution σ is a first-order matching solution for P if for each equation $s \approx_{\gamma} t$ in P it is true that $s\sigma \approx t$.

The idea is that the substitution σ is only applied to one side of the equation. If a term t does not contain variables, $t\sigma$ is simply t and, in this case, the problem of matching $s \approx_{\gamma} t$ is the same as the problem of unifying $s \approx_{\gamma} t$.

Example 2.7. We formalise one of the examples given in the introduction. Suppose X, Y are variables, a, b are constants and $f \in \Sigma$ is a binary function. For the unification problem $P = \{f(X, b) \approx_{\gamma} f(a, Y)\}$, a possible solution is $\sigma = \{X/a, Y/b\}$, since $f(X, b)\sigma = f(a, Y)\sigma = f(a, b)$.

2.2.3 First-Order Unification Modulo C and AC

When we consider the semantics of commutative function symbols, we have unification modulo commutativity (or modulo C, for short). In a similar manner, when we consider the semantics of associative-commutative function symbols, we have unification modulo associativity-commutativity (or modulo AC, for short). The properties of associativity and commutativity are:

- Associativity (A): $f(f(X, Y), Z) \approx f(X, f(Y, Z))$
- Commutativity (C): $f(X, Y) \approx f(Y, X)$

Remark 2.4. When we have associativity, terms such as $f(f(X, Y), Z)$ and $f(X, f(Y, Z))$ are considered equivalent and thus can be represented as $f(X, Y, Z)$. This is called a flattened representation of the term. When we opt for this representation, it is possible to unify associative and associative-commutative function applications rooted by the same function symbol but with different arities after flattened. For instance, the terms $f(f(a, b), c)$ and $f(X, c)$ would be represented as $f(a, b, c)$ and $f(X, c)$ and could be unified by the substitution $\sigma = \{X \rightarrow f(a, b)\}$ even though their arities after flattening (3 and 2 respectively) are different.

Example 2.8 (Adapted from [Sti81] and [dCS19]). We provide an example of the algorithm given by Stickel in [Sti81] to perform the task of first-order AC unification. Let $P = \{f(X, X, Y, a, b, c) \approx_{\gamma} f(b, b, b, c, Z)\}$ be our unification problem, where f is an AC function symbol. Here we opt for the flattened representation of terms, which is possible since we have associativity.

The first step is to eliminate arguments that appear simultaneously on $f(X, X, Y, a, b, c)$ and $f(b, b, b, c, Z)$. Therefore, P is now transformed in $P' = \{f(X, X, Y, a) \approx_{\gamma} f(b, b, Z)\}$.

The second step is to transform our current problem P' into a new one, by replacing constants and subterms headed by different function symbols with new variables. The result is then $P'' = \{f(X, X, Y, W_0) \approx_{\gamma} f(W_1, W_1, Z)\}$.

The third step is to transform the unification problem into a diophantine equation by observing the number of occurrences of each variable. In our case, the diophantine equation obtained would be: $2X + Y + W_0 = 2W_1 + Z$. The substitutions can be represented by non-negative integer solutions to this equation.

The fourth step is to generate the solutions to the system of diophantine equations, as in Table 2.1.

Table 2.1 Solutions for the Equation $2X + Y + W_0 = 2W_1 + Z$

X	Y	W_0	W_1	Z	$2X + Y + W_0$	$2W_1 + Z$	New Variables
0	0	1	0	1	1	1	Z_1
0	1	0	0	1	1	1	Z_2
0	0	2	1	0	2	2	Z_3
0	1	1	1	0	2	2	Z_4
0	2	0	1	0	2	2	Z_5
1	0	0	0	2	2	2	Z_6
1	0	0	1	0	2	2	Z_7

In the fifth step, new variables are associated with every solution (see the last column of Table 2.1). This new variables Z_1, \dots, Z_7 can be interpreted as a “basis” of solutions. Observing the columns of the Table 2.1 one obtains:

$$X = Z_6 + Z_7$$

$$Y = Z_2 + Z_4 + 2Z_5$$

$$W_0 = Z_1 + 2Z_3 + Z_4$$

$$W_1 = Z_3 + Z_4 + Z_5 + Z_7$$

$$Z = Z_1 + Z_2 + 2Z_6$$

Each one of the variables Z_1, \dots, Z_7 can be included or not in the solution, giving us $2^7 = 128$ possible cases. However, since all of X, Y, W_0, W_1, Z must be different than zero, this is restricted to 69 cases. More cases should be dropped: the cases where the variables that in fact represent constants and subterms headed by a different AC function symbol are assigned to more than one of the variables Z_1, \dots, Z_7 . For instance, the solution

$$\sigma_{wrong} = \{X \rightarrow Z_6, Y \rightarrow Z_4, W_0 \rightarrow f(Z_1, Z_4), W_1 \rightarrow Z_4, Z \rightarrow f(Z_1, Z_6, Z_6)\}$$

should be discarded as the variable W_0 , which represents the constant a , must not be assigned to $f(Z_1, Z_4)$. This is the sixth step. After this step, in our example, there are only six remaining cases:

$$\sigma_1 = \{X \rightarrow Z_6, Y \rightarrow Z_4, W_0 \rightarrow Z_4, W_1 \rightarrow Z_4, Z \rightarrow f(Z_6, Z_6)\}$$

$$\sigma_2 = \{X \rightarrow Z_6, Y \rightarrow f(Z_2, Z_4), W_0 \rightarrow Z_4, W_1 \rightarrow Z_4, Z \rightarrow f(Z_2, Z_6, Z_6)\}$$

$$\sigma_3 = \{X \rightarrow Z_6, Y \rightarrow f(Z_5, Z_5), W_0 \rightarrow Z_1, W_1 \rightarrow Z_5, Z \rightarrow f(Z_1, Z_6, Z_6)\}$$

$$\sigma_4 = \{X \rightarrow Z_6, Y \rightarrow f(Z_2, Z_5, Z_5), W_0 \rightarrow Z_1, W_1 \rightarrow Z_5, Z \rightarrow f(Z_1, Z_2, Z_6, Z_6)\}$$

$$\sigma_5 = \{X \rightarrow Z_7, Y \rightarrow Z_2, W_0 \rightarrow Z_1, W_1 \rightarrow Z_7, Z \rightarrow f(Z_1, Z_2)\}$$

$$\sigma_6 = \{X \rightarrow f(Z_6, Z_7), Y \rightarrow Z_2, W_0 \rightarrow Z_1, W_1 \rightarrow Z_7, Z \rightarrow f(Z_1, Z_2, Z_6, Z_6)\}$$

In the seventh step, the variables introduced in step two are replaced by the original term they substituted. This can cause solutions to be lost. For instance, in our example, σ_0 and σ_1 should be discarded, as W_0 and W_1 must be mapped to distinct constants (respectively, a and b).

Finally, in the eighth step, the remaining cases are normalised by replacing the variables in the image of the substitution that also happen in the domain. The final result is the following list of substitutions:

$$\left\{ \begin{array}{l} \{Y \rightarrow f(b, b), Z \rightarrow f(a, X, X)\} \\ \{Y \rightarrow f(Z_2, b, b), Z \rightarrow f(a, Z_2, X, X)\} \\ \{X \rightarrow b, Z \rightarrow f(a, Y)\} \\ \{X \rightarrow f(Z_6, b), Z \rightarrow f(a, Y, Z_6, Z_6)\} \end{array} \right\}$$

This algorithm has been proved terminating, sound and complete (see [Sti81], [Fag87] and [dCS19]).

The Table 2.2, from [dCS19], summarises interesting information about first-order unification, whether syntactic, commutative or associative-commutative. An unification type of 1 means that only one most general solution is generated, while an unification type of ω means that a finite number of most general solutions are generated.

Table 2.2 Information Regarding First-Order Unification

Equational Theory	Unification Type	Complexity Matching	Complexity Unification	Related Work
Syntactic	1	$O(n)$	$O(n)$	[BN99], [MM82]
C	ω	NP-complete	NP-complete	[BN99], [KN87]
AC	ω	NP-complete	NP-complete	[BN99], [KN92]

2.3 The Nominal Setting

The definitions and notations used in this section are based on [ARFFSNS19] and [FG07].

2.3.1 Nominal Terms, Permutations and Substitutions

In the nominal setting, we consider a countable set of atoms $\mathcal{A} = \{a, b, c, \dots\}$. Atoms represent object level variables, and therefore, can be abstracted but not substituted. We also consider a countable set of variables $\mathcal{X} = \{X, Y, Z, \dots\}$ and impose that \mathcal{A} and \mathcal{X} are disjoint. These variables represent meta-level variables and can be substituted, but not abstracted.

Remark 2.5. In the nominal setting, atoms with different names are considered different. For instance, if we consider atoms a and b , it is needless to say that $a \neq b$. This is sometimes called Gabbay's permutative convention.

Renaming of atoms happens through permutations, where a permutation π is a bijection of the form $\pi : \mathcal{A} \rightarrow \mathcal{A}$ such that the set of atoms that are modified by π (also called the domain of π) is finite. Permutations are usually represented as list of swappings, where a swapping $(a b)$ renames a to b and b to a , while leaving all the other atoms fixed. Therefore, a permutation is represented as $\pi = (a_n b_n) :: \dots :: (a_1 b_1) :: nil$.

Definition 2.10. The action of a permutation over an atom is recursively defined as:

$$\begin{aligned}
 nil \cdot c &= c \\
 ((a b) :: \pi) \cdot c &= \begin{cases} a & \text{if } \pi \cdot c = b \\ b & \text{if } \pi \cdot c = a \\ \pi \cdot c & \text{otherwise} \end{cases}
 \end{aligned}$$

Finally, the reverse of a permutation π is denoted by π^{-1} and consists in the reverse of the list of swappings.

With the concepts of atoms, variables and permutations, we are able to define nominal terms:

Definition 2.11 (Nominal Terms). Let Σ be a signature. The set $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{X})$ of nominal terms is generated according to the grammar:

$$s, t ::= \langle \rangle \mid a \mid \pi \cdot X \mid [a]t \mid \langle s, t \rangle \mid ft$$

$\langle \rangle$ is the unit. a is an atom. $\pi \cdot X$ is a suspended variable (the permutation π is said to be suspended on the variable X). $[a]t$ is an abstraction, which allows us to capture the idea of a bound variable on a term (in this case, the atom a is bound on the term t), and the atom a is said to be abstracted in t . Finally, $\langle s, t \rangle$ is a pair and ft is a function application.

Remark 2.6. Although the function application has arity one, this is not a limitation, for we can use the pair to encode tuples with an arbitrary number of arguments. For example, the tuple (t_1, t_2, t_3) could be constructed as $\langle t_1, \langle t_2, t_3 \rangle \rangle$.

Permutations also act on terms:

Definition 2.12 (Permutation Action). The action of a permutation on a term is also defined recursively:

$$\begin{aligned} \pi \cdot \langle \rangle &= \langle \rangle & \pi \cdot (\pi' \cdot X) &= (\pi :: \pi') \cdot X \\ \pi \cdot [a]t &= [\pi \cdot a]\pi \cdot t & \pi \cdot \langle s, t \rangle &= \langle \pi \cdot s, \pi \cdot t \rangle \\ \pi \cdot ft &= f\pi \cdot t \end{aligned}$$

Remark 2.7. Notice that when a permutation π is applied to a suspended variable $\pi' \cdot X$, the permutation π stays suspended. The intuition behind this is that π and π' are waiting for X to be instantiated and will only act when the variable X is instantiated.

Example 2.9. To illustrate the action of a permutation on a term, consider $\pi = (a b) :: (b c) :: (d e) :: \text{nil}$ and $t = f\langle a, \langle d, X \rangle \rangle$. Then, the result of the permutation action is $\pi \cdot t = f\langle b, \langle e, \pi \cdot X \rangle \rangle$.

Finally, a substitution σ is used to instantiate unknowns. Substitutions are represented as lists of nuclear substitutions.

Definition 2.13 (Nuclear Substitution). A nuclear substitution is a pair $[X \rightarrow t]$, where X is a variable and t is a term. Nuclear substitutions also act over terms by induction, as seen below:

$$\langle \rangle[X \rightarrow t] = \langle \rangle$$

$$a[X \rightarrow t] = a$$

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

$$\langle s_1, s_2 \rangle[X \rightarrow t] = \langle s_1[X \rightarrow t], s_2[X \rightarrow t] \rangle \quad (fs)[X \rightarrow t] = f(s[X \rightarrow t])$$

Definition 2.14 (Substitution). A substitution σ is a list of nuclear substitutions $[X_n \rightarrow t_n] :: \dots :: [X_1 \rightarrow t_1] :: Id$, which are applied consecutively to a term:

$$s Id = s, \quad \text{where } Id \text{ is the empty list}$$

$$s(\sigma :: [X \rightarrow t]) = (s[X \rightarrow t])\sigma$$

Remark 2.8. The definition of substitution here presented differs from the traditional view of a substitution as a simultaneous application of nuclear substitutions, being closer to the notion of triangular substitutions [KN10]. Both approaches can correctly represent a substitution [ARFRO16].

Remark 2.9. Substitutions and permutations were defined as lists, and their actions on terms start by the last element in the list and finish with the element in the head of the list. This is advantageous since a composition such as $\pi' \circ \pi$ can be specified in PVS simply as $\text{append}(\pi', \pi)$.

Example 2.10. Let $\sigma = [Y \rightarrow a] :: [X \rightarrow f(Y, b)] :: Id$ and $t = [a]X$. Then, $t\sigma = [a]f(a, b)$.

2.3.2 Freshness and α -equality

Two important notions in the nominal setting are freshness (represented by $\#$) and α -equality (represented by \approx_α):

- $a\#t$ intuitively means that if a occurs in the term t then it does so under an abstractor $[a]$. For example, $a\#b$, since a does not occur in b , and also $a\#[a]a$, since a occurs under an abstractor $[a]$. However, we do not have $a\#a$.
- $s \approx_\alpha t$ means that s and t are α -equivalent, that is, the terms can be made equal by a suitable renaming of bounded atoms. For instance, $[a]a \approx_\alpha [b]b$ but we do not have $a \approx_\alpha b$.

The formal definitions of freshness and α -equivalence are given in Definitions 2.15 and 2.16.

Definition 2.15 (Freshness). A freshness context Δ is a set of constraints of the form $a\#X$. An atom a is said to be fresh on t under a context Δ (which we denote by $\nabla \vdash a\#t$) if it is possible to build a proof using the rules:

$$\begin{array}{c} \frac{}{\Delta \vdash a\#\langle \rangle} \text{(\#}\langle \rangle\text{)} \qquad \frac{}{\Delta \vdash a\#b} \text{(\#atom)} \\ \\ \frac{(\pi^{-1} \cdot a\#X) \in \Delta}{\Delta \vdash a\#\pi \cdot X} \text{(\#X)} \qquad \frac{}{\Delta \vdash a\#[a]t} \text{(\#[a]a)} \\ \\ \frac{\Delta \vdash a\#t}{\Delta \vdash a\#[b]t} \text{(\#[a]b)} \qquad \frac{\Delta \vdash a\#s \quad \Delta \vdash a\#t}{\Delta \vdash a\#\langle s, t \rangle} \text{(\#pair)} \\ \\ \frac{\Delta \vdash a\#t}{\Delta \vdash a\#f t} \text{(\#app)} \end{array}$$

Definition 2.16 (α -equality). Two terms t and s are said to be α -equivalent under the context Δ ($\Delta \vdash t \approx_{\alpha} s$) if it is possible to build a proof using the rules:

$$\begin{array}{c} \frac{}{\Delta \vdash \langle \rangle \approx_{\alpha} \langle \rangle} \text{(\approx}_{\alpha} \langle \rangle\text{)} \qquad \frac{}{\Delta \vdash a \approx_{\alpha} a} \text{(\approx}_{\alpha} \text{atom)} \\ \\ \frac{\Delta \vdash s \approx_{\alpha} t}{\Delta \vdash fs \approx_{\alpha} ft} \text{(\approx}_{\alpha} \text{app)} \qquad \frac{\Delta \vdash s \approx_{\alpha} t}{\Delta \vdash [a]s \approx_{\alpha} [a]t} \text{(\approx}_{\alpha} [a]a\text{)} \\ \\ \frac{\Delta \vdash s \approx_{\alpha} (ab) \cdot t, a\#t}{\Delta \vdash [a]s \approx_{\alpha} [b]t} \text{(\approx}_{\alpha} [a]b\text{)} \qquad \frac{ds(\pi, \pi')\#X \subseteq \Delta}{\Delta \vdash \pi \cdot X \approx_{\alpha} \pi' \cdot X} \text{(\approx}_{\alpha} X\text{)} \\ \\ \frac{\Delta \vdash s_0 \approx_{\alpha} t_0, \Delta \vdash s_1 \approx_{\alpha} t_1}{\Delta \vdash \langle s_0, s_1 \rangle \approx_{\alpha} \langle t_0, t_1 \rangle} \text{(\approx}_{\alpha} \text{pair)} \end{array}$$

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

The derivations rules for freshness and α -equivalence form a sequent calculus, as indicated in Examples 2.11 and 2.12.

Example 2.11. Let's derive $a\#\langle X, [a]Y \rangle$ with context $\Delta = \{a\#X\}$:

$$\frac{a\#X \quad \frac{}{a\#[a]Y} \text{(\#[a]a)}}{a\#\langle X, [a]Y \rangle} \text{(\#pair)}$$

Example 2.12. As commented before, $[a]a \approx_\alpha [b]b$ are α -equivalent. The following derivation proves this:

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

2.4 Extending the Nominal Setting to Handle Commutative and Associative-Commutative Function Symbols

2.4.1 Extending Nominal Terms, Permutations and Substitutions

From now on, we will denote commutative functions by a superscript C and associative-commutative functions by a superscript AC. After explicitly saying that a symbol is C or AC, the superscript may be dropped to simplify the notation.

The definition of nominal term may be extended in order to consider C and AC function symbols, as can be seen in Definition 2.17. The code in PVS to specify a nominal term can be checked on Appendix A, Section A.1.

Definition 2.17 (Nominal Terms with C and AC function symbols). Let Σ be a signature of function symbols, C function symbols and AC function symbols. The set $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{X})$ of nominal terms are given according to the grammar of Definition 2.11 and also the rules:

$$s, t ::= f^C \langle s, t \rangle \mid f^{AC} t$$

Remark 2.10. As in [ARdCSFNS18], we impose that commutative functions always receive a pair as their argument. This was done in order to simplify the analysis, as we do not have to worry about the semantics of a C function symbol with only one argument, or a C function symbol with no arguments. Consider for instance the disjunction operator \vee . We have $a \vee b$ if and only if a or b is true (the case of a C function symbol receiving two arguments), $\vee t$ is true if and only if t is true (the case of a C function symbol receiving only one argument) and $\vee \langle \rangle$ is always false (here the unit is being used to represent the case off a C function symbol receiving no arguments).

It is straightforward to extend the action of permutations and nuclear substitutions to work with C and AC function symbols, as seen in Definitions 2.18 and 2.19. The action of a substitution remains simply a consecutive application of the nuclear substitutions.

Definition 2.18 (Permutation Action with C and AC function symbols). The action of a permutation on a term (Definition 2.12) is extended by the following rules in order to deal with C and AC functions:

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

$$\pi \cdot f^{AC} t = f^{AC} (\pi \cdot t)$$

Definition 2.19 (Nuclear Substitution with C and AC function symbols). The action of a nuclear substitution on a term (Definition 2.12) is extended by the following rules in order to deal with C and AC functions:

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

$$(f^{AC} s)[X \rightarrow t] = f^{AC} (s[X \rightarrow t])$$

2.4.2 Extending Freshness and α -equality

Extending freshness to deal with C and AC functions is straightforward, as seen in Definition 2.20. Extending α -equality (Definition 2.21), by contrast, is not as simple, since the semantics of commutativity and associativity-commutativity play an important role.

Definition 2.20 (Freshness with C and AC function symbols). In addition to the rules already presented in 2.15, when introducing C and AC function symbols, there are the rules:

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

$$\frac{\Delta \vdash a \# t}{\Delta \vdash a \# f^{AC} t} (\#ac - app)$$

Definition 2.21 (α -equality with C and AC function symbols). In addition to the rules already presented in 2.16, when introducing C and AC function symbols, there are the rules:

$$\frac{\Delta \vdash s_0 \approx_\alpha t_i, \Delta \vdash s_1 \approx_\alpha t_{i+1(mod 2)}}{\Delta \vdash f^C \langle s_0, s_1 \rangle \approx_\alpha f^C \langle t_0, t_1 \rangle} \quad i = 0, 1 (\approx_\alpha c - app)$$

$$\frac{\Delta \vdash S_1(f^{AC} s) \approx_\alpha S_i(f^{AC} t) \quad \Delta \vdash D_1(f^{AC} s) \approx_\alpha D_i(f^{AC} t)}{\Delta \vdash f^{AC} s \approx_\alpha f^{AC} t} (\approx_\alpha ac - app)$$

In the rule for an associative-commutative function, $S_n(f^*)$ is an operator that selects the n th argument of the flattened subterm f^* and $D_n(f^*)$ is an operator that deletes the n th argument of the flattened subterm f^* .

Remark 2.11. The rule for commutative function contemplates two possibilities. The first is that in order for $f^C\langle s_0, s_1 \rangle \approx_\alpha f^C\langle t_0, t_1 \rangle$ it is enough to have $s_0 \approx_\alpha t_0$ and $s_1 \approx_\alpha t_1$. The second is that we can also derive $f^C\langle s_0, s_1 \rangle \approx_\alpha f^C\langle t_0, t_1 \rangle$ from $s_0 \approx_\alpha t_1$ and $s_1 \approx_\alpha t_0$, since we have commutativity.

Remark 2.12. If the flattened subterm f^* contains only two arguments, then $D_n(f^*)$ will contain only one argument. Also, if the flattened subterm f^* contains only one argument, then $D_n(f^*)$ returns the unit. For instance, $D_1(f\langle a, b \rangle) = fb$ and $D_1(fb) = \langle \rangle$.

Example 2.13. The following derivation proves that $g(f^C\langle a, b \rangle) \approx_\alpha g(f^C\langle b, a \rangle)$.

$$\frac{\frac{b \approx_\alpha b \quad (\approx_\alpha \text{ atom})}{f^C\langle a, b \rangle \approx_\alpha f^C\langle b, a \rangle} \quad \frac{a \approx_\alpha a \quad (\approx_\alpha \text{ atom})}{(\approx_\alpha c - \text{app})}}{g(f^C\langle a, b \rangle) \approx_\alpha g(f^C\langle b, a \rangle)} \quad (\approx_\alpha \text{ app})$$

Example 2.14. We exemplify the operators $S_n(*)$ and $D_n(*)$. Let f be an AC-function symbol. In the above definition, $S_2(f\langle f\langle a, b \rangle, f\langle [a]X, \pi \cdot Y \rangle \rangle)$ is b , and $D_2(f\langle f\langle a, b \rangle, f\langle [a]X, \pi \cdot Y \rangle \rangle)$ is $f\langle fa, f\langle [a]X, \pi \cdot Y \rangle \rangle$.

Example 2.15. Let f be an AC-function symbol. Notice that $f\langle f\langle a, b \rangle, [a]X \rangle$ and $f\langle [a]X, \langle a, b \rangle \rangle$ are α -equivalent. To see that, apply rule $(\approx_\alpha \text{ ac-app})$ and prove that $S_1(f\langle f\langle a, b \rangle, [a]X \rangle) \approx_\alpha S_2(f\langle [a]X, \langle a, b \rangle \rangle)$ and that $D_1(f\langle f\langle a, b \rangle, [a]X \rangle) \approx_\alpha D_2(f\langle [a]X, \langle a, b \rangle \rangle)$.

We have $S_1(f\langle f\langle a, b \rangle, [a]X \rangle) = a \approx_\alpha a = S_2(f\langle [a]X, \langle a, b \rangle \rangle)$. Finally, since $D_1(f\langle f\langle a, b \rangle, [a]X \rangle) = f\langle b, [a]X \rangle$ and $D_2(f\langle [a]X, \langle a, b \rangle \rangle) = f\langle [a]X, b \rangle$ we have, by another convenient application of rule $(\approx_\alpha \text{ ac-app})$, that $D_1(f\langle f\langle a, b \rangle, [a]X \rangle) \approx_\alpha D_2(f\langle [a]X, \langle a, b \rangle \rangle)$.

2.4.3 Nominal C and AC-Unification

Unification, both in first-order and in nominal syntax, is about discovering a substitution that makes two terms t and s equal. In the absence of equational theories, the only difference is that in first-order theory we consider syntactical equality, while in the nominal approach we consider equality modulo renaming of bound variables, that is, α -equality. By adding C and AC function symbols, we also need to consider equality modulo the semantics of commutativity and associativity-commutativity. We now formally define an unification problem and a solution for a unification problem in the nominal setting. For a comparison with the analogous definitions in the first-order setting, check Definitions 2.6 and 2.7.

Definition 2.22 (Unifiable Terms and Unifiers). Two terms t and s are unifiable if there exists a pair $\langle \nabla, \sigma \rangle$ such that $\nabla \vdash t\sigma \approx_\alpha s\sigma$. The pair $\langle \nabla, \sigma \rangle$ is said to be a unifier of t and s .

Definition 2.23 (Unification Problem with C and AC function symbols). A unification problem is a pair $\langle \Delta, P \rangle$ where Δ is a freshness context and P is a finite set of equations and freshness constraints of the form $s \approx_\gamma t$ and $a \#_\gamma t$, respectively, with s and t terms and a atom.

Example 2.16. Let f be an AC function symbol. A unification problem with empty context and one equation constraint is: $\langle \Delta, P \rangle = \langle \emptyset, f\langle f\langle X, Y \rangle, c \rangle \approx_\gamma f\langle c, f\langle a, b \rangle \rangle \rangle$.

Let ∇ and ∇' be freshness contexts and σ and σ' be substitutions. Before defining a solution to a unification problem, we need the following notation:

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

Definition 2.24 (Solution for a Triple or Problem). A solution for a unification problem $\langle \Delta, P \rangle$ is a solution for the associated triple $\langle \Delta, id, P \rangle$, where id is the identity substitution. We define a solution for a triple $\mathcal{P} = \langle \Delta, \delta, P \rangle$ (where δ is a substitution) as a pair $\langle \nabla, \sigma \rangle$ that fulfills the following four conditions:

- $\nabla \vdash \Delta \sigma$
- $\nabla \vdash a \# t \sigma$, if $a \#_\gamma t \in P$
- $\nabla \vdash s \sigma \approx_\alpha t \sigma$, if $s \approx_\gamma t \in P$
- There exists λ such that $\nabla \vdash \delta \lambda \approx \sigma$

Example 2.17. Consider the unification problem of Example 2.16 and remember that f is an AC function symbol. We have $\langle \Delta, id, P \rangle = \langle \emptyset, id, f\langle f\langle X, Y \rangle, c \rangle \approx_\gamma f\langle c, f\langle a, b \rangle \rangle \rangle$. A possible solution is: $\langle \emptyset, [X \rightarrow a] :: [Y \rightarrow b] :: Id \rangle$.

Definition 2.25. A fixed point equation is an equation of the form $\pi \cdot X \approx_\alpha \pi' \cdot X$.

Fixed point equations are not central to standard nominal unification, but they do play a key role in nominal C or AC-Unification. In standard nominal unification, a fixed point equation like $\pi \cdot X \approx_\alpha \pi' \cdot X$ would be easily solved by requiring that $ds(\pi, \pi') \# \nabla$, where ∇ is the context that composes the solution to the problem (see the rule for α -equivalence of suspended variables in Definition 2.16).

This approach, while still correct in nominal C or AC-unification, is not complete. Take, for instance the equation $(a b) \cdot X \approx_\alpha X$. A solution not captured by this traditional approach is $\langle \emptyset, [X \rightarrow a + b] :: Id \rangle$, where $+$ is an associative-commutative function symbol (here we are using infix notation). Moreover, there is an infinite number of solutions to fixed point equations

[ARdCSFNS18]. Considering the equation above, among the infinite number of solutions we have: $\langle \emptyset, [X \rightarrow a + b] :: Id \rangle$ $\langle \emptyset, [X \rightarrow (a + b) + (a + b)] :: Id \rangle$ and so on.

Consequently, fixed point equations are not solved in nominal C or AC-Unification, instead, they are carried on as part of the solution to the unification problem [ARdCSFNS18]. In this work, we decided to separate the fixed point equations from the set of equational and freshness constraints, which allowed us reduce by one the number of parameters of the lexicographic measure used in the proofs of termination, soundness and completeness. This will be better discussed in Section 3.4.1.

There is a simple extension of the definition of solution for a triple or unification problem (Definition 2.24) in order to consider this separation of fixed point equations. It is shown in Definition 2.26. For further reference, we repeat the common clauses with Definition 2.24.

Definition 2.26 (Solution for a Quadruple). Let FP be a set of fixed point equations. $\langle \nabla, \sigma \rangle$ is a solution to the quadruple $\mathcal{P} = \langle \Delta, \delta, P, FP \rangle$ if the four conditions of the Definition 2.24 are satisfied and we also have:

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

This means that five conditions must be met:

- $\nabla \vdash \Delta \sigma$
- $\nabla \vdash a \# t \sigma$, if $a \#_\gamma t \in P$
- $\nabla \vdash s \sigma \approx_\alpha t \sigma$, if $s \approx_\gamma t \in P$
- There exists λ such that $\nabla \vdash \delta \lambda \approx \sigma$
- $\nabla \vdash \pi \cdot X \sigma \approx_\alpha \pi' \cdot X \sigma$, if $\pi \cdot X \approx_\gamma \pi' \cdot X \in FP$

Remark 2.13. In [AFN19], instead of working with freshness relations, the authors work with the notion of a fixed-point constraint. This alternative formulation has the advantage that C-unification remains finitary. For further reference about this alternative approach, one can check [AFN19].

Chapter 3

A Correct and Complete Algorithm for Functional Nominal C-Unification

In this chapter we detail the algorithm developed for the task of functional nominal C-Unification. In this chapter the nominal setting contains C function symbols, but not AC function symbols. Since we used PVS for specifying and formalising the algorithm, we briefly tell about the PVS files and where to find further information. Next, we present and explain the specification of the algorithm. Then, we give some examples and comment on the most interesting aspects of the formalisation. Following this, we briefly mention our Python implementation of the algorithm, again pointing where to find further information. Finally, we discuss possible applications. The material of this chapter is an extension of [ARFFSNS19] and all theorems and corollaries in this chapters were proved in PVS.

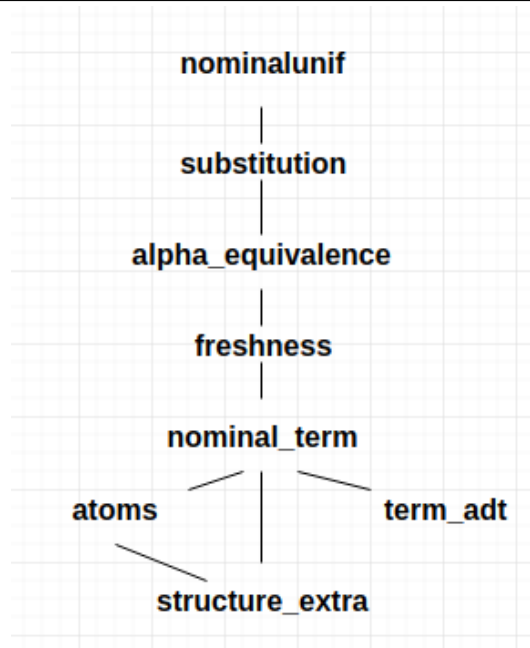
3.1 The PVS Theory Organisation and Where to Find Further Information

As remarked before, the specification and formalisation of the PVS theory are fully available at <http://www.github.com/gabriel951/c-unification>. Additional information can be found by looking at the README or the .pvs files themselves. Version 6 of PVS must be used.

The hierarchy of the PVS files is presented in Figure 3.1.

We succinctly describe the contents of each file:

- `nominalunif.pvs` - It is the main file of the nominal C-unification theory. The file contains the main function `c_unify` and also the specifications for the lemmas of soundness and completeness. As usual, the formalisations of these lemmas are in the correspondent `.prf` file.

Fig. 3.1 Hierarchy of the PVS Theory For Nominal C-Unification.

- `substitution.pvs` - Deals with substitutions and basic properties. Contains a lot of auxiliary lemmas used in the formalisation of the algorithm.
- `alpha_equivalence.pvs` - Concerns α -equivalence, defined via the function “alpha”. It is proved that α -equivalence is indeed an equivalent relation.
- `fresh.pvs` - Contains the definition of freshness and related lemmas.
- `nominal_term.pvs` - Defines a nominal term and functions that deal with nominal terms, such as the action of permutation on nominal terms. Contains the functions that calculates the set of variables in a nominal term and the size of a nominal term, used in the lexicographic measure of the proofs of termination, soundness and completeness.
- `term_adt.pvs` - This file is generated automatically by PVS and contains the necessary datatype definitions in order for the nominal term datatype to work correctly.
- `atoms.pvs` - Defines atoms, permutations and how permutations act on atoms.
- `structure_extra.pvs` - Has basic properties related with lists and membership, to be used by the remaining `.pvs` files.

3.2 Specification

We have developed a recursive algorithm that can unify nominal terms t and s which may contain C function symbols (see Algorithm 1). In the pseudocode, the $+$ denotes a concatenation of lists, the $=$ denotes an assignment, the symbol $==$ checks equality while $!=$ checks inequality.

The algorithm needs to keep track of the current context, the substitutions made so far, the remaining terms to unify and the current fixed point equations. Hence, the algorithm receives as input a quadruple $(\Delta, \sigma, PrbLst, FPEqLst)$, where Δ is the context we are working with, σ is a list of substitutions already done, $PrbLst$ is a list of pairs of terms which we must still unify and $FPEqLst$ is a list of the fixed point equations already computed.

The first call to the algorithm, in order to unify terms t and s is simply: $UNIFY(\emptyset, Id, [(t, s)], \emptyset)$. The algorithm will eventually terminate, returning a list of solutions (which may be empty, if there is no solution) where each solution is of the form: $(\Delta, \sigma, FPEqLst)$.

Although long, the algorithm is simple. It starts by analysing the list of terms it needs to unify. If $PrbLst$ is an empty list, then it has finished and can return the answer computed so far, which is the list $[(\Delta, \sigma, FPEqLst)]$. If $PrbLst$ is not empty, then there are terms to unify and the algorithm begins by trying to unify the terms t and s in the head of $PrbLst$ and only after that it goes to the tail of the list (represented, in Algorithm 1, by $PrbLst'$). The algorithm keeps calling itself on progressively simpler versions of the problem until it finishes.

Remark 3.1. When trying to unify $f^C\langle t_1, t_2 \rangle$ with $f^C\langle s_1, s_2 \rangle$ there are two possible branches to take:

- try to unify t_1 with s_1 and t_2 with s_2
- try to unify t_1 with s_2 and t_2 with s_1

This means there are two paths we must consider, and since each path can generate one solution, we may have more than one solution. That is why Algorithm 1 gives, as output, a list of solutions.

Remark 3.2. As can be seen in line 26 of Algorithm 1, a fixed point equation $\pi \cdot X \approx_\alpha \pi' \cdot X$ is represented as a pair $(\pi'^{-1} \circ \pi, X)$. To convert from this representation to the representation of a unification problem, the function `fix_pnt2unif_prb` was used. A better approach should have been employed: fixed point equations should have been represented in the same way unification problems were, which would avoid the need for a separate function to perform the conversion from one representation to another.

Algorithm 1 - First Part - Functional Nominal C-Unification

```

1: procedure UNIFY( $\Delta, \sigma, PrbLst, FPEqLst$ )
2:   if null( $PrbLst$ ) then
3:     return list( $(\Delta, \sigma, FPEqLst)$ )
4:   else
5:      $(t, s) + PrbLst' = PrbLst$ 
6:     if ( $s == \pi \cdot X$ ) and ( $X$  not in  $t$ ) then
7:        $\sigma' = \{X \rightarrow \pi^{-1} \cdot t\}$ 
8:        $\sigma'' = \sigma' \circ \sigma$ 
9:        $(\Delta', bool1) = \text{fresh\_subs?}(\sigma', \Delta)$ 
10:       $\Delta'' = \Delta \cup \Delta'$ 
11:       $PrbLst'' = (PrbLst')\sigma' + (FPEqLst)\sigma'$ 
12:      if bool1 then return UNIFY( $\Delta'', \sigma'', PrbLst'', null$ )
13:      else return null
14:      end if
15:     else
16:       if  $t == a$  then
17:         if  $s == a$  then
18:           return UNIFY( $\Delta, \sigma, PrbLst', FPEqLst$ )
19:         else
20:           return null
21:         end if
22:       else if  $t == \pi \cdot X$  then
23:         if ( $X$  not in  $s$ ) then
24:            $\triangleright$  Similar to case above where  $s$  is a suspension and  $X$  is not in  $t$ .
25:         else if ( $s == \pi' \cdot X$ ) then
26:            $FPEqLst' = FPEqLst \cup \{((\pi')^{-1} + \pi) \cdot X\}$ 
27:           return UNIFY( $\Delta, \sigma, PrbLst', FPEqLst'$ )
28:         else return null
29:         end if
30:       else if  $t == \langle \rangle$  then
31:         if  $s == \langle \rangle$  then
32:           return UNIFY( $\Delta, \sigma, PrbLst', FPEqLst$ )
33:         else return null
34:         end if
35:       else if  $t == \langle t_1, t_2 \rangle$  then
36:         if  $s == \langle s_1, s_2 \rangle$  then
37:            $PrbLst'' = [(s_1, t_1)] + [(s_2, t_2)] + PrbLst'$ 
38:           return UNIFY( $\Delta, \sigma, PrbLst'', FPEqLst$ )
39:         else return null
40:         end if

```

Algorithm 1 - Second Part - Functional Nominal C-Unification

```

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

```

3.2.1 Auxiliary Functions

In accordance with the approach of [ARFRO16], we separate the treatment of freshness constraints from the main function, by using auxiliary functions. This is advantageous, for the function UNIFY gets smaller, as it only needs to deal with equational constraints. This separate treatment also allowed a reduction on the number of parameters of the lexicographic measure used in the proofs of termination, soundness and completeness, as will be better discussed in Section 3.4. The auxiliary functions used by UNIFY to deal with freshness constraints are:

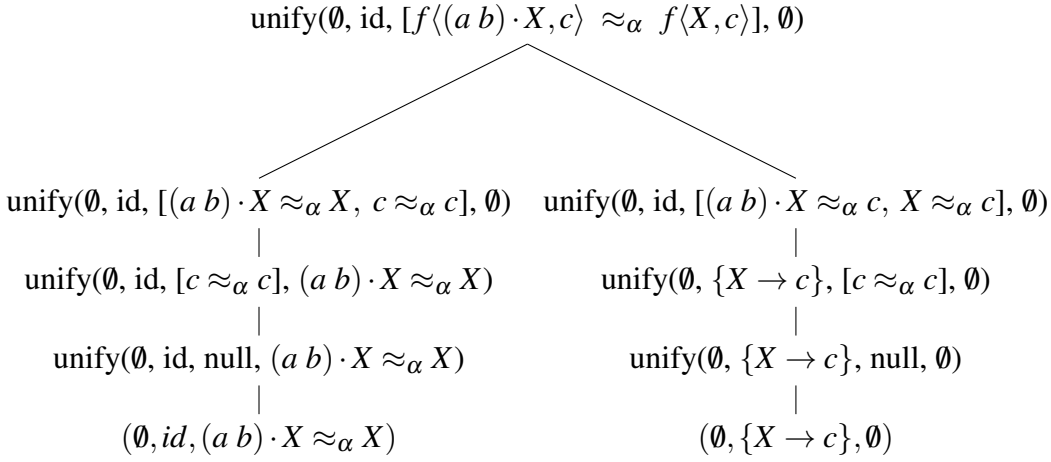
- `fresh_subs?(σ, Δ)`: This function returns the minimal context (Δ' in Algorithm 1) in which $a\#_?X\sigma$ holds, for every $a\#X$ in the context Δ . A boolean (`bool1` in Algorithm 1) is also returned, indicating if it is possible to find the mentioned context. This function can be found in file `substitution.pvs`.
- `fresh?(a, t)`: This function computes and returns the minimal context (Δ' in Algorithm 1) in which a is fresh on t . This function also returns a boolean (`bool1` in Algorithm 1), indicating if it is possible to find the mentioned context. This function can be found in file `freshness.pvs`.

Remark 3.3. The algorithm does not handle directly as input an unification problem that already has a freshness constraint $a\#_?t$. This, however, would not be a difficult extension from what we currently have and could be done in the following manner: if the algorithm receives a freshness constraint $a\#_?t$ apply function `fresh?` directly and if the algorithm receives an equational constraint $s \approx_? t$, apply function UNIFY.

3.3 Examples

An example of the algorithm running is given in Example 3.1. It is possible to observe how commutative function symbols introduces branches, how more than one solution may be given and also how the algorithm keeps calling itself with progressively easier versions of the problem until it finishes. A more intricate example, which uses Example 3.1, is shown in Example 3.2.

Example 3.1. Suppose f is a commutative function symbol. This example shows how the algorithm proceeds in order to unify $f\langle(a\ b) \cdot X, c\rangle$ with $f\langle X, c\rangle$.



In this example, the output of the algorithm would be:

$$[(\emptyset, \text{id}, (a b) \cdot X \approx_{\alpha} X), (\emptyset, \{X \rightarrow c\}, \emptyset)]$$

Example 3.2. Let f and g be commutative function symbols and h be a non-commutative function symbol. This example shows how the algorithm would unify $g\langle h d, f\langle(a b) \cdot X, c\rangle\rangle$ with $g\langle f\langle X, c\rangle, h d\rangle$. Because g is commutative, the algorithm explores two branches:

- On the first branch, the algorithm tries to unify $h d$ with $f\langle X, c\rangle$ and $f\langle(a b) \cdot X, c\rangle$ with $h d$. However, since it is impossible to unify $h d$ with $f\langle X, c\rangle$ (different function symbols), the algorithm returns an empty list, indicating that no solution is possible for this branch.
- On the second branch, the algorithm tries to unify $h d$ with $h d$ and $f\langle(a b) \cdot X, c\rangle$ with $f\langle X, c\rangle$. First, $h d$ unify with $h d$ without any alterations on the context Δ , the substitution σ or the list of fixed point equations $FPEqLst$. Finally, the unification of $f\langle(a b) \cdot X, c\rangle$ with $f\langle X, c\rangle$ was shown in Example 3.1, and gives two solutions: $(\emptyset, \text{id}, (a b) \cdot X \approx_{\alpha} X)$ and $(\emptyset, \{X \rightarrow c\}, \emptyset)$.

The algorithm then concatenates the empty list of the first branch with the list containing two solutions of the second branch, giving as solution:

$$[(\emptyset, \text{id}, (a b) \cdot X \approx_{\alpha} X), (\emptyset, \{X \rightarrow c\}, \emptyset)]$$

3.4 Formalisation

3.4.1 The Lexicographic Measure and Termination of the Algorithm

The termination of the algorithm was proved by proving the TCCs (the concept of TCC can be found in Section 2.1.1) generated by PVS [SORSC01]. In order to do that, we defined the following lexicographic measure and proved that it decreases in every recursive call:

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

$|\text{Vars}(\text{PrbLst}) \cup \text{Vars}(\text{FPEqLst})|$ is the cardinality of the set of variables which occur in PrbLst (the list of unification problems that remain) or in FPEqLst (the list of fixed point equations). To compute the set of variables in a list, we consider the union of the set of variables in all terms of the list. The set of variables is recursively computed as shown in Definition 3.1.

Definition 3.1. The set of variables in a term is recursively defined as:

$$\text{Vars}(a) = \emptyset \qquad \text{Vars}(\langle \rangle) = \emptyset$$

$$\text{Vars}(\pi \cdot X) = \{X\} \qquad \text{Vars}([a]t) = \text{Vars}(t)$$

$$\text{Vars}(\langle t_0, t_1 \rangle) = \text{Vars}(t_0) \cup \text{Vars}(t_1) \qquad \text{Vars}(ft) = \text{Vars}(t)$$

$$\text{Vars}(f^C \langle t_0, t_1 \rangle) = \text{Vars}(t_0) \cup \text{Vars}(t_1)$$

To see how the PVS specification computes this, check function Vars in Appendix A, Section A.1.

The second component of the lexicographic measure is $\text{size}(\text{PrbLst})$, which is the sum of the size of every unification problem. In an arbitrary manner, it was decided that the size of the unification problem (t, s) is the size of t , the first term. The measure would still work had we taken the size of the unification problem (t, s) to be the size of s or even the size of t plus the size of s . The definition for the size of the term t is given in Definition 3.2.

Definition 3.2. The size of a term is recursively computed as as:

$$\text{size}(a) = 1 \qquad \text{size}(\langle \rangle) = 1$$

$$\text{size}(\pi \cdot X) = 1 \qquad \text{size}([a]t) = 1 + \text{size}(t)$$

$$\text{size}(\langle t_0, t_1 \rangle) = 1 + \text{size}(t_0) + \text{size}(t_1) \qquad \text{size}(ft) = 1 + \text{size}(t)$$

$$\text{size}(f^C \langle t_0, t_1 \rangle) = 1 + \text{size}(t_0) + \text{size}(t_1)$$

To see how the PVS specification computes this, check function `size` in Appendix A, Section A.1.

Notice that the lexicographic measure decreases in each recursive call and the component that diminishes depends on the type of the terms t and s that are in the head of the list of problems to unify. If one of them is a suspended variable, say $\pi \cdot X$, and we are not dealing with a fixed point equation, then the algorithm will instantiate the variable X appropriately and the first component of the lexicographic measure, $|Vars(PrbLst) \cup Vars(FPEqLst)|$ will decrease. In any other case, the first component remains the same (since no variable is introduced) and the second component ($size(PrbLst)$) decreases.

Remark 3.4. We reduced the lexicographic measure used in [ARdCSFNS18], from 4 parameters to only 2 parameters. The measure adopted in [ARdCSFNS18] was:

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

where P_{\approx} is the set of equation constraints in P , P_{nfp} is the set of non fixed point equations in P and $P_{\#}$ is the set of freshness constraints in P . Two insights were necessary to accomplish this reduction. The first was to separate the treatment of freshness constraints from equational constraints, and treat freshness constraints separately, with the use of the auxiliary functions of Section 3.2.1. This idea comes from [ARFRO16]. The second insight is to separate the fixed point equations from the equational constraints. In this manner, when a fixed point equation is found in $PrbLst$, it is added to $FPEqLst$ and taken out of $PrbLst$, diminishing $size(PrbLst)$.

Remark 3.5. We simplified the proofs of soundness, completeness and termination by reducing the lexicographic measure used on them.

3.4.2 Soundness and Completeness

Before stating the main theorems of soundness and completeness, we define a valid quadruple. A valid quadruple is an invariant of the UNIFY function (see Algorithm 3.1) with useful properties.

Definition 3.3 (Valid Quadruple). Let Δ be a freshness context, σ a substitution, P a list of unification problems and FP a list of fixed point equations. $\mathcal{P} = \langle \Delta, \sigma, P, FP \rangle$ is a valid quadruple if the following two conditions hold:

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

Remark 3.6. A valid quadruple has two desirable properties: the substitution is idempotent (condition 1) and applying the substitution to P or FP produces no effect (condition 2).

Soundness

The corollary 3.1 enunciates that UNIFY is sound. It follows straightforwardly by application of Theorem 3.1.

Theorem 3.1. Suppose $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \text{UNIFY}(\Delta, \sigma, PrbLst, FPEqLst)$, (∇, δ) is a solution to $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$ and that $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$ is a valid quadruple. Then (∇, δ) is a solution to $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$.

Proof. The proof is by induction on the lexicographic measure, according to the case of the terms t and s that are in the head of $PrbLst$, the list of remaining unification problems. The hardest cases are the ones of suspended variables and abstractions (the same happens when proving completeness, so their explanation is put together in Remark 3.12). We now explain the case of commutative functions.

In the case of commutative function symbols $f\langle t_1, t_2 \rangle$ and $f\langle s_1, s_2 \rangle$, there are no changes in the context nor in the substitution from one recursive call to the next. Therefore, it is trivial to check that we remain with a valid quadruple and it is also trivial to check all but the third condition of Definition 2.26. For the third condition we have either $(\nabla \vdash t_1 \delta \approx_\alpha s_1 \delta$ and $\nabla \vdash t_2 \delta \approx_\alpha s_2 \delta)$ or $(\nabla \vdash t_1 \delta \approx_\alpha s_2 \delta$ and $\nabla \vdash t_2 \delta \approx_\alpha s_1 \delta)$. In any case, we are able to deduce $\nabla \vdash (f\langle t_1, t_2 \rangle) \delta \approx_\alpha (f\langle s_1, s_2 \rangle) \delta$ by noting that $(f\langle t_1, t_2 \rangle) \delta = f\langle t_1 \delta, t_2 \delta \rangle$, $(f\langle s_1, s_2 \rangle) \delta = f\langle s_1 \delta, s_2 \delta \rangle$ and then using rule $(\approx_\alpha c - app)$ for α -equivalence of commutative function symbols. \square

Corollary 3.1 (Soundness of Unify). Suppose $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \text{UNIFY}(\emptyset, Id, [(t, s)], \emptyset)$, (∇, δ) is a solution to $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$. Then (∇, δ) is a solution to $\langle \emptyset, Id, [(t, s)], \emptyset \rangle$.

Proof. Notice that $\langle \emptyset, Id, [(t, s)], \emptyset \rangle$ is a valid quadruple. Then, we apply Theorem 3.1 and prove the corollary. \square

Remark 3.7. An interpretation of Corollary 3.1 is that if (∇, δ) is a solution to one of the outputs of the algorithm UNIFY, then (∇, δ) is a solution to the original problem.

Remark 3.8. It was not possible to do a proof by induction directly in Corollary 3.1 for although in the initial problem we have an empty context, the identity substitution, only one unification problem and an empty list of fixed point equations this is not necessarily the case for the next recursive call.

Completeness

In an analogous manner with the proof of soundness, the Corollary 3.2 enunciates that UNIFY is complete and it follows directly by the application of Theorem 3.2.

Theorem 3.2. Suppose (∇, δ) is a solution to $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$ and that $\langle \Delta, \sigma, PrbLst, FPEqLst \rangle$ is a valid quadruple. Then, there exists $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \text{UNIFY}(\Delta, \sigma, PrbLst, FPEqLst)$ such that (∇, δ) is a solution to $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$.

Proof. The proof is by induction on the lexicographic measure, according to the case of the terms t and s that are in the head of $PrbLst$, the list of remaining unification problems. The hardest cases are again the ones of suspended variables and abstractions (explained in Remark 3.12). Below we explain the case of commutative functions.

In the case of commutative function symbols $f\langle t_1, t_2 \rangle$ and $f\langle s_1, s_2 \rangle$, there are no changes in the context nor in the substitution from one recursive call to the next. Therefore, it is trivial to check that we remain with a valid quadruple and it is also trivial to check all but the third condition of Definition 2.26. For the third condition we have $\nabla \vdash (f\langle t_1, t_2 \rangle)\delta \approx_\alpha (f\langle s_1, s_2 \rangle)\delta$ and must prove that either $(\nabla \vdash t_1\delta \approx_\alpha s_1\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_2\delta)$ or $(\nabla \vdash t_1\delta \approx_\alpha s_2\delta$ and $\nabla \vdash t_2\delta \approx_\alpha s_1\delta)$ happens. This again is solved by noting that $(f\langle t_1, t_2 \rangle)\delta = f\langle t_1\delta, t_2\delta \rangle$, $(f\langle s_1, s_2 \rangle)\delta = f\langle s_1\delta, s_2\delta \rangle$ and then using rule $(\approx_\alpha c - app)$ for α -equivalence of commutative function symbols. \square

Corollary 3.2 (Completeness of Unify). Suppose (∇, δ) is a solution to $\langle \emptyset, Id, [(t, s)], \emptyset \rangle$. Then, there exists $(\Delta_{sol}, \sigma_{sol}, FPEqLst_{sol}) \in \text{UNIFY}(\emptyset, Id, [(t, s)], \emptyset)$ such that (∇, δ) is a solution to $\langle \Delta_{sol}, \sigma_{sol}, \emptyset, FPEqLst_{sol} \rangle$.

Proof. Notice that $\langle \emptyset, Id, [(t, s)], \emptyset \rangle$ is a valid quadruple. Then, we apply Theorem 3.2 and prove the corollary. \square

Remark 3.9. An interpretation of Corollary 3.2 is that if (∇, δ) is a solution to the initial problem, then (∇, δ) is also a solution to one of the outputs of UNIFY.

Remark 3.10. In a similar way as Remark 3.8, it was not possible to do a proof by induction directly in Corollary 3.2 for although in the initial problem we have an empty context, the identity substitution, only one unification problem and an empty list of fixed point equations this is not necessarily the case for the next recursive call.

Remark 3.11. An alternative approach for formalising the correctness and completeness of the Algorithm 1 given would be to prove its functional equivalence with the set of non-deterministic rules presented in [ARdCSFNS18], since this rules were already shown to be correct and complete. That, however, would require a “translation” (which must be assured to be correct) from the Coq specification to the PVS specification (or vice-versa).

3.4.3 Interesting Points

Interesting points of the formalisation are discussed in Remarks 3.12 and 3.13.

Remark 3.12. To prove correctness and completeness of the algorithm, we worked with the terms t and s that were in the head of $PrbLst$. We divided the proof in cases. The hardest case happened when t or s were a suspended variable of the type $\pi \cdot X$ and the variable X did not occur in the other term (see Algorithm 1).

The reason for that is the algorithm receives as arguments Δ , σ , $PrbLst$ and $FPEqLst$ and the next recursive call is made with four different parameters: Δ'' , σ'' , $PrbLst''$ and $null$ (see Algorithm 1). Therefore, all of the five conditions of the Definition 2.26 are not trivially satisfied. Moreover, since in the next recursive call we will be working with a new substitution σ'' and the variable X will no longer be a part of $PrbLst''$, we must assure that the quadruple we are working with remains valid.

The case of unifying $t = [a]t_1$ with $s = [b]s_1$ was also interesting, as both the context and the list of problems to unify suffer modifications in the recursive call. In all the remaining cases, there are no changes in the context nor in the substitution, making them easier. In these remaining cases, only one condition (the third) of the five in Definition 2.26 is not trivially satisfied. Moreover, this one condition can be easily checked to be true (or false, and in this event the algorithm will return an empty list) in the remaining cases by applying the appropriate α -equivalence rule.

Remark 3.13. There were three main novelties in the formalisation which were due to the introduction of commutative function symbols.

- Unification of commutative functions. The unification of commutative functions (for instance, $f(t_1, t_2)$ with $f(s_1, s_2)$) was straightforward (see Algorithm 1). First, the algorithm tries to unify t_1 with s_1 and t_2 with s_2 , generating a list of solutions sol_1 . Then, the algorithm tries unifying t_1 with s_2 and t_2 with s_1 , generating a list of solutions sol_2 . The final result is then simply the concatenation of both lists.
- Management of fixed point equations. This was also straightforward. We keep a separate list of fixed point equation ($FPEqLst$), and when the algorithm recognises a fixed point equation in $PrbLst$ it takes this equation out of $PrbLst$ and puts it on $FPEqLst$.
- Reason about the appropriate data structure for the algorithm and the solutions. This was not straightforward. As mentioned before, since commutativity introduced branches, the recursive calls of the algorithm can be seen as a tree (see Example 3.1). Therefore, initially, an approach using a tree data structure was planned (which would have complicated the analysis). However, since the algorithm simply solves one branch and then solves the other, we realized all that was needed was to do two recursive calls (one for each branch) and append the two lists of solutions generated. Hence, we were able to avoid the tree data structure, working instead with lists, which simplified the specification.

3.5 Implementation

We have implemented in Python the functional algorithm proved correct and complete. The implementation may be found on file `unify.py`, while Examples 3.3 and 3.4 that will be shown next can be run by executing file `tests.py` (both of them also available at <http://www.github.com/gabriel951/c-unification>). Specific instructions can be found on these files. The most recent version of Python, Python 3, should be used. As PVS does not support automatic extraction to Python code, the translation of the PVS specification to the Python implementation was done manually. The implementation follows strictly the PVS specification developed, with the exception of minor differences now detailed. For fragments of the Python code, check appendix B.

Two optional parameters were added to function UNIFY when programming the algorithm, to allow the user to run the algorithm either printing or not the tree of recursive calls. The first parameter (`verb`) is a boolean that sets the verbosity of the algorithm: when set to true, the algorithm prints the tree of recursive calls and when set to false the algorithm does not. The second parameter (`indent_lvl`) is a string that regulates the indentation level. When the algorithm forks into two branches, this parameter is updated, allowing us to clearly distinguish when the algorithm enters a branch (see Examples 3.3 and 3.4).

As mentioned before, a term in the nominal setting can be an atom, a suspended variable, the unit, a pair, an abstraction, a function application or a commutative function application. Each of these options is represented as a class in our implementation. An atom object a has one attribute, its name, represented by the algorithm as a string. A suspended variable object contains two attributes: a permutation and a variable. A permutation is represented as a list, where each element of the list (corresponding to a swapping) is a pair consisting of two strings (to represent the two atoms being swapped) and a variable is represented by the algorithm as a string. The unit is a class with no attributes. A pair object contains two attributes, to represent the two terms of a pair. An abstraction object contains two attributes: the first is an atom and the second a term. A function application and a commutative function application both have two attributes: the first is the function symbol, represented as a string, and the second is their argument, which is another term. In the case of commutative function applications, an assertion verifies that the argument is indeed a pair, as commented in Remark 2.10.

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

Example 3.3. Let f be a commutative function symbol. Here we show how the Python algorithm would proceed to unify the terms in Example 3.1: $f\langle(a b) \cdot X, c\rangle$ and $f\langle X, c\rangle$. Note that each one of the two branches explored generates a solution.

Trying to unify terms: $f\langle[(a b)] * X, c\rangle$ and $f\langle[] * X, c\rangle$

$\langle[], \text{id}, [(f\langle[(a b)] * X, c\rangle) = f\langle[] * X, c\rangle)], [], []\rangle$

|

$\langle[], \text{id}, [([(a b)] * X = [] * X), (c = c)], [], []\rangle$

|

$\langle[], \text{id}, [(c = c)], [(['a', 'b')] * X = X], []\rangle$

|

$\langle[], \text{id}, [], [(['a', 'b')] * X = X], []\rangle$

|

solution: $\langle[], \text{id}, [(['a', 'b')] * X = X], []\rangle$

$\langle[], \text{id}, [([(a b)] * X = c), (c = [] * X)], [], []\rangle$

|

$\langle[], [X \rightarrow c], [(c = c)], [], []\rangle$

|

$\langle[], [X \rightarrow c], [], [], []\rangle$

|

solution: $\langle[], [X \rightarrow c], [], []\rangle$

Finished.

Example 3.4. Let f and g be commutative function symbols, and let h be a non-commutative function symbol. We now show how the Python algorithm would proceed to unify the terms in Example 3.2: $g\langle h d, f\langle(a b) \cdot X, c\rangle\rangle$ and $g\langle f\langle X, c\rangle, h d\rangle$. Note that one branch does not generate any solution.

Trying to unify terms: $g(\langle h(d), f(\langle [(a\ b)] * X, c \rangle) \rangle)$ and $g(\langle f(\langle [] * X, c \rangle), h(d) \rangle)$

```

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

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

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

```

Finished.

Example 3.5. We now give an example in which the context returned is not empty, a substitution has been made and there is a fixed point equation. Imagine we want to unify $h\langle [a](c\ d) \cdot X, (c\ a) \cdot Z \rangle$ and $h\langle [b]Id \cdot Y, (b\ e) \cdot Z \rangle$. The algorithm running is shown below.

Trying to unify terms: $h(\langle [a] [(c\ d)] *X, [(c\ a)] *Z \rangle)$ and
 $h(\langle [b] [] *Y, [(b\ e)] *Z \rangle)$

```

<[], id, [(h(<[a] [(c d)] *X, [(c a)] *Z) = h(<[b] [] *Y, [(b e)] *Z)), ], [] >
|
<[], id, [(<[a] [(c d)] *X, [(c a)] *Z = <[b] [] *Y, [(b e)] *Z>), ], [] >
|
<[], id, [( [a] [(c d)] *X = [b] [] *Y), ([ (c a)] *Z = [(b e)] *Z), ], [] >
|
<[a#Y ], id, [( [(c d)] *X = [] *Y), ([ (c a)] *Z = [(b e)] *Z), ], [] >
|
<[a#X a#Y ], [Y->[(c d)] *X ], [( [(c a)] *Z = [(b e)] *Z), ], [] >
|
<[a#X a#Y ], [Y->[(c d)] *X ], [], [( ('b', 'e'), ('c', 'a')) *Z = Z ] >
|
solution: <[a#X a#Y ], [Y->[(c d)] *X ], [( ('b', 'e'), ('c', 'a')) *Z = Z ] >

```

Finished.

The final answer is $(\Delta, \sigma, FPEqLst) = (\{a\#X, a\#Y\}, \{Y \rightarrow (cd) \cdot X\}, (b\ e) :: (c\ a) \cdot Z \approx_\alpha Z)$.

3.6 Possible Applications

Since unification has applications to logic programming ([BN99]), the nominal C-unification algorithm given could be used in a logic programming language that employs the nominal setting, such as α Prolog (see the work of Cheney and Urban in [CU04]). For systems implemented in α Prolog, there is a tool for checking if the desired properties really occur: α Check [CM17]. Therefore, we speculate that other possible application could be in the α Check program itself, to test properties related with commutativity.

Another application is related to matching (see [BS01], [BN99]). As remarked before, matching two terms t and s can be seen as unification where one of the terms (suppose t , without loss of generality) is not affected by substitutions [ARdCSFNS18]. This can be accomplished by extending the algorithm to use as an additional parameter a set of variables that cannot be instantiated. Then, matching resumes to unifying using as this additional parameter the set of variables in t [ARdCSFNS18]. The C-matching algorithm proposed could then be used to extend the nominal rewriting relation introduced in [FG07] modulo commutativity. Also,

nominal C-unification and C-matching would be relevant in implementing nominal narrowing allowing commutative symbols (see [[AFN16](#)]).

Chapter 4

Formalising Nominal AC-Unification

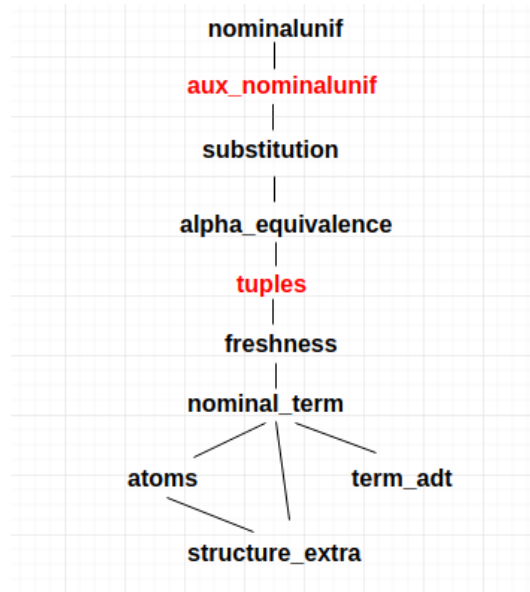
This chapter discusses our work in progress in developing a nominal AC-Unification algorithm. Here, the nominal syntax contains both C and AC function symbols. We begin by explaining the organisation and hierarchy of the PVS files. Then, the current specification of the algorithm is presented. Next, interesting points related with the formalisation are discussed. Finally, we briefly comment about possible applications of the functional nominal AC-unification algorithm. The content of this chapter is an extension of [ARFFS19].

4.1 Hierarchy and Organisation of the PVS theory

The specification and formalisation of the PVS theory are fully available at www.github.com/gabriel951/ac-unification. Other informations can be found by looking at the README file or the .pvs files themselves. The hierarchy of the PVS files is similar to the one described in 3.1 and is shown on Figure 4.1. The names in red are files not previously contained in the PVS theory that formalised nominal C-unification.

The contents of each file are shortly described:

- `nominalunif.pvs` - It is the main file of the nominal AC-unification theory. This file contains the main function `unify` and also the specification of the lemmas of soundness and completeness.
- `aux_nominalunif.pvs` - This new file contains secondary definitions and lemmas used by the file `nominalunif.pvs`, to avoid cluttering it.
- `substitution.pvs` - Deals with substitutions.
- `alpha_equivalence.pvs` - Concerns α -equivalence.

Fig. 4.1 Hierarchy of the PVS theory for Nominal AC-Unification.

- `tuples.pvs` - This new file contains lemmas related with the functions that handle the combinatory part of the problem (introduced by the AC functions, as will be seen in 4.2) and also with the operators $S_n(f^*)$ and $D_n(f^*)$, that select and delete arguments from AC functions.
- `fresh.pvs` - Deals with freshness.
- `nominal_term.pvs` - Contains the datatype definition of a nominal term and related functions.
- `term_adt.pvs` - Generated automatically by PVS, contains necessary definitions for the nominal term datatype to work correctly.
- `atom.pvs` - Concerns atoms and permutations.
- `structure_extra.pvs` - Has basic properties that will be used by the remaining of the `.pvs` files.

Remark 4.1. The hierarchy for the PVS theory for nominal AC-unification is similar to the one for nominal C-unification (see Section 3.1): the only two differences are the files `aux_nominal_unif.pvs` and `tuples.pvs`. The file `aux_nominalunif.pvs` is simply not to clutter the main file `nominalunif.pvs`. Therefore, the only significant alteration in the organisation is the new file `tuples.pvs` necessary to deal with the combinatory part of the problem and the operators $S_n(f^*)$ and $D_n(f^*)$, introduced by associativity-commutativity.

Remark 4.2. Despite its current name, the PVS theory for nominal AC-unification, if completed, would allow unification with both C and AC function symbols.

4.2 Current Specification

We have specified a functional nominal AC-unification algorithm for unifying two terms t and s . The algorithm here presented is an extension of Algorithm 1, to also deal with the case of t or s being AC functions. Hence, we will not explain again how the whole algorithm works, as this can be found on Section 3.2, focusing instead on the case of AC function symbols.

When t or s is rooted by an AC function symbol and the other term is a suspended variable, the algorithm instantiates the suspended variable term appropriately and unifies the terms. Alternatively, when one of the terms is rooted by an AC function symbol and the other is neither a suspended variable nor a term rooted by the same AC function symbol, the algorithm recognises it is a situation where no solution is possible and the list of solutions returned is empty. The interesting case is, therefore, when both t and s are rooted by the same AC function symbol.

In this case, the algorithm first extracts all arguments of t and then generates all *pairings* of these arguments, in any order. After that, the algorithm extracts all arguments of s and then generates all pairings of these arguments, again in any order. Finally, the algorithm tries to unify every pairing of arguments of t with every pairing of arguments of s . An example of the pairings generated is shown on Example 4.1.

Example 4.1. Let f be an AC function symbol and suppose we are trying to unify the terms $f\langle f\langle X, Y \rangle, c \rangle$ and $f\langle c, f\langle a, b \rangle \rangle$:

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

Remark 4.3. Our initial idea to deal with the pairings of the arguments of t and s was to generate all pairings of the arguments of t , preserving the order and all pairings of the arguments of s , in any order. This approach, however, is not complete.

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

substitution σ is found when trying to unify $\langle\langle a, c \rangle, b\rangle$ with $\langle X, b\rangle$ and the arguments of the pairing $\langle\langle a, c \rangle, b\rangle$ are not in the same order that they are in t .

The pseudocode of the algorithm when extended for AC function applications is shown in Algorithm 2. When the algorithm is unifying two AC function applications t and s , the first step is to generate all pairings of t and s and combine them. This step is done by function `gen_unif_prb`. The result is a list of unification problems (each one of these unification problems corresponds to an attempt of unifying a pairing of t and a pairing of s), stored in variable $NewPrbLst$. The algorithm will call the function `unify` recursively, with the same Δ , σ and $FPEqLst$, but with a new list of unification problems, consisting of one unification problem from $NewPrbLst$ concatenated with the list of remaining problems $PrbLst'$. This corresponds to the algorithm exploring one branch in the tree of recursive calls. The algorithm will do this for every unification problem in $NewPrbLst$, thus exploring all branches in the tree of recursive calls. The algorithm does that in two steps. In the first step, the function `get_1st_quad` is called, constructing a list of quadruples, which are stored in variable $LstQuad$. Each quadruple in $LstQuad$ is of the form $(\Delta, \sigma, PrbLst, FPEqLst)$. In the next step, the `map` function applies the function `unify` to every quadruple in $LstQuad$ and since every application of `unify` generates a list of solutions, the returned value of the `map` function is a list of lists of solutions, stored in variable $LstLstSol$. The algorithm then returns a flattened version of this list, via the use of the `flatten` function.

Remark 4.4. To handle the combinatory of the problem, we opted for a naive, brute-force approach that does not take efficiency into consideration. This has been done with the goal of simplifying the formalisation. Indeed compared with the algorithm presented in Example 2.8, the approach here described is much simpler, with the view of facilitating the formalisation.

4.3 Our Work in Progress on the Formalisation

4.3.1 Main Theorems That Must Be Proved

The Corollaries 3.1 and 3.2 must be proved again to guarantee correctness and completeness of the nominal AC-unification algorithm. As before, they are easy consequences of Theorems 3.1 and 3.2, which must be reproved. In relation to the formalisation of C-unification, our only work is extending those proofs to the case where the terms t and s in the head of $PrbLst$ are AC functions. In this case, there are no changes in the context nor in the substitution, so it is trivial to check that we remain with a valid quadruple and it is also trivial to check all but the third condition of Definition 2.26.

Algorithm 2 Functional Nominal AC-Unification

```

1: procedure UNIFY( $\Delta, \sigma, PrbLst, FPEqLst$ )
2:   if null( $PrbLst$ ) then
3:     return list( $(\Delta, \sigma, FPEqLst)$ )
4:   else
5:      $(t, s) + PrbLst' = PrbLst$ 
6:     if  $(s == \pi \cdot X)$  and  $(X$  not in  $t)$  then
7:       see Algorithm 1
8:     else
9:       if  $t == a$  then
10:        see Algorithm 1
11:       else if  $t == \pi \cdot X$  then
12:        see Algorithm 1
13:       else if  $t == \langle \rangle$  then
14:        see Algorithm 1
15:       else if  $t == \langle t_1, t_2 \rangle$  then
16:        see Algorithm 1
17:       else if  $t == [a]t_1$  then
18:        see Algorithm 1
19:       else if  $t == f t_1$  then
20:        see Algorithm 1
21:       else if  $t == f^C \langle t_1, t_2 \rangle$  then
22:        see Algorithm 1
23:       else  $\triangleright t$  is of the form  $f^{AC}t'$ 
24:         if  $s \neq f^{AC} s'$  then return null
25:         else
26:            $NewPrbLst = \text{gen\_unif\_prb}(t, s)$ 
27:            $LstQuad = \text{get\_lst\_quad}(NewPrb, \Delta, \sigma, PrbLst', FPEqLst)$ 
28:            $LstLstSol = \text{map}(\text{unify}, LstQuad)$ 
29:           return FLATTEN( $LstLstSol$ )
30:         end if
31:       end if
32:     end if
33:   end if
34: end procedure

```

Regarding the third condition, Theorems 4.1 and 4.2 must be formalised. As mentioned before, the function $\text{gen_unif_prb}(ft, fs)$, for f AC, generates all pairings of ft and fs (in any order) and then combines them to generate a list of unification problems.

Theorem 4.1 (Soundness of Unifying AC functions). Let ft and fs be AC function applications. Suppose that $(t_1, s_1) \in \text{gen_unif_prb}(ft, fs)$ and that $\nabla \vdash t_1 \sigma \approx_\alpha s_1 \sigma$. Then, $\nabla \vdash (ft)\sigma \approx_\alpha (fs)\sigma$.

Theorem 4.2 (Completeness of Unifying AC functions). Let ft and fs be AC function applications. Suppose that $\nabla \vdash (ft)\sigma \approx_\alpha (fs)\sigma$. Then, there exists $(t_1, s_1) \in \text{gen_unif_prb}(ft, fs)$ such that $\nabla \vdash t_1 \sigma \approx_\alpha s_1 \sigma$.

4.3.2 A Plan to the Proof

The natural way of proving the theorem of soundness of unifying AC functions would be by induction on the size of the term. If we had decided to prove the theorem directly, we would find the i that makes $\nabla \vdash S_1((ft)\sigma) \approx_\alpha S_i((fs)\sigma)$ and then try to use the induction hypothesis to prove that $\nabla \vdash D_1((ft)\sigma) \approx_\alpha D_i((fs)\sigma)$.

This straightforward approach, however, does not work. We would not be able to apply the induction hypothesis, since the term being deleted of $t\sigma$ could be the first term of t but it could have also being introduced by the substitution σ . A similar problem could happen for s .

To get around this problem, we must first eliminate the substitutions from the problem and only then solve a version of the problem without substitutions, by induction. As explained next, Lemmas 4.1 and 4.2, together with a convenient renaming of variables, eliminate the substitution, while Lemma 4.3 solves a simplified version of the problem.

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

Lemma 4.1. Let ft and fs be AC applications. Suppose that $(t_1, s_1) \in \text{gen_unif_prb}(ft, fs)$. Then, $\forall t'_1 \in F_{AO}(t_1 \sigma), s'_1 \in F_{AO}(s_1 \sigma): (t'_1, s'_1) \in \text{gen_unif_prb}(ft\sigma, fs\sigma)$.

Remark 4.5. The operator F_{AO} is needed in the Lemma 4.1 because, although we have the guarantee that t_1 and s_1 are pairings of ft and fs (and therefore do not contain the AC function symbol f), the substitution σ may reintroduce f into the terms $t_1 \sigma$ and $s_1 \sigma$. Since an output of $\text{gen_unif_prb}((ft)\sigma, (fs)\sigma)$ cannot contain the AC function symbol, we must apply the flattener operator F_{AO} to $t_1 \sigma$ and $s_1 \sigma$. A case where this reintroduction of the AC function symbol occurs is illustrated in Example 4.2.

Example 4.2. Let f be an AC function symbol. Consider $ft = fX$, $fs = fY$ and $\sigma = \{X \rightarrow f\langle a, b \rangle, Y \rightarrow f\langle b, a \rangle\}$. Then, $t_1 = X$ and $s_1 = Y$ do not indeed contain the AC function symbol f . However, the substitution σ given reintroduces this AC function symbol and we have $t_1\sigma = f\langle a, b \rangle$ and $s_1\sigma = f\langle b, a \rangle$.

Lemma 4.2. Let ft and fs be AC applications. Suppose that $(t_1, s_1) \in \text{gen_unif_prb}(ft, fs)$ and that $\nabla \vdash t_1\sigma \approx_\alpha s_1\sigma$. Then, $\exists t'_1 \in F_{AO}(t_1\sigma), s'_1 \in F_{AO}(s_1\sigma) : \nabla \vdash t'_1 \approx_\alpha s'_1$.

Using Lemmas 4.1 and 4.2 we can obtain, from our original hypothesis of soundness, the existence of t'_1 and s'_1 such that $(t'_1, s'_1) \in \text{gen_unif_prb}((ft)\sigma, (fs)\sigma)$ and $\nabla \vdash t'_1 \approx_\alpha s'_1$ and we must prove that $\nabla \vdash (ft)\sigma \approx_\alpha (fs)\sigma$. Then, with a convenient renaming of variables, this is equivalent to proving Lemma 4.3, which can be done by induction on the size of the term t .

Lemma 4.3. Let ft and fs be AC function applications, with the same function symbol. Suppose that $(t_1, s_1) \in \text{gen_unif_prb}(ft, fs)$ and that $\nabla \vdash t_1 \approx_\alpha s_1$. Then, $\nabla \vdash ft \approx_\alpha fs$.

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

Remark 4.6. As mentioned in Remark 4.3, our initial approach to the problem was to generate the pairings of the term t (when trying to unify t and s) preserving order. In that previous version of the specification, we have formalised Lemma 4.3 with a proof by induction on the size of ft . We have changed the specification of the algorithm to generate all pairings of t , in any order and we believe a proof by induction on the size of t would still work. After this change on the specification, not one of the lemmas enunciated in this chapter has been fully proved.

4.3.3 A Loose End

Let f be an AC function symbol and imagine we are trying to unify $f\langle X, b \rangle$ with $f\langle a, Y \rangle$. After generating all the pairings and combining them, the only solution our algorithm would find is $\langle \Delta, \sigma, FPEqLst \rangle = \langle \emptyset, \{X \rightarrow a, Y \rightarrow b\}, \emptyset \rangle$. Nevertheless, the solution $\langle \Delta, \sigma, FPEqLst \rangle = \langle \emptyset, \{X \rightarrow f\langle a, U \rangle, Y \rightarrow f\langle b, U \rangle\}, \emptyset \rangle$ is also correct and should have been found. This proves the current algorithm is not complete and should be adapted to find solutions like this.

It appears that our simple but inefficient approach of generating all the pairings is inherently incomplete. Other approaches, for instance the translation of the problem of AC-unification to the one of solving a system of diophantine equations (as seen in Example 2.8), should be considered. The simplicity of potential approaches should also be considered, to make the formalisation easier. As indicated in Chapter 5, this is one possible path of future work.

4.3.4 Interesting Discoveries

Interesting facts found during our work in progress in formalising the algorithm are enumerated below:

1. Prior to the introduction of AC function applications, α -equivalence was a size preserving relation, meaning that if $t \approx_\alpha s$ then $size(t) = size(s)$. However, according to the grammar of the nominal setting, our rule for α -equivalence of AC function applications and the definition of the operators $S_n(f^*)$ and $D_n(f^*)$, this ceases to be the case when AC function symbols are introduced. To see that, let f be an AC function symbol and consider, for instance, $t = f(fa)$ and $s = fa$. Then, t and s are α -equivalent, since $S_1(t) = S_1(f(fa)) = a \approx_\alpha a = S_1(fa) = S_1(s)$ and $D_1(t) = D_1(f(fa)) = \langle \rangle \approx_\alpha \langle \rangle = D_1(fa) = D_1(s)$ but the size of t is greater than the size of s , as there is one more occurrence of the AC function symbol f in t than there is in s . The same phenomenon happens for the depth of a term: α -equivalence ceases to be a depth preserving relation.
2. α -equivalence is an equivalent relation. This is formalised by proving that α -equivalence is reflexive, symmetric and transitive. During the proofs of symmetry and transitivity it was used that α -equivalence preserved size. Hence, these proofs had to be modified in order to avoid using this fact, which is no more valid. The modified proofs were not hard and can be found on file `alpha.prf`.
3. Imagine t and s are terms that do not contain any AC function symbol, in any position. Then, in an initial analysis, from the hypothesis that $S_1(t) \approx_\alpha S_1(s)$ (i.e. the first argument of t is α -equivalent to the first argument of s) and $D_1(t) \approx_\alpha D_1(s)$ (i.e. after deleting the first argument of t and deleting the first argument of s the terms are α -equivalent) it appears that we should be able to conclude that t and s are α -equivalent. This however is not true, as shown by the following counterexample. Let $t = \langle a, \langle b, c \rangle \rangle$ and $s = \langle \langle a, b \rangle, c \rangle$. Then, t and s are not α -equivalent, but $S_1(t) = a \approx_\alpha a = S_1(s)$ and $D_1(t) = \langle b, c \rangle \approx_\alpha \langle b, c \rangle = D_1(s)$ are indeed α -equivalent.

4.4 Possible Applications

The possible applications of a nominal AC-unification algorithm are analogous to the ones of a nominal C-unification algorithm, which are discussed on Section 3.6.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This work presented contributions to nominal unification with C and AC function symbols. Regarding nominal C-unification, a functional algorithm was specified and proven to be correct and complete using PVS and then implemented in Python. In relation to the work of [ARdCSFNS18], there are two important differences. The first is the specification of an indeed functional algorithm that can be directly executed, not a set of non-deterministic rules. The second is the reduction on the lexicographic measure used in the proofs of soundness, completeness and termination, from four parameters to only two. The algorithm could be used in logic programming or be adapted to a matching algorithm, with potential applications in nominal rewriting and nominal narrowing.

Regarding nominal AC-unification, we presented our work in progress to extend the nominal C-unification algorithm to handle AC function symbols as well. The strategy adopted was explained, the problem that remains unsolved was discussed and interesting points found so far during our efforts in the formalisation were presented.

5.2 Future Work

The most urgent future work would be finishing the AC-unification algorithm formalisation, proving its correctness and completeness. After that, a possible path is extending the unification algorithm to handle other equational properties such as associativity, neutral element, idempotence and distributivity.

A future work could be done in the area of complexity of algorithms. Although it is known that the problem of C-unification is NP-complete, one could restrict the set of inputs for the problem and investigate the complexity of the algorithm for this restricted set.

In [\[ARdCSFNS17b\]](#), a method for enumerating all the solutions to fixed point equations is detailed. Another possible future work is formalising that the method proposed in [\[ARdCSFNS17b\]](#) is correct and complete.

References

- [AFN16] Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal Narrowing. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD)*, volume 52 of *LIPICs*, pages 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
- [AFN19] Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. On nominal syntax and permutation fixed points. *CoRR*, abs/1902.08345, 2019.
- [ARdCSFNS17a] Mauricio Ayala-Rincón, Washington de Carvalho-Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. A Formalisation of Nominal α -equivalence with A and AC Function Symbols. *Electronic Notes in Theoretical Computer Science*, 332:21–38, 2017.
- [ARdCSFNS17b] Mauricio Ayala-Rincón, Washington de Carvalho-Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. On Solving Nominal Fixpoint Equations. In *11th International Symposium on Frontiers of Combining Systems (FroCoS)*, volume 10483 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2017.
- [ARdCSFNS18] Mauricio Ayala-Rincón, Washington de Carvalho-Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal C-Unification. In *27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017), Revised Selected Papers*, volume 10855 of *Lecture Notes in Computer Science*, pages 235–251. Springer, 2018.
- [ARDM17] Mauricio Ayala-Rincón and Flávio LC De Moura. *Applied Logic for Computer Scientists: Computational Deduction and Formal Proofs*. Springer, 2017.
- [ARFFS19] Mauricio Ayala-Rincón, Maribel Fernández, and Gabriel Ferreira Silva. Formalising Nominal AC-Unification. 2019. Presented in the International Workshop on Unification UNIF 2919.
- [ARFFSNS19] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. A Certified Functional Nominal C-Unification Algorithm. Available at <http://ayala.mat.unb.br/publications.html>, 2019.
- [ARFNS18] Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Fixed-Point Constraints for Nominal Equational Unification. In *3rd International Conference on Formal Structures for Computation and Deduction*

- (*FSCD*), volume 108 of *LIPICs*, pages 7:1–7:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.
- [ARFRO16] Mauricio Ayala-Rincón, Maribel Fernández, and Ana Rocha-Oliveira. Completeness in PVS of a nominal unification algorithm. *Electronic Notes in Theoretical Computer Science*, 323:57–74, 2016.
- [Baa89] Franz Baader. Characterization of Unification Type Zero. In *Rewriting Techniques and Applications, 3rd International Conference, RTA-89, Chapel Hill, North Carolina, USA, April 3-5, 1989, Proceedings*, pages 2–14, 1989.
- [BBC⁺97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*, 1997.
- [BKL15] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal anti-unification. In *26th International Conference on Rewriting Techniques and Applications (RTA)*, volume 36 of *LIPICs*, pages 57–73. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- [BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.
- [BS01] Franz Baader and Wayne Snyder. Unification Theory. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001.
- [CF08] Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theoretical Computer Science*, 403(2-3):285–306, 2008.
- [CM17] James Cheney and Alberto Momigliano. α Check: A mechanized metatheory model checker. *TPLP*, 17(3):311–352, 2017.
- [Com93] Hubert Comon. Complete Axiomatizations of Some Quotient Term Algebras. *Theoretical Computer Science*, 118(2):167–191, 1993.
- [Con04] Evelyne Contejean. A Certified AC Matching Algorithm. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.
- [CU04] James Cheney and Christian Urban. α -Prolog: A logic programming language with names, binding and α -equivalence. In *20th International Conference on Logic Programming (ICLP)*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004.
- [dCS19] Washington Luis Ribeiro de Carvalho Segundo. *Nominal Equational Problems Modulo Associativity, Commutativity and Associativity-Commutativity*. PhD thesis, Universidade de Brasilia, 2019.

- [dMKA⁺15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.
- [Fag87] François Fages. Associative-Commutative Unification. *J. Symb. Comput.*, 3(3):257–275, 1987.
- [FG07] Maribel Fernández and Murdoch Gabbay. Nominal rewriting. *Information and Computation*, 205(6):917–965, 2007.
- [For87] Albrecht Fortenbacher. An Algebraic Approach to Unification Under Associativity and Commutativity. *J. Symb. Comput.*, 3(3):217–229, 1987.
- [KKN85] Abdelilah Kandri-Rody, Deepak Kapur, and Paliath Narendran. An Ideal-Theoretic Approach to Work Problems and Unification Problems over Finitely Presented Commutative Algebras. In *Rewriting Techniques and Applications, First International Conference, RTA-85, Dijon, France, May 20-22, 1985, Proceedings*, pages 345–364, 1985.
- [KN87] Deepak Kapur and Paliath Narendran. Matching, unification and complexity. *ACM SIGSAM Bulletin*, 21(4):6–9, 1987.
- [KN92] Deepak Kapur and Paliath Narendran. Complexity of Unification Problems with Associative-Commutative Operators. *Journal of Automated Reasoning*, 9(2):261–288, 1992.
- [KN10] Ramana Kumar and Michael Norrish. (Nominal) Unification by Recursive Descent with Triangular Substitutions. In *First International Conference on Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2010.
- [LV08] Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In *19th International Conference on Rewriting Techniques and Applications (RTA)*, volume 5117 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2008.
- [LV10] Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA)*, volume 6 of *LIPICs*, pages 209–226. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.
- [MM82] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.
- [NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
- [OS99] Sam Owre and Natarajan Shankar. The formal semantics of PVS. 1999.

- [OSRSC01] Sam Owre, Natarajan Shankar, John M Rushby, and David W. J. Stringer-Calvert. *PVS System Guide - Version 2.4*, 2001.
- [Pit13] Andrew Pitts. *Nominal sets: names and symmetry in computer science*. Cambridge University Press, 2013.
- [SORSC01] Natarajan Shankar, Sam Owre, John M Rushby, and Dave WJ Stringer-Calvert. PVS prover guide. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1:11–12, 2001.
- [SSKLV17] Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal unification of higher order expressions with recursive let. In *26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016), Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, pages 328–344. Springer, Springer, 2017.
- [Sti81] Mark E. Stickel. A Unification Algorithm for Associative-Commutative Functions. *J. ACM*, 28(3):423–434, 1981.
- [TS00] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic proof theory*. Number 43. Cambridge University Press, 2000.
- [UPG04] Christian Urban, Andrew Pitts, and Murdoch Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.
- [Urb08] Christian Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

Appendix A

The PVS Theory for Nominal C-Unification

In this appendix, we comment on some fragments of the PVS specification and formalisation. As commented before, the PVS code can be found on <http://www.github.com/gabriel951/c-unification>.

A.1 Fragments of the Specification

A.1.1 Defining a Nominal Term in PVS

It was necessary to define a subtype `pair`, in order to guarantee that the argument of a commutative function is a pair. Nevertheless, when working with subtypes in PVS it is required that every term has a subtype. Hence, a subtype `plain` was also defined, and every term that is not a pair has subtype `plain`.

```

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

```

Remark A.1. In the specification of a nominal term in PVS, by defining term as a datatype, it is assumed that all terms must have one of the types given. In other words, all terms are either an atom, or a permutation, or the unit, or an abstraction, or a function application or a commutative function application or an associative-commutative function application.

A.1.2 Specification of the Main Functions in the Lexicographic Measure

To calculate the lexicographic measure, the main functions are the Vars function and the size function. The first calculates the set of variables in a term t , while the second calculates the size of a term t . The specification of the Vars function is:

```

Vars(t): RECURSIVE finite_set[variable] =
  CASES t OF
    at(a): emptyset,
    *(pm,v): singleton(v),
    unit: emptyset,
    pair(t1,t2): union(Vars(t1),Vars(t2)),
    abs(a,bd): Vars(bd),
    app(s1,ag): Vars(ag),
    c_app(s1, ag): Vars(ag)
  ENDCASES
  MEASURE t BY <<

```

While the specification of the `size` function is:

```
size(t): RECURSIVE nat =
  CASES t OF
    at(a): 1,
    *(pm,v): 1,
    unit: 1,
    pair(t1,t2): 1 + size(t1) + size(t2),
    abs(a,bd): 1 + size(bd),
    app(s1,ag): 1 + size(ag),
    c_app(s1, ag): 1 + size(ag)
  ENDCASES
  MEASURE t BY <<
```

A.2 A Fragment of the Formalisation

We now present a fragment of the formalisation. One of the TCCs generated by the algorithm ask us to prove that in the recursive call of the case $s = \pi \cdot X$ and X not in t (i.e., the recursive call of Algorithm 1 in line 12), the lexicographic measure decreases.

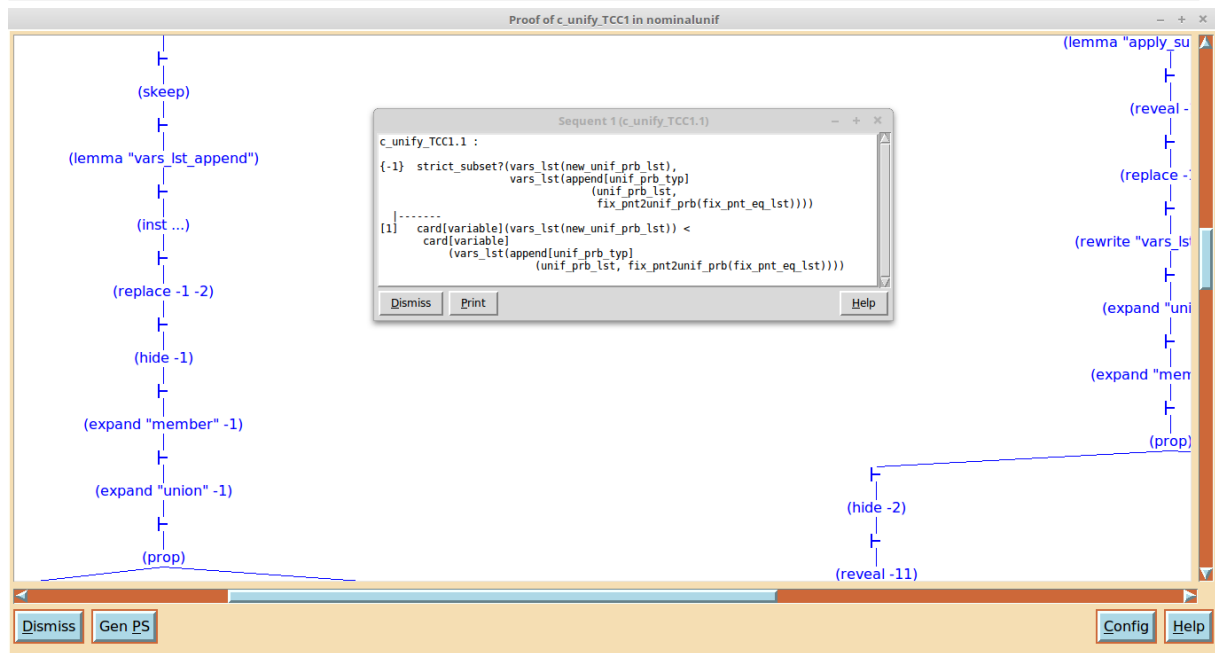
This happens since the cardinality of $\text{Vars}(\text{PrbLst}) \cup \text{Vars}(\text{FPEqLst})$ is reduced. Since the variable X , previously present in s , is now in the domain of the substitution σ'' (see line 8 of Algorithm 1), by the Definition 3.3 of valid quadruples (once we prove that we remain with a valid quadruple) this variable will not be in PrbLst'' or in null (see line 12 of Algorithm 1) and we will have $|\text{Vars}(\text{PrbLst}'') \cup \text{Vars}(\text{null})| < |\text{Vars}(\text{PrbLst}) \cup \text{Vars}(\text{FPEqLst})|$.

Figure A.1 shows a screenshot of one sequent of the proof (and in the background part of the tree of the commands used to prove the TCC).

Additionally, Figure A.2 contains just the sequent (the same sequent of Figure A.1). It is possible to see that we have, as our hypothesis, that $\text{Vars}(\text{new_unif_prb_lst})$ is a strict subset of $\text{Vars}(\text{unif_prb_lst}) \cup \text{Vars}(\text{fix_pnt_eq_lst})$ and we must prove that $|\text{Vars}(\text{new_unif_prb_lst})| < |\text{Vars}(\text{unif_prb_lst}) \cup \text{Vars}(\text{fix_pnt_eq_lst})|$. Comparing Figure A.2 and the pseudocode of Algorithm 1 we have the following correspondences:

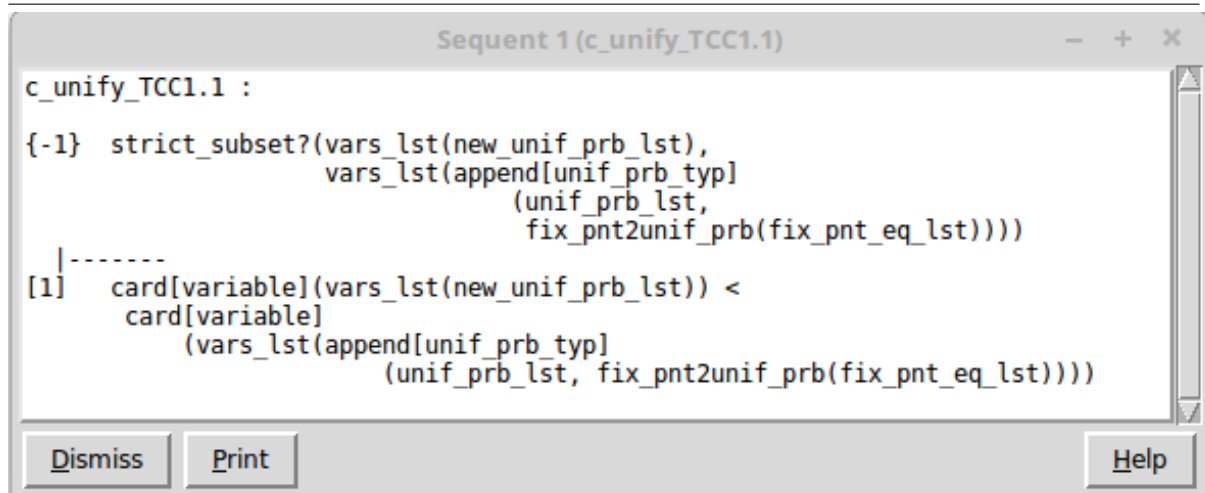
- new_unif_prb_lst corresponds to PrbLst'' .
- unif_prb_lst corresponds to PrbLst .
- fix_pnt_eq_lst corresponds to FPEqLst .

Fig. A.1 One of the Sequents in a Proof of a TCC. In the Back, Part of the Tree of Commands Used in the Proof.



Remark A.2. The function `fix_pnt2unif_prb` simply converts a fixed point equation to an unification problem. This conversion is not important here.

This part of the proof has been done with the use of lemma `card_strict_subset`, available in the prelude file of PVS.

Fig. A.2 One of the Sequents of a Proof of a TCC.

```
Sequent 1 (c_unify_TCC1.1)
c_unify_TCC1.1 :
{-1} strict_subset?(vars_lst(new_unif_prb_lst),
                    vars_lst(append[unif_prb_typ]
                              (unif_prb_lst,
                               fix_pnt2unif_prb(fix_pnt_eq_lst))))
|-----
[1]  card[variable](vars_lst(new_unif_prb_lst)) <
     card[variable]
       (vars_lst(append[unif_prb_typ]
                    (unif_prb_lst, fix_pnt2unif_prb(fix_pnt_eq_lst))))

Dismiss Print Help
```


Appendix B

Fragments of The Python Code For Nominal C-Unification

In this appendix, we give some fragments of the Python implementation of the algorithm here presented for nominal C-unification. As commented before, the Python code can be found at <http://www.github.com/gabriel951/c-unification>.

B.1 The Classes

As mentioned previously, terms are represented as objects of a given class, and there is one class for atoms, one class for suspended variables and so on. Below, we present the code for the Atom class. An atom object has only one attribute, its name. The `__init__` method is called when creating the atom and is responsible for correctly setting the name of the atom (via the attribute name, which should be a string). Finally, the `__str__` method is used when printing an atom object and returns the attribute name.

```
class Atom:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name
```

The code below creates an atom with name *a* and binds the variable `atom_a` to it:

```
atom_a = Atom("a")
```

In the code above, the Python variable `atom_a` is an instance of the Atom class and its only attribute (`name`) is equal to the string `a`.

Next, we present the code for the class `CApplication`. Notice that a `CApplication` object has two attributes: `symbol` (a string to represent the function symbol) and `arg` (a term that corresponds to the argument of the function). Since commutative function applications receive a pair as their argument, we put an assertion in the code forcing the attribute `arg` to be an instance of the `Pair` class. Finally, the `__str__` method correctly prints a commutative function application.

```
class CApplication:
    def __init__(self, symbol, arg):
        assert(isinstance(arg, Pair))
        self.symbol = symbol
        self.arg = arg

    def __str__(self):
        return self.symbol + "(" + (self.arg).__str__() + ")"
```

B.2 Fragments of the Unify Function

The initial fragment of the unify function is given below (this fragment corresponds to lines 1 to 5 of the pseudocode given in Algorithm 1). The parameters `Delta`, `sigma`, `PrbLst` and `FPEqLst` are respectively the context, the substitution, the list of unification problems that remain and the list of fixed point equations. The parameters `verb` and `indent_lvl` are respectively the boolean that indicates whether the algorithm should run in verbose mode or not and a string that regulates the indentation level. They are optional parameters and have as default value `False` and `""` (empty string) respectively. The function `print_quad` prints the quadruple $(\Delta, \sigma, PrbLst, FPEqLst)$ the algorithm receives, with the appropriate indentation and the function `print_sol` prints the solution the algorithm found.

```
def unify(Delta, sigma, PrbLst, FPEqLst, verb=False,
          indent_lvl=""):
    if verb:
        print_quad(Delta, sigma, PrbLst, FPEqLst,
                  indent_lvl)

    if len(PrbLst) == 0:
        if verb:
            print(indent_lvl, end = '')
            print_sol([(Delta, sigma, FPEqLst)])
            print("\n")
        return [(Delta, sigma, FPEqLst)]
    else:
        (t, s) = PrbLst[0]
        PrbLst1 = PrbLst[1:]
```

Next, we present the fragment for the case where the terms t and s in the head of the list are commutative function applications. This code corresponds to lines 61 to 70 of the Algorithm 1. Notice that in the recursive calls correspondent to exploring the two branches introduced by commutativity, the variable `new_indent_lvl` is used. `new_indent_lvl` is defined to be `indent_lvl` (our original indentation) plus four empty spaces, to correctly give the impression that the algorithm has entered a branch.

```
elif isinstance(t, CApplication):
    if (not isinstance(s, CApplication))
        or (t.symbol != s.symbol):
        if verb:
            print(indent_lvl + "No solution\n")
        return []
    else:
        new_indent_lvl = indent_lvl + "  "
        PrbLst_branch1 = [((t.arg).term1, (s.arg).term1)]+
            [((t.arg).term2, (s.arg).term2)]+
            PrbLst1
        sol1 = unify(Delta, sigma, PrbLst_branch1, FPEqLst,
            verb, new_indent_lvl)
        PrbLst_branch2 = [((t.arg).term1, (s.arg).term2)]+
            [((t.arg).term2, (s.arg).term1)]+
            PrbLst1
        sol2 = unify(Delta, sigma, PrbLst_branch2, FPEqLst,
            verb, new_indent_lvl)
        return sol1 + sol2
```