Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Towards Nominal AC-Unification
(Rumo à AC-Unificação Nominal)

Gabriel Ferreira Silva

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Orientador
Prof. Dr. Mauricio Ayala-Rincón

Coorientadora
Prof. Dr. Maribel Fernández

Brasília
2024

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Towards Nominal AC-Unification
(Rumo à AC-Unificação Nominal)

Gabriel Ferreira Silva

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Brasília, 26 de Janeiro de 2024

To my family and friends, you make life worth living.

Acknowledgements

I thank God for all the blessings I have received, among them health, disposition to work and great family/friends.

I thank my mom, my dad and my sister for the amazing support and love I have received during my whole life. You are incredible and the best family I could ask for! Love you!

My grandfather João was always in an optimistic and happy mood, even though he had health problems. My uncle Apu was always there for my family. My grandmother and godmother Gesi was full of wisdom and love. João, Apu and Gesi passed away during my masters/PhD studies, but have inspired me to finish the PhD and are always in my prayers.

I have a lot to thank my family from Brasília and from Patos de Minas for, including the support and the understanding when would I missed/left early a family reunion to study. You are great.

I thank my childhood and best friends! I had amazing and fun moments with my mates Alexander, Manno, Fugi and Felipe while playing videogames, eating fast-food and “boys talking”. And I know we will keep having more great moments in the future!

I thank my superb advisor Ayala for all the help during my master’s and PhD studies! You are the best advisor I could have asked for. Thanks for believing in me, accepting to supervise me even after I failed the analysis exam, and sharing my excitement about the research we did together. I learned a lot during these years and had fun while doing that.

I thank my coadvisor Maribel Fernández for all the help and great feedback that started in my master’s studies and continued during my PhD! Daniele Nantes is a complete professor who researches well, teaches with clarity and kindness and attracts new students to graduate studies. Thanks Daniele for the help in the papers we wrote together and for being a role model of what a professor should be like! I thank Temur Kutsia for the great feedback in the papers we wrote together.

I thank José Meseguer, Christian Urban, César Muñoz and Vander Alves for accepting our invitation and forming the jury of my PhD defense. I also appreciate their valuable feedback that improved this work.

I thank my friends from the group “Muçar? Café? Cota?” for the time we spent together during my PhD. It was great spending time with people who also wanted to finish graduate studies and were willing to work really hard for it. It was a pleasure meeting you! I thank specially MEGS, Mateus, Tharles, Talita, PP, Caio, Gabriel, Gabi, Ismael and Flávia.

I thank the group “Os Vingadores do Jantar”! My friendship with Sensei Santiago, Thiago, Pavel, Geovane and Ariane started in the MsC and continued through the PhD. Our light-hearted conversations made my graduate studies even more pleasant.

Some Saturday mornings were well-spent playing soccer with my dad and other friends. I thank the group “Pelada Millenium” for these soccer matches.

It was great to play board games or chat about academic life (its perks and drawbacks) with MEGS, Gabriel Bufolo and Georginho. Thank you “GGGM - Dr.”!

During the pandemic and also after it, I kept talking with some friends I made during my master’s studies. I thank my friends from the group “Os Tops da Sala Top”: MEGS, Mateus, Tharles, Geovane, Adler, Murilo. It was great having your support and friendship during those tough times and after them!

By the end of my PhD I was lucky to find a job as a data scientists with excellent colleagues at TJDFT. I thank my friends at COCID for their encouragement!

I thank Maximal (Deivid Vale) and the colleagues from the group “Ga^{2}khanda do Mau”: Gabi, Andrés, Santaguida, Ali and Guilherme.

I thank the researchers from NASA formal methods/SRI international for the support whenever I had doubts about PVS. Mariano Moscato helped me a lot, for instance when he explained how I could use a Perl script in NASALib to generate the hierarchy of the whole nominal library. Thanks Mariano! César Muñoz helped us with PVSio and also pointed that we could combine the use of PVSio with semantic attachments. Thanks César!

As part of the CICM 2022 doctoral programme, David Cerna kindly accepted being my mentor. In addition to nice tips on my current research he also gave me insights into the academic life after the PhD and its work/life balance. Thanks David!

According to the rules of the computer science department, I had to publish a journal paper as a prerequisite for defending the PhD. The journal paper we published is an extended version of a previous paper to a special issue of Mathematical Structures in Computer Science (MSCS). I thank Sandra Alves and Renata Wassermann for the invitation to submit this extended version!

I thank Leslie Lamport for inventing a better way (in my opinion) to prove things in mathematics, with his seminal work on structured proofs.

I thank the math department at UnB for offering me a temporary (one semester) scholarship. After this one semester in the math department, the computer science department at UnB offered me a permanent scholarship that allowed me to focus on my research. I thank the computer science department and the CAPES funding agency for this scholarship!

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

Resumo

O paradigma nominal estende a sintaxe de primeira ordem e representa adequadamente o conceito de variáveis ligadas. Para trabalhar com esse vantajoso paradigma faz-se necessário adaptar noções de primeira ordem a ele, como unificação e *matching*. Esta tese é sobre unificação e *matching* no paradigma nominal na presença de uma teoria equacional E e sobre nosso trabalho em progresso em AC-unificação nominal. Inicialmente, generalizamos e formalizamos um algoritmo de C-unificação nominal para realizar *matching* e *equality-checking*, através da adição de um parâmetro $\mathcal{X}$ para lidar com variáveis protegidas. A formalização foi usada para testar uma implementação manual em Python do algoritmo. Em seguida, fornecemos a primeira formalização de um algoritmo de AC-unificação em primeira ordem. Escolhemos formalizar o algoritmo seminal de Stickel e na prova de terminação usamos uma intrincada (mas devidamente motivada) medida lexicográfica, baseada no trabalho de Fages. Depois disso, adaptamos este algoritmo para obter o primeiro algoritmo para AC-*matching* em nominal e verificamos que o algoritmo termina e é correto e completo. Assim como em C-unificação nominal, usamos um parâmetro $\mathcal{X}$ para as variáveis protegidas, o que nos permitiu obter um AC-*equality-checker* como corolário. As 3 formalizações descritas foram feitas no assistente de provas PVS e integram a NASALib, o principal repositório de formalizações do PVS. Para cada uma dessas formalizações descrevemos a estrutura e tamanho dos arquivos que compõem a formalização. Visando obter um algoritmo de AC-unificação nominal, mostramos que o problema tem duas questões interessantes associadas a ele: gerar as soluções para $\pi \cdot X \approx^? X$ e demonstrar terminação. Para a primeira questão, propomos um procedimento não determinístico de enumeração e exemplificamos como este calcula soluções não triviais. Para a segunda questão demonstramos como o problema $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$ gera um *loop* e provamos que é suficiente "entrar no *loop*" uma quantidade limitada de vezes, onde esse limite depende da ordem da permutação π. Acreditamos que a teoria desenvolvida será útil para a formulação de um algoritmo de AC-unificação nominal.

Palavras-chave: Nominal, Métodos Formais, PVS, C-Unificação Nominal, AC-Unificação, AC-Matching Nominal, AC-Unificação Nominal.

Resumo Extendido

O paradigma nominal estende a sintaxe de primeira ordem e representa adequadamente o conceito de variáveis ligadas. Para trabalhar com esse vantajoso paradigma faz-se necessário adaptar noções de primeira ordem a ele, como unificação e *matching*. Esta tese é sobre unificação e *matching* no paradigma nominal na presença de uma teoria equacional E. Além disso abordamos nosso trabalho em progresso em AC-unificação nominal.

Inicialmente, generalizamos e formalizamos um algoritmo de C-unificação nominal para realizar *matching* e *equality-checking*, através da adição de um parâmetro $\mathcal{X}$ para lidar com variáveis protegidas, i.e. variáveis que não podem ser instanciadas. Assim, dado um problema P, pode-se realizar unificação/*matching*/*equality-checking* colocando os respectivos valores para o parâmetro $\mathcal{X}$: $\emptyset$, $\mathit{Vars}(\mathit{rhs}(P))$ e $Vars(P)$. A formalização foi usada para testar (através da ferramenta *PVSio*) uma implementação manual em Python do algoritmo.

Em seguida, fornecemos a primeira formalização de um algoritmo de AC-unificação em primeira ordem. Escolhemos formalizar o algoritmo seminal de Stickel e na prova de terminação usamos uma intrincada (mas devidamente motivada) medida lexicográfica, baseada no trabalho de Fages. Além de terminação, descrevemos as provas de corretude e completude, destacando os seus pontos mais intricados.

Depois disso, adaptamos este algoritmo para obter o primeiro algoritmo para AC-*matching* em nominal e verificamos que o algoritmo termina e é correto e completo. Assim como em C-unificação nominal, usamos um parâmetro $\mathcal{X}$ para as variáveis protegidas, o que nos permitiu obter também um AC-*equality-checker* como corolário.

As 3 formalizações descritas foram feitas no assistente de provas PVS e integram a NASALib, o principal repositório de formalizações do PVS. Para cada uma dessas formalizações listamos o tamanho dos arquivos que compõem a formalização em tabelas e detalhamos a hierarquia entre os arquivos em figuras.

Visando obter um algoritmo de AC-unificação nominal, mostramos que o problema tem duas questões interessantes associadas a ele: gerar as soluções para equações de ponto fixo $\pi{\cdot}X \approx^? X$ e demonstrar terminação. Para a primeira questão, propomos um procedimento não determinístico de enumeração e exemplificamos como este calcula soluções não triviais.

Para a segunda questão demonstramos como o problema $f(X, W) \approx^? f(\pi{\cdot}X, \pi{\cdot}Y)$ gera um *loop* e provamos que é suficiente “entrar no *loop*” uma quantidade limitada de vezes, onde esse limite depende da ordem da permutação π. Realizamos também uma investigação preliminar sobre a conexão entre nominal e *higher-order patterns*, visto que o problema de unificação em *higher-order patterns* já se encontra resolvido. Acreditamos que a teoria desenvolvida será útil para a formulação de um algoritmo de AC-unificação nominal.

Detalhamos a seguir a organização deste trabalho. O Capítulo 1 motiva o tópico e sumariza as contribuições. O Capítulo 2 fornece a base necessária para ler a tese, introduzindo conceitos do paradigma nominal, de AC-unificação em primeira ordem e do provador de teoremas PVS. Depois disso, o Capítulo 3 explica o algoritmo de C-unificação nominal generalizado com variáveis protegidas. Os três capítulos seguintes abordam raciocínio equacional na presença de símbolos de função AC. Primeiramente, o Capítulo 4 reporta a formalização de AC-unificação em primeira ordem e suas aplicações. Depois, o Capítulo 5 mostra como adaptamos a formalização de primeira ordem para nominal e obtivemos um algoritmo verificado para AC-*matching* nominal. O Capítulo 6 discute o nosso trabalho em progresso rumo à AC-unificação nominal. Por fim, os Capítulos 7 e 8 descrevem trabalhos correlatos e apontam direções para trabalho futuro. Neste trabalho incluímos *hyperlinks* coloridos em azul-claro (com o logo ↗) para os pontos de interesse da formalização em PVS.

Palavras-chave: Nominal, Métodos Formais, PVS, C-Unificação Nominal, AC-Unificação, AC-Matching Nominal, AC-Unificação Nominal.

Abstract

The nominal syntax extends first-order syntax and allows us to represent smoothly system with bindings. In order to profit from the nominal setting, we must adapt important notions to it, such as unification and matching. This thesis is about nominal unification/-matching in the presence of an equational theory E and our efforts towards obtaining a nominal AC-unification algorithm. First, we extend and formalise a nominal C-unification algorithm to also handle matching and equality checking by adding an extra parameter $\mathcal{X}$ for protected variables, i.e., variables that cannot be instantiated. The formalised algorithm is used to test a Python manual implementation of the algorithm. Then, as a first step towards nominal AC-unification, we give the first formalisation of a first-order AC-unification algorithm. We choose to verify Stickel's tried-and-tested algorithm. The proof of termination employs an intricate (but duly motivated) lexicographic measure that is based on Fages' proof of termination. Finally, we adapt the first-order AC-unification algorithm to propose the first nominal AC-matching algorithm and formalise it to be terminating, sound and complete. As was the case for nominal C-unification, we used a parameter $\mathcal{X}$ for protected variables and this approach also let us obtain a verified nominal AC-equality checker as a byproduct. The 3 formalisations previously described were done in the PVS proof assistant and are available in NASALib, PVS' main repository of formalisations. In each one of the three formalisations we describe the files that compose the formalisation, pointing out their structure, hierarchy and size. With the aim of obtaining a nominal AC unification algorithm, we studied two interesting questions: generating solutions to $\pi \cdot X \approx^? X$ and proving termination. For the first question we propose a non-deterministic enumeration procedure and exemplify how it can compute non-obvious solution. For the second question we demonstrate that the problem $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$ gives rise to a loop and prove that it is enough to loop a limited amount of times, where this limit depend on the order of the permutation π. We hope these insights will advance the search for a nominal AC unification algorithm.

Keywords: Nominal, Formal Methods, PVS, Nominal C-Unification, AC-Unification, Nominal AC-Matching, Nominal AC-Unification.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Unification is an important topic in computer science, with applications in logic programming languages, theorem provers, type inference algorithms, narrowing and so on [15]. At its core, unification revolves around the task of determining when and how two mathematical expressions can be made equivalent by substituting appropriate values for their variables. 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 $f(a, b)$.

A more practical, although more intricate, example can be given by imagining that we are trying to compute $\int ln(x)dx$ using integration by parts. Recalling that the formula of integration by parts is $\int udv = uv - \int vdu$, in order to use this strategy we must unify $\int ln(x)dx$ with $\int udv$. This is done by instantiating the variable u to $ln(x)$ and v to x. Then, we get:

$$\int ln(x)\ dx = xln(x) - \int x\ d(ln(x))$$

which we can further simplify to obtain:

$$xln(x) - \int x\ d(ln(x)) = xln(x) - \int x * \frac{1}{x}\ dx = x\ ln(x) - x + C.$$

In that preceding case, only one of the terms contained the variables that were replaced by the substitution. The particular case of unification where we only instantiate variables from one of the terms is known as matching. It too has important applications, such as rewriting [15].

More precisely, given terms s and t, syntactic unification is the problem of finding a substitution σ such that $\sigma s = \sigma t$ and syntactic matching is the problem of finding a substitution σ such that $\sigma s = t$. The problem of syntactic unification can be generalised to consider an equational theory E. In this case, called E-unification, we must find a substitution σ such that σs and σt are equal modulo E, which we denote $\sigma s \approx_E \sigma t$ [44].[1]

[1]When E is clear from the context, we may write simply $\sigma s \approx \sigma t$.

Similarly, E-matching is the problem of finding a substitution σ such that $\sigma s \approx_E t$. Since associative and commutative (AC) operators are frequently used in programming languages and theorem provers, tools to support reasoning modulo associativity and commutativity axioms are often required. As an example of E-matching being used in software systems, Eker [37] gave an efficient implementation of AC-matching to handle AC-theories and described experimental results using Maude.

On the other hand, a different concept that is also fundamental in computer science and mathematics is the concept of binding. This concept appears, for instance, when we specify parameters to define functions: in $f : x \mapsto x + 1$, the variable x is said to be bound. Bindings are also present when we use quantifiers. For instance, in $\forall y : P(x, y)$, where P is some property of interest, the variable y is bound, while the variable x is not (x is said to be a free variable).

Since first-order syntax does not handle binding, extensions of it that consider free and bound variables are appealing areas of work. These extensions are not trivial, as it is possible to have expressions that are semantically equal, but syntactically distinct. For example, the formulas $\forall x : x + 1 > 0$ and $\forall y : 1 + y > 0$ should be considered equivalent. We could use indices to represent bound variables, as in explicit substitutions à la de Bruijn (see [1, 45, 59]), but from the user point of view it is simpler to use systems with variables names than systems with indices. The nominal syntax is an extension of the first-order syntax that smoothly represents languages with variable bindings [64]. It does so by using atoms, atom permutations, abstractions and freshness constraints to represent binders more naturally [41].

Although we will only fully explain the concepts of nominal in the next chapter, we now give as an appetizer an example of the nominal approach to handle binders. The formulas $\forall x : x + 1 > 0$ and $\forall y : 1 + y > 0$ would be represented as the nominal terms $\forall[x](x + 1 > 0)$ and $\forall[y](y + 1 > 0)$, where x and y are denoted atoms, 0 and 1 are 0-ary function symbols (constants), the symbol $\forall$ is a unary function symbol, and $+, >$ are binary function symbols that we write infix. These nominal terms are α-equivalent and this is derived as[2]:

$$\dfrac{\dfrac{\dfrac{\dfrac{x \approx x \qquad 1 \approx 1}{x + 1 \approx (x\ y) \cdot y + 1} \qquad 0 \approx 0}{x + 1 > 0 \approx (x\ y) \cdot y + 1 > 0} \qquad \dfrac{\dfrac{x\#1 \qquad x\#y}{x\#y + 1} \qquad x\#0}{x\#y + 1 > 0}}{[x](x + 1 > 0) \approx [y](y + 1) > 0}}{\forall[x](x + 1 > 0) \approx \forall[y](y + 1 > 0)}$$

Given the importance of unification and matching, the development of techniques for unification and matching in the nominal paradigm has been an attractive area of research since the invention of the nominal approach. Nominal unification is the extension of

[2]We have not explained the rules used in this derivation yet, but we will do so in the next chapter.

first-order unification to the nominal syntax, replacing the concept of syntactic equality by α-equivalence, and was first solved by Urban et al. in [75]. From there, research continued in the direction of making algorithm improvements to solve this problem and on considering nominal unification modulo equational theories.

1.1 Contributions

The contributions of this work can be grouped in four parts:

- We extend a functional nominal C-unification algorithm, adding a parameter $\mathcal{X}$ of protected variables, i.e., variables that cannot be instantiated. This nominal C-unification algorithm generalised with protected variables was formalised in the PVS proof assistant and can be used to the task of unification, matching and α-equivalence by correctly setting the parameter $\mathcal{X}$. This extension cannot be formally checked by simple reuse of the original formalisation, requiring additional effort. Moreover, we used the PVS formalisation to test the correctness of a Python manual implementation of the algorithm. This Item is described in our work "Formalising Nominal C-Unification Generalised with Protected Variables" (see [3]) and is the focus of Chapter 3.
- We give the first formalisation of an AC-unification algorithm. We specified the pioneering AC-unification algorithm of Stickel [72, 73] and proved it to be terminating (using an elaborate lexicographic measure, based on Fages' [39, 40] termination proof), sound and complete. We give a detailed description of the formalisation, including explanations of the main steps in the proofs of termination, soundness, and completeness; the files that were created along with their hierarchy and size; and a discussion about our design choices, including the consequences of our choice for the grammar of terms. We also discuss applications of the certified AC-unification algorithm, showing how the formalisation could be used as a starting point to formalise more efficient AC-unification algorithms or to test implementations of AC-unification algorithms. This Item is described in our works "A Certified Algorithm for AC-Unification" (see [9]) and "Certified First-Order AC-Unification and Applications" (see [7]) and is the focus of Chapter 4.
- We extend the certified first-order AC-unification algorithm described in the last Item to solve nominal AC-matching problems. We present the first algorithm for nominal AC-matching and formalise its termination, correctness and completeness. The formalisation enriches the first-order AC-unification algorithm providing structures and mechanisms to deal with the combinatorial aspects of nominal atoms,

permutations and abstractions. Furthermore, by adding a parameter for "protected variables" that cannot be instantiated during the execution, it enables nominal matching. As was the case for nominal C-unification, such general treatment of protected variables also gives rise to a verified nominal AC-equality checker as a byproduct. This Item is described in our work[3] "Nominal AC-Matching" (see [8]) and is the focus of Chapter 5.

- We report our work in progress on the task of nominal AC-unification. We sketch how we can solve fixpoint equations in the presence of AC function symbols and why we were not able to solve equations such as $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$. Additionally we speculate on whether we could use the connection between higher-order pattern and nominal to solve the problem. This Item was shortly described in [8] and is given more consideration in Chapter 6 of this thesis.

  The first three Items mention formalisations of nominal C-unification, first-order AC-unification and nominal AC-matching. Those formalisations were all done using the proof assistant PVS and are part of the Nominal library of the NASALib repository.

1.2 Organisation

Chapter 2 gives the necessary background, explaining concepts from the nominal framework, from first-order AC-unification and offering a summary of the PVS proof assistant. Then, Chapter 3 explains the nominal C-unification algorithm generalised with protected variables. The next 3 chapters are for equational reasoning in the presence of AC function symbols. First, Chapter 4 reports on the first-order AC-unification formalisation and its applications. Then, Chapter 5 shows how we adapted the first-order AC-unification formalisation to the nominal setting to obtain a verified nominal AC-matching algorithm. Chapter 6 discusses our work in progress towards nominal AC-unification. Finally, Chapter 7 describes related work and Chapter 8 concludes this work and outlines directions of future work. We include cyan-coloured hyperlinks (with the ↗ icon) to specific points of interest of the PVS formalisation.

[3]This paper received the Best Paper Award at the CICM 2023 conference.

Chapter 2

Background

2.1 First-Order Syntax and AC-Unification

In this section, we omit the subscript and write that t and s are equal modulo AC as $t \approx s$.

Definition 1 (Terms ↗). *Let Σ be a signature with function symbols and AC-function symbols. Let $\mathbb{X}$ be the set of all variables. The set $T(\Sigma, \mathbb{X})$ is generated by the grammar:*

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

where c denotes a constant [1] *, X is a variable, $\langle\rangle$ is the unit, $\langle s, t\rangle$ is a pair, $f\ t$ is a function application and $f^{AC}t$ is an associative-commutative function application.*

Terms were specified as shown in Definition 1 to make it easier to eventually adapt the formalisation to the nominal setting. That is the reason why the unit (an element in the grammar of the nominal terms) appears in Definition 1. Pairs are used to represent tuples with an arbitrary number of terms. For instance, the pair $\langle t_1, \langle t_2, t_3\rangle\rangle$ represents the tuple (t_1, t_2, t_3). In Definition 1 we imposed that a function application is of the form ft, which is not a limitation since t can be a pair. For instance, the term $f(a, b, c)$ can be represented as $f\langle a, \langle b, c\rangle\rangle$ and its arguments are a, b and c.

Remark 1. *When enumerating the arguments of a function with more than 2 arguments, some care must be taking in how we use pairs to represent it. Consider for instance, the distinct terms $f\langle a, \langle b, c\rangle\rangle$ and $f\langle\langle a, b\rangle, c\rangle$. Which one would we use to represent $f(a, b, c)$? This type of problem can be avoided by establishing as a convention that we always represent the tuple $(t_1, t_2, \ldots, t_n)$ as $\langle t_1, \langle t_2, \langle \ldots, t_n\rangle\rangle \ldots\rangle$. If we follow this convention, we would represent $f(a, b, c)$ as $f\langle a, \langle b, c\rangle\rangle$.*

[1]We represent the constants using the initial letters of the alphabet: a, b, c, …

Remark 2 (Variable Representation). *The variables in our PVS formalisation are represented as natural numbers. Given a variable X we denote by $|X|$ the corresponding natural number and given a set of variables V we define $max(V) = max(\{|X| : X \in V\})$. This notation will be used in Section 4.4.3.*

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

To ease our formalisation (more details in Section 4.6.1), we have restricted the terms in the unification problem that our algorithm receives to well-formed terms. Excluding pairs is natural since they are only used to encode (lists of) arguments of functions.

Definition 3 (AC-Unification problem). *An AC-unification problem is a finite set of equations $P = \{t_1 \approx^? s_1, \dots, t_n \approx^? s_n\}$. The left-hand side of the unification problem P, denoted as $lhs(P)$, is defined as $\{t_1, \dots, t_n\}$ while the right-hand side of P, denoted as $rhs(P)$, is defined as $\{s_1, \dots, s_n\}$.*

Notation 1 (AC-Unification pairs). *When t and s are both* headed by the same AC-function symbol, *we refer to the equation $t \approx^? s$ as an AC-unification pair.*

Notation 2. *When convenient, we may mention that a function symbol f is an AC-function symbol, omit the superscript and write simply f instead of f^{AC}.*

Notation 3 (Flattened form of AC-functions). *When convenient, we may denote in this paper an AC-function in flattened form. For instance, the term $f^{AC}\langle f^{AC}\langle a, b\rangle, f^{AC}\langle c, d\rangle\rangle$ may be denoted simply as $f^{AC}(a, b, c, d)$. In our formalisation (for instance in function $Args_f$), when we manipulate an AC-function term t we are more interested in its arguments than in how they were encoded using pairs.*

Notation 4 (*Vars*). *We denote the set of variables of a term t by $Vars(t)$. Similarly, we denote the set of variables that occur in a unification problem P as $Vars(P)$.*

A substitution σ is a function from variables to terms, such that $\sigma X \neq X$ only for a finite set of variables, called the domain of σ and denoted as $dom(\sigma)$. The image of σ is then defined as $im(\sigma) = \{\sigma X \mid X \in dom(\sigma)\}$. We denote the identity substitution by *id*.

Definition 4 (Well-Formed Substitution). *A substitution σ is said to be well-formed if, for every X, σX is a well-formed term.*

In the proof of completeness of the algorithm, we restrict ourselves to well-formed substitutions (this is explained in the proof of Section 4.4.3).

Notation 5 ($\sigma \subseteq V$). *Let V be a set of variables. If $dom(\sigma) \subseteq V$ and $Vars(im(\sigma)) \subseteq V$ we write $\sigma \subseteq V$.*

Notation 6 ($\sigma =_V \sigma_1$)**.** *Let σ and σ_1 be substitutions and V a set of variables. If $\sigma X = \sigma_1 X$ for every $X \in V$ we write $\sigma =_V \sigma_1$.*

In our PVS code, substitutions are represented by a list, where each entry of the list is called a nuclear substitution and is of the form $\{X \mapsto t\}$. The action of a nuclear substitution and the action of a substitution over terms are shown in Definitions 5 and 6 respectively.

Definition 5 (Nuclear substitution action on terms)**.** *A nuclear substitution $\{X \mapsto s\}$ acts over a term by induction as shown below:*

- $\{X \mapsto s\}a = a.$
- $\{X \mapsto s\}\langle\rangle = \langle\rangle.$
- $\{X \mapsto s\}Y = \begin{cases} s & \text{if } X = Y \\ Y & \text{otherwise.} \end{cases}$
- $\{X \mapsto s\}\langle t_1, t_2\rangle = \langle \{X \mapsto s\}t_1, \{X \mapsto s\}t_2\rangle.$
- $\{X \mapsto s\}(f\ t_1) = f\ (\{X \mapsto s\}t_1).$
- $\{X \mapsto s\}(f^{AC}\ t_1) = f^{AC}\ (\{X \mapsto s\}t_1).$

Definition 6 (Substitution acting on terms)**.** *Since a substitution σ is a list of nuclear substitutions, the action of a substitution is defined as:*

- *$\textsc{nil}\ t = t$, where $\textsc{nil}$ is the null list, used to represent the identity substitution.*
- *$\textsc{cons}(\{X \mapsto s\},\ \sigma)\ t = \{X \mapsto s\}(\sigma t)$.*

The notion of substitution used here differs from the more traditional view of a substitution as a simultaneous application of nuclear substitutions, although both are correct. The way we defined substitution here is closer to triangular substitutions [50]. Notice that in the definition of action of substitutions the nuclear substitution in the head of the list is applied last. This allows us to, given substitutions σ and δ, obtain the substitution $\sigma \circ \delta$ in our code simply as $\textsc{append}(\sigma, \delta)$.

Remark 3 (Substitution in PVS and How We Denote Them)**.** *Although substitutions were defined in PVS as specified in Definition 6, when giving examples we may opt for the more familiar $\{variable \mapsto term, \ldots\}$ notation. For instance, the substitution that is defined in PVS as $\textsc{cons}(\{X \mapsto a\}, \textsc{cons}(\{Y \mapsto b\}, \textsc{nil}))$ may be denoted simply as $\{X \mapsto a, Y \mapsto b\}$. A similar remark applies for substitution on nominal terms.*

Notation 7 (Composition of Substitutions)**.** *When composing two substitutions σ and δ we may omit the composition symbol and write $\sigma\delta$ instead of $\sigma \circ \delta$.*

Definition 7 (Renaming)**.** *A renaming ρ is an injective substitution that always instantiates a variable to a variable.*

Example 1. *The substitution $\rho = \{Z_1 \mapsto Z_2, Z_2 \mapsto Z_3\}$ has $dom(\rho) = \{Z_1, Z_2\}$, $im(\rho) = \{Z_2, Z_3\}$ and is a renaming.*

We now define unifiers, more general substitutions and complete set of unifiers (Definitions 8, 9 and 10).

Definition 8 (Unifiers)**.** *Let P be a unification problem $\{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$. A unifier or solution of P is a substitution σ such that $\sigma t_i \approx \sigma s_i$ for every i from 1 to n. When σ is a unifier for P we say that σ* unifies *P.*

Example 2. *Suppose X, Y are variables, a, b are constants and f is an AC function. For the unification problem $P = \{f(b, X) \approx^? f(a, Y)\}$, a possible solution is*

$$\sigma = \{X \mapsto a, Y \mapsto b\},$$

as

$$\sigma f(b, X) = f(b, a) \approx f(a, b) = \sigma f(a, Y).$$

Definition 9 (More General Substitutions)**.** *A substitution σ is* more general (modulo AC) than *a substitution σ' in a set of variables V if there is a substitution δ such that $\sigma' =_V \delta\sigma$, for all variables $X \in V$. In this case we write $\sigma \leq_V \sigma'$. When V is the set of all variables, we say that σ is more general than σ' and write $\sigma \leq \sigma'$.*

Example 3. *The substitution $\sigma = \{Y \mapsto g(X), W \mapsto b\}$ is more general than $\delta = \{Y \mapsto g(a), X \mapsto a\}$ in the set $V = \{Y, X\}$, since with $\theta = \{X \mapsto a\}$ we have $\delta =_V \theta\sigma$.*

Definition 10 (Complete Set of Unifiers)**.** *With the notion of more general substitution, we can define a complete set $\mathcal{C}$ of unifiers of P as a set that satisfies two conditions:*

- *each $\sigma \in \mathcal{C}$ is an unifier of P.*
- *for every δ that unifies P, there is $\sigma \in \mathcal{C}$ such that $\sigma \leq_{Vars(P)} \delta$.*

We represent an AC-unification problem P as a list in our PVS code, where each element of the list is a pair (t_i, s_i) that represents an equation $t_i \approx^? s_i$. Finally, given a unification problem $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$, we define σP as $\{\sigma t_1 \approx^? \sigma s_1, \ldots, \sigma t_n \approx^? \sigma s_n\}$.

Notation 8. *Since P is a list in our PVS code, we denote by $car(P)$ the equation $t \approx^? s$ in the head of the list P and by $cdr(P)$ the tail of the list P.*

2.1.1 Complexity of α-Equivalence, Matching and Unification in First-Order Modulo Equational Theories

Table 2.1 summarises the complexity of α-equivalence, matching and unification modulo some equational theories in first-order syntax. In the mentioned table ω denotes a theory of type finitary, ∞ denotes a theory of type infinitary and 0 a theory of type nullary (a nullary unification type, also called type zero, means that there are terms for which a minimal complete set of unifiers does not exist). As usual, C stands for commutativity and A stands for associativity. Additionally, AU stands for Associativity with Unit, ACU is Associativity-Commutativity with Unit and AI unification is unification with an Associative and Idempotent function symbol.

Table 2.1: Unification Type and Complexity for Some Equational Theories in First-Order Syntax.

Theory	**Unification Type**	**Complexity of**			**Related Work**
		Equality-Checking	**Matching**	**Unification**	
Syntactic	1	O(n)	O(n)	O(n)	[56, 63, 66]
C	ω	O(n^2)	NP-comp.	NP-comp.	[20, 46]
A	∞	O(n)	NP-comp.	NP-hard	[20, 55]
AU	∞	O(n)	NP-comp.	decidable	[46, 55]
AI	0	O(n)	NP-comp.	NP-comp.	[14, 49, 67]
AC	ω	O(n^3)	NP-comp.	NP-comp.	[20, 46, 47]
ACU	ω	O(n^3)	NP-comp.	NP-comp.	[47]

Remark 4 (Associative Unification in Maude)**.** *Although associative unification is in general infinitary (see Table 2.1), this does not mean that it cannot be treated by computational systems. For instance, Eker [38] describes an algorithm used by Maude for A-unification that generates all possible solutions until a certain bound, chosen by the user.*

2.2 Examples of AC-Unification

Completeness of AC-unification is more complex than it looks at first glance. Stickel [72, 73] was the first to give a complete algorithm to solve unification in the presence of AC-function symbols. He did it by discovering and exploring the connection between unification and solving linear equations in $\mathbb{Z}^+$. Termination is also harder than it appears to be: Stickel's original proof of termination was not valid for the general case, and it took almost a decade for Fages [39, 40] to discover the flaw and propose a (complex) proof of termination. We now give three examples to illustrate the interesting aspects of the problem.

2.2.1 What Makes AC-Unification Hard

Let f be an associative-commutative function symbol. Finding a complete set of unifiers for $\{f(X_1, X_2) \approx^? f(a, Y)\}$ is not as easy as it appears at first sight, since it is not enough to simply compare the arguments of the first term with the arguments of the second term. Indeed, this strategy would give us only

$$\sigma_1 = \{X_1 \mapsto a, Y \mapsto X_2\}$$
$$\sigma_2 = \{X_2 \mapsto a, Y \mapsto X_1\}$$

as solutions, missing for example the substitution $\sigma_3 = \{X_1 \mapsto f(a, W), Y \mapsto f(X_2, W)\}$. This solution would be missed because the arguments of $\sigma_3 Y = f(X_2, W)$ are partially contained in $\sigma_3 X_1 = f(a, W)$ and partially contained in $\sigma_3 X_2 = X_2$.

Remark 5. *In contrast to AC-unification, to guarantee the completeness of AC-matching, it is enough to explore all possible pairings of the arguments of the first term with the arguments of the second term. Evidence of the difficulty of AC-unification is that, although Contejean formalised AC-matching in 2004 and left as future work a formalisation of AC-unification (see [31]), it took 18 years to obtain the first formalisation of AC-unification (see [9]).*

2.2.2 Unifying $f(X, X, Y, a, b, c)$ and $f(b, b, b, c, Z)$

We give a higher-level example (taken from the very accessible [73]) of how we would solve

$$\{f(X, X, Y, a, b, c) \approx^? f(b, b, b, c, Z)\}.$$

In a high-level view, this technique converts an AC-unification problem into a linear Diophantine equation and uses a basis of solutions of the Diophantine equation to get a complete set of AC-unifiers to our original problem.

The first step is to eliminate common arguments in the terms that we are trying to unify. The problem becomes

$$\{f(X, X, Y, a) \approx^? f(b, b, Z)\}.$$

The second step is to associate our unification problem with a linear Diophantine equation, where each argument of our terms corresponds to one variable in the equation (this process is called variable abstraction) and the coefficient of this variable in the equation is the number of occurrences of the argument. In our case, the linear Diophantine equation obtained is: $2X_1 + X_2 + X_3 = 2Y_1 + Y_2$ (variable X_1 was associated with argument X, variable X_2 with the argument Y and so on; the coefficient of variable X_1 is two, since argument X occurs twice in $f(X, X, Y, a)$ and so on).

The third step is to generate a basis of solutions to the equation and associate a new variable (the Z_is) to each solution. As we shall soon see, the unification problem $\{f(X, X, Y, a) \approx^? f(b, b, Z)\}$ may branch into (possibly) many unification problems and the new variables Z_is will be the building blocks for the right-hand side of these unification problems. The result is shown on Table 2.2.

Table 2.2: Solutions for $2X_1 + X_2 + X_3 = 2Y_1 + Y_2$.

$\mathbf{X_1}$	$\mathbf{X_2}$	$\mathbf{X_3}$	$\mathbf{Y_1}$	$\mathbf{Y_2}$	**New Variables**
0	0	1	0	1	Z_1
0	1	0	0	1	Z_2
0	0	2	1	0	Z_3
0	1	1	1	0	Z_4
0	2	0	1	0	Z_5
1	0	0	0	2	Z_6
1	0	0	1	0	Z_7

Observing Table 2.2 we relate the "old variables" (X_is and Y_is) with the "new variables" (Z_is). For instance, the column of variable X_2 has a 0 in the lines that correspond to variables Z_1, Z_3, Z_6, Z_7; a 1 in the lines that correspond to variables Z_2 and Z_4; and a 2 in the line that corresponds to variable Z_5. Hence, the relation between the X_2 with the new variables is: $X_2 = Z_2 + Z_4 + 2Z_5$. All those relations between the "old variables" and the "new variables" are shown below:

$$\begin{aligned} X_1 &= Z_6 + Z_7 \\ X_2 &= Z_2 + Z_4 + 2Z_5 \\ X_3 &= Z_1 + 2Z_3 + Z_4 \\ Y_1 &= Z_3 + Z_4 + Z_5 + Z_7 \\ Y_2 &= Z_1 + Z_2 + 2Z_6. \end{aligned} \tag{2.1}$$

In order to explore all possible solutions, we must consider whether we will include or not each solution of our basis. Since seven solutions compose our basis (one for each variable Z_i), this means that *a priori* there are 2^7 cases to consider. Considering that including a solution of our basis means setting the corresponding variable Z_i to 1 and not including it means setting it to 0, we must respect the constraint that no original variables

$(X_1, X_2, X_3, Y_1, Y_2)$ receive 0. Eliminating the cases that do not respect this constraint[2], we are left with 69 cases [72].

For example, if we decide to include only the solutions represented by the variables Z_1, Z_4 and Z_6, the corresponding unification problem, according to Equations (2.1), becomes:

$$P = \{X_1 \approx^? Z_6, X_2 \approx^? Z_4, X_3 \approx^? f(Z_1, Z_4), Y_1 \approx^? Z_4, Y_2 \approx^? f(Z_1, Z_6, Z_6)\}. \tag{2.2}$$

We can also drop the cases where a variable that does not represent a variable term is paired with an AC-function application. For instance, the unification problem P should be discarded, since the variable X_3 represents the constant a, and we cannot unify a with $f(Z_1, Z_4)$. This constraint eliminates 63 of the 69 potential unifiers.

Finally we replace the variables X_1, X_2, X_3, Y_1, Y_2 by the original arguments they substituted and proceed with the unification. Some unification problems that we will explore will be unsolvable and discarded later, as:

$$\{X \approx^? Z_6, Y \approx^? Z_4, a \approx^? Z_4, b \approx^? Z_4,\ Z \approx^? f(Z_6, Z_6)\}$$

(we cannot unify both a with Z_4 and b with Z_4 simultaneously). In the end, the solutions computed for the original problem $\{f(X, X, Y, a, b, c) \approx^? f(b, b, b, c, Z)\}$ are:

$$\begin{aligned}
\sigma_1 &= \{Y \mapsto f(b, b), Z \mapsto f(a, X, X)\}.\\
\sigma_2 &= \{Y \mapsto f(Z_2, b, b), Z \mapsto f(a, Z_2, X, X)\}.\\
\sigma_3 &= \{X \mapsto b, Z \mapsto f(a, Y)\}.\\
\sigma_4 &= \{X \mapsto f(Z_6, b), Z \mapsto f(a, Y, Z_6, Z_6)\}.
\end{aligned} \tag{2.3}$$

Remark 6. *When using the technique described in this section to unify $f(X, X, Y, a, b, c)$ with $f(b, b, b, c, Z)$, we obtained unification problems that only contain the variables X_1, X_2, X_3, Y_1, Y_2 or AC-functions whose arguments are all variables (for instance P in Equation 2.2). However, this does not mean that our technique cannot be applied to general AC-unification problems, since we eventually replace the variables X_1, X_2, X_3, Y_1, Y_2 by their corresponding arguments (X, Y, a, b, Z respectively) and proceed with unification.*

Remark 7 (Cases on AC1-Unification)**.** *If we were considering AC1-unification, where our signature has an identity `id` function symbol, we could consider only the case where we include all the AC solutions in our basis and instantiate the variables Z_is later on to be `id`.*

[2]Suppose for instance that we set variables $(Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7)$ to $(1, 1, 1, 1, 1, 0, 0)$. Then $X_1 = Z_6 + Z_7$ would be set to 0, so this case does not respect the constraint and is eliminated.

2.2.3 Avoiding Infinite Loops

It is necessary to compose the substeps of solving AC-unification equations with some strategy, as the following example (adapted from [40]) shows.

Example 4 (Looping forever)**.** *Let f be an AC-function symbol. Suppose we want to solve*

$$P = \{f(X,Y) \approx^? f(U,V), X \approx^? Y, U \approx^? V\}$$

and instead of instantiating the variables as soon as we can, we decide to try solving the first equation. When trying to unify $f(X,Y)$ with $f(U,V)$ we obtain as one of the branches the unification problem:

$$\begin{aligned}&\{X \approx^? f(X_1,X_2), Y \approx^? f(X_3,X_4), U \approx^? f(X_1,X_3), V \approx^? f(X_2,X_4)\\ &\ X \approx^? Y, U \approx^? V\}.\end{aligned}$$

We can solve this branch by instantiating X, Y, U and V in the first four equations. After these instantiations, the substitution we have computed and the two remaining equations we have to unify are:

$$\begin{aligned}\sigma &= \{X \mapsto f(X_1,X_2), Y \mapsto f(X_3,X_4), U \mapsto f(X_1,X_3), V \mapsto f(X_2,X_4)\}\\ P' &= \{f(X_1,X_2) \approx^? f(X_3,X_4), f(X_1,X_3) \approx^? f(X_2,X_4)\}\end{aligned}$$

One way of solving the first equation is to decompose it into $\{X_1 \approx^? X_3, X_2 \approx^? X_4\}$, which get us back to

$$P' = \{f(X_1,X_3) \approx^? f(X_2,X_4), X_1 \approx^? X_3, X_2 \approx^? X_4\}$$

which is essentially the same as the unification problem P we started with.

Notice that this infinite loop in our example would not happen if we had instantiated $\{X \mapsto Y\}$ and $\{U \mapsto V\}$ in the beginning. In our first-order AC-unification algorithm, we always instantiate the variables that we can before tackling AC-unification pairs.

2.3 The Nominal Setting

This section lays the background for both the nominal C-unification formalisation and the nominal AC-matching formalisation. The definitions and notations in this section are based on [8] and [3]. We define nominal terms and associated concepts modulo an equational theory E, where $E = C$ (Chapter 3) or $E = AC$ (Chapters 5). The links in this section point to the nominal AC-matching formalisation, but the corresponding concepts in the nominal C-unification formalisation were defined similarly.

2.3.1 Atoms, Variables, Nominal Terms, Substitutions and Permutations

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

Remark 8 (Name Convention)**.** *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 called* name convention *or* Gabbay's permutative convention*.*

Renaming of atoms happens through permutations, where a permutation π is a bijection of the form $\pi : \mathbb{A} \to \mathbb{A}$ 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 $\pi = (a_n\ b_n) :: ... :: (a_1\ b_1) :: \text{NIL}$.

Definition 11 (Action of Permutations on Atoms)**.** *The action of a permutation over an atom is recursively defined as:*

$$NIL \cdot c = c$$

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

Finally, the reverse of a permutation π is denoted by π^{-1} and can be computed by reversing the list of swappings.

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

Definition 12 (Nominal Terms)**.** *The set $\mathcal{T}(\Sigma, \mathbb{A}, \mathbb{X})$ of nominal terms is generated according to the grammar:*

$$s, t \quad ::= \quad a \ \mid\ \pi \cdot X \ \mid\ \langle\rangle \ \mid\ [a]t \ \mid\ \langle s, t\rangle \ \mid\ f\ t \ \mid\ f^E\, t \qquad (2.4)$$

where $\langle\rangle$ is the unit, a is an atom term, $\pi \cdot X$ is a moderated variable or suspension (the permutation π is suspended on the variable X), $[a]t$ is an abstraction (a term with the atom a abstracted), $\langle s, t\rangle$ is a pair, $f\ t$ is a function application and $f^E\ t$ is a E function application.

Remark 9. *We represent moderated variables of the form $id \cdot X$ simply as X.*

The action of permutation on nominal terms is defined recursively as shown in Definition 13.

Definition 13 (Permutation Action on Terms). *The action of permutations on terms is defined recursively:*

- $\pi \cdot \langle\rangle = \langle\rangle$
- $\pi \cdot (\pi' \cdot X) = \textit{APPEND}(\pi, \pi') \cdot X$
- $\pi \cdot [a]t = [\pi \cdot a]\pi \cdot t$
- $\pi \cdot \langle s, t\rangle = \langle \pi \cdot s, \pi \cdot t\rangle$
- $\pi \cdot f\ t = f\ \pi \cdot t$
- $\pi \cdot f^E\ t = f^E\ \pi \cdot t$

Remark 10. *When a permutation π is applied to a suspension $\pi' \cdot X$, 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 5. *To illustrate the action of a permutation on a term, consider $\pi = (a\ b) :: (b\ c) :: (d\ e) :: nil$ and $t = f\langle b, \langle d, X\rangle\rangle$. Then, the result of the permutation action is $\pi \cdot t = f\langle c, \langle e, \pi \cdot X\rangle\rangle$.*

In Definition 5 we define action of nuclear substitutions for first-order terms. Definition 14, is the corresponding definition for nominal terms. As was done in first-order, we use the definition of action of nuclear substitution to define the action of a substitution for terms.

Definition 14 (Nuclear Substitution Acting on Nominal Terms). *The action of a nuclear substitution on a nominal term is defined inductively:*

- $\{X \mapsto t\}\pi \cdot Y = \begin{cases} \pi \cdot Y & \textit{if } X \neq Y \\ \pi \cdot t & \textit{otherwise} \end{cases}$
- $\{X \mapsto t\}\langle\rangle = \langle\rangle$

- $\{X \mapsto t\}\langle s_1, s_2\rangle = \langle \{X \mapsto t\}s_1, \{X \mapsto t\}s_2\rangle$
- $\{X \mapsto t\}([a]s) = [a](\{X \mapsto t\}s)$
- $\{X \mapsto t\}(f\ s) = f\ (\{X \mapsto t\}s)$
- $\{X \mapsto t\}(f^E\ s) = f^E\ (\{X \mapsto t\}s)$

Definition 15 (Substitution acting on terms). *Since a substitution σ is a list of nuclear substitutions, the action of a substitution is defined as:*

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

Example 6. *Let $\sigma = \{Y \mapsto a, X \mapsto f(Y, b)\}$ and $t = [a]X$. Then, $\sigma t = [a]f(a, b)$.*

2.3.2 Freshness and α-Equality

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

- $a\#t$ 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 $a \approx_\alpha b$.

To formally define freshness (Definition 17) we need the definition of freshness context (Definition 16).

Definition 16 (Freshness Context). *A freshness context ∇ is a set of constraints of the form $a\#X$.*

Notation 9. *We denote contexts by letters Δ, Γ, ∇ and so on. Let Γ be an arbitrary context. We denote by Vars(Γ) the set $\{X \mid a\#X \in \Gamma, \text{for some atom } a\}$.*

Notation 10 (Difference Set ds). *We define the difference set between two permutations π and π' as $ds(\pi, \pi') = \{a \in \mathbb{A} \mid \pi \cdot a \neq \pi' \cdot a\}$. Thus, $ds(\pi, \pi')\#X$ is the set containing every constraint of the form $a\#X$ for $a \in ds(\pi, \pi')$.*

Definition 17 (Freshness). *An atom a is said to be fresh on t under a context Δ (which we denote by $\Delta \vdash a\#t$) if it is possible to build a proof using the rules:*

$$\frac{}{\Delta \vdash a\#\langle\rangle}\ (\#\langle\rangle) \qquad \frac{}{\Delta \vdash a\#b}\ (\#atom)$$

$$\frac{(\pi^{-1}\cdot a\#X) \in \Delta}{\Delta \vdash a\#\pi \cdot X}\ (\#X) \qquad \frac{}{\Delta \vdash a\#[a]t}\ (\#[a]a)$$

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

$$\frac{\Delta \vdash a\#t}{\Delta \vdash a\#f\ t}\ (\#app) \qquad \frac{\Delta \vdash a\#t}{\Delta \vdash a\#f^E\ t}\ (\#app)$$

Example 7. *Let's derive $a\#\langle X,[a]Y\rangle$ with context $\Delta = \{a\#X\}$:*

$$\frac{\dfrac{a\#X \in \Delta}{\Delta \vdash a\#X}\ (\#X) \quad \dfrac{}{\Delta \vdash a\#[a]Y}\ (\#[a]a)}{\Delta \vdash a\#\langle X,[a]Y\rangle}\ (\#pair)$$

With the notion of freshness, one can define α-equality in the nominal setting. To define α-equality with AC operators (Definition 18) we used operators S_n and D_n, defined as follows. Let f be an AC function symbol, $S_n(f\,t)$ be an operator that selects the nth argument of $f\,t$ (considering the flattened form) and $D_n(f\,t)$ be an operator that deletes the nth argument of $f\,t$ (considering the flattened form).

Example 8. *Let f be an AC-function symbol and $t = f\langle f\langle a,b\rangle, f\langle [a]X, \pi\cdot Y\rangle\rangle$. In the above definition, $S_2(f,t) = b$ and $D_2(f,t) = f\langle f a, f\langle [a]X, \pi \cdot Y\rangle\rangle\rangle)$.*

Definition 18 (α-Equality with AC operators). *If there exist i and j such that $\Delta \vdash S_i(f^{AC}s) \approx_\alpha S_j(f^{AC}t)$ and $\Delta \vdash D_i(f^{AC}s) \approx_\alpha D_j(f^{AC}t)$, then $\Delta \vdash f^{AC}s \approx_\alpha f^{AC}t$. In other words, the rule of α-equality for an AC-function application is:*

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

Two terms t and s are said to be α-equivalent under the freshness context Δ ($\Delta \vdash t \approx_\alpha s$) if it is possible to build a proof using rule ($\approx_\alpha AC$) and the rules:

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

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

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

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

Example 9. *Let f be an AC function symbol. One can derive that $\emptyset \vdash f\langle f\langle a,b\rangle, c\rangle \approx^? f\langle c, f\langle b,a\rangle\rangle$ by first noticing that*

$$S_1(f\langle f\langle a,b\rangle,c\rangle) = a \textit{ and } S_3(f\langle c, f\langle b,a\rangle\rangle) = a$$
$$D_1(f\langle f\langle a,b\rangle,c\rangle) = f\langle fb, c\rangle \textit{ and } D_3(f\langle c, f\langle b,a\rangle\rangle) = f\langle c, fb\rangle$$

and then noticing that

$$S_1(f\langle fb,c\rangle) = b \textit{ and } S_2(f\langle c, fb\rangle) = b$$
$$D_1(f\langle fb,c\rangle) = fc \textit{ and } D_2(f\langle c, fb\rangle) = fc$$

More precisely the derivation tree that proves $\emptyset \vdash f\langle f\langle a,b\rangle,c\rangle \approx^? f\langle c, f\langle b,a\rangle\rangle$ is shown below. To make the derivation more compact, we omit the name of the rules that are not related to AC-operators.

$$\dfrac{\dfrac{}{\emptyset \vdash a \approx^? a} \quad \dfrac{\dfrac{}{\emptyset \vdash b \approx^? b} \quad \dfrac{\dfrac{}{\emptyset \vdash c \approx^? c} \quad \dfrac{}{\emptyset \vdash \langle\rangle \approx^? \langle\rangle}}{\emptyset \vdash fc \approx^? fc}\,(\approx_\alpha AC,\ (i,j)=(1,1))}{\emptyset \vdash f\langle fb, c\rangle \approx^? f\langle c, fb\rangle}\,(\approx_\alpha AC,\ (i,j)=(1,2))}{\emptyset \vdash f\langle f\langle a,b\rangle,c\rangle \approx^? f\langle c, f\langle b,a\rangle\rangle}\,(\approx_\alpha AC,\ (i,j)=(1,3))$$

Example 10. *The notion of "exists a number that is greater than 0" could be represented as the nominal term $\exists[a]a > 0$ or as $\exists[b]b > 0$ and these two representations are equivalent. Here is is how we would derive $\exists[a]a > 0 \approx \exists[b]b > 0$:*

$$\dfrac{\dfrac{\dfrac{a \approx_\alpha a \qquad 0 \approx_\alpha 0}{a > 0 \approx_\alpha (a\ b)\cdot b > 0}\,(\approx_\alpha app) \qquad \dfrac{a\#b \qquad a\#0}{a\#b > 0}\,(\#app)}{[a]a > 0 \approx_\alpha [b]b > 0}\,(\approx_\alpha [a]b)}{\exists[a]a > 0 \approx_\alpha \exists[b]b > 0}\,(\approx_\alpha app)$$

Definition 19 is α-equality with C operators. Notice that the rule for α-equality of two AC function applications is very different from the rule for α-equality of two C function applications.

Definition 19 (α-Equality with C operators). *α-equality under the presence of commutative function symbols is defined by using all the rules of Definition 18 with the exception of rule ($\approx_\alpha AC$) and adding the rule ($\approx_\alpha C$):*

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

Example 11. *The following derivation proves that $g(f^C\langle a,b\rangle) \approx_\alpha g(f^C\langle b,a\rangle)$.*

$$\dfrac{\dfrac{\dfrac{}{b \approx_\alpha b}\,\textit{(}\approx_\alpha \textit{atom)} \qquad \dfrac{}{a \approx_\alpha a}\,\textit{(}\approx_\alpha \textit{atom)}}{f^C\langle a,b\rangle \approx_\alpha f^C\langle b,a\rangle}\,\textit{(}\approx_\alpha C\textit{)}}{g(f^C\langle a,b\rangle) \approx_\alpha g(f^C\langle b,a\rangle)}\,\textit{(}\approx_\alpha \textit{app)}$$

2.3.3 Solution to a Quintuple and Additional Notation

For the proofs of soundness and completeness of nominal AC-matching and nominal C-unification, we need the notion of a solution to a quintuple (Definition 22). This definition depends on a parameter $\mathcal{X}$, a set of "protected variables", i.e., variables that cannot be instantiated. Before presenting this concept, we introduce some notation and recall the concept of a solution in nominal syntactic unification (Definition 20).

Notation 11 (Equational Constraints)**.** *In the nominal setting, $t \approx^? s$ is denoted an equational constraint or an equation. $a\#_?t$ is denoted a freshness constraint.*

Notation 12. *Let ∇ and ∇' be freshness contexts and σ and σ' substitutions. We need the following notation to define a solution to a quintuple:*

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

Definition 20 (Solution in Nominal Syntactic Unification)**.** *Let P be a finite set of equational and freshness constraints of the form $t \approx^? s$ and $a\#_?t$. In nominal syntactic unification, i.e., nominal unification with no symbols from any equational theory E, the solution to P is a pair (Δ, δ) such that*

1. *if $a\#_?t \in P$ then $\Delta \vdash a\#\delta t$.*
2. *if $t \approx^? s \in P$ then $\Delta \vdash \delta t \approx_\alpha \delta s$.*

We now define a general notion of unification problem with protected variables (Definition 21) and a solution to a quintuple (Definition 22). Then, the definition for a nominal E-unification/matching/equality problem are obtained immediately from the corresponding definitions of unification by correctly setting the parameter $\mathcal{X}$.

Definition 21 (Unification Problem With Protected Variables)**.** *A unification problem with protected variables is a triple $(\Gamma, P, \mathcal{X})$ where Γ is a freshness context; P is a finite set of equational and freshness constraints of the form $t \approx^? s$ and $a\#_?t$, respectively; and $\mathcal{X}$ is a set of variables.*

When $\mathcal{X} = \emptyset$, Definition 21 corresponds to an E-unification problem. When $\mathcal{X} = \mathit{Vars}(\mathit{rhs}(P))$, Definition 21 corresponds to an E-matching problem and when $\mathcal{X} = \mathit{Vars}(P)$ the mentioned definition corresponds to an E-equality checking problem.

Definition 22 (Solution for a Quintuple). *Suppose that Γ is a context, P is a set of freshness constraints (of the form $a\#_?t$) and equational constraints (of the form $t \approx^? s$), σ is a substitution, V is a set of variables and $\mathcal{X}$ is a set of protected variables that cannot be instantiated. A solution for a quintuple $(\Gamma, P, \sigma, V, \mathcal{X})$ is a pair (Δ, δ), where the following conditions are satisfied:*

1. $\Delta \vdash \delta\Gamma$.
2. *if* $a\#_?t \in P$ *then* $\Delta \vdash a\#\delta t$.
3. *if* $t \approx^? s \in P$ *then* $\Delta \vdash \delta t \approx_\alpha \delta s$.
4. *there exists* λ *such that* $\Delta \vdash \lambda\sigma \approx_V \delta$.
5. $dom(\delta) \cap \mathcal{X} = \emptyset$.

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

Definition 23 (Solution for an E-unification/matching/equality problem). *A solution for an E-unification problem with protected variables $(\Gamma, P, \mathcal{X})$ is a solution for the associated quintuple $(\Gamma, P, id, Vars(P), \mathcal{X})$. When $\mathcal{X} = Vars(rhs(P))$, we have the definition for an AC-matching problem and when $\mathcal{X} = Vars(P)$ we have the definition of solution to an AC-equality checking problem.*

2.3.4 Fixpoint Equations

An equational constraint of the form $\pi \cdot X \approx^? \pi' \cdot X$ is denoted a *fixpoint equation*. Since every equational constraint $\pi \cdot X \approx^? \pi' \cdot X$ can be rewritten as $\pi'^{-1}\pi \cdot X \approx^? X$, we usually represent a generic fixpoint equation simply as $\pi \cdot X \approx^? X$.

In nominal unification, we can solve a fixpoint equation $\pi \cdot X \approx^? X$ by adding $\{a\#X \mid a \in dom(\pi)\}$ to our context. As shown in Ayala-Rincón et al. [5], this same approach is not complete in nominal C-unification. Consider for instance the equational constraint $(a\ b) \cdot X \approx^? X$ and let $+$ be a commutative function symbol. The pair $(\Delta, \delta) = (\{a\#X \mid a \in dom(\pi)\},\ id)$ is a solution to the equation, but it is not the only one; other examples are $(\emptyset, \{X \mapsto a + b\})$, $(\emptyset, \{X \mapsto (a + b) + (a + b)\})$ and so on. Indeed, as shown in [5] the problem of nominal C-unification is infinitary if we express the solutions as pairs $(context, substitution)$ and in order for us to obtain an algorithm for nominal C-unification it is necessary to add a parameter for the fixpoint equations in the solution, i.e., the output is a triple $(context, substitution, fixpoint\ equations)$. The problem of fixpoint equations in nominal AC-unification is not yet published, but appears to be similar to nominal C-unification (see our work in progress in Chapter 6).

2.3.5 Complexity of Unification and Matching Modulo Equational Theories in the Nominal Setting

As was done for the first-order syntax (Table 2.1), Table 2.3 shows the unification type and complexity of unification, matching and α-equivalence for some equational theories in the nominal setting. A question mark was put for questions that are open.

Table 2.3: Unification Type and Complexity for Some Equational Theories in the Nominal Syntax.

Theory	**Unification Type**	**Complexity of**			**Related Work**
		Equality-Check	**Matching**	**Unification**	
Syntactic	1	O(n log n)	O(n log n)	O(n^2)	[6, 27, 53, 75]
C	∞	O(n^2 log n)	NP-comp.	NP-comp.	[4, 6]
A	?	O(n log n)	?	?	[6]
AC	?	O(n^2 log n)	NP-comp.	?	[6, 8]

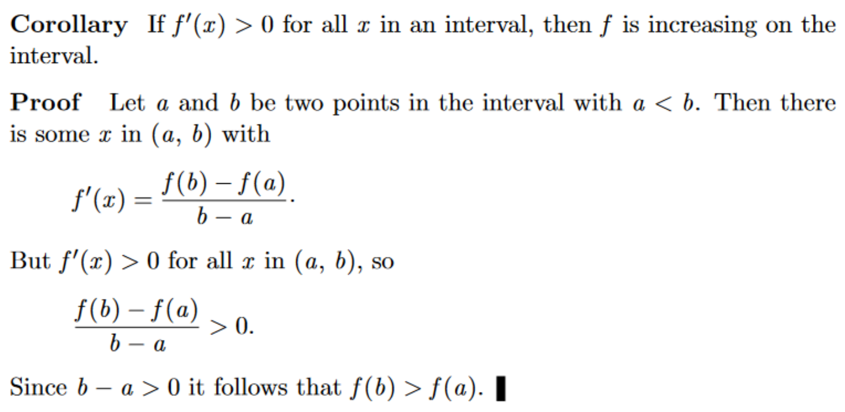
2.4 Structured Proofs

In this section we present the concept of structured proofs and argue about the advantages of using them. The discussion in this section is also given in [71] and more information on the topic can be found in [13, 51, 52].

When mathematicians or computer scientists write a standard proof of some theorem, they have to decide on which level of detail they will present their proof. If they provide few details, readers may spend a lot of time filling the holes left or, even worse, may not understand why a specific step of the proof is correct. However, more argumentation is not necessarily always better, since this may obscure the "big picture" and some readers may be more interested in seeing the "big picture" than in checking every tiny detail of the proof. Hence, the ideal level of detail in a mathematical proof varies from reader to reader, as it depends on the reader's background and the reader's intention (checking that every step of the proof is correct vs seeing the "big picture" and getting an intuition on why it works).

How to appease every type of reader? By using proof sketches, lemmas, and *structured proofs* it is possible to get close to this goal.

We illustrate the advantages of structured proofs via an example, comparing them to normal proofs. This example comes from [52] and it's about a corollary to the Mean Value Theorem, which is a theorem from calculus. Here is the corollary and the proof as presented in Spivak's Calculus textbook:



Corollary If $f'(x) > 0$ for all x in an interval, then f is increasing on the interval.

Proof Let a and b be two points in the interval with $a < b$. Then there is some x in (a, b) with

$$f'(x) = \frac{f(b) - f(a)}{b - a}.$$

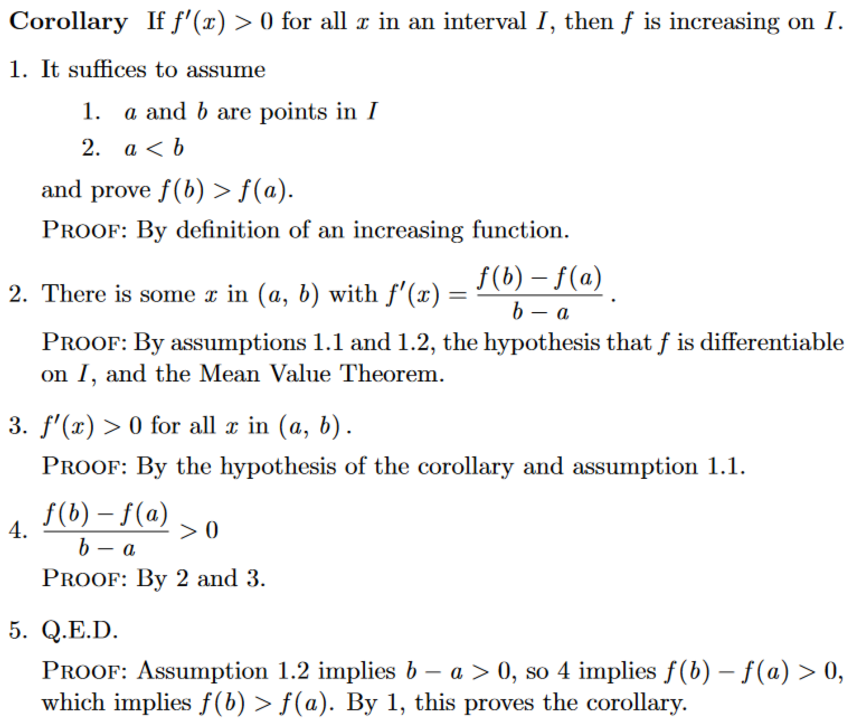
But $f'(x) > 0$ for all x in (a, b), so

$$\frac{f(b) - f(a)}{b - a} > 0.$$

Since $b - a > 0$ it follows that $f(b) > f(a)$. ▮

Figure 2.1: An Example of a Standard Non-Structured Proof.

A correspondent structured proof presented by Leslie Lamport in [52] is shown below:



Corollary If $f'(x) > 0$ for all x in an interval I, then f is increasing on I.

1. It suffices to assume
   1. a and b are points in I
   2. $a < b$

   and prove $f(b) > f(a)$.

   PROOF: By definition of an increasing function.
2. There is some x in (a, b) with $f'(x) = \dfrac{f(b) - f(a)}{b - a}$.

   PROOF: By assumptions 1.1 and 1.2, the hypothesis that f is differentiable on I, and the Mean Value Theorem.
3. $f'(x) > 0$ for all x in (a, b) .

   PROOF: By the hypothesis of the corollary and assumption 1.1.
4. $\dfrac{f(b) - f(a)}{b - a} > 0$

   PROOF: By 2 and 3.
5. Q.E.D.

   PROOF: Assumption 1.2 implies $b - a > 0$, so 4 implies $f(b) - f(a) > 0$, which implies $f(b) > f(a)$. By 1, this proves the corollary.

Figure 2.2: The Corresponding Structured Proof.

Notice that in the structured proof it is easier for a reader to see what were the necessary steps and the justification for each one. As an example, suppose that reader "A" is studying calculus for the first time and wants to make sure he understands everything. Then, reader "A" can read steps 1–5 to get the "big picture" and then understand each step by reading the proof for that step. What about a reader "B", who (let's say) already knows a lot of calculus? "B" can just read steps 1–5 and be convinced that the proof works. Now imagine that a struggling reader "C" did not understand the justification of step 2. What can we do? We simply expand the justification of step 2 in a structured manner, as shown below (this also comes from Lamport [52]):

2. There is some x in $(a,\ b)$ with $f'(x) = \dfrac{f(b) - f(a)}{b - a}$.
   - 2.1. f is differentiable on $[a,\ b]$.
     PROOF: By 1.1, since f is differentiable on I by hypothesis.
   - 2.2. f is continuous on $[a,\ b]$.
     PROOF: By 2.1 and Theorem 1.
   - 2.3. Q.E.D.
     PROOF: By 2.1, 2.2, and the Mean Value Theorem.

Figure 2.3: Expanding the Proof of Step 2 in a Structured Way.

If we used the conventional, non-structured way of writing proofs, we would not be able to please everyone, since everyone would be reading the proof in the same way and the level of detail would not satisfy readers “A” “B” and “C” simultaneously.

Due to the advantages discussed above, in this thesis we present the proofs in a structured manner.

2.5 The PVS Proof Assistant

An interactive theorem prover, also called proof assistant, is a software used to help humans with the development of formal proofs. A verified proof has all of its steps accepted as correct by the proof assistant, which diminishes the probability of wrong proofs. However, it is important to point out that wrong proofs can still occur, for instance if a given mathematical theory is wrongly defined in the proof assistant. Examples of interactive proof assistants are Coq [17], Isabelle/Hol [60], PVS [62] and Lean [34].

PVS[3] is an interactive proof assistant based on higher-order logic developed at SRI International since 1990 [62,70]. It extends Church’s simply typed higher-order logic with practical features such as algebraic data types, dependent predicate subtypes, parametric theories and theory interpretations. Additionally, it has effective proof automation (for instance by using SMT or other decision procedures).

The formalisations described in this work were done using the proof assistant PVS. Specifications of mathematical definitions and statement of lemmas/theorems/corollaries are kept in `.pvs` files, while the corresponding formalisations are found in `.prf` files.

[3]PVS officially stands for “Prototype Verification System” but it is sometimes extraoficially called “People’s Verification System”, an acronym that was created by John Rushby [70].

Although there were many different proof assistants we could have chosen to do the formalisation, we opted for PVS for three different reasons. The first was to reuse a great portion of definitions and lemmas from [12] (a formalisation of nominal C-unification that is *not* generalised with protected variables), instead of proving them from scratch. The second is the support offered by PVS to specify functional algorithms. Finally, we had previous experience using PVS.

2.5.1 TCCs - Type Correctness Conditions

When specifying functions and theorems, PVS may generate proof obligations that must be satisfied. These proof obligations are called Type Correctness Conditions (TCCs) and the PVS system includes several pre-defined proof strategies that automatically discharge simple TCCs. The more elaborate TCCs that PVS cannot automatically prove must be proved manually by the user.

2.5.2 PVSio

PVS does not support code extraction to a functional programming language like Haskell or OCaml. Nevertheless, it has the PVSio package which extends the capabilities of the ground evaluator with a predefined library of imperative programming language features, among them input and output operators [58].

This implies that *in some cases* we can run the formalised algorithm inside the PVS environment passing the input we want and seeing the output returned. However, some code fragments cannot be handled by PVSio. For instance, the function `divides` is used in our formalisation when solving the Diophantine equations and is defined as follows:

```
divides(n, m): bool = EXISTS x : m = n * x
```

PVSio cannot be used when the algorithm relies on code fragments such as `divides` that use the PVS reserved word `EXISTS`. Hence, fragments of the algorithm that rely on this should be replaced by equivalent fragments specified in a “procedural manner”. Specifying the equivalent fragments should be straightforward, but proving that the two fragments are indeed the same for every case requires some effort. For the case of `divides`, one could specify and use instead `divides_alt`:

```
divides_alt(n, m): RECURSIVE bool =
  IF m = 0 OR m - n = 0 THEN TRUE
  ELSIF m - n < 0 THEN FALSE
  ELSE divides_alt(n, m-n)
  ENDIF
```

```
MEASURE m
```

Specifying the equivalent fragments is usually straightforward, but proving that the two fragments give the same result under any circumstance requires more effort.

PVSio can be combined with semantic attachments in cases where the code is not fully executable (see [32, 36]). This allows us to animate a specification, i.e. make a specification actually perform a calculation. For instance, in the case of `divides` one could use the following semantic attachment:

```
(defattach |divides.divides| (m n)
  "Returns TRUE if n divides m"
  (multiple-value-bind (mod rem) (floor m n) (= rem 0)))
```

Then, going back to PVSio one would obtain:

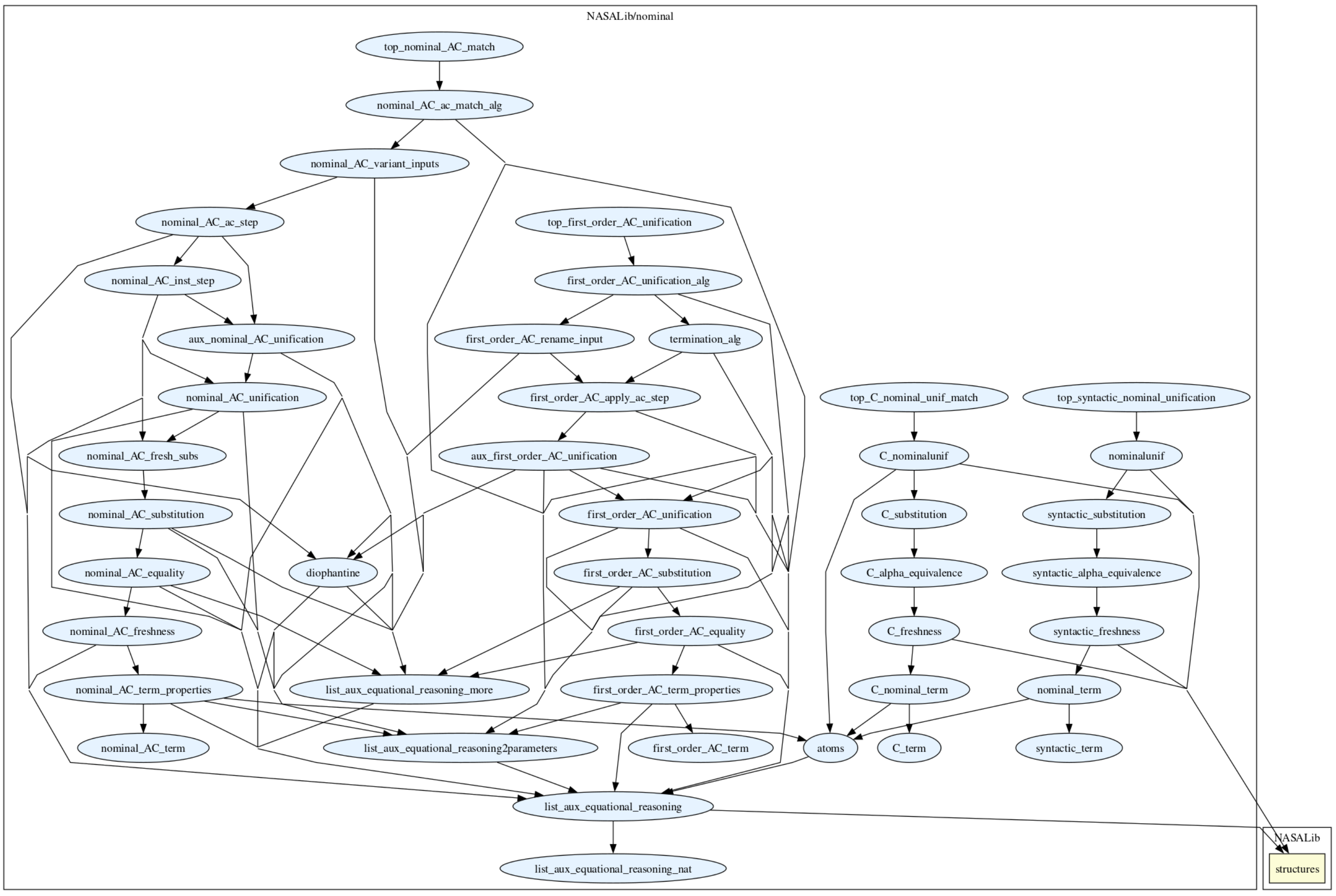
```
<PVSio> divides(10,2);
==>
TRUE

<PVSio> divides(10,3);
==>
FALSE
```

2.6 NASALib and the Nominal Library

NASALib is the main repository for PVS formalisations. It consists of over 60 top-level libraries, with over 38K proven formulas. The formalisations described in this thesis are part of the nominal library of NASALib, which consists of four main results:

- A sound and complete Nominal Syntactic Unification Algorithm. This formalisation is described in [11] and is not part of this work.
- A sound and complete Nominal C-Unification Algorithm generalised with protected variables. This formalisation is described in [3] and in Chapter 3.
- A sound and complete First-order AC-Unification Algorithm. This formalisation is described in [9] and in Chapter 4.
- A sound and complete Nominal AC-Matching Algorithm. This formalisation is described in [8] and in Chapter 5.


NASALib/nominal
top_nominal_AC_match
nominal_AC_ac_match_alg
nominal_AC_variant_inputs
nominal_AC_ac_step
top_first_order_AC_unification
nominal_AC_inst_step
first_order_AC_unification_alg
aux_nominal_AC_unification
first_order_AC_rename_input
termination_alg
nominal_AC_unification
first_order_AC_apply_ac_step
top_C_nominal_unif_match
top_syntactic_nominal_unification
nominal_AC_fresh_subs
aux_first_order_AC_unification
C_nominalunif
nominalunif
nominal_AC_substitution
first_order_AC_unification
C_substitution
syntactic_substitution
nominal_AC_equality
diophantine
first_order_AC_substitution
C_alpha_equivalence
syntactic_alpha_equivalence
nominal_AC_freshness
first_order_AC_equality
C_freshness
syntactic_freshness
nominal_AC_term_properties
list_aux_equational_reasoning_more
first_order_AC_term_properties
C_nominal_term
nominal_term
nominal_AC_term
list_aux_equational_reasoning2parameters
first_order_AC_term
atoms
C_term
syntactic_term
list_aux_equational_reasoning
list_aux_equational_reasoning_nat
NASALib
structures
Figure 2.4: Hierarchy of the Nominal library.

The hierarchy of the files in the nominal library is shown in Figure 2.4.

In Chapters 3, 4 and 5 we detail files that are specific to Items (2), (3) and (4), but we comment here some files that are common to all the theories:

- `atoms` - Definition and properties about permutations and their actions on atoms.
- `Diophantine` - Code to solve Diophantine equations (used by the formalisation of first-order AC-unification and nominal AC-matching).
- `list_aux_equational_reasoning`, `list_aux_equational_reasoning2parameters`, `list_aux_equational_reasoning_more` and `list_aux_equational_reasoning_nat` - Set of parametric theories that define specific functions for the task of equational reasoning (most of them operating on lists).
- `structures` - This is a different library that is being used by the nominal library, with results about data structures.

Remark 11. *The files `top_nominal_AC_match.pvs`, `top_first_order_AC_unification.pvs`, `top_C_nominal_unif_match.pvs` and `top_syntactic_nominal_unification.pvs` only contain high-level descriptions of the formalisations of nominal AC-matching, first-order AC-unification, nominal C-unification generalised with protected variables and nominal syntactic unification. These files do not contain theorem specifications and therefore, there is no `.prf` files associated with them, as there is no proof of any theorem.*

Finally, in Table 2.4 we give the main information for every file in the NASALib library. Since there are many files, we separated the rows in the table in 5 parts (this separation between parts is done by a solid lines). The first part consists of the first 6 rows and corresponds to files that were used by more than one of the 4 formalisations. The second part consists of the files of the nominal syntactic unification formalisation, the third part consists of the files of the nominal C-unification formalisation, the fourth part consists of the files of the first-order AC-unification formalisation and the fifth part consists of the files of the nominal AC-matching formalisation. We omit from Table 2.4 the four files `top_<name_of_the_formalisation>.pvs` since those files only contain high-level descriptions (see Remark 11).

Table 2.4: Information for Every File in the Nominal Library.

Theory	**Theorems**	**TCCs**	**Size**		
			`.pvs`	`.prf`	%
`list_aux_equational_reasoning_nat`	3	5	3 kB	0.01 MB	< 0.1 %
`list_aux_equational_reasoning`	210	84	45 kB	1 MB	1.3%
`list_aux_equational_reasoning_more`	34	13	8 kB	1.1 MB	1.5%
`list_aux_equational_reasoning2parameters`	17	6	5 kB	0.04 MB	0.1%
`atoms`	14	3	5 kB	0.03 MB	< 0.1%
`Diophantine`	73	44	24 kB	1.1 MB	1.5%
`nominalunif`	2	17	4 kB	0.6 MB	0.8 %
`syntactic_substitution`	38	7	13 kB	0.4 MB	0.5 %
`syntactic_alpha_equivalence`	15	7	6 kB	0.3 MB	0.4 %
`syntactic_freshness`	9	6	5 kB	0.08 MB	0.1%
`nominal_term`	7	4	5 kB	0.04 MB	0.1%
`C_nominalunif`	29	24	21 kB	6.3 MB	8.5%
`C_substitution`	73	14	22 kB	0.6 MB	0.8%
`C_alpha_equivalence`	14	8	5 kB	0.3 MB	0.4%
`C_freshness`	9	7	5 kB	0.04 MB	0.1%
`C_nominal_term`	9	7	6 kB	0.04 MB	0.1%
`first_order_AC_unification_alg`	10	19	6 kB	2.3 MB	3.1%
`first_order_AC_renamed_inputs`	21	23	10 kB	2.7 MB	3.6%
`first_order_AC_termination_alg`	80	35	23 kB	11 MB	14.8%
`first_order_AC_apply_ac_step`	29	12	15 kB	9.7 MB	13.1%
`aux_first_order_AC_unification`	204	58	59 kB	8.2 MB	11.0%

Table 2.4: Information for Every File in the Nominal Library.

Theory	**Theorems**	**TCCs**	**Size**		
			`.pvs`	`.prf`	%
`first_order_AC_unification`	86	14	20 kB	1.0 MB	1.3%
`first_order_AC_substitution`	144	22	27 kB	2.4 MB	3.2%
`first_order_AC_AC_equality`	67	18	12 kB	1.1 MB	1.5%
`first_order_AC_terms`	131	48	28 kB	1.1 MB	1.5%
`nominal_AC_ac_match_alg`	22	35	12 kB	2.6 MB	3.5%
`nominal_AC_variant_inputs`	22	5	8 kB	1.4 MB	1.9%
`nominal_AC_ac_step`	48	11	13 kB	1.6 MB	2.2%
`nominal_AC_inst_step`	75	17	21 kB	2.1 MB	2.8%
`aux_nominal_AC_unification`	152	52	49 kB	7.1 MB	9.6%
`nominal_AC_unification`	120	13	28 kB	1.8 MB	2.4%
`nominal_AC_fresh_subs`	38	5	12 kB	0.6 MB	0.8%
`nominal_AC_substitution`	175	36	30 kB	2.6 MB	3.5%
`nominal_AC_equality`	83	20	15 kB	1.7 MB	2.3%
`nominal_AC_freshness`	15	10	5 kB	0.1 MB	0.1 %
`nominal_AC_terms`	147	53	30 kB	1.2 MB	1.6 %
Total	2225	762	605 kB	74.3MB	100%

Chapter 3

Nominal C-Unification Generalised With Protected Variables

This chapter describes how we extended the functional nominal C-unification[1] algorithm from Ayala-Rincón et al. [11], adding a parameter $\mathcal{X}$ for variables that cannot be instantiated, and obtained a nominal C-unification algorithm generalised with protected variables. Given a unification problem P, this generalised algorithm can be used to the task of unification, matching and α-equality checking by correctly setting the parameter $\mathcal{X}$ to $\emptyset$, $Vars(rhs(P))$ or $Vars(P)$ respectively. The extended algorithm has been formalised in the PVS proof assistant. Moreover, we tested the correctness of a Python manual implementation of the algorithm using the PVS formalisation, through the PVSio feature (see Section 2.5.2). Most of the content of this Chapter is described in Ayala-Rincón et al. [3].

Remark 12 (Difference Between This Chapter and [3])**.** *Although both [3] and this Chapter describe a nominal C-unification algorithm generalised with protected variables, there are two differences between them. The main difference is that [3] also describes how a set of inductive rules for nominal C-unification specified in Coq can be extended to handle matching and α-equivalence. A second difference is that this Chapter describes the statistics of the PVS formalisation and the hierarchy of the PVS files (see Section 3.5) in more details than [3].*

[1]A set X equipped with a commutative operator $+$ that is closed over X but not necessarily associative defines an algebraic structure $(X, +)$ called commutative magma or commutative groupoid. Commutative magmas have been used to model a variety of problems, including the NAND logic gate and the rock-paper-scissors game.

3.1 Specifying Unification Via Set of Rules and Via Algorithms

As mentioned, Ayala-Rincón et al. [3] showed how nominal C-unification could be formalised as a set of non-deterministic inference rules in Coq and as a recursive algorithm in PVS. In this section we discuss the advantages and drawbacks of both approaches.

On one hand, in a rule-based specification, the unification problem is progressively transformed into a simpler one by the rules. This elegant approach has a higher level of abstraction than the algorithmic way, which can simplify the analysis of some computational properties such as correctness and completeness of solutions.

On the other hand, the rule-based approach has the drawback that from a specification of these non-deterministic rules we cannot extract executable code directly. Instead, from a set of non-deterministic inductive rules one usually obtains a recursive algorithm by providing a heuristic on how to apply the rules and then extracts executable code. In this case, one can use the formalised computational properties of the non-deterministic rules (soundness, completeness, termination...) to prove the corresponding properties for the algorithm. Finally, once we have extracted executable code of an algorithm, there are two possibilities: we can use it directly or use it to test the correctness of manual implementations of the algorithm, which may contain optimisations and are usually faster.

Finally, the choice of proof assistant may play a role in the approach used to formalise unification. The Coq and the PVS proof assistants support both approaches to formalise unification, although inductive formalisations via set of rules are more common in Coq (e.g. [4, 75]) and recursive formalisations are more common in PVS (e.g. [11, 12]).

3.2 Main Algorithm

Algorithm 1 is a functional algorithm for nominal C-unification that let us unify two terms t and s. By using the appropriate set of protected variables, the algorithm can be adapted to do C-matching and C-equality checking. The algorithm is recursive and keeps track of the protected variables, the current context, the substitutions done so far, the remaining terms left to unify and the current fixpoint equations. Therefore, the algorithm receives as input a quintuple $(\mathcal{X}, \Delta, \sigma, P, FP)$, where $\mathcal{X}$ is the set of protected variables, Δ is the context we are working with, σ represents the substitutions already made, P is a list of equations we must still solve (each equation $t \approx^? s$ is represented as a pair (t, s) in Algorithm 1) and FP is a list of fixpoint equations we have already computed.

The first call to the algorithm in order to unify the terms t and s is done with $\mathcal{X} = \emptyset$, $\Delta = \emptyset$, $\sigma = id$, $P = \{t \approx^? s\}$ and $FP = \emptyset$. The algorithm eventually terminates, returning a list (possibly empty) of triples of the form (Δ, σ, FP).

Although long, the algorithm is simple. It starts by analysing the list of terms it needs to unify. If P is an empty list, then it has finished and can return the answer computed so far, which is a list with only one element: (Δ, σ, FP). If P is not empty, then there are terms to unify, and the algorithm starts by trying to unify the terms t and s in the head of the list. The algorithm calls itself on progressively simpler versions of the problem until it finishes.

The pseudocode for the algorithm is presented in Algorithm 1. Although in the PVS specification all fixpoint equations are stored in FP, in the pseudocode here presented we show how fixpoint equations $\pi \cdot X \approx^? X$ with $X \in \mathcal{X}$ can be solved. In relation to the algorithm presented in [12], there are three changes. First, the addition of the parameter $\mathcal{X}$ for a set of protected variables, which remains constant in the execution of the algorithm. Second there is the check to see if X is in $\mathcal{X}$ or not in lines 5 and 14 to decide whether there will be an instantiation or not. Third, the algorithm solves fixpoint equations with protected variables in lines 16-17.

Remark 13 (Minor Changes on the Pseudocode of CUnif)**.** *In comparison to [3] we made minor stylistic changes in the pseudocode of Algorithm 1, to follow the same style of Algorithm 2 (certified first-order AC-unification) and Algorithm 5 (nominal AC-matching).*

Remark 14 (Terms in Nominal C-unification)**.** *In the formalisation of first-order AC unification (Chapter 4) and of nominal AC-matching (Chapter 5) we restrict ourselves to well-formed terms. In this formalisation we do not, although we impose in the grammar of term that every commutative function application receives a pair, i.e. for every (sub)term of the form $f^C t$, t is necessarily a pair.*

3.2.1 Auxiliary functions

Following the approach of [11], freshness constraints are handled by auxiliary functions, making the main function CUnif smaller. To deal with the freshness constraints, the following auxiliary functions, which come from [11] and were extended to also handle commutative function applications, were used:

- freshSubs?(σ, Δ) recursively returns the minimal context (Δ' in Algorithm 1) in which $a \#_? X\sigma$ holds, for every $a \# X$ in the context Δ, and a boolean (*flag* in Algorithm 1), indicating if it was possible to find the mentioned context.

Algorithm 1 Functional Nominal C-Unification

```
procedure CUnif($\mathcal{X}, \Delta, \sigma, P, FP$)
    if nil?($P$) then  cons(($\Delta, \sigma, FP$), NIL)
    else
        let cons(($t, s$), $P_1$) = $P$ in
        if ($s$ matches $\pi \cdot X$) and ($X$ not in $t$) and ($X$ not in $\mathcal{X}$) then
            let $\sigma_1 = \{X \mapsto \pi^{-1} \cdot t\}$,
                $(\Delta_1, \mathit{flag})$ = FRESHSUBS?$(\sigma_1, \Delta)$ in
            if $\mathit{flag}$ then CUnif($\mathcal{X}$,  $\Delta_1 \cup \Delta$,  $\sigma_1\sigma$,  $\sigma_1 P_1 \cup \sigma_1 FP$,  NIL)
            else NIL
        else
            if $t$ matches $a$ and $s$ matches $a$ then CUnif($\mathcal{X}, \Delta, \sigma, P_1, FP$)

            else if $t$ matches $\pi \cdot X$ then
                if $X$ not in $s$ and $X$ not in $\mathcal{X}$ then
                                        ▷ Similar to case of lines 5-9, swapping $t$ and $s$
                else if $s$ matches $\pi' \cdot X$ and $X$ in $\mathcal{X}$ then
                    CUnif($\mathcal{X}$,  $\Delta \cup ds(\pi, \pi')\#X$,  $\sigma$,  $P_1$,  $FP'$)
                else if $s$ matches $\pi' \cdot X$ and $X$ not in $\mathcal{X}$ then
                    CUnif($\mathcal{X}, \Delta, \sigma, P', FP \cup \{\pi \cdot X \approx^? \pi' \cdot X\}$)
                else NIL

            else if $t$ matches $\langle\rangle$ and $s$ matches $\langle\rangle$ then CUnif($\mathcal{X}, \Delta, \sigma, P_1, FP$)

            else if $t$ matches $\langle t_1, t_2\rangle$ and $s$ matches $\langle s_1, s_2\rangle$ then
                CUnif($\mathcal{X}, \Delta, \sigma, \{t_1 \approx^? s_1, t_2 \approx^? s_2\} \cup P_1, FP$)

            else if $t$ matches $[a]t_1$ and $s$ matches $[a]s_1$ then
                CUnif($\mathcal{X}, \Delta, \sigma, \{t_1 \approx^? s_1\} \cup P_1, FP$)

            else if $t$ matches $[a]t_1$ and $s$ matches $[b]s_1$ then
                let $(\Delta_1, \mathit{flag})$ = FRESH?$(a, s_1)$ in
                if $\mathit{flag}$ then CUnif($\mathcal{X}, \Delta_1 \cup \Delta, \sigma, \{t_1, \approx^? (a\ b) \cdot s_1)\} \cup P_1, FP$)
                else NIL

            else if $t$ matches $f\ t_1$ and $s$ matches $f\ s_1$ then        ▷ f is not commutative
                CUnif($\mathcal{X}, \Delta, \sigma, \{t_1 \approx^? s_1\} \cup P_1, FP$)

            else if $t$ matches $f^C(t_1, t_2)$ and $s$ matches $f^C(s_1, s_2)$ then
                let $sol_1$ = CUnif($\mathcal{X}, \Delta, \sigma, \{t_1 \approx^? s_1, t_2 \approx^? s_2\} \cup P_1, FP$),
                    $sol_2$ = CUnif($\mathcal{X}, \Delta, \sigma, \{t_1 \approx^? s_2, t_2 \approx^? s_1\} \cup P_1, FP$) in
                APPEND$(sol_1, sol_2)$
            else NIL
```

- FRESH?(a, t) recursively computes and returns the minimal context (Δ' in Algorithm 1) in which a is fresh in t, and a boolean (*flag* in Algorithm 1), indicating if it was possible to find the mentioned context.

3.3 Interesting Points on Adapting the Algorithm to Handle Protected Variables

3.3.1 Termination

The proof of termination was straightforward and follows the same reasoning of [12]. First, we need the notion of the size of a unification problem P (Definition 24).

Definition 24 (Size of P). *The size of a unification problem P is the sum of the size of every equational constraint $t \approx^? s$ in P. The size of an equational constraint $t \approx^? s$ was defined to be size(t), recursively defined as follows:*

- $\mathit{size}(a) = 1.$
- $\mathit{size}(\pi \cdot X) = 1.$
- $\mathit{size}(\langle\rangle) = 1.$
- $\mathit{size}([a]t_1) = 1 + \mathit{size}(t_1)$
- $\mathit{size}(\langle t_1, t_2\rangle) = 1 + \mathit{size}(t_1) + \mathit{size}(t_2).$
- $\mathit{size}(f\ t_1) = 1 + \mathit{size}(t_1).$
- $\mathit{size}(f^C\ t_1) = 1 + \mathit{size}(t_1).$

The lexicographic measure used was:

$$\mathit{lex} = (|\mathit{Vars}(P) \cup \mathit{Vars}(FP)|, \mathit{size}(P)).$$

In comparison with the Coq specification of nominal C-unification with protected variables (see [3]), we were able to reduce the number of components in the lexicographic measure from four parameters to only two, simplifying the proof of termination. The two extra components in the lexicographic measure of the Coq formalisation count the "size of all the freshness constraints" and the "number of equations in the problem that are not fixpoint equations". First, by using separate functions to solve freshness constraints (see their description in Section 3.2.1) instead of solving them in the main function CUnif we got rid of the first extra component. Second, by separating fixpoint equations in a different parameter FP as soon we identify them in P, the component $size(P)$ also diminishes when we "move" a fixpoint equation from P to FP.

Table 3.1 shows which component decrease in each recursive call of Algorithm 1.

Table 3.1: Decrease of the Components of the Lexicographic Measure.

Recursive Call	$\lvert Vars(\mathbf{P}) \cup Vars(\mathbf{FP})\rvert$	$\mathbf{size(P)}$
lines 11, 17, 19, 22, 25, 28, 32, 36, 39, 40	$\leq$	$<$
line 8, 15	$<$	

3.3.2 Valid Quintuples and Solution to an Input in Nominal C-Unification

Before presenting the proofs of soundness and completeness, we need the notions of valid quintuples and solution to an input.

Valid Quintuples

Valid quintuples have valuable properties and are preserved between the recursive calls of CUnif. The corresponding concept in the formalisation of first-order AC-unification (Definition 34) and in the formalisation of nominal AC-matching (Definition 37) is the notion of nice inputs. If we compare the definitions of nice inputs (Definition 34 and 37) with the definition of the valid quintuples, we see that less items were necessary in the definition of valid quintuples.

Definition 25 (Valid Quintuple). *The input* $(\mathcal{X}, \Delta, \sigma, P, FP)$ *is a valid quintuple if:*

- σ *is idempotent.*
- $dom(\sigma) \cap (Vars(P) \cup Vars(FP)) = \emptyset$.

Solution to an Input in Nominal C-Unification

Definition 22 is solution for a quintuple $(\Gamma, P, \sigma, V, \mathcal{X})$ and was used in the formalisation of nominal AC-matching. Since the nominal C-unification algorithm works with input $(\mathcal{X}, \Gamma, \sigma, P, FP)$ the notion of solution of an input is basically the same of Definition 22, with changes essentially in the input order.

Definition 26 (Solution to an Input in Nominal C-Unification). In Nominal C-Unification *we say that* (Δ, δ) *is a solution to an input* $(\mathcal{X}, \Gamma, \sigma, P, FP)$ *if:*

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

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

3. *if $t \approx^? s \in P$ or $t \approx^? s \in FP$ then $\Delta \vdash \delta t \approx_\alpha \delta s$.*

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

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

Explaining Definition 26 in terms of Definition 22, this means that (Δ, δ) will be a solution to input $(\mathcal{X}, \Gamma, \sigma, P, FP)$ if (Δ, δ) is a solution to the associated quintuple $(\Gamma, P \cup FP, \sigma, \mathbb{X}, \mathcal{X})$. Recall that $\mathbb{X}$ is the set of all variables and hence $\Delta \vdash \lambda\sigma \approx_{\mathbb{X}} \delta$ is equivalent to $\Delta \vdash \lambda\sigma \approx \delta$.

3.3.3 Soundness

We formalised soundness of unification and matching (Corollaries 2 and 3). These corollaries rely on Theorem 1.

Theorem 1 (Main Theorem for Soundness of CUNIF). *Suppose that $(\Delta_{sol}, \sigma_{sol}, FP_{sol}) \in$ CUNIF$(\mathcal{X}, \Delta, \sigma, P,\ FP)$, (∇, δ) is a solution to $(\mathcal{X}, \Delta_{sol}, \sigma_{sol}, \emptyset, FP_{sol})$ and $(\mathcal{X}, \Delta, \sigma, P, FP)$ is a valid quintuple. Then (∇, δ) is a solution to $(\mathcal{X}, \Delta, \sigma, P, FP)$.*

PROOF SKETCH: The proof is done by induction and is essentially the same as the corresponding theorem for C-unification in [12].

Corollary 2 (Soundness of CUNIF for Unification). *Suppose (∇, δ) is a solution to $(\emptyset, \Delta_{sol}, \sigma_{sol}, \emptyset, FP_{sol})$, and $(\Delta_{sol}, \sigma_{sol}, FP_{sol}) \in$ CUNIF$(\emptyset, \emptyset, id, \{t \approx^? s\}, \emptyset)$. Then (∇, δ) is a solution to $(\emptyset, \emptyset, id, \{t \approx^? s\}, \emptyset)$.*

PROOF: Notice that $(Vars(s), \emptyset, id, \{t \approx^? s\}, \emptyset)$ is a valid quintuple. Then, we apply Theorem 1 and prove the corollary.

Corollary 3 (Soundness of CUNIF for Matching). *Suppose (∇, δ) is a solution to $(Vars(s), \Delta_{sol}, \sigma_{sol}, \emptyset, FP_{sol})$, and $(\Delta_{sol}, \sigma_{sol}, FP_{sol}) \in$ CUNIF$(Vars(s), \emptyset, id, \{t \approx^? s\}, \emptyset)$. Then (∇, δ) is a solution to $(Vars(s), \emptyset, id, \{t \approx^? s\}, \emptyset)$.*

PROOF: Notice that $(Vars(s), \emptyset, id, \{t \approx^? s\}, \emptyset)$ is a valid quintuple. Then, we apply Theorem 1 and prove the corollary.

An interpretation of Corollary 2 (3) is that if (∇, δ) is a unification (matching) solution to one of the outputs of the algorithm CUNIF, then it is a unification (matching) solution to the original problem.

3.3.4 Completeness

We formalised completeness of unification and matching (Corollaries 5 and 6). They rely on Theorem 4.

Theorem 4 (Main Theorem for Completeness of CUnif). *Suppose* (∇, δ) *is a solution to* $(\mathcal{X}, \Delta, \sigma, P, FP)$ *and that* $(\mathcal{X}, \Delta, \sigma, P, FP)$ *is a valid quintuple. Then, there exists* $(\Delta_{sol}, \sigma_{sol}, FP_{sol})$ *such that:*

1. $(\Delta_{sol}, \sigma_{sol}, FP_{sol}) \in \textsc{CUnif}(\mathcal{X}, \Delta, \sigma, P, FP)$.
2. (∇, δ) *is a solution to* $(\mathcal{X}, \Delta_{sol}, \sigma_{sol}, \emptyset, FP_{sol})$.

Proof sketch: The proof is done by induction and is essentially the same as the corresponding theorem for C-unification in [12].

Corollary 5 (Completeness of CUnif for Unification). *Suppose* (∇, δ) *is a solution to the input quintuple* $(\emptyset, \emptyset, id, \{t \approx^? s\}, \emptyset)$*. Then, there exists* $(\Delta_{sol}, \sigma_{sol}, FP_{sol})$ *such that:*

1. $(\Delta_{sol}, \sigma_{sol}, FP_{sol}) \in \textsc{CUnif}(\emptyset, \emptyset, id, \{t \approx^? s\}, \emptyset)$.
2. (∇, δ) *is a solution to* $(\emptyset, \Delta_{sol}, \sigma_{sol}, \emptyset, FP_{sol})$.

Proof: Notice that $(\emptyset, \emptyset, id, \{t \approx^? s\}, \emptyset)$ is a valid quintuple. Then, we apply Theorem 4 and prove the corollary.

Corollary 6 (Completeness of CUnif for Matching). *Suppose* (∇, δ) *is a solution to the input quintuple* $(\mathit{Vars}(s), \emptyset, id, \{t \approx^? s\}, \emptyset)$*. Then, there exists* $(\Delta_{sol}, \sigma_{sol}, FP_{sol}) \in$ *such that:*

1. $(\Delta_{sol}, \sigma_{sol}, FP_{sol}) \in \textsc{CUnif}(\mathit{Vars}(s), \emptyset, id, \{t \approx^? s\}, \emptyset)$.
2. (∇, δ) *is a solution to* $(\mathit{Vars}(s), \Delta_{sol}, \sigma_{sol}, \emptyset, FP_{sol})$.

Proof: Notice that $(\mathit{Vars}(s), \emptyset, id, \{t \approx^? s\}, \emptyset)$ is a valid quintuple. Then, we apply Theorem 4 and prove the corollary.

An interpretation of Corollary 5 (6) is that if (∇, δ) is a unification (matching) solution to the original problem, then it is a solution to one of the outputs of CUnif.

3.3.5 Possible Pitfalls

Finally, possible pitfalls when adapting a recursive formalisation of C-unification to also handle C-matching are described in Remarks 15 and 16.

Remark 15 (Equational Constraints with Protected Variables)**.** *If the algorithm encounters non fixpoint equations of the form $\pi \cdot X \approx^? s$, where X in $\mathcal{X}$, it cannot simply return an empty list, since their solubility depends on the form of s. Indeed, if s is a non protected moderated variable, say $\pi' \cdot Y$, the equation $\pi \cdot X \approx^? \pi' \cdot Y$ has solutions of the form $Y \mapsto (\pi'^{-1} :: \pi) \cdot X$.*

Remark 16 (Considerations on the Parameter $\mathcal{X}$)**.** *The theorems of soundness and completeness of the algorithm had to be specified again, as the algorithm now has a new parameter $\mathcal{X}$ for the protected variables. If one is interested only in C-matching, one might wonder if it is not possible to plug in Vars(rhs(P)) as the set of protected variables $\mathcal{X}$ directly. However, since the proofs of correctness and completeness are done by induction and from one recursive call of the algorithm to another the set Vars(rhs(P)) may change, this does not work. The correct way to proceed is to prove the soundness and completeness of the algorithm with an arbitrary set of protected variables $\mathcal{X}$ and then, by a suitable choice of $\mathcal{X}$, obtain as corollaries the correctness of the algorithm for unification and matching.*

3.3.6 Examples of CUnif

Example 12 illustrates the execution of the algorithm for unification, while Examples 13 and 14 illustrate the execution of the algorithm for matching.

Example 12 (Recursive Nominal C-unification)**.** *This example shows how the algorithm proceeds in order to unify $f^C\langle (a\ b)\cdot X, a\rangle$ and $f^C\langle a, b\rangle$. Notice we have $\mathcal{X} = \emptyset$ in all calls to the function* CUnif.

$$\begin{tikzcd}
 & \mathit{CU\small{NIF}}(\emptyset,\emptyset,id,\{f^C\langle (a\ b)\cdot X, a\rangle \approx^? f^C\langle a,b\rangle\},\emptyset) \arrow[dl, dash] \arrow[dr, dash] & \\
\mathit{CU\small{NIF}}(\emptyset,\emptyset,id,\{(a\ b)\cdot X \approx^? a,\ a \approx^? b\},\emptyset) \arrow[d, dash] & & \mathit{CU\small{NIF}}(\emptyset,\emptyset,id,\{(a\ b)\cdot X \approx^? b,\ a \approx^? a\},\emptyset) \arrow[d, dash] \\
\mathit{CU\small{NIF}}(\emptyset,\emptyset,\{X \mapsto b\},\{a \approx^? b\},\emptyset) \arrow[d, dash] & & \mathit{CU\small{NIF}}(\emptyset,\emptyset,\{X \mapsto a\},\{a \approx^? a\},\emptyset) \arrow[d, dash] \\
\mathit{NIL} & & \mathit{CU\small{NIF}}(\emptyset,\emptyset,\{X \mapsto a\},\mathit{NIL},\emptyset) \arrow[d, dash] \\
 & & (\emptyset,\{X \mapsto a\},\emptyset)
\end{tikzcd}$$

Example 13 (Recursive Nominal C-Matching). *Suppose our matching problem is:*

$$\{[a]\langle f(Z),\ [b](X * Y)\rangle \approx^? [b]\langle f(Z),\ [a](a * X)\rangle\},$$

where $*$ is a commutative function symbol that we write infix. This results in the execution of a nominal C-matching algorithm, with recursive function calls, as shown below:

```
$CUnif(\{X, Z\},\ \emptyset,\ id,\ \{[a]\langle f(Z),\ [b](X * Y)\rangle \approx^? [b]\langle f(Z),\ [a](a * X)\rangle\},\ \emptyset)$
  $FRESH?(a,\ \langle f(Z),\ [a](a * X)\rangle)$
    Branch 1:
    $FRESH?(a,\ f(Z))$
    $FRESH?(a,\ Z)$
      $RETURN(\{a\#Z\},\ true)$

    Branch 2:
    $FRESH?(a,\ [a](a * X))$
      $RETURN(\emptyset,\ true)$
  $RETURN(\{a\#Z\},\ true)$

$CUnif(\{X, Z\},\ \{a\#Z\},\ id,\ \{\langle f(Z),\ [b](X * Y)\rangle \approx^? \langle f((a\ b) \cdot Z),\ [b](b * (a\ b) \cdot X)\rangle\},\ \emptyset)$

$CUnif(\{X, Z\},\ \{a\#Z\},\ id,\ \{f(Z) \approx^? f((a\ b) \cdot Z),\ [b](X * Y) \approx^? [b](b * (a\ b) \cdot X)\},\ \emptyset)$

$CUnif(\{X, Z\},\ \{a\#Z\},\ id,\ \{Z \approx^? (a\ b) \cdot Z,\ [b](X * Y) \approx^? [b](b * (a\ b) \cdot X)\},\ \emptyset)$

$CUnif(\{X, Z\},\ \{a\#Z, b\#Z\},\ id,\ \{[b](X * Y) \approx^? [b](b * (a\ b) \cdot X)\},\ \emptyset)$

$CUnif(\{X, Z\}, \{a\#Z, b\#Z\},\ id,\ \{(X * Y) \approx^? (b * (a\ b) \cdot X)\},\ \emptyset)$

  Branch 1:
  $CUnif(\{X, Z\},\ \{a\#Z, b\#Z\},\ id,\ \{(X \approx^? b,\ Y \approx^? (a\ b) \cdot X)\},\ \emptyset)$
    RETURN NIL

  Branch 2:
  $CUnif(\{X, Z\},\ \{a\#Z, b\#Z\},\ id,\ \{(X \approx^? (a\ b) \cdot X,\ Y \approx^? b\},\ \emptyset)$

  $CUnif(\{X, Z\},\ \{a\#Z, b\#Z, a\#X, b\#X\},\ id,\ \{Y \approx^? b\},\ \emptyset)$
```

```
    $CU\textsc{nif}(\{X, Z\},\ \{a\#Z, b\#Z, a\#X, b\#X\},\ \{Y \mapsto b\},\ \emptyset,\ \emptyset)$
        RETURN $(\{a\#Z, b\#Z, a\#X, b\#X\},\ \{Y \mapsto b\},\ \emptyset)$

RETURN $(\{a\#Z, b\#Z, a\#X, b\#X\},\ \{Y \mapsto b\},\ \emptyset)$
```

Notice that the algorithm bifurcates in two branches of recursive calls when it encounters an equation constraint $t \approx^? s$ such that t and s are commutative functions headed by the same symbol. The first branch has no solutions, since $X \in \mathcal{X}$ cannot be instantiated to b, and therefore the algorithm returns NIL *for this branch. The second branch gives as solution:*

$$\langle\{a\#Z, b\#Z, a\#X, b\#X\},\ \{Y \mapsto b\},\ \emptyset\rangle$$

which is, since the first branch gives no solution, the output returned by the algorithm. The theorems of correctness and completeness guarantee that (∇, δ) is a matching solution to the input problem $(\{X, Z\}, \emptyset,\ id,\ \{[a]\langle f(Z),\ [b](X * Y)\rangle \approx^? [b]\langle f(Z), [a](a * X)\rangle\}, \emptyset)$ if, and only if, (∇, δ) is a matching solution to output returned by CUnif*: $(\{a\#Z, b\#Z, a\#X, b\#X\},\ \{Y \mapsto b\},\ \emptyset)$.*

Example 14 (Unsolvable Equational Constraints)**.** *This example shows how our algorithm handles an unsolvable equational constraints and compares it to the Coq non-deterministic inference rules approach. Let the equational constraint be:*

$$\langle a,\ f\langle(b\ d)\cdot X, [d]d\rangle\rangle \approx^? \langle b,\ f\langle X, [d]d\rangle\rangle)$$

This results in the execution of the nominal C-matching algorithm, with recursive calls as shown below:

```
$CU\textsc{nif}(\{X\},\ \emptyset,\ id,\ \{\langle a,\ f\langle(b\ d)\cdot X, [d]d\rangle\rangle \approx^? \langle b,\ f\langle X, [d]d\rangle\rangle\},\ \emptyset)$

$CU\textsc{nif}(\{X\},\ \emptyset,\ id,\ \{a \approx^? b,\ f\langle(b\ d)\cdot X, [d]d\rangle \approx^? f\langle X, [d]d\rangle\},\ \emptyset)$
    RETURN NIL
```

Notice that as soon as there is an unsolvable equation constraint in the head of P (the equational constraints we must still solve) the algorithm returns NIL *communicating that there are no solutions possible for our unification problem. This contrasts with the Coq non-deterministic inference rules approach for nominal C-unification also described in [3], where the rules to the unification problem are applied until it is no longer possible (if there is an unsolvable equation constraint in the problem but the rules can be applied to other equational constraints they continue to be applied).*

3.3.7 Preserving Information Regarding Protected Variables

In our approach, we keep freshness constraints related with protected variables. Such freshness information might be useful in applications, since nominal (C-)matching has direct application in nominal rewriting (which has applications in software engineering, programming languages, etc - see [41]). Consider, for instance, the nominal rewriting rule $\oplus\langle Z, Z\rangle \to 0$ and the terms $\lambda a.a\ X$ and $\lambda b.b\ X$ from the λ-calculus extended with meta-variables. In the nominal framework these terms can be represented as $\mathtt{lam}([a]\mathtt{app}\langle a, X\rangle)$ and $\mathtt{lam}([b]\mathtt{app}\langle b, X\rangle)$. Then, to check whether we can use the mentioned nominal rewriting rule to reduce term $\oplus\langle \mathtt{lam}[b]\mathtt{app}\langle b, X\rangle, \mathtt{lam}[a]\mathtt{app}\langle a, X\rangle\rangle$ one needs to solve the C-matching problem

$$\mathcal{P} = \langle \{X\},\ \emptyset,\ id,\ \{\oplus\langle Z, Z\rangle \approx^? \oplus\langle \mathtt{lam}[a]\mathtt{app}\langle a, X\rangle, \mathtt{lam}[b]\mathtt{app}\langle b, X\rangle\rangle\}\rangle.$$

Applying the algorithm for nominal C-unification, one obtains as output:

$$\langle\{a\#X, b\#X\}, \{Z \mapsto \mathtt{lam}[a]\mathtt{app}\langle a, X\rangle\}, \emptyset\rangle$$

and

$$\langle\{a\#X, b\#X\}, \{Z \mapsto \mathtt{lam}[b]\mathtt{app}\langle b, X\rangle\}, \emptyset\rangle$$

Notice that the additional freshness information about protected variables obtained during the generalised C-unification algorithm is necessary. Indeed, by condition (3) of Definition 26 we must have $\Delta \vdash \delta((a\ b) \cdot X) \approx \delta X$ and since $dom(\delta) \cap \mathit{Vars}(\mathit{rhs}(P)) = \emptyset$ this means that $\Delta \vdash (a\ b) \cdot X \approx X$. According to the rules for the α-equivalence relation, this only holds if $\{a\#X, b\#X\} \subseteq \Delta$.

3.4 Testing the Python algorithm

We have manually implemented Algorithm 1 in Python and used the input and output capabilities provided by PVSio (see Section 2.5.2) to test if the manual Python algorithm and the formalised algorithm give the same output when run with the same input.

We investigated the literature but could not find a database for unification problems. In Ayala-Rincón et al. [6] experiments are made for nominal equality-check in the presence of A, C and AC function symbols, while in Calvès and Fernández [27] experiments are made for syntactic nominal matching and nominal α-equivalence. In [6], the terms generated are ground and arbitrary choices were made with respect to the size of unification problems, the number of different atoms and the different function symbols. The focus was on the running time of the algorithm. After a term t is randomly generated, the term

s of the unification problem $t \approx^? s$ is generated by swapping arguments of commutative functions and changing the atom being abstracted in an abstraction.

In [27], experiments were made with syntactic nominal α-equivalence and ground matching problems (i.e. matching problems where there are no variables on the right hand side). The experiments were restricted to solvable problems. The focus was seeing how the running time of the algorithm depends on the size of the unification problem and the type of task (α-equivalence or matching).

In both cases, the terms generated were synthetic and some arbitrary choices were made (although these choices can be manually altered in the code, if one wants). In our tests, some arbitrary choices are also made during the term generation, which we describe now. Our approach covers the approach of [6] as we swap arguments of commutative functions and change atoms being abstracted in an abstraction. In contrast with [27] we generate both solvable and unsolvable unification problems.

To compare the Python and the PVS implementation, we generated 2000 unification problems, consisting of terms t and s to be unified and ran the implementations. By printing the Python results in the same way as the PVS implementation prints, it was possible to check whether the implementations match. We generate the term t randomly, with the same probability of generating each component of the grammar of nominal terms, i.e, the probability of generating an atom is the same as the probability of generating a moderated variable and so on. The number of different atoms, variables, function symbols and commutative function symbols was defined arbitrarily to be 10. When generating a permutation for a moderated variable the number of swappings is a random number between 0 and 10.

Finally, we generate the term s as a "copy with modifications" of the term t. These modifications and their corresponding probabilities (chosen arbitrarily) are:

- With a 10% probability we substitute part of the term t by a random moderated variable.
- With a 50% probability, if we encounter a commutative function application in t we change the order of the two arguments.
- With a 50% probability, if we encounter an abstraction $[a]t'$ we change it to a term $[b](a\ b) \cdot t'$.
- With a 10% probability, if we encounter an atom we change it to another atom. Notice that this may result in generating non unifiable terms t and s. This is precisely what we hoped to accomplish, since we also want to test the implementations when the terms are not unifiable.

Both implementations gave the same result for all 2000 unification problems, suggesting that our Python manual implementation is correct. As expected from a manual implementation, the Python code executed faster.

3.4.1 Preliminar Experiments Comparing PVSio and Python

We have made preliminar experiments comparing the time the PVS certified algorithm and the Python manual algorithm took to execute. The machine that ran the experiments has the following specifications:

- Operating System - MacOS High Sierra
- Processor - 3,6GHz Intel Core i7
- Memory - 16GB 2400 MHz DDR4
- Graphics - Radeon Pro 560 4096 MB

The running time, according to the number of terms being unified is shown in Table 3.2.

Table 3.2: Time PVSio and Python Took to Unify

Number of unification problems	**Python**	**PVSio**
1000	< 1s	43s
2000	< 1s	1min24s
10000	3s	Error - stack overflow

3.5 Statistics of the PVS Formalisation

The formalisation described in this chapter extends the functional nominal C-unification formalisation described in [12] by adding a parameter $\mathcal{X}$ for protected variables. Extending the functions and the proofs to take into account this extra parameter is not automatic. In other words, the task is interactive theorem proving, and not automated reasoning.

A similar comment applies to the nominal C-unification formalisation of [12]: it extends the formalisation of [11], adding commutative function symbols. To give an example on how the reuse of proofs for the formalisation of [12] was done, consider that a lot of proofs in [11] were done by induction on the structure of a term t. Since the grammar of terms is extended with commutative function symbols, this means that for the proofs by induction on the structure of a term t we can reuse the cases where t is not a commutative function but must complete the proof by adding the case of when t is indeed a commutative function.

Below we describe the main theories that are part of the nominal C-unification generalized with protected variables.

- `top_C_nominal_unif_match` - High Level description of the nominal C-unification formalisation.
- `C_nominalunif` - Contains function CUnif, the lemmas of soundness and completeness of unification and matching for CUnif and Definition 26 (solution to an input in nominal C-unification)
- `C_substitution` - Definition and properties about substitutions. Contains function freshSubs?.
- `C_alpha_equivalence` - The notion of equality in the nominal setting modulo commutative functions, Definition 19.
- `C_freshness` - Definition and properties about freshness. Contains function fresh?.
- `C_nominal_term` - Basic properties about terms.
- `C_terms` - The grammar of terms.
- `atoms` - Definition and properties of permutations and their actions on atoms.
- `list_aux_equational_reasoning`, `list_aux_equational_reasoning_nat` - Set of parametric theories that define specific functions for the task of equational reasoning (most of them operating on lists).

Figure 3.1 shows the dependency diagram for the PVS theories that compose our formalisation. Besides the nominal C-unification formalisation, there are other 3 formalisations in the nominal library, which we again represent in the picture as orange ellipses. As shown in Figure 3.1, some of them use theories that are also used by the nominal C-unification formalisation.

Table 3.3 shows the number of theorems and TCCs proved for each file, along with the theory's approximate size and percentage of the total size. In contrast to Table 2.4, the percentage of the total size shown here is only with respect to the files that are part of the nominal C-unification formalisation, and not the whole NASALib theory. We group theories `list_aux_equational_reasoning` and `list_aux_equational_reasoning_nat` under the name `list` since the specifics of each one are not relevant to our discussion. Finally, PVS theories `C_nominal_term` and `C_terms` are the only ones that are actually in the same file, so we group them together under the name `terms` in Table 3.3.

Table 3.3 shows that most of the effort of the formalisation is in file `C_nominalunif`. Hence, if one wants to "balance" the formalisation in the future, a possible solution

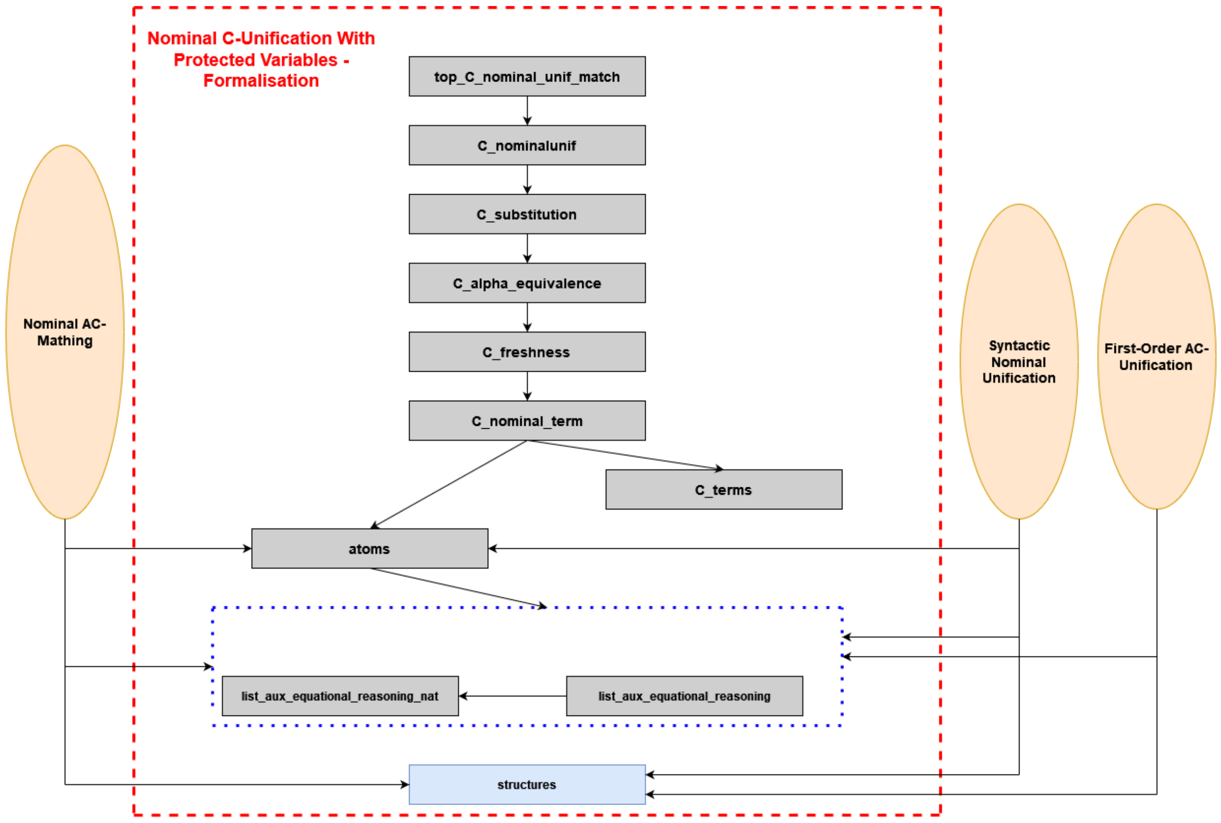

Nominal C-Unification With Protected Variables - Formalisation
top_C_nominal_unif_match
C_nominalunif
C_substitution
C_alpha_equivalence
C_freshness
C_nominal_term
C_terms
atoms
list_aux_equational_reasoning_nat
list_aux_equational_reasoning
structures
Nominal AC-Mathing
Syntactic Nominal Unification
First-Order AC-Unification

Figure 3.1: PVS formalisation of Nominal C-Unification With Protected Variables.

would be studying if some lemmas or definitions could be moved to different files. For instance, the definition of solution to input in nominal C-unification (Definition 26) along with its associated lemmas could be moved to a separate theory. Finally, although `list` responds for the 12% of the size of the formalisation in Table 3.3 this data is a bit misleading: most functions and theorems in `list_aux_equational_reasoning` and `list_aux_equational_reasoning_nat` are not used in the nominal C-unification formalisation, but instead in the first-order AC-unification and in the nominal AC-matching formalisation.

Table 3.3: Information for Every File in the Nominal C-Unification Formalisation.

Theory	**Theorems**	**TCCs**	**Size**		
			`.pvs`	`.prf`	%
`list`	213	89	48 kB	1 MB	12 %
`atoms`	14	3	5 kB	0.03 MB	<1 %
`C_nominalunif`	29	24	21 kB	6.3 MB	76 %
`C_substitution`	73	14	22 kB	0.6 MB	7 %
`C_alpha_equivalence`	14	8	5 kB	0.3 MB	4 %
`C_freshness`	9	7	5 kB	0.04 MB	1%
`terms`	9	7	6 kB	0.04 MB	1 %
Total	361	152	112 kB	8.31 MB	100 %

Chapter 4

Certified First-Order AC-Unification

This chapter describes how we gave the first formalisation of a first-order AC-unification algorithm. Our approach involved specifying Stickel's groundbreaking AC-unification algorithm and proving its termination (using an intricate lexicographic measure, rooted on Fages' termination proof), as well as its soundness and completeness. We provide a comprehensive account of the formalisation process, including explanations of the key steps in the proofs of termination, soundness, and completeness. Additionally, we delve into the files that compose the formalisation, detailing their structure, hierarchy and size. Furthermore we discuss our design choices, including the consequences of our choice for the grammar of terms. We also discuss applications of the certified AC-unification algorithm, showing how the formalisation could be used as a starting point to formalise more efficient AC-unification algorithms or to test implementations of AC-unification algorithms. The content of this chapter is also described in [9] and in [7].

4.1 Algorithm

For readability, we present the pseudocode of the algorithms, instead of the actual PVS code. We have formalised Algorithm 2 to be terminating, sound and complete. Moreover, the algorithm is functional and keeps track of the current unification problem P, the substitution σ computed so far, and the variables V that are/were in the problem. The algorithm's output is a list of substitutions, where each substitution δ in this list is a unifier of P. The first call to the algorithm, in order to unify two terms t and s, is done with $P = \{t \approx^? s\}$, $\sigma = id$ (because we have not computed any substitution yet), and $V = Vars(t, s)$.

Remark 17. *In the PVS code notation, this means that the initial call is done with parameters* $P = cons((t, s), \textsc{nil})$, $\sigma = \textsc{nil}$, *and* $V = Vars(t, s)$.

Algorithm 2 Algorithm to Solve an AC-Unification Problem P

```
procedure ACUnif($P, \sigma, V$)
    if nil?($P$) then $cons(\sigma, \text{NIL})$
    else let $((t, s), P_1) = \text{CHOOSEEQ}(P)$ in
        if ($s$ matches $X$) and ($X$ not in $t$) then
            let $\sigma_1 = \{X \mapsto t\}$ in ACUnif($\sigma_1 P_1, \sigma_1\sigma, V$)

        else
            if $t$ matches $a$ and $s$ matches $a$ then ACUnif($P_1, \sigma, V$)

            else if $t$ matches $X$ then
                if $X$ not in $s$ then
                    let $\sigma_1 = \{X \mapsto s\}$ in ACUnif($\sigma_1 P_1, \sigma_1\sigma, V$)
                else if $s$ matches $X$ then ACUnif($P_1, \sigma, V$)
                else NIL

            else if $t$ matches $\langle\rangle$ and $s$ matches $\langle\rangle$ then ACUnif($P_1, \sigma, V$)

            else if $t$ matches $f\ t_1$ and $s$ matches $f\ s_1$ then
                let $(P_2, \mathit{flag}) = \text{DECOMPOSE}(t_1, s_1)$ in
                if $\mathit{flag}$ then ACUnif($P_2 \cup P_1, \sigma, V$)
                else NIL

            else if $t$ matches $f^{AC}\ t_1$ and $s$ matches $f^{AC}\ s_1$ then
                let $\mathit{InputLst} = \text{APPLYACSTEP}(P, \text{NIL}, \sigma, V)$,
                    $\mathit{LstResults} = \text{MAP}(\text{ACUnif}, \mathit{InputLst})$ in
                FLATTEN($\mathit{LstResults}$)

            else NIL
```

The algorithm explores the structure of terms. It starts by analysing the list P of terms to unify. If it is empty (line 2), we have finished, and the algorithm returns a list containing only one element: the substitution σ computed so far. Otherwise, the algorithm calls the auxiliary function CHOOSEEQ (line 3), which returns a pair (t, s) and a unification problem P_1, such that $P = \{t \approx^? s\} \cup P_1$. The algorithm will try to simplify our unification problem P by simplifying $\{t \approx^? s\}$, and it does that by seeing what the form of t and s is.

Remark 18. *The algorithm does not check the arity consistency of the input.*

4.1.1 Function chooseEq

The function CHOOSEEQ ↗ selects a unification pair from the input problem, avoiding AC-unification pairs if possible. This means that we will only enter on the **else if** of line 23 of ACUnif (see Algorithm 2) when $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$ is such that for every i, $t_i \approx^? s_i$ is an AC-unification pair. This heuristic aids us in the proof of termination. It makes the algorithm more efficient since it guarantees that we only enter the AC-part of the algorithm when we need it (the AC-part is the computationally heaviest). Also, it is not a significant deviation from Stickel's algorithm [73].

4.1.2 Function decompose

If the function DECOMPOSE receives two terms t and s and these terms are both pairs, it recursively tries to decompose them, returning a tuple (P, flag), where P is a unification problem and *flag* is a boolean that is *True* if the decomposition was successful. If neither t nor s is a pair, the unification problem returned is just $P = \{t \approx^? s\}$ and $\mathit{flag} = \mathit{True}$. If one of the terms is a pair and the other is not, the function returns $(\textsc{nil}, \mathit{False})$. In Algorithm 2, we call DECOMPOSE $(t_1,\ s_1)$ when we encounter an equation of the form $ft_1 \approx^? fs_1$ and therefore guarantee that all the terms in the unification problem remain well-formed. Although it would have been correct to simplify an equation of the form $ft_1 \approx^? fs_1$ to $t_1 \approx^? s_1$, if t_1 or s_1 were pairs, we would not respect our restriction that only well-formed terms are in our unification problem.

Example 15. *Below, we give examples of the function* DECOMPOSE.

- DECOMPOSE$(\langle a, \langle b, c\rangle\rangle,\ \langle c, \langle X, Y\rangle\rangle) = (\{a \approx^? c,\ b \approx^? X,\ c \approx^? Y\},\ \mathit{True})$.
- DECOMPOSE$(a, Y) = (\{a \approx^? Y\},\ \mathit{True})$.
- DECOMPOSE$(X, \langle c, d\rangle) = (\textsc{nil},\ \mathit{False})$.

4.1.3 The AC-part of the Algorithm

The AC-part of Algorithm 2 relies on function APPLYACSTEP (Section 4.1.3), which depends on two functions: SOLVEAC (Section 4.1.3) and INSTANTIATESTEP (Section 4.1.3). Since there are multiple possibilities for simplifying each AC-unification pair, APPLYACSTEP will return a list (*InputLst* in Algorithm 2), where each entry of the list corresponds to a branch Algorithm 2 will explore (line 24). Each entry in the list is a triple that will be given as input to ACUnif, where the first component is the new AC-unification problem, the second component is the substitution computed so far and the third component is the new set of variables that are/were in use. After ACUnif calls APPLYACSTEP, it explores every branch generated by calling itself recursively on every input in *InputLst* (line 25 of Algorithm 2). The result of calling MAP(ACUnif, *InputLst*) is a list of lists of substitutions. This result is then flattened into a list of substitutions and returned.

Function solveAC

The function SOLVEAC does what was illustrated in the example of Section 2.2.2. While APPLYACSTEP or ACUNIF take as part of the input the whole unification problem, SOLVEAC takes only two terms t and s. It assumes that both terms are headed by the same AC-function symbol f. It also receives as input the set of variables V that are/were in the problem. Since SOLVEAC will introduce new variables, we must know the ones that are/were already in use.

The first step is eliminating common arguments of t and s. This is done by the function ELIMCOMARG, which returns the remaining arguments and their multiplicity.

To ease the formalisation we do not calculate a basis of solutions for the linear Diophantine equation, but a spanning set (which is not necessarily linearly independent). To generate this spanning set, it suffices to calculate all the solutions until an upper bound, computed by the function CALCULATEUPPERBOUND. Given a linear Diophantine equation $a_1X_1 + \ldots + a_mX_m = b_1Y_1 + \ldots + b_nY_n$, our upper bound (taken from [72]) is the maximum of m and n times the maximum of all the least common multiples (lcm) obtained by pairing each one of the a_is with each one of the b_js. In other words, our upper bound is:

$$max(m, n) * max_{i,j}(lcm(a_i, b_j)).$$

The Table 2.2 of the Example in Section 2.2.2 is represented in our code as the matrix D (see Matrix 4.1). This matrix is obtained by calling function DIOSOLVER, which receives as input the multiplicity of the arguments of t and s and the upper bound calculated by CALCULATEUPPERBOUND. Each row of D is associated with one solution and thus with one of the new variables. Each column of D is associated with one of the arguments of t or s. Modifying DIOSOLVER to calculate a basis of solutions (for instance, by using the method described in [30]) instead of a spanning set would certainly improve the algorithm's efficiency.

$$D = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{4.1}$$

To explore all possible cases, we must decide whether or not we will include each solution. In our code, this translates to considering submatrices of D by eliminating some rows. In the example of Section 2.2.2, we mentioned that we should observe two constraints:

1. no "original variable" (the variables $X_1, \ldots, X_m, Y_1, \ldots, Y_n$ associated with the arguments of t and s) should receive the value 0.
2. an "original variable" which does not represent a variable term cannot be paired with an AC-function application.

As noted by Fages in [40], in terms of our Diophantine matrix D, these two constraints are:

1. every column has at least one coefficient different from 0.
2. a column corresponding to one non-variable argument has one coefficient equal to 1 and all the remaining coefficients equal to 0.

The function in our PVS code that extracts (a list of) the submatrices of D that satisfies these constraints is extractSubmatrices. Let *SubmatrixLst* be this list.

Finally, we translate each submatrix D_1 in *SubmatrixLst* into a new unification problem P_1, by calling function dioMatrix2acSol. For instance, the unification problem

$$P_1 = \{X \approx^? Z_6, Y \approx^? Z_4, a \approx^? Z_4, b \approx^? Z_4, Z \approx^? f(Z_6, Z_6)\}$$

would be obtained from submatrix D_1:

$$D_1 = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 2 \end{pmatrix}.$$

Notice that this is the submatrix associated with a solution including only rows 4 and 6 (of the variables Z_4, Z_6).

The function dioMatrix2acSol also updates the variables that are/were in the unification problem, to include the new variables Z_is introduced. In our example, the new set of variables that are/were in the problem is $V_1 = \{X, Y, Z, Z_4, Z_6\}$. Therefore, the output of dioMatrix2acsol is a pair, where the first component is the new unification problem (in our example P_1) and the second component is the new set of variables that are/were in use (in our example V_1). The output of solveAC is the list of pairs obtained by applying dioMatrix2acSol to every submatrix in *SubmatrixLst*.

Remark 19 (New Variables Introduced by solveAC)**.** *As mentioned in Remark 2, variables in our formalisation are represented as natural numbers. When introducing new variables $Z_1, Z_2, Z_3, \ldots$ solveAC checks the parameter V to compute max(V) and internally* *represents these new variables with natural numbers* $max(V) + 1, max(V) + 2, max(V) + 3, \ldots$

Common Structure of Unification Problems Returned by solveAC

Suppose function SOLVEAC receives the terms u and v as input, both headed by the same AC-function symbol f. Let $u_1, \ldots, u_m$ be the different arguments of u and let $v_1, \ldots, v_n$ be the different arguments of v, after eliminating the common arguments of u and v. If $P_1 = \{t_1 \approx^? s_1, \ldots, t_k \approx^? s_k\}$ is one of the unification problems generated by function SOLVEAC, when it receives as input u and v then:

1. $k = m+n$ and the left-hand side of this unification problem (i.e., the terms $t_1, \ldots, t_k$) are the different arguments of u and v:

$$t_i = \begin{cases} u_i, & \text{if } i \leq m \\ v_{i-m} & \text{otherwise.} \end{cases}$$

2. The terms in the right-hand side of this problem (i.e., the terms $s_1, \ldots, s_k$) are introduced by SOLVEAC and are either new variables Z_is or AC-functions headed by f whose arguments are all new variables Z_is (This is how we obtained the problem in Equation (2.2)).

3. A term s_i is an AC-function headed by f only if the corresponding term t_i is a variable.

Function instantiateStep

After the application of function SOLVEAC, we instantiate the variables that we can by calling function INSTANTIATESTEP. For the particular case of equations $t \approx^? s$ where both t and s are variables, INSTANTIATESTEP instantiates s to t. This decision prioritizes instantiating the variables on the right-hand side and keeping the variables on the left-hand side. Recall that in the unification problems obtained immediately after calling SOLVEAC (see Section 4.1.3), the variables on the right-hand side are the new variables, while the variables in the left-hand side are variables that were in the problem before calling SOLVEAC. Indeed, as shown in Example 4, it is necessary to compose the substeps of the algorithm with some strategy to avoid infinite loops. To prevent loops such as the one of Example 4 from happening, Algorithm 2 only handles AC-unification pairs when there are no equations $t \approx^? s$ of other type left, and as soon as we apply the function SOLVEAC we immediately call function INSTANTIATESTEP.

Algorithm 3 Algorithm That Instantiates When Possible

```
procedure INSTANTIATESTEP($P_1, P_2, \sigma$)
    if nil?($P_1$) then ($P_2, \sigma, False$)
    else
        let $(t, s) = car(P_1)$, $P_1' = cdr(P_1)$ in
        if $s$ matches $X$ and $X$ not in $t$ then
            let $\sigma_1 = \{X \mapsto t\}$ in INSTANTIATESTEP($\sigma_1 P_1', \sigma_1 P_2, \sigma_1\sigma$))

        else if $t$ matches $X$ and $X$ not in $s$ then
            let $\sigma_1 = \{X \mapsto s\}$ in INSTANTIATESTEP($\sigma_1 P_1', \sigma_1 P_2, \sigma_1\sigma$)

        else if $t$ matches $X$ and $X$ matches $s$ then
            INSTANTIATESTEP($P_1', P_2, \sigma$)

        else if ($t$ matches $X$ and $X$ in $s$) or ($s$ matches $X$ and $X$ in $t$) then
            (NIL, $\sigma$, $True$) ▷ the terms $t$ and $s$ are impossible to unify

        else INSTANTIATESTEP($P_1', \{t \approx^? s\} \cup P_2, \sigma$) ▷ we skip the equation
```

Algorithm 3 is the pseudocode for INSTANTIATESTEP. It receives as input a unification problem P_1 (the part of our unification problem which we have not yet inspected), a unification problem P_2 (the part of our unification problem we have already inspected) and σ, the substitution computed so far. Therefore, the first call to this function in order to instantiate the variables in the unification problem P is with $P_1 = P$, $P_2 =$ NIL and $\sigma =$ NIL. The algorithm returns a triple, where the first component is the remaining unification problem; the second component is the substitution computed by this step; and the third component is a Boolean to indicate if we found an equation $t \approx^? s$ which is not unifiable (in this case the Boolean is $True$) or not (in this case the Boolean is $False$). The only kind of equations that INSTANTIATESTEP identifies as not unifiable are those where one of the terms is a variable, and the other term is a non-variable term that contains this variable. The algorithm works by progressively inspecting every equation $t \approx^? s \in P_1$ and deciding whether:

- One of the terms is a variable and we can instantiate (lines 5-9).
- Both terms are the same variable and we can eliminate this equation from the problem (lines 11-12).
- The terms are impossible to unify (lines 14-15).
- Neither term is a variable, and so we do not act on this equation (line 17).

Function applyACStep

Function APPLYACSTEP relies on functions SOLVEAC and INSTANTIATESTEP, and is called by Algorithm 2 when all the equations $t \approx^? s \in P$ are AC-unification pairs. In a very high-level view, it applies functions SOLVEAC and INSTANTIATESTEP to every AC-unification pair in the unification problem P.

It receives as input a unification problem, which is partitioned into sets P_1 and P_2, a substitution σ, and the set of variables to avoid V. P_1 and P_2 are, respectively, the subset of the unification problem for which functions SOLVEAC and INSTANTIATESTEP have not been called, and the subset to which we have already called these functions. The substitution σ is the substitution computed so far. Therefore, the first call to this function is with $P_2 = \text{NIL}$, and as the function recursively calls itself, P_1 diminishes while P_2 increases.

We now describe APPLYACSTEP in more details (Algorithm 4). The first thing APPLYACSTEP does is check if P_1 is the null list. If it is (line 2), we have finished applying functions SOLVEAC and INSTANTIATESTEP and we return a list with only one element: the triple (P_2, σ, V).

If P_1 is not the null list, we get the AC-unification pair in the head of the list (let us call it (t, s)) and examine whether we already have $t \approx s$. If that is indeed the case (line 4), we simply remove this equation, calling APPLYACSTEP with $(cdr(P_1), P_2, \sigma, V)$.

If t is not equal (modulo AC) to s, we call function SOLVEAC. This function will return a list of unification problems $PLst$ (line 8). Next we apply the function INSTANTIATESTEP to every problem P in $PLst$, obtaining a list $ACInstLst$ (lines 10-11), where each entry is a pair (P', δ). P' is the unification problem after we instantiate the variables and δ is the substitution computed by this function. It may happen that INSTANTIATESTEP “discovers” that a unification problem is actually unsolvable (this is communicated to APPLYACSTEP via the Boolean value that is part of the output of INSTANTIATESTEP) and in this case this problem is not included in $ACInstLst$.

We check if $ACInstLst$ is null (in this case there are no solutions to the first AC-unification pair, and therefore there are no solutions to the problem) and return NIL if it is. If $ACInstLst$ is not null (lines 14-19), there will be branches to explore. Given an entry (P', δ) of $ACInstLst$, the part of the unification problem to which we must call functions SOLVEAC and INSTANTIATESTEP is now $\delta\ cdr(P_1)$ and the part of the unification problem we have already explored is $P' \cup \delta P_2$. The substitution computed so far is $\delta\sigma$. We take care to update the set of variables that are/were in the problem to include the new variables introduced by SOLVEAC (in Algorithm 4 we change V to V'). In short, we make an input list $InputLst$ of all the branches we need to explore and each entry (P', δ) of $ACInstLst$ gives rise to an entry $(\delta cdr(P_1), P' \cup \delta P_2, \delta\sigma, V')$ in $InputLst$.

Algorithm 4 applyACStep

```
procedure applyACStep($P_1, P_2, \sigma, V$)
    if nil?($P_1$) then $cons((P_2, \sigma, V), \textsc{nil})$
    else let $(t, s) = car(P_1)$ in
        if $t \approx s$ then applyACStep($cdr(P_1), P_2, \sigma, V$)

        else
            ▷ assuming $t$ and $s$ are headed by the same function symbol $f$
            $PLst$ = solveAC($t, s, f, V$)

            ▷ Call instantiateStep in every $P$ in $PLst$ obtaining a list $ACInstLst$,
            ▷ where each entry in this list is a pair $(P', \delta)$.

            if nil?($ACInstLst$) then nil
            else
                ▷ make an input list $InputLst$ of all the branches we need to explore.
                ▷ For each $(P', \delta)$ in $ACInstLst$, the quadruple in $InputLst$ will be
                ▷ $(\delta cdr(P_1), P' \cup \delta P_2, \delta\sigma, V')$ to applyACStep
                ▷ recursively explore all the branches
                Flatten(map(applyACStep, $InputLst$))
```

Finally, applyACStep calls itself recursively taking as argument every input in *InputLst*. This is done by calling map(applyACStep, *InputLst*) and the output is flattened using function flatten.

Remark 20 (Eliminating $t \approx^? s$ When $t \approx s$)**.** *In function* applyACStep*, we eliminate equations $t \approx^? s$ from our unification problem if $t \approx s$ (line 4). This was done because if we called function* solveAC *in line 10 of Algorithm 4 passing as parameter two equal terms (modulo AC), the value returned would be $PLst$ =* nil*.* applyACStep *would interpret that as meaning that the unification pair had no solution (when actually every substitution σ is a solution to $\{t \approx^? s\}$) and also return* nil*. To prevent this corner case, we eliminate those trivial equations from our unification problem before calling* solveAC*. In our code, the function* equal? *tests equality (modulo AC) between terms t and s, returning $True$ if the terms are equal and $False$ otherwise.*

4.2 Proving Termination

4.2.1 The Lexicographic Measure

To prove termination in PVS, we must define a measure and show that this measure decreases at each recursive call the algorithm makes. We have chosen a lexicographic measure with four components:

$$lex = (|V_{NAC}(P)|,\ |V_{>1}(P)|,\ |AS(P)|,\ size(P)),$$

where $V_{NAC}(P)$, $V_{>1}(P)$, $AS(P)$, $size(P)$ are given in Definitions 27, 29, 31 and 32, respectively. Table 4.1 shows which components do not increase (represented by $\leq$) and which components strictly decrease (represented by $<$) for each recursive call that Algorithm 2 makes.

Definition 27 ($V_{NAC}(P)$). *We denote by $V_{NAC}(P)$ the set of variables that occur in the problem P, excluding those that only occur as arguments of AC-function symbols.*

Example 16. *Let f be an AC-function symbol and g be a standard function symbol. Let*

$$P = \{X \approx^? a, f(X, Y, W, g(Y)) \approx^? Z\}.$$

Then $V_{NAC}(P) = \{X, Y, Z\}$.

Before defining $V_{>1}(P)$, we need to define the subterms of a unification problem.

Definition 28 ($Subterms(P)$). *The subterms of a unification problem P are given as:*

$$Subterms(P) = \bigcup_{t \in P} Subterms(t),$$

where the notion of $Subterms(t)$ of a term t excludes all pairs and is defined recursively as follows:

- $Subterms(a) = \{a\}$.
- $Subterms(Y) = \{Y\}$.
- $Subterms(\langle\rangle) = \{\langle\rangle\}$.
- $Subterms(\langle t_1, t_2\rangle) = Subterms(t_1) \cup Subterms(t_2)$.
- $Subterms(f\ t_1) = \{f\ t_1\} \cup Subterms(t_1)$.
- $Subterms(f^{AC} t_1) = \bigcup_{t_i \in Args(f^{AC} t_1)} Subterms(t_i) \cup \{f^{AC} t_1\}$.

Here, $Args(f^{AC} t_1)$ denote the arguments of $f^{AC}\ t_1$.

Remark 21 (Subterms of AC and non-AC functions). *The definition of subterms for non-AC functions cannot be used for AC functions, as the following counterexample shows. Let f be an AC-function symbol and consider the term $t = f\langle f\langle a, b\rangle, f\langle c, d\rangle\rangle$. Then*

$$Subterms(t) = \{t, a, b, c, d\}.$$

However, if we had used the definition of subterms for non-AC functions, we would obtain

$$Subterms(t) = \{t, f\langle a, b\rangle, f\langle c, d\rangle, a, b, c, d\}.$$

Definition 29 ($V_{>1}(P)$ ↗)**.** *We denote by $V_{>1}(P)$ the set of variables that are arguments of (at least) two terms t and s such that t and s are headed by different function symbols and t and s are in $Subterms(P)$. The informal meaning is that if $X \in V_{>1}(P)$, then X is an argument to at least two different function symbols.*

Example 17. *Let f be an AC-function symbol and g be a standard function symbol. Let*

$$P = \{X \approx^? a, g(X) \approx^? h(Y), f(Y, W, h(Z)) \approx^? f(c, W)\}.$$

In this case $V_{>1}(P) = \{Y\}$.

We define proper subterms in order to define admissible subterms in Definition 31.

Definition 30 (Proper Subterms ↗)**.** *If t is not a pair, we define the proper subterms of t, denoted as $PSubterms(t)$ as:*

$$PSubterms(t) = \{s \mid s \in Subterms(t) \text{ and } s \neq t\}.$$

We define the proper subterm of a pair $\langle t_1, t_2\rangle$ as:

$$PSubterms(\langle t_1, t_2\rangle) = PSubterms(t_1) \cup PSubterms(t_2).$$

Definition 31 (Admissible Subterm AS ↗)**.** *We say that s is an admissible subterm of a term t if s is a proper subterm of t and s is not a variable. The set of admissible subterms of t is denoted as $AS(t)$. The set of admissible subterms of a unification problem P, denoted as $AS(P)$, is defined as*

$$AS(P) = \bigcup_{t \in P} AS(t).$$

Example 18. *If $P = \{a \approx^? f(Z_1, Z_2), b \approx^? Z_3, g(h(c), Z) \approx^? Z_4\}$ then $AS(P) = \{h(c), c\}$.*

Definition 32 (Size of a Unification Problem ↗)**.** *We define the size of a term t ↗ recursively as follows:*

- *$size(a) = 1$.*
- *$size(Y) = 1$.*

- *$size(\langle\rangle) = 1$.*
- *$size(\langle t_1, t_2\rangle) = 1 + size(t_1) + size(t_2)$.*
- *$size(f\ t_1) = 1 + size(t_1)$.*
- *$size(f^{AC}\ t_1) = 1 + size(t_1)$.*

Given a unification problem $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$, the size of P is defined as:

$$size(P) = \sum_{1 \le i \le n} size(t_i) + size(s_i).$$

Remark 22 ($s \in AS(t) \implies size(s) < size(t)$)**.** *If $s \in AS(t)$, we have that s is a proper subterm of t, and therefore the size of s is less than the size of t.*

Table 4.1: Decrease of the Components of the Lexicographic Measure.

Recursive Call	$\|\mathbf{V}_{NAC}(\mathbf{P})\|$	$\|\mathbf{V}_{>1}(\mathbf{P})\|$	$\|AS(\mathbf{P})\|$	$size(\mathbf{P})$
line 5, 12	$<$			
lines 8, 13, 16, 20	$\le$	$\le$	$\le$	$<$
case 1 - line 26	$\le$	$<$		
case 2 - line 26	$\le$	$\le$	$<$	
case 3 - line 26	$\le$	$\le$	$\le$	$<$

4.2.2 Proof Sketch for Termination

Non AC Cases

To prove the termination of syntactic unification, we can use a lexicographic measure lex_s consisting of two components: $lex_s = (|Vars(P)|, size(P))$, where $Vars(P)$ is the set of variables in the unification problem. We adapted this idea to our proof of termination by using $|V_{NAC}(P)|$ as our first component and $size(P)$ as the fourth. The proof of termination for all the cases of Algorithm 2 except AC (line 26) is similar to the proof of termination of syntactic unification, with two caveats.

First, we need to use $|V_{NAC}(P)|$ instead of $|Vars(P)|$ to avoid taking into account the variables that are arguments of the AC-function terms introduced by SOLVEAC (see Section 4.1.3). The variable terms introduced by SOLVEAC do not increase $|V_{NAC}(P)|$, since they will be instantiated by function INSTANTIATESTEP and therefore eliminated from the problem.

Second, in some of the recursive calls (lines 8, 13, 16, 20), we must ensure that the components introduced to prove termination in the AC-case ($|V_{>1}(P)|$ and $|AS(P)|$) do not increase. This is straightforward.

The AC-case

Our proof of termination for the AC-case uses the components $|V_{>1}(P)|$ and $|AS(P)|$, proposed in [40]. To explain the choice for the components of the lexicographic measure, let us start by considering the restricted case where $P = \{t \approx^? s\}$. The idea of the proof of termination is to define the set of admissible subterms of a unification problem $AS(P)$ in a way that when we call function solveAC to terms t and s, every problem P_1 generated will satisfy $|AS(P_1)| < |AS(P)|$.

Let $t_1, \ldots, t_m$ be the arguments of t and let $s_1, \ldots, s_n$ be the arguments of s. Then, as described in Section 4.1.3, the left-hand side of P_1 is $\{t_1, \ldots, t_m, s_1, \ldots, s_n\}$. Denote by $\{t'_1, \ldots, t'_m, s'_1, \ldots, s'_n\}$ the right-hand side of P_1, which means that $P_1 = \{t_1 \approx^? t'_1, \ldots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \ldots, s_n \approx^? s'_n\}$. This is what motivated our definition of admissible subterms: every term t'_i of the right-hand side of P_1 will have $AS(t'_i) = \emptyset$. Therefore, $AS(P_1) \subseteq AS(P)$ always holds.

If we are also in a situation where at least one of the terms on the left-hand side of P_1 is not a variable, we can prove that $|AS(P_1)| < |AS(P)|$. To see that, let u be the non-variable term in the left-hand side of P_1 of the greatest size (if there is a tie, pick any term with the greatest size). Then, u is an argument of either t or s and therefore $u \in AS(P)$. We also have $u \notin AS(P_1)$: otherwise there would be a term u' in P_1 such that $u \in AS(u')$, which would mean that the size of u' is greater than u (see Remark 22), contradicting our hypothesis that no term in P_1 has size greater than u. Combining the fact that $AS(P_1) \subseteq AS(P)$ and the fact that there is a term u with $u \in AS(P)$ and $u \notin AS(P_1)$ we obtain that $|AS(P_1)| < |AS(P)|$.

Example 19. *In the example of Section 2.2.2, after we eliminated the common arguments, we had*

$$P = \{f(X, X, Y, a) \approx^? f(b, b, Z)\}\}.$$

Notice that we had $AS(P) = \{a, b\}$. After applying solveAC*, one of the unification problems that is generated is:*

$$P_1 = \{X \approx^? Z_6, Y \approx^? f(Z_5, Z_5), a \approx^? Z_1, b \approx^? Z_5, Z \approx^? f(Z_1, Z_6, Z_6)\},$$

where $AS(P_1) = \emptyset$.

What happens if all the arguments of t and s are variables? In this case, we would have $AS(P_1) = AS(P) = \emptyset$, but this is not a problem since after function solveAC is called, the function instantiateStep would execute (receiving as input P_1), and it would instantiate all the arguments. The result, call it P_2 would be an empty list and we would have $AS(P_2) = AS(P) = \emptyset$ and $size(P_2) < size(P)$.

Therefore, all that is left in this simplified example with only one equation $t \approx^? s$ in the unification problem P is to make sure that when we call INSTANTIATESTEP in a unification problem P_1 and obtain as output a unification problem P_2 we maintain $|AS(P_2)| \leq |AS(P_1)|$. However, this does not necessarily happen, as Example 20 shows.

Example 20 (A case where INSTANTIATESTEP increases $|AS|$)**.** *Let f and g be AC-function symbols and*

$$P_1 = \{X \approx^? f(Z_1, Z_2), g(X, W) \approx^? g(a, c)\}.$$

Calling INSTANTIATESTEP with input P_1 we obtain

$$P_2 = \{g(f(Z_1, Z_2), W) \approx^? g(a, c)\}.$$

In this case we have $AS(P_1) = \{a, c\}$ while $AS(P_2) = \{f(Z_1, Z_2), a, c\}$ and therefore $|AS(P_2)| > |AS(P_1)|$.

This problem motivated the inclusion of the measure $|V_{>1}(P)|$ in our lexicographic measure, as we now explain. First, notice that if we changed Example 20 to make it so that X only appears as an argument of AC-functions headed by f, then instantiating X to an AC-function headed by f would not increase the cardinality of the set of admissible subterms. This is illustrated in Example 21.

Example 21 (A case where INSTANTIATESTEP does not increase $|AS|$)**.** *If we change slightly the problem from Example 20 to*

$$P'_1 = \{X \approx^? f(Z_1, Z_2), f(X, W) \approx^? g(a, c)\}$$

and apply INSTANTIATESTEP we would obtain:

$$P'_2 = \{f(Z_1, Z_2, W) \approx^? g(a, c)\},$$

and we would have $AS(P'_1) = AS(P'_2) = \{a, c\}$.

Let's return to our original example of $P = \{t \approx^? s\}$ and $P_1 = \{t_1 \approx^? t'_1, \dots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \dots, s_n \approx^? s'_n\}$, and denote by P_2 the unification problem obtained by calling INSTANTIATESTEP passing as input P_1. We will show that in the cases where $|AS(P_2)|$ may be greater than $|AS(P)|$ we necessarily have $|V_{>1}(P)| > |V_{>1}(P_2)|$.

Consider an arbitrary variable term X on the left-hand side of P_1. If X was instantiated by INSTANTIATESTEP, it would be instantiated to an AC-function headed by f (see Section 4.1.3) and therefore would only contribute to increasing $|AS(P_2)|$ in relation with

$|AS(P_1)|$ if it also occurred as an argument to a function term (let's call it t^*) headed by a different symbol than f (let's say g). Since X is in the left-hand side of P_1 this means that it was an argument of t or s in P (suppose t, without loss of generality) and remember that both t and s are headed by the same symbol f. Then X is an argument of t^* and t and therefore, by definition, $X \in V_{>1}(P)$. However X was instantiated by INSTANTIATESTEP and therefore it is not in $V_{>1}(P_2)$. The new variables introduced by SOLVEAC will not make any difference in favour of $|V_{>1}(P_2)|$: when they occur as arguments of function terms, the terms are always headed by the same symbol f. Therefore $|V_{>1}(P)| > |V_{>1}(P_2)|$. Accordingly, to fix our problem we include the measure $|V_{>1}(P)|$ before $|AS(P)|$, obtaining the lexicographic measure described in Section 4.2.1.

The situation described is similar when our unification problem P has multiple equations. Let's say $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$. The only difference is that it is insufficient to call function SOLVEAC and then function INSTANTIATESTEP in only the first equation $t_1 \approx^? s_1$: we need to call function APPLYACSTEP and simplify every equation $t_i \approx^? s_i$.

To see how things may go wrong, notice that in our previous explanation, when the unification problem P had just one equation, a call to SOLVEAC might reduce the admissible subterms by removing a given term (we called it u). However, now that P has more than one equation, if u is also present in other equations of the original problem P, calling SOLVEAC only in the first equation no longer removes u from the set of admissible subterms.

4.3 A Structured Proof of Termination for applyACStep

In this Section we detail how we proved termination for function APPLYACSTEP. The proof of termination (Theorem 16) is based on Lemmas 8, 9, 14 and 15. Before presenting the mentioned results and its proofs, we first introduce some prior notation.

4.3.1 Notation for the Proof of Termination

Algorithm 2 calls APPLYACSTEP with input $(P, \text{NIL}, \sigma, V)$. Recall that P is represented as a list and is not NIL. Let $t \approx^? s$ be the equation in the head of the list P and $n \geq 1$ the number of equations in P. Denote by P_i an arbitrary unification problem (recall that there may be many, since at each call to SOLVEAC the algorithm branches) obtained after we apply SOLVEAC and INSTANTIATESTEP to the first i equations, with $0 \leq i \leq n$. Hence, $P = P_0$. Denote by P_i^* a unification problem obtained after calling SOLVEAC with input P_i, but before we call INSTANTIATESTEP. Schematically, this means that:

$$P_i \xrightarrow{\text{SOLVEAC}} P_i^* \xrightarrow{\text{INSTANTIATESTEP}} P_{i+1}$$

Finally, we denote by P_i^C only the part of the unification problem P_i^* that replaces equation $t_i \approx^? s_i$ when we call SOLVEAC(t_i, s_i, V_i, f_i).

The substitution computed when we go from problem P_i to problem P_j is denoted by σ_{ij}. Given a substitution σ, we consider the function $\psi_\sigma : \mathbb{X} \to \mathbb{X}$ such that:

$$\psi_\sigma(X) = \begin{cases} \sigma X & \text{if } \sigma X \text{ is a variable} \\ X & \text{otherwise} \end{cases}$$

ψ_{ij} is syntactic sugar for $\psi_{\sigma_{ij}}$.

Example 22. *Let f be an AC-function symbol and g a syntactic function symbol. Suppose that $P = P_0 = \{f(X,Y) \approx^? f(a,b),\ f(W,g(U)) \approx^? f(g(c),d)\}$. After SOLVEAC but before INSTANTIATESTEP, one branch may be:*

$$P_0^* = \{X \approx^? Z_1,\ Y \approx^? Z_2,\ a \approx^? Z_1, b \approx^? Z_2,\ f(W,g(U)) \approx^? f(g(c),d)\},$$

where $P_0^C = \{X \approx^? Z_1,\ Y \approx^? Z_2,\ a \approx^? Z_1, b \approx^? Z_2\}$. After INSTANTIATESTEP, we have:

$$\begin{aligned} P_1 &= \{f(W,g(U)) \approx^? f(g(c),d)\} \\ \sigma_{01} &= \{Z_1 \mapsto a, Z_2 \mapsto b, X \mapsto a, Y \mapsto b\} = \psi_{01} \end{aligned}$$

APPLYACSTEP will call itself again, this time with P_1. After calling SOLVEAC in one branch we will have

$$P_1^* = \{W \approx^? Z_3,\ g(U) \approx^? Z_4,\ g(c) \approx^? Z_4,\ d \approx^? Z_3\} = P_1^C$$

and finally after INSTANTIATESTEP we have:

$$\begin{aligned} P_2 &= \{g(U) \approx^? g(c)\} \\ \sigma_{12} &= \{Z_3 \mapsto d, W \mapsto d\} = \psi_{12} \\ \sigma_{02} &= \sigma_{12}\sigma_{01} = \{Z_1 \mapsto a, Z_2 \mapsto b, X \mapsto a, Y \mapsto b, Z_3 \mapsto d, W \mapsto d\} = \psi_{02} \end{aligned}$$

At this point, APPLYACSTEP would return control to ACUNIF.

Notation 13. *If t and s are functions headed by the same function symbol, we represent this as $t \sim_{fsym} s$. If t and s are functions headed by different function symbols, we represent this as $t \not\sim_{fsym} s$.*

Notation 14. *We denote by $NVS(t)$ the set of non-variable subterms of P.*

Remark 23 (Signature of $\textsc{instantiateStep}$)**.** *Function $\textsc{instantiateStep}$ is recursive and receives as input a unification problem P_1 (the part of our unification problem which we have not yet inspected), a unification problem P_2 (the part of our unification problem we have already inspected) and σ, the substitution computed so far. Therefore, the first call to this function in order to instantiate the unification problem P is with $P_1 = P$, $P_2 = \textsc{nil}$ and $\sigma = \textsc{nil}$.*

The algorithm returns a triple $(P', \delta, bool)$, where the first component is the remaining unification problem; the second component is the substitution computed by this step; and the third component is a Boolean to indicate if we found an equation $t \approx^? s$ which is not unifiable (in this case the Boolean is $True$) or not (in this case the Boolean is $False$).

Notation 15. *Denote by $[\![\textsc{instantiateStep}(P_1, P_2, \sigma)]\!]_n$ the n-th component ($n = 1, 2, 3$) of the triple $(P', \delta, bool)$ returned by $\textsc{instantiateStep}(P_1, P_2, \sigma)$.*

4.3.2 Auxiliary Lemmas

Lemma 7. ↗ *$(P', \sigma', V') \in \textsc{applyACStep}(P^A, P^B, \sigma, V)$ if and only if $(P', \sigma'', V') \in \textsc{applyACStep}(P^A, P^B, \textsc{nil}, V)$, where $\sigma' = \sigma'' \circ \sigma$.*

Lemma 8 (V_{NAC} in $\textsc{applyACStep}$ ↗)**.** *Let $P_0 = P_0^A \cup P_0^B$ and let $(P_n, \sigma_{0n}, V_n) \in \textsc{applyACStep}(P_0^A, P_0^B, \textsc{nil}, V)$. Then*

$$V_{NAC}(P_n) \subseteq \psi_{0n}(V_{NAC}(P_0)).$$

$\langle 1 \rangle$1. We proceed by induction on the number of equations in P_0^A. SUFFICES: to prove that $V_{NAC}(P_1) \subseteq \psi_{01}(V_{NAC}(P_0))$.

PROOF: The induction hypothesis give us $V_{NAC}(P_n) \subseteq \psi_{1n}(V_{NAC}(P_1))$ and $\psi_{0n} = \psi_{1n} \circ \psi_{01}$.

COMMENT: The next recursive call will be $\textsc{applyACStep}(P_1^A, P_1^B, \sigma_{01}, V_1)$, where $P_1 = P_1^A \cup P_1^B$. The third component of the input is not $\textsc{nil}$ anymore, but we can fix that by using Lemma 7 to prove that if $(P_n, \sigma_{0n}, V') \in \textsc{applyACStep}(P_1^A, P_1^B, \sigma_{01}, V_n)$ then there is $(P_n, \sigma_{1n}, V_n) \in \textsc{applyACStep}(P_1^A, P_1^B, \textsc{nil}, V1)$ such that $\sigma_{0n} = \sigma_{1n} \circ \sigma_{01}$. A similar reasoning happens when we prove Lemmas 9, 14.

⟨1⟩2. From now until the rest of this proof, we denote σ_{01} as σ and ψ_{01} as ψ. Let Y be an arbitrary variable in $V_{NAC}(P_1)$. Then, exists some term t_1 in P_1 such that $Y \in V_{NAC}(t_1)$. A term t_1 in P_1 is not a variable and can be written as $t_1 = \sigma t_2$, where t_2 is a subterm in P_0^*.

Proof: t_1 is not a variable because P_1 is obtained from P_0^* by applying instantiateStep.

⟨1⟩3. $Y \in V_{NAC}(\sigma t_2)$ implies either:

1. exists X in $V_{NAC}(t_2)$ such that $\sigma X = Y$.
2. Y in $V_{NAC}(im(\sigma))$.

⟨1⟩4. Case: exists X in $V_{NAC}(t_2)$ such that $\sigma X = Y$. Then we have $Y \in \psi(V_{NAC}(P_0))$.

Proof: We have X in $V_{NAC}(P_0^*)$. Therefore, X in $V_{NAC}(P_0)$ and $\psi X = \sigma X = Y \in \psi(V_{NAC}(P_0))$.

⟨1⟩5. Case: Y in $V_{NAC}(im(\sigma))$. Then $Y \in \psi(V_{NAC}(P_0))$.

Proof: $Y \in V_{NAC}(im(\sigma))$ implies there exists X such that $\sigma X = Y$ and $X \in V_{NAC}(P_0^*)$. If $X \in V_{NAC}(P_0^*)$ then $X \in V_{NAC}(P_0)$. Finally, $\psi X = \sigma X = Y \in \psi(V_{NAC}(P_0))$.

Lemma 9 ($V_{>1}$ in applyACStep ↗). *Let* $P_0 = P_0^A \cup P_0^B$ *and let* $(P_n, \sigma_{0n}, V_n) \in$ *applyACStep*$(P_0^A, P_0^B, \text{nil}, V)$. *Then*

$$V_{>1}(P_n) \subseteq \psi_{0n}(V_{>1}(P_0)).$$

⟨1⟩1. We prove by induction on the number of equations in P_0^A. Suffices: to prove that $V_{>1}(P_1) \subseteq \psi_{01}(V_{>1}(P_0))$.

Proof: The induction hypothesis give us $V_{>1}(P_n) \subseteq \psi_{1n}(V_{>1}(P_1))$ and $\psi_{0n} = \psi_{1n} \circ \psi_{01}$.

⟨1⟩2. From now until the rest of this proof, we denote ψ_{01} by ψ and σ_{01} by σ. Let: Y be an arbitrary variable in $V_{>1}(P_1)$. Suffices: to prove that $Y \in \psi(V_{>1}(P_0))$.

⟨1⟩3. Since $Y \in V_{>1}(P_1)$, there exist t_1 and s_1 such that Y is an argument of t_1 (for short $Y \in Args(t_1)$) and Y is an argument of s_1, where $t_1 \not\sim_{fsym} s_1$ and t_1 and s_1 are subterms of P_1.

⟨1⟩4. There exists some subterm t_2 of P_0^* such that $t_2 \sim_{fsym} t_1$ and there exists $X \in Args(t_2)$ with $\sigma X = Y$. Similarly, there exists some subterm s_2 of P_0^* such that $s_2 \sim_{fsym} s_1$ and there exists $W \in Args(s_2)$ with $\sigma W = Y$. Since $t_1 \not\sim_{fsym} s_1$, we get $t_2 \not\sim_{fsym} s_2$.

Proof:

⟨2⟩1. We prove the existence of t_2 and X. The case for s_2 and W is analogous.

⟨2⟩2. Since $t_1 \in \mathit{Subterms}(P_1)$, there exists some t_1' in P_1 such that $t_1 \in \mathit{Subterms}(t_1')$. This t_1' can be written as σt_3, with t_3 in P_0^*. Hence, $t_1 \in \mathit{Subterms}(\sigma t_3)$.

⟨2⟩3. $t_1 \in \mathit{Subterms}(\sigma t_3)$ and t_1 is a function, which means that either:

1. $t_1 = \sigma t_4$ with $t_4 \in \mathit{Subterms}(t_3)$ and $t_4 \sim_{fsym} t_1$.
2. $t_1 \in \mathit{Subterms}(\mathit{im}(\sigma))$.

⟨2⟩4. Case: $t_1 \in \mathit{Subterms}(\mathit{im}(\sigma))$. If Y is an argument of a term t_1 in $\mathit{Subterms}(\mathit{im}(\sigma))$, then there exists a term t_4 (same symbol as t_1) in $\mathit{Subterms}(P^C)$ and a variable X_1 immediately under t_4 such that $\sigma X_1 = Y$. Pick t_2 as t_4 and X as X_1.

⟨2⟩5. Case: $t_1 = \sigma t_4$ with $t_4 \in \mathit{Subterms}(t_3)$ and $t_4 \sim_{fsym} t_1$. Then $Y \in \mathit{Args}(\sigma t_4)$ and either:

1. There is a variable $X_1 \in \mathit{Args}(t_4)$ with $\sigma X_1 = Y$. Pick X as X_1 and t_2 as t_4.
2. There is a variable $X_1 \in \mathit{Args}(t_4)$ and σX_1 is an AC-function with Y as one of its argument. In this case, Y is an argument of a term t_5, where $t_5 \in \mathit{Subterms}(\mathit{im}(\sigma))$. Hence, the reasoning in Step ⟨2⟩4 apply.

⟨1⟩5. Let: $t \approx^? s$ be the first unification pair in P_0. Let: f be the function symbol they are both headed.

⟨1⟩6. We divide our proof in four cases, according to whether X is equal to Y or not and according to whether W is equal to Y or not. The two following facts will be used:

1. $\sigma Y = Y$.
2. If $t' \in \mathit{Subterms}(P_0^*)$ and is headed by a symbol different than f, then $t' \in \mathit{Subterms}(P_0)$.

Proof:

⟨2⟩1. Recall that $Y \in \mathit{Args}(t_1)$. The term $t_1 \in \mathit{Subterms}(P_1)$ can be written as σt_3, where $t_3 \in \mathit{Subterms}(P_0^*)$. If we had $Y \in \mathit{dom}(\sigma)$, then Y would not happen in $t_1 = \sigma t_3$ (recall that σ is idempotent). Therefore, $Y \notin \mathit{dom}(\sigma)$, i.e. $\sigma Y = Y$.

⟨2⟩2. If a term t' is in $\mathit{Subterms}(P_0^*) - \mathit{Subterms}(P_0)$ it is necessarily in the right hand side of P_0^C. All function terms in the right hand side of P_0^C are headed by f.

⟨1⟩7. Case: $X = Y$ and $W = Y$, i.e. $Y \in \mathit{Args}(t_2)$ and $Y \in \mathit{Args}(s_2)$. Then $\psi(Y) \in \psi(V_{>1}(P_0))$.

Proof:

⟨2⟩1. Case: $t_2 \sim_{fsym} t$. Then, $s_2 \not\sim_{fsym} t$ and, by Step ⟨1⟩6, $s_2 \in \mathit{Subterms}(P_0)$. Since $Y \in \mathit{Args}(s_2)$, this implies $Y \in \mathit{Vars}(P_0)$. From that and the fact that

$Y \in Vars(t_2)$ we get that $t_2 \in Subterms(P_0)$. Hence, we have that $Y \in V_{>1}(P_0)$ and therefore $\psi(Y) \in \psi(V_{>1}(P_0))$.

⟨2⟩2. CASE: $t_2 \not\sim_{fsym} t$. We repeat the reasoning of Step ⟨2⟩1, exchanging the roles of t_2 and s_2.

⟨1⟩8. CASE: $X = Y$ and $W \neq Y$.

PROOF:

⟨2⟩1. Since $\sigma W = Y$, both W and Y are in P_0^C.

⟨2⟩2. Y must be in the left-hand side of P_0^C.

PROOF: Indeed if Y were in the right-hand side of P_0^C it would have been instantiated by σ (see the description of INSTANTIATESTEP in Section 4.1.3), which contradicts the fact that $\sigma Y = Y$ (see Step ⟨1⟩6).

⟨2⟩3. Since Y is in the left-hand side of P_0^C, it is an argument of either t or s (the terms in the first unification pair). LET: t_3 be the term Y is an argument.

⟨2⟩4. SUFFICES: to assume that $t_2 \sim_{fsym} t_3$.

PROOF: If $t_2 \not\sim_{fsym} t_3$ then $t_2 \in Subterms(P_0)$ (see Step ⟨1⟩6). t_3 is either t or s, hence $t_3 \in Subterms(P_0)$. By definition (PICK t_2 and t_3) we have $Y \in V_{>1}(P_0)$ and therefore $\psi Y \in \psi(V_{>1}(P_0))$. Finally, from Step ⟨1⟩6 and from the definition of ψ we have $\psi Y = \sigma Y = Y$, which allow us to conclude that $Y \in \psi(V_{>1}(P_0))$.

⟨2⟩5. If $t_2 \sim_{fsym} t_3$ then $s_2 \not\sim_{fsym} t_3$. Then, $s_2 \in Subterms(P_0)$ (Fact from ⟨1⟩6). Since $W \in Args(s_2)$ this means that $W \in Vars(P_0)$. Together with Step ⟨2⟩1, this let us conclude that W is in the left-hand side of P_0^C. Therefore, it is an argument of one of the terms of the first unification pair. LET: s_3 be this term.

⟨2⟩6. CASE: $s_2 \not\sim_{fsym} s_3$. Then by definition (PICK s_2 and s_3) we have $W \in V_{>1}(P_0)$. Therefore $\psi W = \sigma W = Y \in \psi(V_{>1}(P_0))$.

⟨2⟩7. CASE: $s_2 \sim_{fsym} s_3$. Together with $t_2 \sim_{fsym} t_3$ and $t_2 \not\sim_{fsym} s_2$ we conclude that $s_3 \not\sim_{fsym} t_3$. This however contradicts the fact that both s_3 and t_3 are terms of the first equation, functions headed by f.

⟨1⟩9. CASE: $X \neq Y$ and $W = Y$. Proof is analogous with Step ⟨1⟩8.

⟨1⟩10. CASE: $X \neq Y$ and $W \neq Y$.

⟨2⟩1. $\sigma X = Y$ let us conclude that X and Y are in P_0^C. $\sigma W = Y$ let us conclude that W is in P_0^C.

⟨2⟩2. Y must be in the left-hand side of P_0^C.

PROOF: By contradiction. If Y were in the right-hand side of P_0^C it would have been instantiated by σ, which contradicts the fact that $Y = \sigma Y = \psi(Y)$ (Fact from Step ⟨1⟩6).

$\langle 2 \rangle 3$. Since Y is in the left-hand side of P_0^C, it is an argument of either t or s. LET: t' be the term Y is an argument of P_0.

$\langle 2 \rangle 4$. CASE: $t_2 \not\sim_{fsym} t'$. Then, $t_2 \in \mathit{Subterms}(P_0)$ (Fact from $\langle 1 \rangle 6$). Since X is in $\mathit{Args}(t_2)$ we have $X \in \mathit{Vars}(P_0)$. This, together with the fact that X is in P_0^C let us conclude that X is in the left-hand side of P_0^C. It is therefore an argument of one of the terms of the first unification pair (t or s). LET: t_3 be this term. Then, by definition (PICK t_2 and t_3) we have $X \in V_{>1}(P_0)$ and hence $\psi X = \sigma X = Y \in \psi(V_{>1}(P_0))$.

$\langle 2 \rangle 5$. CASE: $s_2 \not\sim_{fsym} t'$. Then, $s_2 \in \mathit{Subterms}(P_0)$ (Fact from $\langle 1 \rangle 6$). Since W is in $\mathit{Args}(s_2)$ we have $W \in \mathit{Vars}(P_0)$. This, together with the fact that W is in P_0^C let us conclude that W is in the left-hand side of P_0^C. It is therefore an argument of one of the terms of the first unification pair (t or s). LET: s_3 be this term. Then, by definition (PICK s_2 and s_3) we have $W \in V_{>1}(P_0)$ and hence $\psi W = \sigma W = Y \in \psi(V_{>1}(P_0))$.

$\langle 2 \rangle 6$. By $\langle 2 \rangle 4$ and $\langle 2 \rangle 5$ all that is left is to consider the case where $t_2 \sim_{fsym} t'$ and $s_2 \sim_{fsym} t'$. This, however, would mean that $s_2 \sim_{fsym} t_2$, contradicting $\langle 1 \rangle 4$.

Lemma 10 (Admissible Subterms of σt). *Let σ be a substitution and let $t_s \in AS(\sigma t)$. We have one of 3 things*

1. *$t_s \in \sigma AS(t)$.*
2. *$t_s \in AS(im(\sigma))$.*
3. *There is $t_1 \in \mathit{Subterms}(t)$ and $X \in \mathit{Args}(t_1)$ such that $\sigma X = t_s$ and if t_s is an AC function symbol, then $t_1 \not\sim_{fsym} t_s$.*

Lemma 11. *Let $\sigma = [\![\text{INSTANTIATESTEP}(P, \text{NIL}, \text{NIL})]\!]_2$. If σX is not a variable, then there exists a non-variable term $t \in P$ such that $\sigma X = \sigma t$.*

Next, we introduce the definition of a nice unification problem with respect to f (Definition 33). It let us prove Lemma 13, which is used in Lemma 14.

Definition 33 (Nice Unification Problem with respect to f). *Let P be a unification problem, f be a function symbol and $\sigma = [\![\text{INSTANTIATESTEP}(P, \text{NIL}, \text{NIL})]\!]_2$. Suppose that for every function term $t \in \mathit{Subterms}(P)$, if there is a variable $X \in \mathit{Args}(t)$ such that σX is not a variable then t is an AC function headed by f. In this case we say that P is* nice with respect to *f.*

Lemma 12 (Terms after AC-step). *Suppose that*

$$(P_n, \sigma_{0n}, V') \in \textsc{applyACStep}(P_u, P_s, \textsc{nil}, V) \text{ and } V_{>1}(P_n) = \psi_{0n}(V_{>1}(P))$$

A term $t_n \in P_n$ can be written as $\sigma_{0n}t_0$ where $t_0 \in P_s$ or t_0 is a non-variable argument of some term $t \in P_u$.

Remark 24. *Recall that the first time we call* $\textsc{applyACStep}$ *we have $P_0 = P_u$ and $P_s = \textsc{nil}$.*

Lemma 13 (AS of the Substitution in the output of $\textsc{instantiateStep}$). *Let $\sigma = [\![\textsc{instantiateStep}(P, \textsc{nil}, \textsc{nil})]\!]_2$. Let P^A be the set of terms of P that are AC functions headed by f and let P^B be the remaining terms of P. Suppose P is nice with respect to f. Then, $AS(im(\sigma)) \subseteq \sigma AS(P^A) \cup \sigma NVS(P^B)$.*

Lemma 14 (AS in $\textsc{applyACStep}$). *Let $P_0 = P_0^A \cup P_0^B$ and let $(P_n, \sigma_{0n}, V_n) \in \textsc{applyACStep}(P_0^A, P_0^B, \textsc{nil}, V)$. If*

$$V_{>1}(P_n) = \psi_{0n}(V_{>1}(P_0))$$

then

$$AS(P_n) \subseteq \sigma_{0n}(AS(P_0)).$$

Proof:

$\langle 1\rangle 1.$ We do a proof by induction. By induction hypothesis, we get that when $V_{>1}(P_n) = \psi_{1n}(V_{>1}(P_1))$ we have $AS(P_n) \subseteq \sigma_{1n}(AS(P_1))$.

$\langle 1\rangle 2.$ $V_{>1}(P_n) = \psi_{1n}(V_{>1}(P_1))$.

Proof:

$\langle 2\rangle 1.$ By Lemma 9, we have $V_{>1}(P_n) \subseteq \psi_{1n}(V_{>1}(P_1))$. Hence, it suffices to prove that $\psi_{1n}(V_{>1}(P_1)) \subseteq V_{>1}(P_n)$.

$\langle 2\rangle 2.$ Since $V_{>1}(P_n) \subseteq \psi_{1n}(V_{>1}(P_1))$ we get

$$\psi_{1n}(V_{>1}(P_1)) \subseteq \psi_{1n} \circ \psi_{01}(V_{>1}(P_0)) = \psi_{0n}(V_{>1}(P_0)).$$

Since by hypothesis $\psi_{0n}(V_{>1}(P_0)) = V_{>1}(P_n)$ we get $\psi_{1n}(V_{>1}(P_1)) \subseteq V_{>1}(P_n)$.

$\langle 1\rangle 3.$ By induction hypothesis, we obtain $AS(P_n) \subseteq \sigma_{1n}(AS(P_1))$. Since we want to prove $AS(P_n) \subseteq \sigma_{0n}(AS(P_0))$, it suffices to prove $AS(P_1) \subseteq \sigma_{01}(AS(P_0))$.

$\langle 1\rangle 4.$ From now until the remaining of the proof, we denote σ_{01} by σ and ψ_{01} by ψ.

$\langle 1\rangle 5.$ Let: $t_{1s} \in AS(P_1)$. Suffices: to prove that t_{1s} in $\sigma(AS(P_0))$. There exists $t_1 \in P_1$ such that $t_{1s} \in AS(t_1)$. Then, there exists $t_2 \in P_0^*$ such that $t_1 = \sigma t_2$. Hence, $t_{1s} \in AS(\sigma t_2)$ and by Lemma 10 we have 3 possibilities:

1. $t_{1s} \in \sigma(AS(t_2))$.
2. $t_{1s} \in AS(\mathit{im}(\sigma))$
3. There is $t_3 \in \mathit{Subterms}(t_2)$ and $X \in \mathit{Args}(t_3)$ such that $\sigma X = t_{1s}$ and if t_{1s} is an AC function symbol, then $t_3 \not\approx_{fsym} t_{1s}$.

⟨1⟩6. LET: $t \approx^? s$ be the first equation of P_0 and f be the function symbol that both t and s are headed. P_0^C is a nice problem with respect to f.

PROOF:

⟨2⟩1. By contradiction. Suppose that P_0^C is not nice, then there exists a term $t' \in \mathit{Subterms}(P_0^C)$ that is not an AC-function term headed by f and a variable X such that $X \in \mathit{Args}(t')$, $\sigma X = t_3$ and t_3 is not a variable.

⟨2⟩2. $X \in V_{>1}(P_0)$ and therefore $X = \psi_{0n}(X) \in \psi_{0n}(V_{>1}(P_0))$.

PROOF: Since t' is not an AC-function term headed by f, we get that $t' \in \mathit{Subterms}(\mathit{lhs}(P_0^C))$ and therefore $X \in \mathit{Subterms}(\mathit{lhs}(P_0^C))$. This, along with the fact that $X \in \mathit{dom}(\sigma)$, let us conclude that $X \in \mathit{Args}(t) \cup \mathit{Args}(s)$. Suppose without loss of generality that $X \in \mathit{Args}(t)$. Then, $X \in V_{>1}(P_0)$ (PICK t and t') and therefore, by the definition of ψ_{0n} we have $X = \psi_{0n}(X) \in \psi_{0n}(V_{>1}(P_0))$.

⟨2⟩3. $X \notin V_{>1}(P_n)$.

PROOF: If we had $X \in V_{>1}(P_n)$ there would be some term $t_3 \in \mathit{Subterms}(P_n)$ such that $X \in Vars(t_3)$. However, every term in P_n can be written as $\sigma_{0n}t_4$, where $t_4 \in \mathit{Subterms}(P_0)$. Hence we would get $X \in Vars(\sigma_{0n}t_4)$. This cannot happen because $X \in \mathit{dom}(\sigma_{0n})$ and σ_{0n} is idempotent.

⟨2⟩4. From Steps ⟨2⟩2 and ⟨2⟩3 we would get $V_{>1}(P_n) \neq \psi_{0n}(V_{>1}(P_0))$, which contradicts our hypothesis.

⟨1⟩7. CASE: $t_{1s} \in \sigma AS(t_2)$. Then $t_{1s} \in \sigma AS(P_0)$.

PROOF: It suffices to prove that $t_2 \in P_0$. We have $t_2 \in P_0^*$. If t_2 was in $P_0^* - P_0$ we would have $t_2 \in \mathit{rhs}(P_0^C)$ and therefore $AS(t_2) = \emptyset$, which contradicts the fact that $t_{1s} \in \sigma AS(t_2)$.

⟨1⟩8. CASE: $t_{1s} \in AS(\mathit{im}(\sigma))$. Then $t_{1s} \in \sigma AS(P_0)$.

PROOF:

⟨2⟩1. LET: $P^A = \mathit{rhs}(P_0^C)$ and $P^B = \mathit{lhs}(P_0^C)$. We can apply Lemma 13 and obtain that $t_{1s} \in \sigma AS(P^A) \cup \sigma NVS(P^B)$.

⟨2⟩2. Since $AS(\mathit{rhs}(P_0^C)) = \emptyset$ we conclude that $t_{1s} \in \sigma NVS(\mathit{lhs}(P_0^C))$.

⟨2⟩3. $NVS(\mathit{lhs}(P_0^C)) \subseteq AS(P_0)$ and therefore $t_{1s} \in \sigma AS(P_0)$.

⟨1⟩9. CASE: There is $t_3 \in \mathit{Subterms}(t_2)$ and $X \in \mathit{Args}(t_3)$ such that $\sigma X = t_{1s}$ and if t_{1s} is an AC function symbol, then $t_3 \not\approx_{fsym} t_{1s}$. Then $t_{1s} \in \sigma AS(P_0)$.

PROOF:

⟨2⟩1. $t_{1s} \in \mathit{im}(\sigma)$, which implies that there exists a non-variable term $t_4 \in P_0^C$ such that $t_{1s} = \sigma t_4$.

⟨2⟩2. SUFFICES: to consider the case where $t_4 \in \mathit{rhs}(P_0^C)$.

PROOF: If $t_4 \in \mathit{lhs}(P_0^C)$ then it is in $\mathit{Args}(t) \cup \mathit{Args}(s)$ and therefore $t_4 \in \mathit{AS}(P_0)$. Hence $t_{1s} = \sigma t_4 \in \sigma \mathit{AS}(P_0)$.

⟨2⟩3. t_4 is an AC-function headed by f and therefore $t_{1s} = \sigma t_4$ is an AC-function headed by f.

PROOF: Since $t_4 \in \mathit{rhs}(P_0^C)$, it is either a variable or an AC-function headed by f. By Step ⟨2⟩1, t_4 is not a variable.

⟨2⟩4. $X \in V_{>1}(P_0)$ and therefore $X = \psi_{0n}(X) \in \psi_{0n}(V_{>1}(P_0))$.

PROOF:

⟨3⟩1. $X \in P_0^C$, since $X \in \mathit{dom}(\sigma)$.

⟨3⟩2. Notice that since t_{1s} is headed by an AC-function symbol and $t_{1s} \not\sim_{fsym} t_3$ we get that t_3 is a function that is not headed by f. Hence, $t_3 \in \mathit{Subterms}(P_0)$ and therefore $X \in \mathit{Subterms}(P_0)$. Since $X \in P_0^C$, we conclude that $X \in \mathit{lhs}(P_0^C)$.

⟨3⟩3. $X \in \mathit{Args}(t) \cup \mathit{Args}(s)$. Suppose without loss of generality that $X \in \mathit{Args}(t)$. Then by picking t and t_3 we get that $X \in V_{>1}(P_0)$.

⟨3⟩4. Since $\sigma X = t_{1s}$ which is not a variable, we have that $\sigma_{0n} = \sigma_{1n}\sigma X$ is not a variable. Therefore, by the definition of ψ, we have $X = \psi_{0n}(X) \in \psi_{0n}(V_{>1}(P_0))$.

⟨2⟩5. $X = \psi_{0n}(X) \notin V_{>1}(P_n)$.

PROOF: We have $\sigma X = t_{1s}$, which is not a variable. Then, $\sigma_{0n} X = \sigma_{1n}\sigma X$ is not a variable and therefore $X \in \mathit{dom}(\sigma_{0n})$. If we had $X \in V_{>1}(P_n)$ there would be some term $t_5 \in \mathit{Subterms}(P_n)$ such that $X \in \mathit{Vars}(t_5)$. There exists some term $t_6 \in \mathit{Subterms}(P_0)$ such that $t_5 = \sigma_{0n} t_6$. Hence, $X \in \mathit{Vars}(\sigma_{0n} t_6)$. This however, contradicts the fact that $X \in \mathit{dom}(\sigma_{0n})$ and σ_{0n} is idempotent.

⟨2⟩6. Steps ⟨2⟩4 and ⟨2⟩5 let us conclude that $V_{>1}(P_n) \neq \psi_{0n}(V_{>1}(P_0))$, contradicting our hypothesis.

Lemma 15 (Decrease of *AS* in APPLYACSTEP). *Let $P_0 = P_0^A \cup P_0^B$ and let $(P_n, \sigma_{0n}, V_n) \in$ APPLYACSTEP$(P_0^A, P_0^B, \text{NIL}, V)$. If*

$$V_{>1}(P_n) = \psi_{0n}(V_{>1}(P_0)) \text{ and } P_n \neq \text{NIL}.$$

Then

$$|AS(P_n)| < |AS(P_0)|.$$

PROOF:

⟨1⟩1. By Lemma 14, we have $AS(P_n) \subseteq \sigma_{0n}(AS(P_0))$.

⟨1⟩2. PICK a term $t' \in P_n$ with the biggest size. Notice that $t' \notin AS(P_n)$.

PROOF: Since $P_n \neq$ NIL, it is possible to pick a term $t' \in P_n$ with the biggest size. If $t' \in AS(P_n)$, there would be some term $t'' \in P_n$ such that $t' \in AS(t'')$. But then $size(t'') > size(t')$, which contradicts our hypothesis that $t' \in P_n$ has the biggest size.

⟨1⟩3. By Lemma 12, the term t' in P_n can be written as σt_1, where t_1 is a non-variable argument of some term $t \in P_0$. So, $t' = \sigma t_i \in \sigma_{0n} AS(P_0)$.

⟨1⟩4. By Steps ⟨1⟩2 and ⟨1⟩3, we conclude that $\sigma_{0n}(AS(P_0)) \nsubseteq AS(P_n)$. Along with $AS(P_n) \subseteq \sigma_{0n} AS(P_0)$ this let us conclude that $|AS(P_n)| < |\sigma_{0n} AS(P_0)|$. Since $|\sigma_{0n} AS(P_0))| \leq |AS(P_0)|$, the result follows.

4.3.3 Termination of applyACStep

Theorem 16 (Termination of APPLYACSTEP). *Suppose that Algorithm 2 is called with the nice input (P, σ, V) and enters the branch of* APPLYACSTEP *(lines 24-26). Let $(P_n, \sigma', V_n) \in$ APPLYACSTEP$(P, \text{NIL}, \sigma, V)$. Then*

$$(|V_{NAC}(P_n)|, |V_{>1}(P_n)|, |AS(P_n)|, size(P_n)) <_{lex} (|V_{NAC}(P)|, |V_{>1}(P)|, |AS(P)|, size(P))$$

PROOF:

⟨1⟩1. By Lemma 7 we have that $(P_n, \sigma_{0n}, V_n) \in$ APPLYACSTEP$(P, \text{NIL}, \text{NIL}, V)$, where $\sigma' = \sigma_{0n}\sigma$.

⟨1⟩2. By Lemma 8 we have $V_{NAC}(P_n) \subseteq \psi_{0n}(V_{NAC}(P))$. Hence

$$|V_{NAC}(P_n)| \leq |\psi_{0n}(V_{NAC}(P))| \leq |V_{NAC}(P)|.$$

⟨1⟩3. By Lemma 9 we have $V_{>1}(P_n) \subseteq \psi_{0n}(V_{>1}(P))$. Hence

$$|V_{>1}(P_n)| \leq |\psi_{0n}(V_{>1}(P))| \leq |V_{>1}(P)|.$$

⟨1⟩4. CASE: $V_{>1}(P_n) = \psi_{0n}(V_{>1}(P))$.

PROOF:

⟨2⟩1. CASE: $P_n =$ NIL. Then $|AS(P_n)| = 0 \leq AS(P)$ and

$$size(P_n) = 0 < size(P),$$

since P is not null.

⟨2⟩2. CASE: $P_n \neq$ NIL. Then by Lemma 15 we have $|AS(P_n)| < |AS(P)|$

⟨1⟩5. CASE: $V_{>1}(P_n) \neq \psi_{0n}(V_{>1}(P))$. Then, $V_{>1}(P_n) \subsetneq \psi_{0n}(V_{>1}(P))$ and hence
$$|V_{>1}(P_n)| < |\psi_{0n}(V_{>1}(P))| \leq |V_{>1}(P)|.$$

4.4 Proving Soundness and Completeness

4.4.1 Nice Inputs

As mentioned, to unify terms t and s we use Algorithm 2 with $P = \{t \approx^? s\}$, $\sigma = id$ and $V = Vars((t, s))$. However, since the parameters of ACUNIF may change between the recursive calls, we cannot directly prove soundness (Corollary 21) by induction. We must prove the more general Theorem 20, with generic parameters for the unification problem P, the substitution σ, and the set V of variables that are/were in use. To aid us in this proof, we notice that while the recursive calls of ACUNIF may change P, σ, and V, some nice relations between them are preserved. These relations between the three components of the input are captured by Definition 34.

Definition 34 (Nice input). *Given an input (P, σ, V), we say that this input is nice if:*

1. *σ is idempotent.*
2. *$Vars(P) \cap dom(\sigma) = \emptyset$.*
3. *$\sigma \subseteq V$.*
4. *$Vars(P) \subseteq V$.*

4.4.2 Soundness

As mentioned, once we prove Theorem 20, then soundness (Corollary 21) is obtained immediately. In order to prove Theorem 20, we used Theorem 18 and Theorem 19. Finally, to establish Theorem 18 (soundness of APPLYACSTEP), we used Theorem 17 (soundness of SOLVEAC).

Theorem 17 (Soundness of SOLVEAC). *Suppose that $(P_1, V_1) \in$ SOLVEAC(t, s, V, f), that δ unifies P and that t and s are AC-function applications headed by the same symbol f. Then δ unifies $\{t \approx^? s\}$.*

Theorem 18 (Soundness of APPLYACSTEP). *Suppose that $(P', \sigma', V') \in$ APPLYACSTEP(P_1, P_2, σ, V), that δ unifies P', that $\exists \sigma_1 : \delta = \sigma_1\sigma'$, that $dom(\sigma) \subseteq V$ and that $dom(\sigma) \cap (Vars(P_1) \cup Vars(P_2)) = \emptyset$. Then δ unifies P_1.*

Remark 25. *Hypotheses $dom(\sigma) \subseteq V$ and $dom(\sigma) \cap (Vars(P_1) \cup Vars(P_2)) = \emptyset$ of Theorem 18 are immediately satisfied when* ACUNIF *calls* APPLYACSTEP*, since in this case we have $P_1 = P$, $P_2 = \emptyset$ and (P, σ, V) is a nice input.*

Theorem 19 (Soundness of Variable Instantiation). *Suppose that (P, σ, V) is a nice input, $\sigma_1 = \{X \mapsto t\}$, $P = \{X \approx^? t\} \cup P_1$, $X \notin Vars(t)$ and $\delta \in ACUnif(\sigma_1 P_1, \sigma_1\sigma, V)$. If δ unifies $\sigma_1 P_1$, then δ unifies $\{X \approx^? t\}$ and δ unifies P_1.*

Theorem 20 (Soundness for Nice Inputs). *Let (P, σ, V) be a nice input, and $\delta \in ACUnif(P, \sigma, V)$. Then, δ unifies P.*

Theorem 20 was proved by induction on the lexicographic measure we used for termination. It branches in many cases, according to the type of the equation $t \approx^? s$ selected by CHOOSEEQ (see Algorithm 2). There are two interesting cases. The first case is in lines 24-26 when we only have AC-unification pairs (in that case, we used the soundness of APPLYACSTEP, i.e. Theorem 18). The second case happens when we instantiate a variable (lines 5 and 12) and is solved by using Theorem 19.

Corollary 21 (Soundness of ACUNIF). *If $\delta \in ACUnif(\{t \approx^? s\}, id, Vars((t, s)))$ then δ unifies $t \approx^? s$.*

4.4.3 Completeness

A Structured Proof of Completeness of solveAC

Theorem 22 is completeness for SOLVEAC. Recalling the structure of a unification problem obtained after APPLYACSTEP (Section 4.1.3), we see that the hypothesis $\delta \subseteq V$ of Theorem 22 means that the substitution δ will only impact the left-hand side of P_1 (since $\delta \subseteq V$ and the variables in the left-hand side of P_1 are all in V). Theorem 22 guarantees that the substitution γ will only impact the new variables introduced by SOLVEAC, since $dom(\gamma) \subseteq V_1 - V$. In terms of P_1, this means that γ will only impact the right-hand side of P_1.

We give a structured proof (*à la* Leslie Lamport [51, 52]) of the completeness of SOLVEAC (Theorem 22). In a structured proof, the main steps are numbered in the form $\langle 1 \rangle x.$, and they may decompose into substeps (of the form $\langle 2 \rangle .y$) and so on.

Theorem 22 (Completeness of SOLVEAC). *Suppose that t and s are AC-function applications headed by the same symbol f, t and s are not equal modulo AC, δ unifies $\{t \approx^? s\}$, $\delta \subseteq V$, and that $Vars((t, s)) \subseteq V$. Then, there is $(P_1, V_1) \in$ SOLVEAC(t, s, V, f) and a substitution γ such that $\gamma\delta$ unifies P_1, $dom(\gamma) \subseteq V_1 - V$, and $Vars(im(\gamma)) \subseteq V_1$.*

PROOF:

⟨1⟩1. It suffices to consider the case where t and s do not share common arguments.

PROOF: Let t^* and s^* be the terms obtained after eliminating the common arguments of t and s. Notice that if δ unifies $\{t^* \approx^? s^*\}$ then δ unifies $\{t \approx^? s\}$. Also, since the first step of SOLVEAC is to eliminate the common arguments, the output of SOLVEAC(t, s, V, f) is the same as SOLVEAC(t^*, s^*, V, f).

⟨1⟩2. Let $t \equiv f(t_1, \ldots, t_m)$ and $s \equiv f(s_1, \ldots, s_n)$, where each t_i occurs a_i times as an argument of t and each s_j occurs b_j times as an argument of s. The associated linear Diophantine equation is:

$$a_1X_1 + \ldots + a_mX_m = b_1Y_1 + \ldots + b_nY_n.$$

Let $|t|_A$ be the number of times the term A (or some term that is equal to A modulo AC) appears in the list of arguments of t, i.e. in $Args_f(t)$. Let $Args(\delta t) = \{A_1, \ldots, A_k\}$ be the set of all the different arguments (modulo AC) of δt.

⟨1⟩3. Since $\delta t \approx \delta s$, for each A_i, we have $|\delta t|_{A_i} = |\delta s|_{A_i}$. Therefore:

$$a_1|\delta t_1|_{A_i} + \ldots + a_m|\delta t_m|_{A_i} = b_1|\delta s_1|_{A_i} + \ldots + b_n|\delta s_n|_{A_i}$$

⟨1⟩4. Let D be the matrix obtained when SOLVEAC calls DIOSOLVER and let $\overrightarrow{Z'_1}, \ldots, \overrightarrow{Z'_{l'}}$ be the rows of D. Then $\{\overrightarrow{Z'_1}, \ldots, \overrightarrow{Z'_{l'}}\}$ is a spanning set of solutions.

COMMENT: since DIOSOLVER calculates all the solutions until an upper bound, this relies on the proof that our bound is correct.

⟨1⟩5. Let $\overrightarrow{n_{A_i}}$ be the vector $(|\delta t_1|_{A_i}, \ldots, |\delta t_m|_{A_i}, |\delta s_1|_{A_i}, \ldots, |\delta s_n|_{A_i})$. Since $\overrightarrow{n_{A_i}}$ solves the Diophantine equation, it can be written as a linear combination of the spanning set of solutions:

$$\overrightarrow{n_{A_i}} = c'_{i1}\overrightarrow{Z'_1} + \ldots + c'_{il'}\overrightarrow{Z'_{l'}}.$$

We can do that for every equation:

$$\begin{aligned}\overrightarrow{n_{A_1}} &= c'_{11}\overrightarrow{Z'_1} + \ldots + c'_{1l'}\overrightarrow{Z'_{l'}}\\ &\vdots\\ \overrightarrow{n_{A_k}} &= c'_{k1}\overrightarrow{Z'_1} + \ldots + c'_{kl'}\overrightarrow{Z'_{l'}}.\end{aligned}$$

Let $C = [c'_{ij}]$ be the matrix of coefficients.

⟨1⟩6. Let D_1 be the Diophantine submatrix of D that includes row $\overrightarrow{Z'_j}$ if and only if the j-th column of C is not the zero column. Let C_1 be the submatrix of C that includes column j if and only if it is not the zero column. Denoting the entries of C_1 by c_{ij}

and the rows of D_1 by $\overrightarrow{Z_1}, \dots, \overrightarrow{Z_l}$, we have:

$$\begin{aligned} \overrightarrow{n_{A_1}} &= c_{11}\overrightarrow{Z_1} + \ldots + c_{1l}\overrightarrow{Z_l} \\ &\vdots \\ \overrightarrow{n_{A_k}} &= c_{k1}\overrightarrow{Z_1} + \ldots + c_{kl}\overrightarrow{Z_l}. \end{aligned} \tag{4.2}$$

Let's denote by $z_{i1}, \dots, z_{i(m+n)}$ the entries of the vector $\overrightarrow{Z_i}$, for $i = 1, \dots, l$. Notice that $D_1 = (\overrightarrow{Z_1}, \dots, \overrightarrow{Z_l}) = [z_{ij}]$ is a $l \times (m+n)$ matrix.

$\langle 1 \rangle 7$. Let (P_1, V_1) be the output of DIOMATRIX2ACSOL when called with matrix D_1. The problem P_1 is of the form:

$$P_1 = \{t_1 \approx^? t'_1, \dots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \dots, s_n \approx^? s'_n\}.$$

$\langle 1 \rangle 8$. Every column of D_1 has at least one coefficient different than zero.

PROOF:

$\langle 2 \rangle 1$. Let's prove for the arbitrary column j. Recall that the j-th term of the vector $(t_1, \dots, t_m, s_1, \dots s_n)$ is associated with column j of D_1. Let's denote by t_j this term.

$\langle 2 \rangle 2$. There exists an A_i such that $|\delta t_j|_{A_i} > 0$.

$\langle 2 \rangle 3$. Analysing the j-th component of i-th equality in Equation 4.2, we have

$$|\delta t_j|_{A_i} = c_{i1}z_{1j} + \ldots + c_{il}z_{lj}.$$

Therefore, there exists some z_{xj} greater than zero, i.e. the j-th column of D_1 has at least one coefficient different than zero.

$\langle 1 \rangle 9$. Define γ such that

$$\gamma Z_j = \begin{cases} A_i, \text{ if } c_{ij} = 1 \text{ and } c_{ix} = 0 \text{ for } k \neq j. \\ f(\underbrace{A_1, \dots A_1}_{c_{1j}}, \dots, \underbrace{A_k, \dots, A_k}_{c_{kj}}), \text{ otherwise} \end{cases}$$

for the new variables Z_j's and for all the other variables X, $\gamma X = X$. Notice that $dom(\gamma) \subseteq V_1 - V$ and that $Vars(im(\gamma)) \subseteq V_1$.

PROOF:

$\langle 2 \rangle 1$. Due to Step $\langle 1 \rangle 8$, this γ is well-defined, as we will never have a case where $c_{1j}, \dots, c_{kj}$ are all zero.

$\langle 2 \rangle 2$. $dom(\gamma) \subseteq V_1 - V$ since the new variables Z_is introduced by SOLVEAC are in $V_1 - V$.

$\langle 2 \rangle 3$. The variables in $im(\gamma)$ are the variables in $A_1, \dots, A_k$. These are the variables occurring in δt (see Step $\langle 1 \rangle 2$). By hypothesis, $Vars(t) \subseteq V$ and $\delta \subseteq V$, which let us conclude that $im(\gamma) \subseteq V$. Since $V \subseteq V_1$ we get that $im(\gamma) \subseteq V_1$.

$\langle 1 \rangle 10$. $\gamma\delta$ unifies P_1.

PROOF:

$\langle 2\rangle 1.$ It suffices to prove that for an arbitrary i we have $\gamma\delta t_i \approx \gamma\delta t'_i$.

$\langle 2\rangle 2.$ This can be simplified to $\delta t_i \approx \gamma t'_i$.

PROOF:

$\langle 3\rangle 1.$ On one hand, since

$$Vars(\delta t_i) \subseteq (Vars(im(\delta)) \cup Vars(t_i)) \subseteq V$$

and $dom(\gamma) \cap V = \emptyset$ we have $\gamma\delta t_i = \delta t_i$.

$\langle 3\rangle 2.$ On the other hand, since $Vars(t'_i) \cap V = \emptyset$ and $dom(\delta) \subseteq V$, we have $\delta t'_i = t'_i$ and therefore $\gamma\delta t'_i = \gamma t'_i$.

$\langle 2\rangle 3.$ It suffices to prove that the list of arguments $Args_f(\delta t_i)$ is a permutation of $Args_f(\gamma t'_i)$. It suffices to prove that for an arbitrary term u, we have $|\delta t_i|_u = |\gamma t'_i|_u$.

COMMENT: from the hypothesis that $Args_f(\delta t_i)$ is a permutation of $Args_f(\gamma t'_i)$, it is only possible to conclude that $\delta t_i \approx^? \gamma t'_i$ because neither δt_i nor $\gamma t'_i$ is a pair. This is guaranteed here because we restrict ourselves to well-formed terms (Definitions 2 and 4) and substitutions.

$\langle 2\rangle 4.$ It suffices to consider the case where u is equal (modulo AC) to one of the A_js. Otherwise we would have $|\delta t_i|_u = |\gamma t'_i|_u = 0$.

$\langle 2\rangle 5.$ Let $u \approx A_j$. Since

$$\overrightarrow{n_{A_j}} = c_{j1}\overrightarrow{Z_1} + \ldots + c_{jl}\overrightarrow{Z_l},$$

we analyse the i-th entry of this vectorial equality and conclude that

$$|\delta t_i|_u = |\delta t_i|_{A_j} = c_{j1}z_{1i} + \ldots + c_{jl}z_{li}.$$

$\langle 2\rangle 6.$ Recall that Z_1 will appears z_{1i} times in $Args_f(t'_i)$, Z_2 will appear z_{2i} times in $Args_f(t'_i)$ and so on - see Section 4.1.3, specially the part about DIOMATRIX2ACSOL . Therefore,

$$|\gamma t'_i|_u = |\gamma t'_i|_{A_j} = z_{1i}|\gamma Z_1|_{A_j} + \ldots + z_{li}|\gamma Z_l|_{A_j} = c_{j1}z_{1i} + \ldots + c_{jl}z_{li}.$$

$\langle 2\rangle 7.$ Comparing the expressions in $\langle 2\rangle 6$ and $\langle 2\rangle 5$, we conclude that $|\delta t_i|_u = |\delta t'_i|_u$.

$\langle 1\rangle 11.$ $(P_1, V_1) \in$ SOLVEAC(t, s, V, f).

PROOF:

$\langle 2\rangle 1.$ All that is left to prove is that EXTRACTSUBMATRICES does not discard the matrix D_1. It is enough to show that D_1 satisfies the two constraints mentioned in Section 4.1.3.

$\langle 2\rangle 2.$ As proved in Step $\langle 1\rangle 8$, D_1 satisfies the first constraint: every column has one coefficient greater than 0.

$\langle 2\rangle 3.$ D_1 satisfies constraint 2: a column corresponding to a non-variable argument will only have one coefficient equal to 1, and the others are 0.

PROOF:

$\langle 3\rangle 1.$ We will prove for the arbitrary column j, associated with the j-th element of the vector $(t_1, \ldots, t_m, s_1, \ldots, s_n)$. Denote this term by t_j. By our hypothesis, t_j is a non-variable argument.

$\langle 3\rangle 2.$ Since t_j is an argument of either t or s, it is not an AC-function application headed by f. Additionally, since t_j is also a non-variable term, for any substitution σ, σt_j is not an AC-function headed by f.

$\langle 3\rangle 3.$ One of the equations in P_1 is $t_j \approx^? t'_j$. Suppose by contradiction that in j-th column of matrix D_1 there is not exactly one coefficient equal to 1, and the others are zero. Then t'_j cannot be a new variable Z_i, and it is instead an AC-function application headed by f whose arguments (at least two) are the new variables Z_is. This means that for any substitution σ we would have that $\sigma t'_j$ is an AC-function application headed by f.

$\langle 3\rangle 4.$ According to Steps $\langle 3\rangle 2$ and $\langle 3\rangle 3$, it would be impossible to unify $t_j \approx^? t'_j$ and therefore P_1. This, however, contradicts Step $\langle 1\rangle 10$.

Completeness of applyACStep

Theorem 23 is completeness for applyACStep.

Theorem 23 (Completeness of applyACStep). *Suppose that δ unifies $P_1 \cup P_2$, P_1 consists of only AC-unification pairs, $\delta \subseteq V$, $\sigma \leq \delta$ and $(P_1 \cup P_2, \sigma, V)$ is a nice input. Then, there exists $(P', \sigma', V') \in$ applyACStep(P_1, P_2, σ, V) and a substitution γ such that $\gamma\delta$ unifies P', $dom(\gamma) \subseteq V' - V$, $im(\gamma) \subseteq V'$ and $\sigma' \leq \gamma\delta$.*

Completeness of ACUnif

Lemma 26 states the completeness of Algorithm 2 with an arbitrary parameter V and an extra hypothesis $\delta \subseteq V$. Similarly to the soundness case, it is proved immediately once we prove Lemma 25.

Lemma 24 (Completeness for Variable Instantiation). *Suppose that (P, σ, V) is a nice input, $\sigma_1 = \{X \mapsto t\}$, $P = \{X \approx^? t\} \cup P_1$, $X \notin Vars(t)$ and $\sigma \leq \delta$. If δ unifies P, then $\sigma_1\sigma \leq \delta$ and δ unifies $\sigma_1 P_1$.*

Lemma 25 (Completeness for Nice Inputs). *Let (P, σ, V) be a nice input, δ unifies P, $\sigma \leq \delta$, and $\delta \subseteq V$. Then, there is a substitution $\gamma \in$ ACUnif(P, σ, V) such that $\gamma \leq_V \delta$.*

Lemma 25 was proved by induction on the lexicographic measure we used for termination. It branches in many cases, according to the type of the equation $t \approx^? s$ selected by

CHOOSEEQ (see Algorithm 2). There are two interesting cases. The first case is in lines 24-26 when we only have AC-unification pairs (in that case, we used the completeness of APPLYACSTEP, i.e. Lemma 23). The second case happens when we instantiate a variable (lines 5 and 12) and is solved by using Lemma 24.

To see the need for hypothesis $\sigma \leq \delta$ in Lemma 25, consider the case where $P = \emptyset$ and recall that in this case, ACUNIF returns a list with only one substitution: σ. Then, any δ unifies P, and if we did not have the hypothesis that $\sigma \leq \delta$ we would not be able to prove our thesis.

Lemma 26 (Completeness of ACUNIF with $\delta \subseteq V$). *Let V be a set of variables such that $\delta \subseteq V$ and $Vars((t, s)) \subseteq V$. If δ unifies $t \approx^? s$, then ACUNIF computes a substitution more general than δ, i.e., there is a substitution $\gamma \in ACUNIF(\{t \approx^? s\}, id, V)$ such that $\gamma \leq_V \delta$.*

In the proof of Lemma 26, the hypothesis $\delta \subseteq V$ is a technicality that was put in order to guarantee that the new variables introduced by the algorithm do not clash with the variables in $dom(\delta)$ or in the terms in $im(\delta)$ and could be replaced by a different mechanism that guarantees that the variables introduced by the AC-part of ACUNIF are indeed new.

As an example, let's go back to the substitutions (see Equation 2.3) computed in the example of Section 2.2.2 and notice that the set of variables in the original problem is $V = \{X, Y, Z\}$. If

$$\delta = \{X \mapsto f(Z_2, a, b), Z \mapsto f(a, Y, Z_2, a, Z_2, a), Z_4 \mapsto c\}$$

there is some overlap between the variables in $dom(\delta)$ and in the terms in $im(\delta)$ and the ones introduced by the algorithm, but the substitution

$$\sigma_4 = \{X \mapsto f(Z_6, b), Z \mapsto f(a, Y, Z_6, Z_6)\}$$

that we computed is still more general than δ (restricted to the variables in V). Indeed, if we take $\delta_1 = \{Z_6 \mapsto f(Z_2, a)\}$ then $\delta W = \delta_1 \sigma_4 W$ for all variables $W \in V$.

Finally, had we considered the set $V' = \{X, Y, Z, Z_1, Z_2, Z_3, Z_4\}$ instead of $V = \{X, Y, Z\}$ we would have $\delta \subseteq V'$ and the set of solutions would be:

$$\begin{aligned}
\sigma'_1 &= \{Y \mapsto f(b, b), Z \mapsto f(a, X, X)\}.\\
\sigma'_2 &= \{Y \mapsto f(Z_4, b, b), Z \mapsto f(a, Z_4, X, X)\}.\\
\sigma'_3 &= \{X \mapsto b, Z \mapsto f(a, Y)\}.\\
\sigma'_4 &= \{X \mapsto f(Z_4, b), Z \mapsto f(a, Y, Z_{10}, Z_{10})\}.
\end{aligned}$$

instead of

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

Notice that the difference between the two sets of solutions is just in the name given to the new variables.

First, we give a high-level description of how to remove hypothesis $\delta \subseteq V$ from Lemma 26. The key step to prove completeness of ACUNIF (an improvement of Lemma 26 where $V = \mathit{Vars}(t,s)$ and without the hypothesis $\delta \subseteq V$) is to prove that the substitutions computed when we call ACUNIF with input (P, σ, V) "differ only by a renaming" from the substitutions computed when we call ACUNIF with input (P, σ, V'), where $\delta \subseteq V'$. Formalising this intuitive reasoning is harder than it appears at first sight. This cannot be proven by induction directly because if V and V' differ and ACUNIF enters the AC-part, the new variables introduced for each input may "differ only by a renaming", i.e. the first component of the two inputs, will also "differ only by a renaming". Once ACUNIF instantiates variables, it may happen that the substitutions computed so far, i.e. the second component of the two inputs, will also "differ only by a renaming." The solution is to prove by induction the more general statement that if the inputs (P, σ, V) and (P', σ', V') "differ only by a renaming" then the substitutions computed when we call ACUNIF with (P, σ, V) "differ only by a renaming" from the substitutions computed when we call ACUNIF with (P', σ', V').

The idea of two inputs differing only by a renaming is captured in the definition of renamed inputs (Definition 35). The number of items in this definition may seem excessive, but they were all used in our proof, as will be explained in Remark 27.

Definition 35 (Renamed Inputs Fixing ψ). *We say that (P, σ, V) and (P', σ', V') are renamed inputs fixing ψ, if there is a renaming ρ such that:*

1. $P' = \rho P$.
2. $\sigma' =_\psi \rho\sigma$.
3. $max(V) \leq max(V')$.
4. $\psi \subseteq V$.
5. $dom(\rho) \subseteq V$
6. $\mathit{Vars}(im(\rho)) \subseteq V'$.
7. *If* $X \in \mathit{Vars}(im(\rho))$ *and* $X \notin dom(\rho)$ *then* $X \notin V$

Example 23. *Consider the inputs*

$$(\{X \approx^? g(Z_2)\}, \{Y \mapsto f(Z_1, Z_3)\}, \{X, Y, Z_1, Z_2, Z_3\}) \text{ and}$$
$$(\{X \approx^? g(Z_3)\}, \{Y \mapsto f(Z_2, Z_4)\}, \{X, Y, Z_2, Z_3, Z_4\})$$

Notice that they are renamed inputs fixing $\psi = \{X, Y\}$, where we pick the renaming $\rho = \{Z_1 \mapsto Z_2, Z_2 \mapsto Z_3, Z_3 \mapsto Z_4\}$.

Remark 26 (On the Name Renamed Inputs)**.** *Let (P, σ, V) and (P', σ', V') be renamed inputs fixing ψ. The name "Renamed Inputs" comes from the fact that P' is a renaming of P (Item 1) and that, restricted to the set ψ, σ' is a renaming of σ (Item 2). However, the only necessary relation between V and V' (the third component of the inputs) in the Definition of Renamed Inputs is that $max(V) \leq max(V')$. An alternative name for Definition 35 could have been "Variant Inputs".*

We can state Theorem 29 with this definition. The proof of Theorem 29 is done by induction, and the hardest cases are when we instantiate a variable (Lemma 27) and, inside the function APPLYACSTEP, when we call SOLVEAC (Lemma 28). We give a structured proof (*à la* Leslie Lamport) of the mentioned lemmas below.

Lemma 27 (Correctness of Renamed Inputs - Variable Instantiation)**.** *Let $\sigma_1 = \{X \mapsto t\}$ and $\sigma'_1 = \{\rho X \mapsto \rho t\}$. Suppose that $P_1 \subseteq P$, $P'_1 = \rho P_1$, $X \notin Vars(t)$, $X \in P$, $t \in P$ and (P, σ, V) and (P', σ', V') are renamed inputs fixing ψ with renaming ρ. Then, $(\sigma_1 P_1, \sigma_1 \sigma, V)$ and $(\sigma'_1 P'_1, \sigma'_1 \sigma', V')$ are renamed inputs fixing ψ with renaming ρ.*

PROOF:

$\langle 1 \rangle 1$. First we prove that $\sigma'_1 \rho =_V \rho \sigma_1$.

PROOF:

$\langle 2 \rangle 1$. SUFFICES: to prove that for every variable $Z \in V$ we have $\sigma'_1 \rho Z = \rho \sigma_1 Z$, i.e., that $[\rho X \mapsto \rho t]\rho Z = \rho[X \mapsto t]Z$.

$\langle 2 \rangle 2$. CASE: $Z = X$. Then both sides are equal to ρt.

$\langle 2 \rangle 3$. CASE: $Z \neq X$.

PROOF:

$\langle 3 \rangle 1$. The right-hand side is $\rho[X \mapsto t]\rho Z = \rho Z$, which means that it suffices to prove that $[\rho X \mapsto \rho t]\rho Z$ (the left-hand side) is also equal to ρZ. To do that, it suffices to prove that $\rho Z \neq \rho X$.

$\langle 3 \rangle 2$. Suppose by contradiction that $\rho Z = \rho X$.

$\langle 3 \rangle 3$. CASE: $X \in dom(\rho)$ and $Z \in dom(\rho)$. Since ρ is a renaming, $\rho Z = \rho X$ and both Z and X are in $dom(\rho)$ we must have $X = Z$. This, however, contradicts the fact that we are in the case where $Z \neq X$.

⟨3⟩4. CASE: $X \notin dom(\rho)$ and $Z \in dom(\rho)$. We have $\rho Z = \rho X = X$, which means that $X \in Vars(im(\rho))$. Since we also have that $X \notin dom(\rho)$, by Item 7 of the Definition of Renamed Inputs, we get that $X \notin V$. However, $X \in P$ and $Vars(P) \subseteq V$ (see item 4 of the Definition of Nice Input). This means that $X \in V$. Contradiction.

⟨3⟩5. CASE: $X \in dom(\rho)$ and $Z \notin dom(\rho)$. Similar to the previous case, exchanging the roles of X and Z and noticing that $Z \in V$ is one of our hypotheses (Step ⟨2⟩1).

⟨3⟩6. CASE: $X \notin dom(\rho)$ and $Z \notin dom(\rho)$. Then $\rho Z = \rho X \mapsto Z = X$, which contradicts the fact that we are in the case where $Z \neq X$.

⟨1⟩2. Item 1 in the Definition of Renamed Inputs is satisfied: $\sigma'_1 P'_1 = \rho\sigma_1 P_1$.

PROOF:

⟨2⟩1. Let t_i be an arbitrary term in P_1 and let t'_i be the correspondent in P'_1. It suffices to prove that $\sigma'_1 t'_i = \rho\sigma_1 t_i$. Since $P'_1 = \rho P_1$ we have $t'_i = \rho t_i$, which means that we must prove $\sigma'_1 \rho t_i = \rho\sigma t_i$.

⟨2⟩2. It suffices to prove that for every variable $Z \in Vars(t_i)$ we have $\sigma'_1\rho Z = \rho\sigma_1 Z$. This follows from $\sigma'_1\rho =_V \rho\sigma_1$ (Step ⟨1⟩1), since $Z \in Vars(P_1) \subseteq Vars(P)$ and $Vars(P) \subseteq V$ (this last one is because of the definition of nice input).

⟨1⟩3. Item 2 in the Definition of Renamed Inputs is satisfied: $\sigma'_1\sigma' =_\psi \rho\sigma_1\sigma$.

PROOF:

⟨2⟩1. Since (P, σ, V) and (P', σ', V') are renamed inputs, by Item 2 of the definition, we have $\sigma' =_\psi \rho\sigma$. Therefore $\sigma'_1\sigma' =_\psi \sigma'_1\rho\sigma$.

⟨2⟩2. Since $\sigma'_1\rho =_V \rho\sigma_1$ (by Step ⟨1⟩1) and $Vars(im(\sigma)) \subseteq V$ (By Item 3 of the Definition of Nice Input) we have $\sigma'_1\rho\sigma =_V \rho\sigma_1\sigma$. Since $\psi \subseteq V$ (Item 4 of the Definition of Renamed Inputs), we have $\sigma'_1\rho\sigma =_\psi \rho\sigma_1\sigma$.

⟨1⟩4. The remaining items to prove that $(\sigma_1 P_1, \sigma_1\sigma, V)$ and $(\sigma'_1 P'_1, \sigma'_1\sigma', V')$ are renamed inputs depend only on ψ, ρ, V and V' and therefore are immediately proved from the fact that (P, σ, V) and (P', σ', V') are renamed inputs.

Lemma 28 (Correctness of Renamed Inputs - SOLVEAC). *Let $(P_1 \cup P_2, \sigma, V)$ be a renamed input of $(P'_1 \cup P'_2, \sigma', V')$ fixing ψ with renaming ρ, let $car(P_1) = t \approx^? s$ be the unification problem that we will apply* SOLVE*AC, where t and s are rooted by the same function symbol f. Let V_1 be the new set of variables to avoid after we call* SOLVE*$AC(t, s, V, f)$ and V'_1 the new set of variables to avoid after we call* SOLVE*$AC(\rho t, \rho s, V', f)$. Let P'_c be a unification problem in* SOLVE*$AC(\rho t, \rho s, V', f)$. Then, there exists P_c in* SOLVE*$AC(t, s, V, f)$*

such that $(cdr(P_1) \cup P_c \cup P_2, \sigma, V_1)$ *and* $(cdr(P'_1) \cup P'_c \cup P'_2, \sigma', V'_1)$ *fixing* ψ.

PROOF:

⟨1⟩1. LET: $Z'_1, \dots, Z'_l$ be the l new variables introduced by SOLVEAC$(\rho t, \rho s, V', f)$. When we call SOLVEAC(t, s, V, f), it will also introduce l new variables, which we denote by $Z_1, \dots, Z_l$. Notice that

$$V_1 = V \cup \{Z_1, \dots, Z_l\}$$
$$V'_1 = V' \cup \{Z'_1, \dots, Z'_l\}.$$

Finally, notice that:

$$|Z_i| = max(V) + i$$
$$|Z'_i| = max(V') + i$$

for every $1 \leq i \leq l$.

⟨1⟩2. DEFINE: ρ_1 as

$$\rho_1 X = \begin{cases} Z'_i & \text{if } X = Z_i \text{ for } i = 1, \dots, l \\ \rho X & \text{otherwise.} \end{cases}$$

Notice that $\rho_1 =_V \rho$.

COMMENT: Recall that in our PVS code, substitutions are defined as a list, where each entry is of the form $\{X \mapsto t\}$. To define ρ_1 in our formalisation, first we defined $\rho^* = \{Z_1 \mapsto Z'_1, \dots, Z_l \mapsto Z'_l\}$. Then, the renaming ρ_1 is defined in our formalisation as $\rho_1 =$ APPEND(ρ, ρ^*). This was of constructing ρ_1 only works due to the fact that $dom(\rho) \subseteq V$ (Item 5 of the Definition of Renamed Inputs) and that $\{Z'_1, \dots, Z'_l\} \cap V = \emptyset$.

⟨1⟩3. If P'_c is a unification problem in SOLVEAC $(\rho t, \rho s, V, f)$, there exists a unification problem P_c in SOLVEAC (t, s, V, f) such that $P'_c = \rho_1 P_c$.

PROOF:

⟨2⟩1. The Diophantine equation associated with both calls of SOLVEAC will be the same, and so will be the matrix returned by DIOSOLVER. As a consequence there exists a unification problem P_C in SOLVEAC(t, s, V, f) such that the only difference between the terms in the right-hand side of P_c and P'_c will be in the name of the variables: they will be $Z_1, \dots, Z_l$ in P_c and correspondingly $Z'_1, \dots, Z'_l$ in P'_c. Therefore, given a term u' in the right-hand side of P'_c, its correspondent term u in P_c is such that $u' = \rho_1 u$.

⟨2⟩2. LET: $t_1, \dots, t_m$ be the arguments of t and $s_1, \dots, s_n$ be the arguments of s. The terms in the left-hand side of every unification problem returned by SOLVEAC(t, s, V, f) will be respectively $t_1, \dots, t_m, s_1, \dots, s_n$. Similarly, the terms in the left-hand side of every unification problem returned by SOLVEAC$(\rho t, \rho s, V, f)$ will be respectively $\rho t_1, \dots, \rho t_m, \rho s_1, \dots, \rho s_n$. Therefore, given a term u' in the left-hand side of P'_c, its correspondent term u in P_c is such that $u' = \rho u$. Additionally, since

$\rho_1 =_V \rho$ we have $u' = \rho_1 u$.

⟨1⟩4. Item 1 of the Definition of Renamed Inputs holds:

$$cdr(P'_1) \cup P'_c \cup P'_2 = \rho_1(cdr(P_1) \cup P_c \cup P_2).$$

PROOF: We have that $(P'_1 \cup P'_2, \sigma', V')$ is a renamed input of $(P_1 \cup P_2, \sigma, V)$ fixing ψ with renaming ρ, which gives us $cdr(P'_1) = \rho\ cdr(P_1)$ and $P'_2 = \rho P_2$ (Item 1 of the Definition of Renamed Inputs). Since $\rho_1 =_V \rho$ we get $cdr(P'_1) = \rho_1\ cdr(P_1)$ and $P'_2 = \rho_1 P_2$. Finally, by Step ⟨1⟩3, $P'_c = \rho_1 P_c$.

⟨1⟩5. Item 2 of the Definition of Renamed Inputs holds: $\sigma' =_\psi \rho_1 \sigma$.

PROOF: Since $\psi \subseteq V$ and $\rho_1 =_V \rho$, it suffices to prove that $\sigma' =_\psi \rho\sigma$. This holds since $(P'_1 \cup P'_2, \sigma', V')$ is a renamed input of $(P_1 \cup P_2, \sigma, V)$ fixing ψ with renaming ρ (Item 2 of the Definition of Renamed Inputs).

⟨1⟩6. Item 3 of the Definition of Renamed Inputs holds: $max(V_1) \leq max(V'_1)$.

PROOF: We have

$$max(V_1) = |Z_l| = l + max(V)$$
$$max(V'_1) = |Z'_l| = l + max(V').$$

Since $max(V) \leq max(V')$ we obtain $max(V_1) \leq max(V'_1)$.

⟨1⟩7. Item 4 of the Definition of Renamed Inputs holds: $\psi \subseteq V_1$.

PROOF: This follows from $\psi \subseteq V$ (from the Definition of Renamed Inputs in our hypothesis) and from $V \subseteq V_1$.

⟨1⟩8. Item 5 of the Definition of Renamed Inputs holds: $dom(\rho_1) \subseteq V_1$.

PROOF: We have

$$dom(\rho_1) \subseteq dom(\rho) \cup \{Z_1, \ldots, Z_l\}.$$

Since $dom(\rho) \subseteq V$ (Item 5 of the Definition of Renamed Inputs in our hypothesis) and $V_1 = V \cup \{Z_1, \ldots, Z_l\}$ the result follows.

⟨1⟩9. Item 6 of the Definition of Renamed Inputs holds: $Vars(im(\rho_1)) \subseteq V'_1$.

PROOF: We have

$$Vars(im(\rho_1)) \subseteq Vars(im(\rho)) \cup \{Z'_1, \ldots, Z'_l\}.$$

Since $Vars(im(\rho)) \subseteq V'$ (Item 6 of the Definition of Renamed Inputs in our hypothesis) and $V'_1 = V' \cup \{Z'_1, \ldots, Z'_l\}$ the result follows.

⟨1⟩10. Item 7 of the Definition of Renamed Inputs holds: If $X \in im(\rho_1)$ and $X \notin dom(\rho_1)$ then $X \notin V_1$.

PROOF:

⟨2⟩1. CASE: $max(V) = max(V')$.

PROOF:

⟨3⟩1. $Z_i = Z'_i$ for every $1 \leq i \leq l$ and therefore $\rho_1 = \rho$.

⟨3⟩2. We have $X \in im(\rho)$ and $X \notin dom(\rho)$. Hence, by Item 7 of the Definition of Renamed Inputs, $X \notin V$.

⟨3⟩3. Since $V_1 = V \cup \{Z_1, \ldots, Z_l\}$, all there is to prove is that $X \notin \{Z_1, \ldots, Z_l\}$. Due to Step ⟨3⟩1, it suffices to prove that $X \notin \{Z'_1, \ldots, Z'_l\}$.

⟨3⟩4. Suppose by contradiction that $X \in \{Z'_1, \ldots, Z'_l\}$. Then, $X \notin V'$. However, this contradicts the fact that $X \in im(\rho)$, by Item 6 of the Definition of Renamed Inputs.

⟨2⟩2. Case: $max(V) < max(V')$.

Proof:

⟨3⟩1. We have

$$\begin{aligned} dom(\rho_1) &= dom(\rho) \cup \{Z_1, \ldots, Z_l\} \\ im(\rho_1) &= im(\rho) \cup \{Z'_1, \ldots, Z'_l\} \\ V_1 &= V \cup \{Z_1, \ldots, Z_l\}. \end{aligned}$$

⟨3⟩2. Case: $X \in im(\rho)$. We also have $X \notin dom(\rho)$ and hence, by Item 7 of the Definition of Renamed Inputs, $X \notin V$. Since $X \in V_1$, this implies $X \in \{Z_1, \ldots, Z_l\}$. This, however, contradicts the fact that $X \notin dom(\rho_1)$.

⟨3⟩3. Case: $X \notin im(\rho)$. Then, $X \in \{Z'_1, \ldots, Z'_l\}$. We have

$$|X| > max(V') > max(V)$$

and hence $X \notin V$. Additionally, $X \notin \{Z_1, \ldots, Z_l\}$ because otherwise we would have $X \in dom(\rho_1)$. Hence, we get that $X \notin V_1$.

With Lemmas 27 and 28 it is possible to prove Theorem 29, shown below.

Theorem 29 (Correctness of Renamed Inputs). *Let (P, σ, V) and (P', σ', V') be renamed inputs fixing ψ and suppose $\gamma' \in ACU\textsc{nif}(P', \sigma', V')$. Then, there exist a renaming ρ and a substitution $\gamma \in ACU\textsc{nif}(P, \sigma, V)$ such that $\gamma' =_\psi \rho\gamma$.*

Proof sketch:

⟨1⟩1. The proof is by induction using the lexicographic measure we used in the proof of termination, for P'.

⟨1⟩2. Case: $nil?(P')$ (line 2 of Algorithm 2).
Then, $ACU\textsc{nif}(P', \sigma', V')$ returns and we have $\gamma' = \sigma'$. Due to Item 1 of the Definition of Renamed Inputs, $P = \rho P' = \emptyset$ and hence $ACU\textsc{nif}(P, \sigma, V)$ returns σ, i.e, $\gamma = \sigma$. Then, $\gamma' = \sigma' =_\psi \rho\sigma = \rho\gamma$, due to Item 2 of the Definition of Renamed Inputs.

⟨1⟩3. If P' is not null, let $((t', s'), P'_1) = \textsc{chooseEq}(P')$. The proof is divided into cases according to the structure of t and s, as Algorithm 2.

$\langle 1\rangle 4$. CASE: (s' matches X) and (X not in t') (lines 4-5 of Algorithm 2).

$\langle 2\rangle 1$. Then,

$$\text{ACUNIF}(P', \sigma', V') = \text{ACUNIF}(\sigma_1' P_1', \sigma_1'\sigma', V')$$
$$\text{ACUNIF}(P, \sigma, V) = \text{ACUNIF}(\sigma_1 P_1, \sigma_1\sigma, V).$$

$\langle 2\rangle 2$. By Lemma 27, $(\sigma_1 P_1, \sigma_1\sigma, V)$ and $(\sigma_1' P_1', \sigma_1'\sigma', V')$ are renamed inputs fixing χ and therefore we can apply the induction hypothesis and conclude.

$\langle 1\rangle 5$. CASE: $t' \approx^? s'$ is an AC-unification pair (lines 24-26 of Algorithm 2).

$\langle 2\rangle 1$. Since $\gamma' \in \text{ACUNIF}(P', \sigma', V')$ there will be

$$(P_*', \sigma_*', V_*') \in \text{APPLYACSTEP}(P', \sigma', V')$$

such that $\gamma' \in \text{ACUNIF}(P_*', \sigma_*', V_*')$.

$\langle 2\rangle 2$. We can prove that there will be

$$(P_*, \sigma_*, V_*) \in \text{APPLYACSTEP}(P, \sigma, V)$$

such that (P_*, σ_*, V_*) and (P_*', σ_*', V_*') are renamed inputs. Since function APPLY-ACSTEP calls functions SOLVEAC and INSTANTIATESTEP to every AC-unification pair in the unification problem, this result is established as soon as we prove the lemmas of the correctness of functions SOLVEAC and INSTANTIATESTEP for renamed inputs. For function SOLVEAC, this is Lemma 28. Finally, since function INSTANTIATESTEP only performs variable instantiation, the corresponding Lemma is proved in the same manner as Lemma 27.

$\langle 2\rangle 3$. Hence, we apply the induction hypothesis and conclude.

$\langle 1\rangle 6$. The case when (t' matches X) and (X not in s') (lines 11-12 of Algorithm 2) is similar to Step $\langle 1\rangle 4$. The remaining cases are straightforward.

Remark 27 (Necessity of Every Item in Definition of Renamed Inputs)**.** *Items 1 and 2 of the Definition of Renamed Inputs (Definition 35) are used in the main proof of Theorem 29. However, to prove that we stay with renamed inputs during the recursive calls ACUNIF makes, we needed to add Items 3 through 7, as explained next. Notice that Items 4 and 7 were used in Lemma 27 to prove that* $(\sigma_1 P_1, \sigma_1\sigma, V)$ *and* $(\sigma_1' P_1', \sigma_1'\sigma', V')$ *satisfy Items 1 and 2 of the Definition 35 and hence should be included in the definition. Finally, in Lemma 28, we used Items 3, 5, 6 to prove that* $(cdr(P_1) \cup P_c \cup P_2, \sigma, V_1)$ *and* $(cdr(P_1') \cup P_c' \cup P_2', \sigma', V_1')$ *satisfy Item 7 of Definition 35.*

Finally, Theorem 29 is used along with Lemma 26 to prove the completeness of ACUNIF (Theorem 30).

Theorem 30 (Completeness of ACUnif). *If δ unifies $t \approx^? s$, then ACUnif computes a substitution more general than δ, i.e., there is a substitution $\gamma \in$ ACUnif$(\{t \approx^? s\}, id, Vars(t,s))$ such that $\gamma \leq_{Vars(t,s)} \delta$.*

Proof:

⟨1⟩1. Let: $V = Vars(t,s)$ and $V' = V \cup dom(\delta) \cup Vars(im(\delta))$. By Theorem 26 we have that there exists a substitution $\gamma' \in$ ACUnif$(\{t \approx^? s\}, id, V')$ such that $\gamma' \leq_{V'} \delta$. Hence, there exists δ_1 such that $\delta =_{V'} \delta_1\gamma'$

⟨1⟩2. Notice that the inputs $(\{t \approx^? s\}, id, V)$ and $(\{t \approx^? s\}, id, V')$ are renamed inputs fixing V with renaming id. We can apply the Theorem of Renamed Inputs and obtain that there exists a renaming ρ and a substitution $\gamma \in$ ACUnif$(\{t \approx^? s\}, id, V)$ such that $\gamma' =_V \rho\gamma$.

⟨1⟩3. $\delta =_{V'} \delta_1\gamma' =_V= \delta_1\rho\gamma$. Therefore, $\gamma \leq_V \delta$.

Remark 28 (The parameter ψ in the Definition of Renamed Inputs). *The parameter ψ in the definition of Renamed Inputs is used in the proof of Theorem 30 as*

$$\psi = Vars(t,s) = V = Vars(P).$$

One may wonder if we could have eliminated this parameter from the Definition of Renamed Inputs and used instead V or $Vars(P)$ in its place. The answer is "no" because ψ is unaffected by the recursive calls ACUnif makes and, hence, can perfectly represent the variables in the original unification problem. P and V are the first and third parameters of ACUnif and, therefore, can change as the algorithm calls itself recursively. Hence, neither one can be used to replace ψ in the Definition of Renamed Inputs.

4.5 Statistics of the PVS Formalisation

Below we describe the main theories that are part of the first-order AC-unification formalisation. Since the nominal library contain other 3 formalisations, the theories that are only part of the first-order AC-unification formalisation have a `first_order_AC_` prefix to their names (see Table 2.4), which we omit in this Section in order not to clutter the presentation.

- `top_first_order_AC_unification` - High Level description of the first-order AC-unification formalisation.
- `unification_alg` - Function ACUnif (Algorithm 2) and the theorems of soundness and completeness.

- `renamed_inputs` - The Definition of Renamed Inputs and auxiliary lemmas to establish correctness.
- `termination_alg` - Definitions and theorems necessary for proving termination.
- `apply_ac_step` - Function APPLYACSTEP, the Definition of Nice Inputs and its properties.
- `aux_unification` - Auxiliary functions such as SOLVEAC, CHOOSEEQ and INSTANTIATESTEP and its properties.
- `Diophantine` - Code to solve Diophantine equations.
- `unification` - Definition of a unification problem and basic properties.
- `substitution` - Properties about substitutions.
- `equality` - Properties about equality modulo AC.
- `term_properties` - Basic properties about terms.
- `terms` - The grammar of terms.
- `list_aux_equational_reasoning`, `list_aux_equational_reasoning2parameters`, `list_aux_equational_reasoning_more` and `list_aux_equational_reasoning_nat` - Set of parametric theories that define specific functions for the task of equational reasoning (most of them operating on lists).
- `structures` - This is a different library that is being used by the formalisation, with results about data structures.

Figure 5.1 shows the dependency diagram for the PVS theories that compose the first-order AC-unification formalisation. Besides the nominal AC-matching formalisation, there are other 3 formalisations in the nominal library, which we again represent in the picture as orange ellipses. As shown in Figure 5.1, some of them use theories that are also used by the nominal AC-matching formalisation.

As mentioned in Section 2.5.1, when specifying functions and theorems, PVS may generate proof obligations to be discharged by the user. These proof obligations are called Type Correctness Conditions (TCCs), and the PVS system includes several pre-defined proof strategies that automatically try to discharge TCCs. In our code, several simple TCCs related to the well-typedness and termination of functions were proved by PVS automatically. However, manual proofs were still required for more elaborated functions (see Example 24).

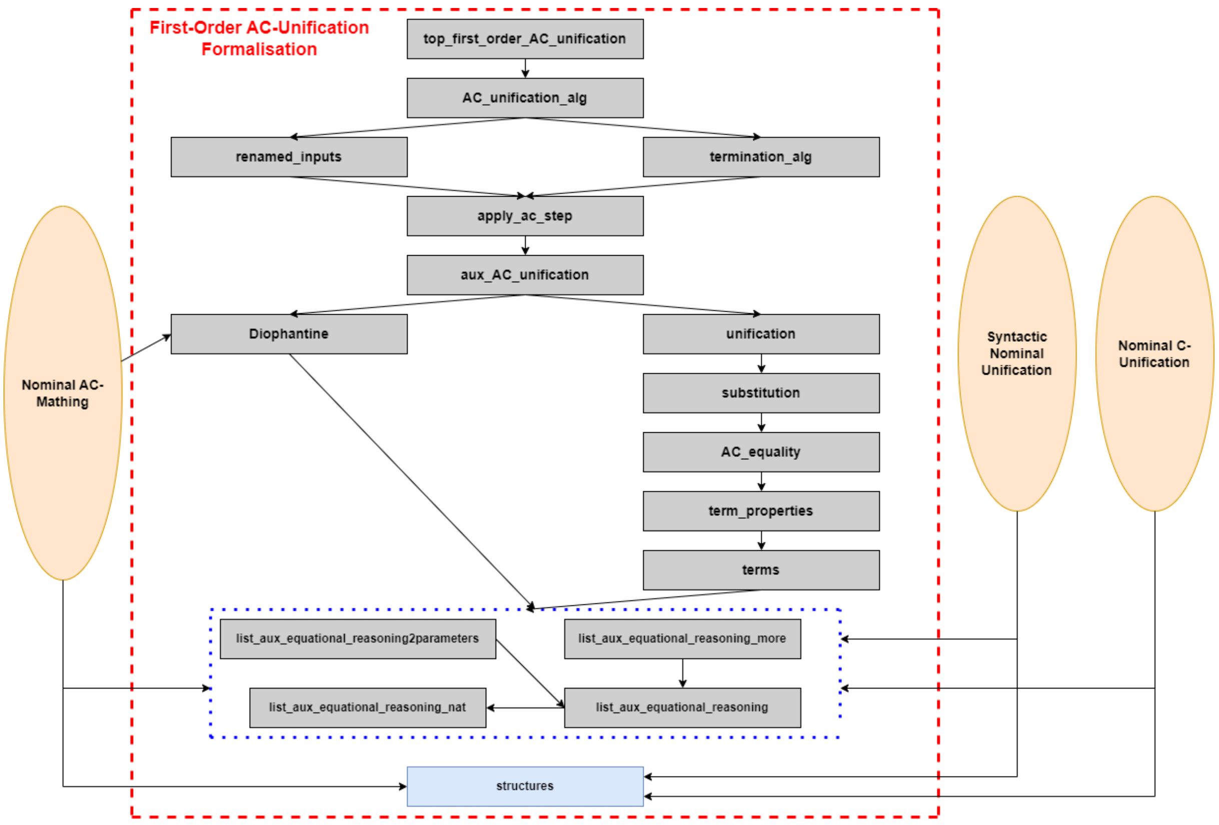

First-Order AC-Unification Formalisation
top_first_order_AC_unification
AC_unification_alg
renamed_inputs
termination_alg
apply_ac_step
aux_AC_unification
Diophantine
unification
substitution
AC_equality
term_properties
terms
Nominal AC-Mathing
Syntactic Nominal Unification
Nominal C-Unification
list_aux_equational_reasoning2parameters
list_aux_equational_reasoning_more
list_aux_equational_reasoning_nat
list_aux_equational_reasoning
structures

Figure 4.1: PVS formalisation of First-Order AC-Unification.

Example 24 (Automatic and Manual TCCs in PVS)**.** *Below, we give an example of how PVS can handle simple TCCs. Recall that a substitution σ in our code is specified as a list of nuclear substitutions. For instance, the substitution $\sigma = \{X \mapsto a, Y \mapsto b\}$ would be represented as* $\textsc{cons}((X, a), \textsc{cons}((Y, b), \textsc{nil}))$*. Consider the function* `supset_dom` *defined below, which computes a superset of the domain of σ, returning a finite set of variables.*

```
supset_dom(sigma): RECURSIVE finite_set[variable] =
  IF null?(sigma) THEN emptyset
  ELSE LET (X, t) = car(sigma) IN add(X, supset_dom(cdr(sigma)))
  ENDIF
MEASURE sigma BY <<
```

PVS extends high-order logic with predicate subtyping, allowing the definition of a new type as a subset $\{x : T \mid p(x)\}$ of a type T that satisfies a predicate p over T. Subtyping is used when defining a ***`finite_set`*** *(as a subtype of a set) and PVS profits from this concept in the case of our function* ***`supset_dom`****: it is able to automatically check that the set returned by* ***`supset_dom`*** *is indeed finite (it does not even generate a TCC), and automatically proves the TCC regarding termination of this function.*

In contrast to that, consider the definition of the domain of a substitution σ in PVS:

```
dom(sigma): finite_set[variable] = {X | subs(sigma)(X) /= variable(X)}
```

PVS generates a proof obligation (slightly simplified below) saying that we must prove that this set is indeed finite:

```
% Subtype TCC generated (at line 120, column 35) for
    % X | subs(sigma)(X) /= variable(X)
    % expected type  finite_set[variable]
  % unfinished
dom_TCC1: OBLIGATION
  FORALL (sigma: sub):
    is_finite[variable]({X |  subs(sigma)(X) /= variable(X)});
```

PVS cannot discharge this TCC automatically. We must prove it manually. To prove this TCC, we first show that the set computed by `supset_dom(sigma)` is indeed a superset of `dom(sigma)`. Then, we argue that a subset of a finite set is necessarily finite.

The number of theorems and TCCs proved for each theory, along with each theory's approximate size and percentage of the total size, is shown in Table 4.2. For this table, we omit file `top_first_order_AC_unification` since it contains only a high-level description of the formalisation and library `structures` as it is a separated library. We group theories `list_aux_equational_reasoning`, `list_aux_equational_reasoning2parameters`, `list_aux_equational_reasoning_more` and `list_aux_equational_reasoning_nat` under the name `list`, since the specifics of each one is not relevant to our discussion. Finally, PVS theories `term_properties` and `terms` are the only ones that are actually in the same file, so we group them together under the name `terms` in Table 4.2.

Remark 29 (Hardest Proofs of the Formalisations)**.** *The two hardest parts of the three formalisations described in Chapters 3, 4 and 5 were both about first-order AC-unification: the proof of termination for* APPLYACSTEP *(Section 4.3) and the proof of completeness (Section 4.4.3).*

4.6 Additional Information on the Formalisation

4.6.1 Grammar of Terms and the Need for Well-Formed Terms

First we explain function $Args_f$. This function acts recursively on the structure of a term (see Example 25) and is used to obtain a list of arguments of an AC-function headed

Table 4.2: Information for Every File In the First-Order AC-Unification Formalisation.

Theory	**Theorems**	**TCCs**	**Size**		
			`.pvs`	`.prf`	%
`unification_alg`	10	19	6 kB	2.3 MB	5%
`renamed_inputs`	21	23	10 kB	2.7 MB	6%
`termination_alg`	80	35	23 kB	11 MB	26%
`apply_ac_step`	29	12	15 kB	9.7 MB	22%
`aux_unification`	204	58	59 kB	8.2 MB	19%
`Diophantine`	73	44	24 kB	1.1 MB	3%
`unification`	86	14	20 kB	1.0 MB	2%
`substitution`	144	22	27 kB	2.4 MB	6%
`AC_equality`	67	18	12 kB	1.1 MB	3%
`terms`	131	48	28 kB	1.1 MB	3%
`list`	268	108	60 kB	2.2 MB	5%
Total	1113	401	284 kB	42.8 MB	100%

by f.

Example 25. *Some examples to illustrate the behaviour of $Args_f$.*

- $Args_f(a) = (a)$.
- $Args_f(Y) = (Y)$.
- $Args_f(\langle a, \langle b, c\rangle\rangle) = (a, b, c)$.
- $Args_f(f\langle c, b\rangle) = (c, b)$.
- $Args_f(f\ f\langle c, b\rangle) = (c, b)$.
- $Args_f(g\langle c, b\rangle) = (g\langle c, b\rangle)$.

As mentioned before, terms were defined as shown in Definition 1 in order to make it easier to eventually adapt the formalisation to the nominal setting (previous papers in the subject, such as Nominal Unification [75] by Urban et al. and Nominal C-unification [3] by Ayala et al. use a similar grammar). However, two issues arose in the formalisation that motivated us to define well-formed terms (Definition 2) and restrict the terms in the unification problem that our algorithm receive to well-formed terms.

The first issue concerns AC-functions that receive only one argument, something allowed in the grammar of terms. Let f be an AC-function symbol and consider Example 26, which shows that $ff\langle a, b\rangle \approx^? f\langle a, b\rangle$. This is problematic because it means that a unification problem such as $P = \{X \approx^? fX\}$ has a solution, for instance $\sigma = \{X \mapsto f\langle a, b\rangle\}$. Notice that if Algorithm 2 received this unification problem P, it would return NIL (line

14). In defining well-formed terms, we avoid this problem by requiring that every AC-function application $f^{AC}s$ that is a subterm of a well-formed term t does not receive only one argument.

Example 26. *Let f be an AC-function symbol. Consider the terms $t \equiv f f\langle a, b\rangle$ and $s \equiv f\langle a, b\rangle$. Two AC function applications are equal (modulo AC) if and only if their list of arguments are permutations of each other. In our particular case we have $Args_f(t) = (a, b) = Args_f(s)$ and therefore $t \approx s$.*

The second issue is with terms that are pairs. As mentioned before, pairs are to be used inside a term t to encode a tuple of arguments to a function. If t and s are not pairs and $Args_f(t)$ and $Args_f(s)$ are permutations of each other, then it is possible to prove that $t \approx s$. This result we just described was used in the proof of completeness of SOLVEAC (see the proof for Theorem 22) and is the reason why we imposed that a well-formed term t is not a pair.

Example 27 (Well-Formed Terms and Non Well-Formed Terms)**.** *Let f be an AC-function symbol and g be a syntactic function symbol. The following terms are well-formed terms:*

- $f\langle a, \langle b, c\rangle\rangle$.
- $f\ f\langle a, \langle b, c\rangle\rangle$ *(here* $Args_f(f\ f\langle a, \langle b, c\rangle\rangle) = (a, b, c)$*).*
- a.
- $g(Y)$.

The following terms are not well-formed terms:

- fX.
- $\langle a, b\rangle$.

4.6.2 Equal Terms May Not Have the Same Size

A drawback of our grammar of terms is that we can have well-formed terms that are equal modulo AC but do not have the same size. Let f be an AC-function symbol and consider, for instance, the terms $t \equiv f\langle f\langle a, b\rangle, c\rangle$ and $s \equiv f\langle\langle a, b\rangle, c\rangle$. These terms are equal modulo AC. Indeed $Args_f(t) = (a, b, c) = Args_f(s)$ but according to the definition of $size$ we have $size(t) = 7$ and $size(s) = 6$. An alternative definition of $size$, called $size_2$, which has this property (Theorem 31) is given below.

Definition 36 ($size_2$ ↗)**.** *We define the $size_2$ of a term t recursively as follows:*

- $size_2(a) = 1$
- $size_2(Y) = 1$
- $size_2(\langle\rangle) = 1$
- $size_2(\langle t_1, t_2\rangle) = size_2(t_1) + size_2(t_2)$
- $size_2(f t_1) = 1 + size_2(t_1)$
- $size_2(f^{AC} t_1) = \sum\limits_{t_i \in Args_f(f^{AC} t_1)} size_2(t_i)$

Theorem 31. *If $t \approx s$ then $size_2(t) = size_2(s)$.*

Theorem 31 is used to prove that if $X \in \mathit{Vars}(s)$ and s is a well-formed term that is not equal to X, then $X \approx^? s$ is not unifiable. This is used in the proof of completeness of our algorithm to argue that if δ unifies $\{X \approx^? s\}$ then we do not enter the `else` of line 14.

4.7 Applications

In this section, we discuss two applications of our certified AC-unification algorithm. First, how it can be used as a first step to formalise more efficient first-order AC-unification algorithms. Second, how it could be to test implementations of AC-unification.

4.7.1 Formalising More Efficient AC-Unification Algorithms

Our formalisation could be used as a starting point to prove the correctness of more efficient algorithms. For instance, when we solve a linear Diophantine equation, we generate a spanning set of solutions instead of a basis. If we modify the corresponding code to generate a basis of solutions, there would be fewer branches to explore. A second possible path to sharpen our formalisation has to do with the bound used to compute solutions to the linear Diophantine equations: we use a bound proved sufficient by Stickel [73], but we can adapt the formalisation to use a smaller bound, such as the one mentioned by Clausen and Fortenbacher [30]. Finally, a third way to be more efficient when solving the mentioned Diophantine equation is to use the graph approach also described in [30].

There are efficient algorithms for AC-unification that rely on using directed acyclic graphs (DAGs) to represent terms (e.g., Boudet's [21]). Hence, a different path would be to adapt our formalisation to formalise those algorithms. The dependency diagram of Figure 4.1 hints at why adapting our formalisation to prove the correctness of algorithms representing terms as DAGs should give us more work than solving the linear Diophantine equations more efficiently. Changing the representation of terms would impact mostly `terms.pvs` but would also require modification in lemmas from other files proved by induction on terms. This means file changes that depend on `terms.pvs`, especially the ones that more closely depend on `terms.pvs`, such as `equality.pvs`, `substitution.pvs` and `unification.pvs`. In contrast, solving the linear Diophantine equations more efficiently should effectively only require changes in `Diophantine.pvs`.

To further illustrate the additional work of changing the term representation in comparison to solving the linear Diophantine equations more efficiently, let's consider the proof of termination of ACUnif, described in Section 4.2.1, which is effectively done in file `termination_alg.pvs` (one of the hardest parts of our formalisation, see Table 4.2). Recalling that the lexicographic measure used is:

$$lex = (|V_{NAC}(P)|,\ |V_{>1}(P)|,\ |AS(P)|,\ size(P))$$

we see that the procedure used to solve the linear Diophantine equations plays no role in this proof. In contrast to that, $V_{NAC}(P)$, $V_{>1}(P)$, $AS(P)$, $size(P)$ depend respectively on $V_{NAC}(t)$, $Subterms(t)$ and $size(t)$ which were all defined inductively on the structure of terms and would need to be adjusted in case we changed the way we represent terms.

4.7.2 Testing Implemented AC-Unification Algorithms

Although PVS does not support code extraction to a programming language such as OCaml or Haskell, we can use our formalisation to test implementations of first-order AC-unification algorithms in two different manners. The first approach is to manually translate our implementation to a programming language of our choice (Python, for instance) and then run both the manual translation of the formalised algorithm and the nominal AC-unification algorithm we wish to test against the same examples, comparing the results.

The second approach is to use the PVSio feature of PVS. As mentioned in Section 2.5.2, PVSio is a PVS package that extends the capabilities of the ground evaluator with a predefined library of imperative programming language features, among them input and output operators. This implies that *sometimes* we can run the formalised algorithm inside the PVS environment passing the input we want and seeing the output returned. However, some code fragments of our formalisation would need to be adapted in order to use this resource or semantic attachments must be provided (see Section 2.5.2 for a description of what types of code fragments must be adapted). Compared to the first approach (manually translating to a programming language), the second approach (using the PVSio feature) is less error-prone but requires more effort.

Chapter 5

Nominal AC-Matching

This chapter describes how we extended the first-order AC-unification formalisation (Chapter 4), providing the mechanisms to deal with atoms, permutations, suspended variables, abstraction, and obtained a formalisation of nominal AC-matching. To the best of our knowledge, this is the first algorithm for nominal AC-matching. As was done in Chapter 3, the formalisation uses a parameter $\mathcal{X}$ for protected variables. By correctly setting this parameter, in addition to a nominal AC-matching algorithm, it was possible to obtain a nominal AC-equality checker as a byproduct. The results of this Chapter are described in less details in Ayala-Rincón et al. [8].

5.1 Algorithm

Following the approach of Chapters 3 and 4, we present the algorithm's pseudocode instead of the actual PVS code for readability. We developed a functional algorithm (Algorithm 5) for matching terms t and s. The algorithm is recursive and needs to keep track of the current context Γ, the equational constraints P that we have to unify, the substitution σ computed so far, the set of variables V that are/were in the problem and the set of protected variables $\mathcal{X}$. Hence, its input is a quintuple $(\Gamma, P, \sigma, V, \mathcal{X})$. The output is a list of solutions, each of the form (Γ_1, σ_1). The freshness constraints are treated by auxiliary functions (see Section 5.1.3), and the equational constraints P are represented as a list in our PVS code, where each element of the list is a pair (t_i, s_i) that represents an equation $t_i \approx^? s_i$.

The first call to the algorithm, in order to match t to s, is done with $P = \{t \approx^? s\}$; $\Gamma = \emptyset$; $\sigma = id$ (because we have not computed any freshness constraint or substitution yet); $V = Vars(t, s)$ and $\mathcal{X} = Vars(s)$.

Remark 30. *In the PVS code, this means that the initial call is done with parameters* $P = cons((t, s), \textsc{nil})$, $\Gamma = \textsc{nil}$, $\sigma = \textsc{nil}$, $V = Vars(t, s)$ *and* $\mathcal{X} = Vars(s)$.

Algorithm 5 Nominal AC-Matching

```
procedure ACMATCH($\Gamma, P, \sigma, V, \mathcal{X}$)
    if nil?($P$) then $cons((\Gamma, \sigma), \text{NIL})$
    else let $((t, s), P_1) = \text{CHOOSEEQ}(P)$ in
        if $t$ matches $a$ and $s$ matches $a$ then ACMATCH($\Gamma, P_1, \sigma, V, \mathcal{X}$)

        else if $t$ matches $\pi \cdot X$ and $X \notin Vars(s)$ and $X \notin \mathcal{X}$ then
            let $\sigma_1 = \{X \mapsto \pi^{-1} \cdot s\}$,
                $(\Gamma_1, flag) = \text{FRESHSUBS?}(\sigma_1, \Gamma)$ in
            if $flag$ then ACMATCH($\Gamma_1 \cup \Gamma, \sigma_1 P_1, \sigma_1 \sigma, V, \mathcal{X}$)
            else NIL

        else if $t$ matches $\pi \cdot X$ and $s$ matches $\pi' \cdot X$ then
            let $\Gamma_1 = ds(\pi, \pi')\#X \cup \Gamma$ in ACMATCH($\Gamma_1, P_1, \sigma, V, \mathcal{X}$)

        else if $t$ matches $\langle\rangle$ and $s$ matches $\langle\rangle$ then ACMATCH($\Gamma, P_1, \sigma, V, \mathcal{X}$)

        else if $t$ matches $f\ t_1$ and $s$ matches $f\ s_1$ then
            let $(P_2, flag) = \text{DECOMPOSE}(t_1, s_1)$ in
            if $flag$ then ACMATCH($\Gamma, P_2 \cup P_1, \sigma, V, \mathcal{X}$)
            else NIL

        else if $t$ matches $[a]\ t_1$ and $s = [a]\ s_1$ then
            let $(P_2, flag) = \text{DECOMPOSE}(t_1, s_1)$ in
            if $flag$ then ACMATCH($\Gamma, P_2 \cup P_1, \sigma, V, \mathcal{X}$)
            else NIL

        else if $t$ matches $[a]\ t_1$ and $s = [b]s_1$ then
            let $(\Gamma_1, flag_1) = \text{FRESH?}(a, s_1)$,
                $(P_2, flag_2) = \text{DECOMPOSE}(t_1, (a\ b) \cdot s_1)$ in
            if $flag_1$ and $flag_2$ then ACMATCH($\Gamma \cup \Gamma_1, P_2 \cup P_1, \sigma, V, \mathcal{X}$)
            else NIL

        else if $t$ matches $f^{AC}\ t_1$ and $s$ matches $f^{AC}\ s_1$ then
            let $InputLst = \text{APPLYACSTEP}\ (\Gamma, cons((t, s), P_1), \sigma, V, \mathcal{X})$,
                $LstResults = \text{MAP}(\text{ACMATCH}, InputLst)$ in FLATTEN($LstResults$)

        else NIL
```

Although extensive, Algorithm 5 is simple. It starts by analysing the list P of terms to match. If it is empty (line 2), it has finished and can return the answer computed so far, a list with a unique element: (Γ, σ). Otherwise, the algorithm calls the auxiliary function CHOOSEEQ (line 3), which returns a pair (t, s) and a list of equational constraints P_1 such that $P = \{t \approx^? s\} \cup P_1$. Then, P is updated by simplifying $\{t \approx^? s\}$ and this is done by seeing the form of t (an atom, a moderated variable, a unit, and so on).

5.1.1 Function chooseEq

The function CHOOSEEQ(P) selects an equational constraint $t \approx^? s$ in P, picking the one with the biggest size. Notice that the behaviour of this function is different from its first-order AC-unification counterpart, since in that formalisation CHOOSEEQ (see Section 4.1.1) avoids selecting AC-unification pairs, instead of picking the equational constraint with the biggest size. The design of CHOOSEEQ, in both formalisations, helped us in the proof of termination (see Sections 4.2 and 5.2.3).

5.1.2 Function decompose

The function DECOMPOSE (lines 18, 23 and 29) works as its corresponding one in the first-order AC-unification formalisation (see Section 4.1.2). It receives two terms t and s, and if they are both pairs, it recursively tries to decompose them, returning a tuple (P, flag), where P is a list of equational constraints and flag is a boolean that is $True$ if the decomposition was successful. If neither t nor s is a pair, the unification problem returned is just $P = \{t \approx^? s\}$ and $\mathit{flag} = True$. If one of the terms is a pair and the other is not, the function returns (NIL, $False$). In Algorithm 5, we call DECOMPOSE(t_1, s_1) when we encounter equations such as $ft_1 \approx^? fs_1$ to guarantee that all the terms in the unification problem remain well-formed. Although it would have been correct to simplify an equation of the form $ft_1 \approx^? fs_1$ to $t_1 \approx^? s_1$, if t_1 or s_1 were pairs, we would not respect our restriction that only well-formed terms are in the matching problem.

Example 28. *Examples of the function DECOMPOSE acting on nominal terms are given below.*

- *DECOMPOSE*$(\langle a, \langle b, c\rangle\rangle,\ \langle c, \langle \pi \cdot X, Y\rangle\rangle) = (\{a \approx^? c,\ b \approx^? \pi \cdot X,\ c \approx^? Y\},\ True)$.
- *DECOMPOSE*$(a, [a]Y) = (\{a \approx^? [a]Y\},\ True)$.
- *DECOMPOSE*$(X, \langle c, [b]\pi \cdot Z\rangle) = (NIL,\ False)$.

5.1.3 Handling Freshness Constraints - Functions freshSubs? and fresh?

As described in Chapter 3, we followed the approach of [11] and handled freshness constraints separately by using the auxiliary functions FRESH? and FRESHSUBS? . These functions were already implemented in [11], and extending them to handle AC-functions is straightforward.

- FRESHSUBS?(σ, Γ) returns the minimal context (Γ_1 in Algorithm 5) in which $a\#_?\sigma X$ holds, for every $a\#X$ in the context Γ and a boolean (*flag* in Algorithm 5), indicating if it was possible to find the mentioned context.

- FRESH?$(a,\ t)$ computes and returns the minimal context (Γ_1 in Algorithm 5) in which a is fresh for t and a boolean (*flag* in Algorithm 5), indicating if it was possible to find the aimed context.

5.1.4 The Function applyACStep

The function APPLYACSTEP was adapted from the formalisation of first-order AC-unification (see [9]). In contrast to its first-order counterpart, it handles only one equational constraint $t \approx^? s$, where t and s are rooted by the same AC function symbol. As we will see, termination for nominal AC-matching is a simpler problem than for first-order AC-unification and that makes it is sufficient for APPLYACSTEP to handle only one equational constraint. Hence, in a high-level overview, APPLYACSTEP will apply SOLVEAC to the first equational constraint and then call function INSTANTIATESTEP in the result, instantiating the variables that it can. This function returns a list (*InputLst* in line 34 of Algorithm 5) with each entry in this list corresponding to a branch ACMATCH will explore. ACMATCH explores every branch generated by calling itself recursively on every input in *InputLst* (line 34 of the algorithm). The algorithm's output is a list of solutions of the form (Γ, σ), where Γ is a context and σ is a substitution. In addition, the result of calling MAP(ACMATCH, *InputLst*), *LstResults* in line 35 of Algorithm 5, is a list of lists of solutions. Hence, *LstResults* is flattened and then returned.

Remark 31 (SOLVEAC and INSTANTIATESTEP)**.** *APPLYACSTEP relies on two functions: SOLVEAC and INSTANTIATESTEP, which are fully described in Chapter 4. Recall that, in short, the function SOLVEAC finds the linear Diophantine equational system associated with the AC-matching equational constraint, generates the basis of solutions, and uses these solutions to generate the new AC-matching equational constraints. The function INSTANTIATESTEP instantiates the moderated variables that it can.*

5.1.5 Modifications to Adapt the Algorithm to the Nominal Setting

The example of Section 2.2.2 describes the process of trying to unify two terms $t \equiv f(t_1, \ldots, t_m)$ and $s \equiv f(s_1, \ldots, s_n)$, where f is an AC-function symbol. In this Section we detail the four modifications that were necessary to adapt this process to the nominal setting, referring to the example of Section 2.2.2 when convenient.

The first is related to eliminating common arguments: we do not eliminate arguments t_i and s_j of t and s if they are equal modulo AC, we eliminate them if they are α-equivalent (modulo AC) under the context Γ that we are working with, i.e., if $\Gamma \vdash t_i \approx s_j$. If we have as hypothesis that (Δ, δ) is the solution to the quintuple we are working with (see Definition 22), the correctness of this step boils down to proving that from $\Gamma \vdash t_i \approx s_j$ we have $\Delta \vdash \delta t_i \approx \delta s_j$. This is possible to prove by using the fact that $\Delta \vdash \delta\Gamma$ (item 1 of Definition 22).

The second change is related with the new variables (Z_is in the Section 2.2.2) introduced and the fact that in the nominal setting a moderated variable $\pi \cdot X$ always has a permutation π suspended on the variable X. What should be the permutation π suspended on the new variables? Since the ultimate goal of these new variables is to outline the combinatory between the arguments of t and the arguments of s, we put the identity permutation suspended on the new variables. For instance, in the example of Section 2.2.2 we would have the moderated variables $id \cdot Z_1, \dots, id \cdot Z_7$, which we would write (see Remark 9) simply as $Z_1, \dots, Z_7$.

In example of Section 2.2.2, we have variables X_1, X_2, X_3, Y_1, Y_2 to represent respectively the arguments X, Y, a, b, Z and we say that when generating the new unification problems we can discard the ones "where a variable that does not represent a variable term is paired with an AC-function application". Here, we can also discard problems where a moderated variable $\pi \cdot X$, with $X \in \mathcal{X}$, is paired with an AC-function application. This is the third change to adapt to the nominal setting.

Finally, we must guarantee that the new variables Z_is introduced by the algorithm can be instantiated. Since those new variables are not in the set V, we ensure that by putting the restriction that $\mathcal{X} \subseteq V$ in the definition of nice inputs (Definition 37).

5.1.6 Common Structures of Equational Constraints Returned by solveAC

Suppose function SOLVEAC is called to simplify $s \approx^? t$, where both s and t are headed by the same AC-function symbol f. Let $s_1, \dots, s_m$ be the different arguments of s and let $t_1, \dots, t_n$ be the different arguments of t, after eliminating the common arguments of s and t. An arbitrary set of equational constraints P_1 obtained after we apply SOLVEAC is of the form

$$\{t_1 \approx^? t'_1, \dots, t_n \approx^? t'_n, s_1 \approx^? s'_1, \dots, s_m \approx^? s'_m\}$$

where:

1. A term t'_i in the right-hand side is a new variable Z_j.

2. A term s'_i in the right-hand side is either a new variable Z_j or an AC-function headed by f whose arguments are all new variables Z_j.

A term s'_i will only be an AC-function headed by f if the corresponding term s_i is an unprotected variable. This is the reason why t'_i cannot be an AC-function headed by f: the corresponding term t_i is part of the right-hand side of $s \approx^? t$ and therefore is not an unprotected variable.

After calling SOLVEAC, we must instantiate the variables that we can. We must handle first the equational constraints $t_i \approx^? t'_i$ and only then go to the equational constraints $s_i \approx^? s'_i$, in order to keep all the protected variables in the right-hand side, as shown in Example 29.

Example 29. *Let f be an AC function symbol and g be a syntactic function symbol. Consider the equational constraint $P_0 = \{f(g(X), g(W)) \approx^? f(g(a), g(b))\}$, and let $\mathcal{X} = \emptyset$ be our set of protected variables. After we apply SOLVEAC, one of the branches has the following equational constraints:*

$$\begin{aligned} g(X) &\approx^? Z_1 \\ g(Y) &\approx^? Z_2 \\ g(a) &\approx^? Z_2 \\ g(b) &\approx^? Z_1 \end{aligned}$$

If INSTANTIATESTEP instantiates the variables that it can in the order shown above, we get as result the equational constraints

$$\{g(a) \approx^? g(Y), g(b) \approx^? g(X)\},$$

and there would be unprotected variables in the right-hand side of the problem, i.e. we would not have a matching problem anymore. To prevent this situation from happening, every equational constraints $t_i \approx^? t'_i$ is handled by INSTANTIATESTEP before any equational constraint $s_i \approx^? s'_i$.

Recall that the set of equational constraints

$$\{t_1 \approx^? t'_1, \dots, t_n \approx^? t'_n, s_1 \approx^? s'_1, \dots, s_m \approx^? s'_m\}$$

is represented in the PVS code as a list. Then, this order of which equational constraint to handle first is done in the PVS code by putting equations $t_i \approx^? t'_i$ before equations $s_i \approx^? s'_i$ in the list and having INSTANTIATESTEP iterate through the list.

5.2 Formalisation

As was done in the formalisation of first-order AC-unification (see Chapter 4), to help us in the proofs of termination (Section 5.2.3), soundness (Section 5.2.4) and completeness (Section 5.2.5) we define the notion of a *nice input* (Section 5.2.2). Before diving in these proofs, we show how every new moderated variable Z_i introduced by SOLVEAC is instantiated by INSTANTIATESTEP.

5.2.1 Instantiation of the New Variables Introduced By solveAC

In contrast to first-order AC-unification, in nominal AC-matching it is possible to prove that the new variables $Z_1, \dots, Z_n$ introduced by SOLVEAC are instantiated by INSTANTIATESTEP. This fundamental result was used to prove that the notion of nice inputs (Definition 37) is preserved between recursive calls of ACMATCH and to establish termination of nominal AC-matching.

Indeed, let f be an AC function symbol and suppose that the equational constraint that SOLVEAC is simplifying is $f(s_1, \dots, s_m) \approx^? f(t_1, \dots, t_n)$. Since we are dealing with matching, there are no unprotected variables in the right-hand side and therefore every t_i is either a protected variable or a non-variable term. The new variable Z_i introduced by SOLVEAC is associated with row i of the Diophantine matrix D, and this row is a non-zero solution to the Diophantine equation. Hence, at least one column $j + m$, that corresponds to a term t_j on the right-hand side, will have its i-th entry non-zero. In other words, there is some j such that $D_{i,j+m} \neq 0$.

This means that one of our equational constraints after we apply SOLVEAC will be $t_j \approx^? Z_i$ and the new variable Z_i will be instantiated. Could we have instead:

$$t_j \approx^? f(Z_i, \text{other arguments})?$$

No. Since t_j is not an unprotected variable and is not an AC function headed by f this kind of equation has no solution and is eliminated by our algorithm.

5.2.2 Nice Input

Nice input is an invariant under the action of the ACMATCH function with valuable properties. Notice that Item 7 of Definition 37 would need to be removed for the proofs of termination, soundness, and completeness to be used in unification.

Definition 37 (Nice Input). *An input $(\Gamma, P, \sigma, V, \mathcal{X})$ is said to be nice if:*

1. *σ is idempotent.*

2. $\mathit{Vars}(P) \cap \mathit{dom}(\sigma) = \emptyset$.

3. $\sigma \subseteq V$.

4. $\mathit{Vars}(P) \subseteq V$.

5. $\mathit{Vars}(\Gamma) \subseteq V$.

6. $\mathcal{X} \subseteq V$.

7. $\mathit{Vars}(\mathit{rhs}(P)) \subseteq \mathcal{X}$.

Items 1 to 4 were present in the definition of nice input for the formalisation of first-order AC-unification, while Items 5 to 7 were added. Item 5 of Definition 34 was expected, since we already have similar hypotheses for P and σ. Item 6 guarantees that the new variables introduced by the algorithm can be instantiated (see Section 5.1.5).

Preservation of $\boldsymbol{\mathit{Vars}(\mathit{rhs}(\mathbf{P})) \subseteq \mathcal{X}}$

Although proving that Items 1-6 are preserved between the recursive calls of ACMatch is straightforward, this is not the case for Item 7. The complicated case is when ACMatch calls applyACStep to simplify $t \approx^? s$. Recall that applyACStep essentially calls solveAC and then instantiateStep in the results returned by solveAC. The issue is that after calling solveAC (see Section 5.1.6) and before calling instantiateStep we do not have $\mathit{Vars}(\mathit{rhs}(P)) \subseteq \mathcal{X}$.

To prove that after applyACStep we still have Item 7, we proceed in two steps:

1. Let P_0 be the equational constraints before we apply solveAC. $\mathit{Vars}(\mathit{rhs}(P_0)) \subseteq \mathcal{X}$. Let P_0^* be the equational constraints after solveAC but before we call instantiateStep. Although $\mathit{Vars}(\mathit{rhs}(P_0^*)) \not\subseteq \mathcal{X}$, we proved that P_0^* satisfies some valuable properties, that we encapsulate in the definition of *matching condition* (Definition 39).

2. Let P_1 be the equational constraints after we call instantiateStep on P_0^*. We proved that if P_0^* satisfies the matching condition then P_1 satisfies $\mathit{Vars}(\mathit{rhs}(P_1)) \subseteq \mathcal{X}$.

The motivation for the definition of matching condition is to guarantee that every unprotected variable introduced by solveAC gets instantiated to a term with protected variables. It relies on the definition of a matching equation (Definition 38).

Definition 38 (Matching Condition Equation). *We say that $t \approx^? s$ is a matching condition equation if s is a variable and $\mathit{Vars}(t) \subseteq \mathcal{X}$.*

Definition 39 (Matching Condition). *We say that the equational constraints P satisfy the matching condition with respect to $\mathcal{X}$ if for every variable $X \in rhs(P)$ that is not in $\mathcal{X}$ there exists i such that:*

- *For every $j < i$, X is not in the variables of the j-th equation of P.*
- *X is a member of the i-th equation of P.*
- *The i-th equation is a matching condition equation.*

Notice that Item 1 of Definition 39 guarantee that X won't be instantiated before the i-th equation, while Items 2 and 3 guarantee that X will be instantiated in the i-th equation to a term that only contains protected variables.

5.2.3 Termination

For the lexicographic measure used in the proof of termination, we need the definition of the size of an equational constraint $t \approx^? s$ (Definition 40).

Definition 40 (Size of an Equational Constraint). *The size of an equational constraint $t \approx^? s$ is $size(t) + size(s)$, where the size of a term t is recursively defined as follows:*

- $size(a) = 1$.
- $size(\pi \cdot X) = 1$.
- $size(\langle\rangle) = 1$.
- $size(\langle t_1, t_2\rangle) = 1 + size(t_1) + size(t_2)$.
- $size(f\ t_1) = 1 + size(t_1)$.
- $size(f^{AC}\ t_1) = 1 + size(t_1)$.
- $size([a]t_1) = 1 + size(t_1)$.

Although the nominal AC-matching algorithm is based on the first-order AC-unification algorithm, the proof of termination was done from scratch and it was much easier than the corresponding one for first-order AC-unification. It was possible to prove that for the particular case of matching (unlike unification) all the new moderated variables introduced by SOLVEAC are instantiated by INSTANTIATESTEP, which greatly simplified the proof. This allowed us to use a simpler lexicographic measure than the one used for first-order AC-unification (see Chapter 4).

The lexicographic measure used has as its first component the number of variables in the equational constraints P and as a second component the multiset order of the size of each equation $t \approx^? s \in P$. Although PVS does not directly implement multiset orders, this part can be emulated easily by analysing the maximum size n of all equations $t \approx^? s$ in P and the number of equations $t \approx^? s$ in P with maximal size (in this order). Algorithm 5 always selects an equation with maximal size to simplify (the heuristic selection is enforced by the function CHOOSEEQ).

Let $MS(P)$ be the maximum size n of all equations $t \approx^? s$ in P and let $NMS(P)$ be the number of equations $t \approx^? s$ whose size is equal to $MS(P)$. The lexicographic measure is then

$$lex = (|Vars(P)|, MS(P), NMS(P)).$$

Table 5.1 shows which components do not increase (represented by $\leq$) and which components strictly decrease (represented by $<$).

Table 5.1: Decrease of the Components of the Lexicographic Measure.

Recursive Call	**|Vars(P)|**	**MS(P)**	**NMS(P)**
line 4, 13, 15, 19, 24, 30, 35 (case 1)	$\leq$	$\leq$	$<$
line 4, 13, 15, 19, 24, 30, 35 (case 2)	$\leq$	$<$	
line 9	$<$		

5.2.4 Soundness

As mentioned, to match terms t and s we first call Algorithm 5 with parameters $\Gamma = \emptyset$, $P = \{t \approx^? s\}$, $\sigma = id$, $V = Vars(t, s)$ and $\mathcal{X} = Vars(s)$. However, since the parameters of ACMATCH change after recursive calls, the proof of soundness (Corollary 33) cannot be done directly by induction, and we must instead prove first Theorem 32 with generic parameters Γ, P, σ, V and $\mathcal{X}$. Once Theorem 32 is proved, it is also immediate to adapt the algorithm to solve nominal AC-equality checking, by setting $\mathcal{X} = Vars(t, s)$, and prove its soundness (Corollary 34).

Theorem 32 (Soundness for Nice Inputs). *Let the pair (Γ_1, σ_1) be an output of ACMATCH$(\Gamma, P, \sigma, V, \mathcal{X})$ and suppose that $(\Gamma, P, \sigma, V, \mathcal{X})$ is a nice input. If (Δ, δ) is a solution to $(\Gamma_1, \emptyset, \sigma_1, \mathbb{X}, \mathcal{X})$ then (Δ, δ) is a solution to $(\Gamma, P, \sigma, \mathbb{X}, \mathcal{X})$.*

Corollary 33 (Soundness for AC-Matching). *Let the pair (Γ_1, σ_1) be an output of ACMATCH$(\emptyset, \{t \approx^? s\}, id, Vars(t, s), Vars(s))$. If (Δ, δ) is an instance of (Γ_1, σ_1) that does not instantiate the variables in s, then (Δ, δ) is a solution to $(\emptyset, \{t \approx^? s\}, id, \mathbb{X}, Vars(s))$.*

Corollary 34 (Soundness for AC-Equality Checking). *Let (Γ_1, σ_1) be an output of ACMATCH$(\emptyset, \{t \approx^? s\}, id, Vars(t, s), Vars(t, s))$. If (Δ, δ) is an instance of (Γ_1, σ_1) that doesn't instantiate variables in t or s, then (Δ, δ) is a solution to $(\emptyset, \{t \approx^? s\}, id, \mathbb{X}, Vars(t, s))$.*

Remark 32. *An interpretation of Corollary 33 is that if* (Δ, δ) *is an AC-matching instance to one of the outputs of* ACMATCH, *then* (Δ, δ) *is an AC-matching solution to the original problem. Corollary 34 has a similar interpretation, replacing AC-matching with AC-equality checking.*

The proof of soundness was mainly a straightforward adaptation from the proof of soundness of first-order AC-unification (see Section 4.4.2). The soundness of FRESH? and FRESHSUBS? were straightforward adaptations from the work of [11], since the only case not covered in [11] (the case of AC-functions) is similar to the case of syntactic functions.

5.2.5 Completeness

Completeness of Algorithm 5 with extra hypotheses $\delta \subseteq V$ and $Vars(\Delta) \subseteq V$ is given by Corollary 36 and similarly to the soundness proof, it is derived easily after proving Theorem 35.

Theorem 35 (Completeness for Nice Inputs). *Let* $(\Gamma, P, \sigma, V, \mathcal{X})$ *be a nice input. Suppose that* (Δ, δ) *is a solution to* $(\Gamma, P, \sigma, \mathbb{X}, \mathcal{X})$*, that* $\delta \subseteq V$ *and that* $Vars(\Delta) \subseteq V$*. Then, there exists* (Γ_1, σ_1) *such that:*

1. $(\Gamma_1, \sigma_1) \in$ ACMATCH$(\Gamma, P, \sigma, V, \mathcal{X})$.
2. (Δ, δ) *is an instance (restricted to the variables of* V*) of* (Γ_1, σ_1) *that does not instantiate the variables in* $\mathcal{X}$.

Corollary 36 (Completeness for AC-Matching With Arbitrary V). *Suppose that* (Δ, δ) *is a solution to* $(\emptyset, \{t \approx^? s\}, id, \mathbb{X}, Vars(s))$*, that* $\delta \subseteq V$ *and that* $Vars(\Delta) \subseteq V$*. Then, there exists* (Γ_1, σ_1) *such that:*

1. $(\Gamma_1, \sigma_1) \in$ ACMATCH$(\emptyset, \{t \approx^? s\}, id, V, Vars(s))$.
2. (Δ, δ) *is an instance (restricted to the variables of* V*) of* (Γ_1, σ_1) *that does not instantiate the variables of* s.

Corollary 37 (Completeness for AC-equality Checking With Arbitrary V). *Suppose* (Δ, δ) *is a solution to* $(\emptyset, \{t \approx^? s\}, id, \mathbb{X}, Vars(t, s))$ *satisfying* $\delta \subseteq V$ *and* $Vars(\Delta) \subseteq V$*. Then, there exists* (Γ_1, σ_1) *such that:*

1. $(\Gamma_1, \sigma_1) \in$ ACMATCH$(\emptyset, \{t \approx^? s\}, id, V, Vars(t, s))$.
2. (Δ, δ) *is an instance (restricted to the variables of* V*) of* (Γ_1, σ_1) *that does not instantiate the variables of* t *or* s.

As was the case for first-order AC-unification (see Section 4.4.3), the hypothesis $\delta \subseteq V$ in the proof of completeness is a technicality that was put in order to guarantee the new variables introduced by the algorithm in the AC-part do not clash with the variables in $dom(\delta)$ or in the terms in $im(\delta)$. This mechanism could be replaced by a different one that assures that the variables introduced by the AC-part of ACMATCH are indeed new. When going from the first-order setting to the nominal setting, we go from having a unifier δ to a pair (Δ, δ) and hence we must add the hypothesis $Vars(\Delta) \subseteq V$.

First, we give a high-level description of how to remove hypotheses $\delta \subseteq V$ and $Vars(\Delta) \subseteq V$ from Lemma 36. The critical step to prove a variant of Theorem 35 with $V = Vars(t, s)$ and without the hypotheses $\delta \subseteq V$ and $Vars(\Delta) \subseteq V$ is to prove that the outputs computed when we call ACMATCH with input $(\Gamma, P, \sigma, V, \mathcal{X})$ "differ only by the name of the new variables" from the outputs computed when we call ACMATCH with input $(\Gamma, P, \sigma, V', \mathcal{X})$. However, this cannot be proved directly by induction because if V and V' differ and ACMATCH enters in the AC-part, the new variables introduced for each input may "differ only by the name of the new variables" and once we instantiate those variables, it may happen that the substitutions computed so far (the third component in the input quintuple) will also "differ only by the name of the new variables". The solution is to prove the more general statement that if the inputs $(\Gamma, P, \sigma, V, \mathcal{X})$ and $(\Gamma, P, \sigma', V', \mathcal{X})$ "differ only by the name of the new variables", then the output of ACMATCH with the first input "differ only by the name of the new variables" from the output of ACMATCH with the second input.

The third and fourth components of the input "differ only by the name of the new variables", but in contrast to what happened in first-order AC-unification, the remaining components stay equal between the recursive calls of ACMATCH. This occurs due to every variable Z_i introduced by SOLVEAC being instantiated by INSTANTIATESTEP.

The idea described in the previous paragraphs has been formalised and it relies on the concept of Variant Inputs (Definition 41). Another crucial concept is the Variant Input Condition (Definition 42) .

Definition 41 (Variant Inputs). *We say that $(\Gamma, P, \sigma, V, \mathcal{X})$ and $(\Gamma', P', \sigma', V', \mathcal{X}')$ are variant inputs fixing Ψ if:*

1. $\Gamma = \Gamma'$, $P = P'$, $\mathcal{X} = \mathcal{X}'$.
2. $\sigma' =_{\Psi} \sigma$.
3. $\Psi \subseteq V$ *and* $\Psi \subseteq V'$.
4. $Vars(P) \subseteq \Psi$ *and* $Vars(\Gamma) \subseteq \Psi$ *and* $\mathcal{X} \subseteq \Psi$.
5. $max(V) \leq max(V')$.

Definition 42 (Variant Input Condition). *We say that $(\Gamma, P, \sigma, V, \mathcal{X}, \sigma', \Psi, \rho, V_1)$ satisfy the variant input condition if:*

1. *P satisfies the matching condition with respect to $\mathcal{X}$.*
2. *$\mathcal{X} \subseteq \Psi$, $\mathit{Vars}(\Gamma) \subseteq \Psi$, $\mathit{Vars}(\mathit{lhs}(P)) \subseteq \Psi$.*
3. *$\sigma =_\Psi \sigma'$.*
4. *$\mathit{dom}(\rho) \cap V = \emptyset$.*
5. *$\mathit{Vars}(\mathit{im}(\sigma)) \subseteq V$.*
6. *$\mathit{Vars}(P) \subseteq V_1$ and $V \subseteq V_1$.*
7. *If $X \in \mathit{Vars}(\mathit{im}(\rho))$ and $X \notin \mathit{dom}(\rho)$ then $X \notin V_1$.*

We can state Theorem 38 will be fundamental to prove completeness for the tasks of nominal AC-matching (Corollary 39) and nominal AC-equality checking (Corollary 40).

Theorem 38 (Correctness of Variant Inputs). *Let $(\Gamma, P, \sigma, V, \mathcal{X})$ and $(\Gamma, P, \sigma', V', \mathcal{X})$ be variant inputs fixing Ψ. If $(\Gamma_1, \sigma'_1) \in \text{ACMatch}(\Gamma, P, \sigma', V', \mathcal{X})$ then exists σ_1 such that:*

1. *$(\Gamma_1, \sigma_1) \in \text{ACMatch}(\Gamma, P, \sigma, V, \mathcal{X})$.*
2. *$\sigma_1 =_\Psi \sigma'_1$.*

The hardest part of the proof of Theorem 38 is when ACMatch calls applyACStep. Recall that applyACStep essentially calls solveAC in an equational constraint $t \approx^? s$ and then instantiateStep in the results returned by solveAC.

Let input_0 and input'_0 be the two inputs before we call solveAC. Let input_1 and input'_1 be the two inputs after we call solveAC and before we call instantiateStep. Finally, let input_2 and input'_2 be the two inputs after we call instantiateStep. We prove that input_2 and input'_2 are variant inputs fixing ψ in two steps:

1. Although we cannot prove that input_1 and input'_1 are variant inputs, we can prove that the components of inputs input_1 and input'_1 satisfy some valuable properties, that we encapsulate in the definition of *variant input condition* (Definition 42).
2. We prove that if we call instantiateStep on the equational constraints of input_1 and on the equational constraints of input'_1, obtaining respectively inputs input_2 and input'_2 then input_2 and input'_2 are variant inputs.

Remark 33 (Two-Step Strategy in $\mathit{Vars}(\mathit{rhs}(P)) \subseteq \mathcal{X}$ and in Variant Inputs)**.** *Notice that both the proof that* APPLY*AC*STEP *preserves the condition* $\mathit{Vars}(\mathit{rhs}(P)) \subseteq \mathcal{X}$ *between recursive calls (Section 5.2.2) and the proof that* APPLY*AC*STEP *preserves variant inputs fixing* ψ *follows a two-step strategy. The issue in both cases is that although our problem* P_0 *has a certain condition* X *that is present before* APPLY*AC*STEP *and we must prove that it remains present in the problem* P_1 *obtained after* APPLY*AC*STEP*, this condition* X *does not hold after we call* SOLVE*AC and before we call* INSTANTIATE*S*TEP*. The solution is to prove that after* SOLVE*AC we have a problem* P_0^* *with a weaker condition* Y*, and that if we call* INSTANTIATE*S*TEP *in* P_0^**, we get back a problem* P_1 *with the condition* X *that we wanted. Schematically:*

$$P_0 \ (\textit{Condition } X) \xrightarrow{\text{SOLVE}AC} P_0^* \ (\textit{Condition } Y) \xrightarrow{\text{INSTANTIATE}S\text{TEP}} P_1 \ (\textit{Condition } X)$$

Finally, we used Theorem 38 and Corollaries 36 and 37 to prove completeness of Algorithm 5 for matching (Corollary 39) and AC-equality checking (Corollary 40).

Corollary 39 (Completeness for AC-Matching)**.** *Suppose that* (Δ, δ) *is a solution to the quintuple* $(\emptyset, \{t \approx^? s\}, id, \mathbb{X}, \mathit{Vars}(s))$*. Then, there exists* (Γ_1, σ_1) *such that:*

1. $(\Gamma_1, \sigma_1) \in \mathit{AC}\textsc{Match}(\emptyset, \{t \approx^? s\}, id, \mathit{Vars}(t, s), \mathit{Vars}(s))$.
2. (Δ, δ) *is an instance (restricted to the variables of* $\mathit{Vars}(t, s)$*) of* (Γ_1, σ_1) *that does not instantiate the variables of* s.

PROOF:

⟨1⟩1. LET: $V = \mathit{Vars}(t, s)$ and $V' = V \cup \mathit{dom}(\delta) \cup \mathit{Vars}(\mathit{im}(\delta)) \cup \mathit{Vars}(\Delta)$. By Corollary 39, there exists $(\Gamma_1, \sigma_1') \in \text{AC}\textsc{Match}(\emptyset, \{t \approx^? s\}, id, V', \mathit{Vars}(s))$ such that (Δ, δ) is an instance (restricted to the variables of V') of (Γ_1, σ_1') that does not instantiate the variables of s.

⟨1⟩2. The inputs $(\emptyset, \{t \approx^? s\}, id, V, \mathit{Vars}(s))$ and $(\emptyset, \{t \approx^? s\}, id, V', \mathit{Vars}(s))$ are variant inputs fixing V. Hence, by Theorem 38 there is σ_1 such that

1. $(\Gamma_1, \sigma_1) \in \text{AC}\textsc{Match}(\emptyset, \{t \approx^? s\}, id, V, \mathit{Vars}(s))$.
2. $\sigma_1 =_V \sigma_1'$.

⟨1⟩3. If (Δ, δ) is an instance (restricted to the variables of V') of (Γ_1, σ_1') that does not instantiate the variables of s and $\sigma_1 =_V \sigma_1'$ then (Δ, δ) is an instance (restricted to the variables of V) of (Γ_1, σ_1') that does not instantiate the variables of s.

PROOF:

⟨2⟩1. It is immediate that $\Delta \vdash \delta\Gamma_1$.

⟨2⟩2. There exists λ such that $\Delta \vdash \lambda\sigma_1 \approx_V \delta$.

PROOF: Notice that since (Δ, δ) is an instance (restricted to the variables of V') of (Γ_1, σ'_1), there is λ such that $\Delta \vdash \lambda\sigma'_1 \approx_{V'} \delta$. PICK this same λ and notice that since $\sigma_1 =_V \sigma'_1$ and $V \subseteq V'$ the result follow.

⟨2⟩3. It is immediate that $dom(\delta) \cap Vars(s) = \emptyset$.

Corollary 40 (Completeness for AC-Equality Checking). *Suppose that* (Δ, δ) *is a solution to* $(\emptyset, \{t \approx^? s\}, id, \mathbb{X}, Vars(t, s))$. *Then, there exists* (Γ_1, σ_1) *such that:*

1. $(\Gamma_1, \sigma_1) \in ACMATCH(\emptyset, \{t \approx^? s\}, id, Vars(t, s), Vars(t, s))$
2. (Δ, δ) *is an instance (restricted to the variables of* $Vars(t, s)$*) of* (Γ_1, σ_1) *that does not instantiate the variables of* t *or* s.

PROOF SKETCH: Similar to the proof of Corollary 39.

Remark 34. *An interpretation of Corollary 39 is that if* (Δ, δ) *is an AC-matching solution to the initial problem, then* (Δ, δ) *is an AC-matching instance of one of the outputs of ACMATCH. Corollary 40 has a similar interpretation, replacing AC-matching with AC-equality checking.*

5.3 Statistics of the PVS Formalisation

Below we describe the main theories that are part of the nominal AC-matching formalisation. Since the nominal library contain other 3 formalisations, the theories that are only part of the nominal AC-matching formalisation have a `nominal_AC_` prefix to their names (see Table 2.4), which we omit in this Section order not to clutter the presentation.

- `top_nominal_ac_match_alg` - High level description of the nominal AC-matching formalisation.
- `ac_match_alg` - contains the function ACMATCH and the lemmas of soundness and completeness.
- `variant_inputs` - Definition of variant inputs and related lemmas.
- `ac_step` - contains function APPLYACSTEP and lemmas about its properties.
- `inst_step` contains function INSTANTIATESTEP and related lemmas.
- `aux_unification` contains functions SOLVEAC (with lemmas about its properties) and the main functions called by SOLVEAC (with lemmas about its properties).

- `diophantine` - definitions and properties about solving linear Diophantine equations.
- `unification` - definition of solution to a quintuple and lemmas about unification.
- `fresh_subs` - definition and properties of FRESHSUBS?.
- `substitution` - definition and properties about substitutions.
- `equality` - definition and properties about nominal AC-equality checking.
- `freshness` - definition and properties about freshness. Contains function FRESH?.
- `terms` - definition and properties about terms.
- `atoms` - definition and properties of permutations and their actions on atoms.
- `list` - Set of parametric theories that define specific functions for the task of equational reasoning (most of them operating on lists).

Figure 5.1 shows the dependency diagram for the PVS theories that compose the nominal AC-matching formalisation. Besides the nominal AC-matching formalisation, there are other 3 formalisations in the nominal library, which we again represent in the picture as orange ellipses. As shown in Figure 5.1, some of them use theories that are also used by the nominal AC-matching formalisation.

Table 5.2 shows the number of theorems and TCCs proved for each file, along with the theory's approximate size and percentage of the total size. In contrast to Table 2.4, the percentage of the total size shown here is only with respect to the files that are part of the nominal AC-matching formalisation, and not the whole NASALib theory. We group theories `list_aux_equational_reasoning`, `list_aux_equational_reasoning2parameters`, `list_aux_equational_reasoning_nat`, `list_aux_equational_reasoning_more` under the name `list` since the specifics of each one are not relevant to our discussion. Finally, PVS theories `term_properties` and `terms` are the only ones that are actually in the same file, so we group them together under the name `terms` in Table 5.2.

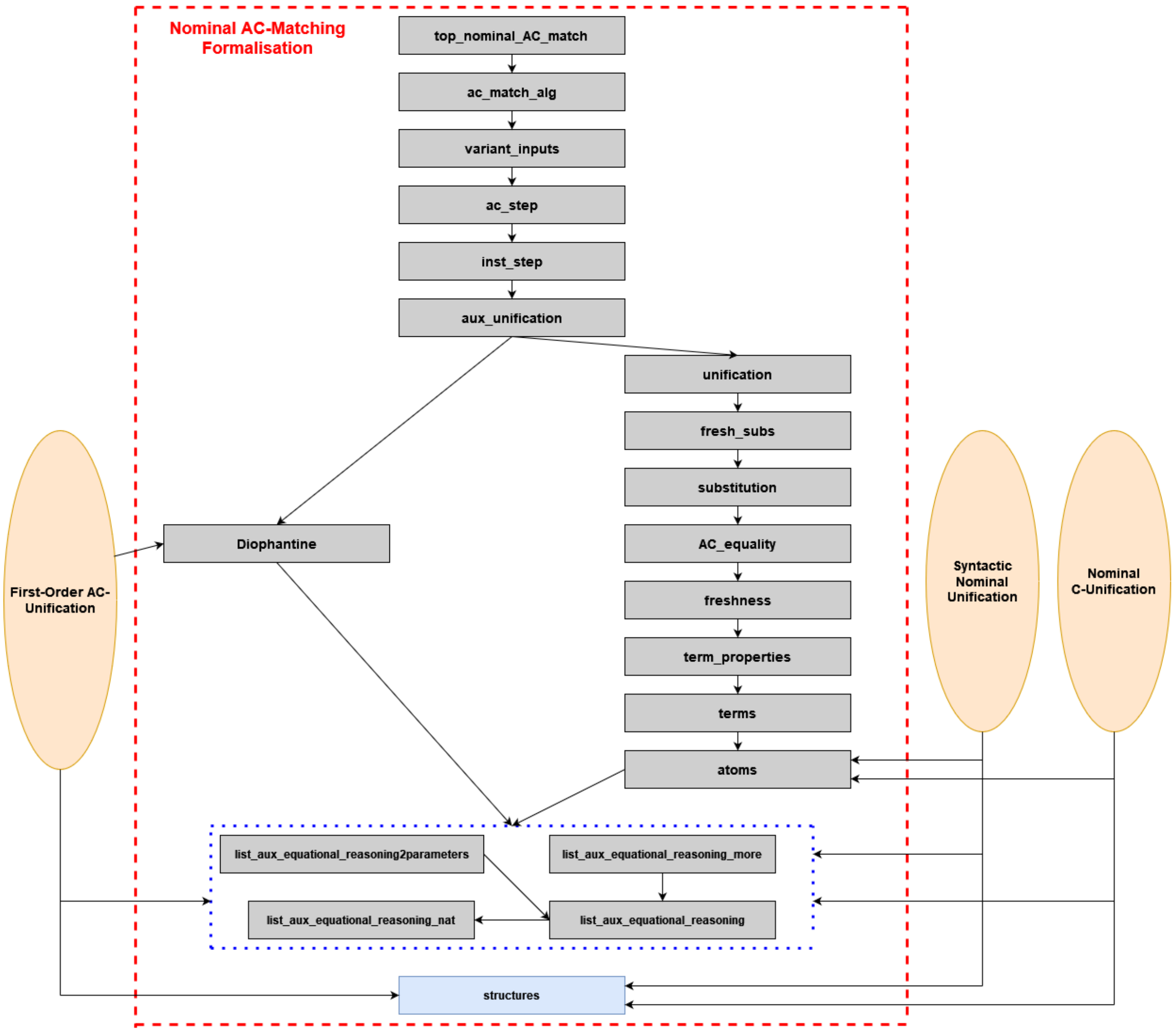

Nominal AC-Matching Formalisation
top_nominal_AC_match
ac_match_alg
variant_inputs
ac_step
inst_step
aux_unification
unification
fresh_subs
substitution
AC_equality
freshness
term_properties
terms
atoms
Diophantine
First-Order AC-Unification
Syntactic Nominal Unification
Nominal C-Unification
list_aux_equational_reasoning2parameters
list_aux_equational_reasoning_more
list_aux_equational_reasoning_nat
list_aux_equational_reasoning
structures

Figure 5.1: PVS formalisation of Nominal AC-Matching.

Table 5.2: Information for Every File in the Nominal AC-Matching Formalisation.

Theory	**Theorems**	**TCCs**	**Size**		
			`.pvs`	`.prf`	%
`ac_match_alg`	22	35	12 kB	2.6 MB	10%
`variant_inputs`	22	5	8 kB	1.4 MB	5%
`ac_step`	48	11	13 kB	1.6 MB	6%
`inst_step`	75	17	21 kB	2.1 MB	8%
`aux_unification`	152	52	49 kB	7.1 MB	27%
`Diophantine`	77	44	24 kB	1.1 MB	4%
`unification`	120	13	28 kB	1.8 MB	7%
`fresh_subs`	38	5	12 kB	0.6 MB	2%
`substitution`	175	36	30 kB	2.6 MB	10%
`equality`	83	20	15 kB	1.7 MB	6%
`freshness`	15	10	5 kB	0.1 MB	< 1 %
`terms`	147	53	30 kB	1.2 MB	5 %
`atoms`	14	3	4 kB	0.1 MB	< 1 %
`list`	263	109	60 kB	2.2 MB	8 %
Total	1251	413	311 kB	26.2 MB	100%

Chapter 6

Towards Nominal AC-Unification

In this Chapter we discuss our work in progress towards obtaining a nominal AC-unification algorithm. Drawing from our experience on formalising nominal AC-matching, we believe that the two main challenges in order to go from nominal AC-matching to nominal AC-unification are correctly solving fixpoint equations (Section 6.1) and proving termination of problems such as $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$ (Section 6.2). A fragment of the discussion in this Chapter regarding termination of nominal AC-unification was briefly mentioned in Ayala-Rincón et al. [8].

Notation 16. *In this Chapter, we may write a moderated variable $\pi \cdot X$ simply as πX.*

6.1 Fixpoint Equations $\pi \cdot X \approx^? X$

In nominal AC-unification, as in nominal C-unification, there may be infinite pairs (Γ, σ) that are solutions to $\pi X \approx^? X$. Hence, as was the case for nominal C-unification, when devising a nominal AC-unification algorithm, we should include fixpoint equations as part of the solution.

A different, but also valuable goal, is to find an efficient enumeration procedure to list every solution (Γ, σ) to a fixpoint equation $\pi X \approx^? X$. Notice that the problem of solving $\pi X \approx^? X$ is equivalent to finding all the terms t such that there exists a context Γ where $\Gamma \vdash \pi \cdot t \approx t$. Therefore, a trivial enumeration procedure is to list every term t and compute (if possible) the minimal context Γ such that $\Gamma \vdash \pi \cdot t \approx t$. The corresponding solution is then $(\Gamma, \{X \mapsto t\})$.

The enumeration procedure sketched in the last paragraph is trivial and has as its main flaw the generation of solutions that are less general. For instance, if we are solving $(a\ b)X \approx^? X$ and g is a syntactic function symbol, the solution $(\emptyset, \{X \mapsto g(c)\})$ should not be included, since it is less general than the solution $(\{a\#X, b\#X\}, id)$.

Notation 17. *We denote a solution* (Δ, δ) *being less general than* (Γ, σ) *by* $(\Gamma, \sigma) \leq (\Delta, \delta)$.

An even more interesting example happens when we are solving $(a\ b)X \approx^? X$ and the AC symbol $+$ is part of our signature. At first glance one would think that the solution $(\emptyset, \{X \mapsto a+b\})$ should be included in the list of generated solutions. Indeed, if we think only about the previous solution, $(\{a\#X, b\#X\}, id)$ we do not have $(\emptyset, \{X \mapsto a+b\})$ less general than $(\{a\#X, b\#X\}, id)$[1] nor the opposite. However, if consider instead the solution $(\emptyset, \{X \mapsto X_1 + (a\ b)X_1\})$ we can see that:

$$(\emptyset, \{X \mapsto X_1 + (a\ b)X_1\}) \leq (\emptyset, \{X \mapsto a+b\}).$$

Notation 18 (Cycle Decomposition Notation)**.** *In this Section, we may represent a permutation* π *using cycle decomposition notation (see [35]). In this notation, we show how* π *acts on atoms by expressing* π *as a product of disjoint cycles. A cycle is a sequence of elements, where each element is mapped to the next one in the sequence by* π *and the last element is mapped to the first. For instance, the cycle* (abc) *means that* $\pi \cdot a = b$, $\pi \cdot b = c$ *and* $\pi \cdot c = a$. *As an example of a permutation* π *written as a product of disjoint cycles consider:*

$$\pi = (abd)(ce).$$

Here $\pi \cdot a = b$, $\pi \cdot b = d$, $\pi \cdot d = a$, $\pi \cdot c = e$ *and* $\pi \cdot e = c$.

6.1.1 Motivation For The Rules of An Enumeration Procedure

We have devised a non-deterministic enumeration procedure given as a set of rules to generate all the solutions of a fixpoint equation $\pi X \approx^? X$ (see Section 6.1.2). Before presenting the rules, we motivate them. Suppose that (Γ, σ) is a solution, let's denote σX as t and think inductively of the form of t.

t **is an atom** a_i**.** In this base case, we necessarily have $a_i \notin dom(\pi)$, and the solution is of the form:

$$(\Gamma, \sigma) = (\emptyset, X \mapsto a_i).$$

Notice however, that this solution is less general than $(dom(\pi)\#X, id)$, which suggest that we may drop the base case where t is an atom.

t **is a moderated variable.** In this base case, the solution is of the form

$$(\Gamma, \sigma) = (dom(\pi)\#X, id).$$

[1] We cannot argue that $(\emptyset, \{X \mapsto a+b\})$ is less general than $(\{a\#X, b\#X\}, id)$ by considering the instantiation $X \mapsto a+b$ because we do not have $a\#a+b$ or $b\#a+b$.

t **is the unit, i.e.** $t \equiv \langle\rangle$**.** In this base case the solution is of the form:

$$(\Gamma, \sigma) = (\emptyset, X \mapsto \langle\rangle).$$

Notice that this solution is less general than $(dom(\pi)\#X, id)$.

t **is a syntactic function application.** Let g be a syntactic function symbol with arity m. Suppose that our solution (Γ, σ) is such that $\sigma X = t = g(t_1, \ldots, t_m)$. Since it is a solution of $\pi X \approx^? X$ we have:

$$\Gamma \vdash \pi \cdot g(t_1, \ldots, t_m) \approx^? g(t_1, \ldots, t_m)$$

which means:

$$\begin{aligned} &\Gamma \vdash \pi \cdot t_1 \approx t_1 \\ &\Gamma \vdash \pi \cdot t_2 \approx t_2 \\ &\;\;\vdots \\ &\Gamma \vdash \pi \cdot t_m \approx t_m \end{aligned} \tag{6.1}$$

Looking at each individual equation of the set of Equations 6.1, it is as if we computed $\sigma' = \{X \mapsto g(X_1, \ldots, X_m)\}$ and then inductively proceeded to solve $\pi X_i \approx^? X_i$. Once we find every (Γ_i, σ_i) that solves $\pi X_i \approx^? X_i$ we generate the solution to our original problem as:

$$(\Gamma, \sigma) = (\bigcup_{1 \leq i \leq n} \Gamma_i, \sigma_n \ldots \sigma_1 \sigma').$$

t **is an abstraction** $[a]t_1$**, where** $a \notin$ ***dom***(π)**.** Our solution (Γ, σ) is such that: $\Gamma \vdash \pi \cdot [a]t_1 \approx [a]t_1$ which means $\Gamma \vdash \pi \cdot t_1 \approx t_1$. Let $\sigma' = \{X \mapsto [a]X_1\}$. Our solution (Γ, σ) could be written in terms of σ' and some (Γ_1, σ_1) that solves $\pi X_1 \approx^? X_1$:

$$(\Gamma, \sigma) = (\Gamma_1, \sigma_1 \sigma').$$

t **is an abstraction** $[a]t_1$**, where** $a \in$ ***dom***(π)**.** Let $b = \pi \cdot a$. Our solution (Γ, σ) is such that:

$$\Gamma \vdash [b]\ \pi \cdot t_1 \approx [a]t_1,$$

which means that

$$\begin{aligned} &\Gamma \vdash a\#\pi \cdot t_1 \\ &\Gamma \vdash t_1 \approx ((a\ b) \circ \pi) \cdot t_1 \end{aligned}$$

Let $\sigma' = X \mapsto [a]X_1$ and $\pi' = ((a\ b) \circ \pi)$. Suppose that we generate every solution (Γ_1, σ_1) to $\pi' X_1 \approx^? X_1$ and for each one we computed the minimal context Γ_2 such that $\Gamma_2 \vdash a \# \pi{\cdot}\sigma_1 X_1$, dropping the cases where there is no context under which $\Gamma_2 \vdash a \# \pi{\cdot}\sigma_1 X_1$ holds. Our solution (Γ, σ) could be written in terms of a particular choice of Γ_1, Γ_2 and σ_1 as:

$$(\Gamma, \sigma) = (\Gamma_1 \cup \Gamma_2, \sigma_1\sigma').$$

***t* is an associative commutative function application.** Let f be an AC function symbol and m be the number of arguments of the flattened version of $\sigma X = t = f(t_1, \ldots, t_m)$. Since $\Gamma \vdash \pi \cdot t \approx t$, then

$$\Gamma \vdash f(\pi \cdot t_1, \ldots, \pi \cdot t_m) \approx f(t_1, \ldots, t_m)$$

and hence there exists a permutation $\psi : \{1, \ldots, m\} \to \{1, \ldots, m\}$ such that

$$\Gamma \vdash t_i \approx \pi \cdot t_{\psi(i)}.$$

We'll represent ψ using cycle decomposition notation. Let

$$\psi = (x_1 x_2 \ldots x_{m_1})(x_{m_1+1} x_{m_1+2} \ldots x_{m_2}) \ldots (x_{m_{k-1}+1} x_{m_{k-1}+2} \ldots x_{m_k})$$

and let $l_1, \ldots, l_k$ be the length of the cycles. Notice that $x_1, x_2, \ldots, x_{m_k}$ are numbers from 1 to m. Let's analyse the first cycle and suppose, without loss of generality, that $x_1 = 1$. Then,

$$\begin{aligned} &\Gamma \vdash t_1 \approx \pi \cdot t_{x_2} \\ &\Gamma \vdash t_{x_2} \approx \pi \cdot t_{x_3} \\ &\ \vdots \\ &\Gamma \vdash t_{x_{m_1}} \approx \pi \cdot t_{x_1} = \pi t_1 \end{aligned} \tag{6.2}$$

Equations 6.2 show that once we know $t_1 = t_{x_1}$ it is possible to find $t_{x_2}, \ldots, t_{x_{m_1}}$. Moreover, if we combine all these equations we get:

$$\Gamma \vdash t_1 \approx \pi^{l_1} \cdot t_1,$$

where l_1 is the length of cycle $(x_1x_2\dots x_{m_1})$. This last equation suggests [2] that we must solve recursively $X_1 \approx^? \pi^{l_1}X_1$. Repeating this reasoning for every cycle of

$$\psi = (x_1x_2\dots x_{m_1})(x_{m_1+1}x_{m_1+2}\dots x_{m_2})\dots(x_{m_{k-1}+1}x_{m_{k-1}+2}\dots x_{m_k})$$

and remembering that $l_1,\dots,l_k$ are the length of the cycles it seems that we must solve $\pi^{l_1}X_1 \approx^? X_1$, ..., $\pi^{l_k}X_k \approx^? X_k$ and then assemble the solutions together via the substitution

$$\sigma' = \{X \mapsto f(\underbrace{X_1, \pi^1X_1\dots,\pi^{l_1-1}X_1}, \dots, \underbrace{X_k, \pi^1X_k, \dots, \pi^{l_k-1}X_k})\}.$$

6.1.2 A Non-Deterministic Enumeration Procedure to Solve Fixpoint Equations

The enumeration procedure is given as a set of non-deterministic rules that operate on triples of the form (Γ, σ, FP), where Γ is the context already computed, σ represents the instantiations we have done so far and FP is the set of fixpoint equations and of freshness problems we still have to solve. Hence, the initial triple to solve $\pi \cdot X \approx^? X$ is simply $(\emptyset, id, \{\pi X \approx^? X\})$.

To handle freshness problems such as $a\#_?t$ we may use what is in the standard literature of nominal unification (see [3, 75]). To handle the set of fixpoint equations we use the rules:

- `(Var)`
- `(Func)`
- `(Abs a)` and `(Abs b)`
- `(AC Func)`

The `(Var)` Rule

$$(\Gamma, \sigma, \{\pi X \approx^? X\} \cup FP) \xRightarrow{(Var)} (\Gamma \cup dom(\pi)\#X, \sigma, FP).$$

The `(Func)` Rule

Let g be an arbitrary syntactic function symbol of arity m and let $\sigma' = \{X \mapsto g(X_1,\dots,X_m)\}$, where $X_1,\dots,X_m$ are new variables. The `(Func)` rule is:

$$(\Gamma, \sigma, \{\pi X \approx^? X\} \cup FP) \xRightarrow{(Func)} (\Gamma, \sigma'\sigma, \{\pi X_1 \approx^? X_1, \dots, \pi X_m \approx^? X_m\} \cup \sigma' FP).$$

[2]recall that solving $\pi X \approx^? X$ is equivalent to finding all the terms t such that there is a context Γ where $\Gamma \vdash \pi t \approx t$.

The `(Abs a)` Rule

Suppose $a \notin dom(\pi)$ and let $\sigma' = \{X \mapsto [a]X_1\}$, where X_1 is a new variable. The `(Abs a)` rule is:

$$(\Gamma, \sigma, \{\pi X \approx^? X\} \cup FP) \xrightarrow{(Abs\ a)} (\Gamma, \sigma'\sigma, \{\pi X_1 \approx^? X_1\} \cup \sigma' FP).$$

The `(Abs b)` Rule

Suppose $a \in dom(\pi)$. Let $\pi \cdot a = b$, $\pi' = (a\ b)\ \pi$, $\sigma' = \{X \mapsto [a]X_1\}$, where X_1 is a new variable. The `(Abs b)` rule:

$$(\Gamma, \sigma, \{\pi X \approx^? X\} \cup FP) \xrightarrow{(Abs\ b)} (\Gamma, \sigma'\sigma, \{\pi' X_1 \approx^? X_1\} \cup \sigma' FP \cup \{a\#_? \pi X_1\}).$$

The `(AC Func)` Rule

Let m be an arbitrary number and let ψ be an arbitrary permutation from $\{1, \dots, m\}$ to $\{1, \dots, m\}$, such that:

$$\psi = (x_1 x_2 \dots x_{m_1})(x_{m_1+1} x_{m_1+2} \dots x_{m_2}) \dots (x_{m_{k-1}+1} x_{m_{k-1}+2} \dots x_{m_k}),$$

where $l_1, \dots, l_k$ are the length of the cycles. Let

$$\sigma' = \{X \mapsto f(\underbrace{X_1, \pi^1 X_1 \dots, \pi^{l_1-1} X_1}, \dots, \underbrace{X_k, \pi^1 X_k, \dots, \pi^{l_k-1} X_k})\}$$

The `(AC Func)` rule is:

$$(\Gamma, \sigma, \{\pi X \approx^? X\} \cup FP) \xrightarrow{(AC)} (\Gamma, \sigma'\sigma, \{\pi^{l_1} X_1 \approx^? X_1, ..., \pi^{l_k} X_k \approx^? X_k\} \cup \sigma' FP)$$

6.1.3 Examples of The Enumeration Procedure

In this section we give examples of how our rule-based enumeration procedure could be used to find certain solutions to some fixpoint equations. In these examples, we use syntactic sugar and denote atoms $a_1, a_2, \dots$ by numbers $1, 2, \dots$ We also assume that $+$ and $*$ are AC function symbols and represent permutations using cycle decomposition notation.

Example 30. *Let* $\pi = (123456)$ *and consider* $\pi X \approx^? X$. *Can our enumeration procedure generate the solution*

$$(\emptyset, \{X \mapsto *(+(1,4), +(2,5), +(3,6))\})?$$

A more general solution than the one presented would be found by the following application of the rules:

1. *First, we would apply* **(AC Func)***, considering $m = 3$ and the permutation $\psi = (123)$. Let $\sigma_1 = \{X \mapsto *(X_1, \pi^1 X_1, \pi^2 X_1)\}$ and notice that the triples before and after* **(AC Func)** *are:*

$$(\emptyset, id, \{\pi X \approx^? X\}) \xrightarrow{(AC)} (\emptyset, \sigma_1, \{\pi^3 X_1 \approx^? X_1\}).$$

2. *Then, we would apply* **(AC Func)***, considering $m = 2$ and the permutation $\psi = (12)$. Let $\sigma_2' = \{X_1 \mapsto +(X_2, \pi^3 X_2)\}$ and notice that the triples before and after* **(AC Func)** *are:*

$$(\emptyset, \sigma_1, \{\pi^3 X_1 \approx^? X_1\}) \xrightarrow{(AC)} (\emptyset, \sigma_2\sigma_1, \{(\pi^3)^2 X_2 \approx^? X_2\}).$$

3. *We would eliminate equation $\{(\pi^3)^2 X \approx^? X\}$ since $(\pi^3)^2 = \pi^6 = Id$ and return* [3]*:*

$$(\emptyset, \sigma_2\sigma_1) = (\emptyset, \{X \mapsto *(+(X_2, \pi^3 X_2), +(\pi X_2, \pi^4 X_2), +(\pi^2 X_2, \pi^5 X_2))\}).$$

Notice that the particular solution:

$$(\emptyset, \{X \mapsto *(+(1,4), +(2,5), +(3,6))\})$$

can be obtained from:

$$(\emptyset, \{X \mapsto *(+(X_2, \pi^3 X_2), +(\pi X_2, \pi^4 X_2), +(\pi^2 X_2, \pi^5 X_2))\})$$

by instantiating X_2 to 1.

Example 31. *Consider again $\pi = (123456)$ and the fixpoint equation $\pi X \approx^? X$. Can our enumeration procedure generate the solution:*

$$(\emptyset, \{X \mapsto *(+(1,3,5), +(2,4,6))\})?$$

A more general solution than the one presented would be found by the following applications of the rules:

1. *First, we would apply* **(AC Func)***, considering $m = 2$ and the permutation $\psi = (12)$. Let $\sigma_1 = X \mapsto *(X_1, \pi X_1)$ and notice that the triples before and after* **(AC Func)**

[3]We omit the instantiations of X_1 and X_2, since the only variable in our original fixpoint equation $\pi X \approx^? X$ is X.

are:

$$(\emptyset, id, \{\pi X \approx^? X\}) \xRightarrow{(AC)} (\emptyset, \sigma_1, \{\pi^2 X_1 \approx^? X_1\}).$$

2. *Then, we would apply of* **(AC Func)** *we may consider $m = 3$ and the permutation $\psi = (123)$. Let $\sigma_2 = X_1 \mapsto +(X_2, \pi^2 X_2, \pi^4 X_2)$ and notice that the triples before and after* **(AC Func)** *are:*

$$(\emptyset, \sigma_1, \{\pi^2 X_1 \approx^? X_1\}) \xRightarrow{(AC)} (\emptyset, \sigma_2\sigma_1, \{(\pi^2)^3 X_2 \approx^? X_2\}).$$

3. *We would eliminate equation $\{(\pi^3)^2 X_2 \approx^? X_2\}$ since $(\pi^3)^2 = \pi^6 = Id$ and return:*

$$(\emptyset, \sigma_2\sigma_1) = (\emptyset, \{X \mapsto *(+(X_2, \pi^2 X_2, \pi^4 X_2), +(\pi X_2, \pi^3 X_2, \pi^5 X_2))\}).$$

Notice that the particular solution:

$$(\emptyset, \{X \mapsto *(+(1,3,5), +(2,4,6))\})$$

can be obtained from

$$(\emptyset, \{X \mapsto *(+(X_2, \pi^2 X_2, \pi^4 X_2), +(\pi X_2, \pi^3 X_2, \pi^5 X_2))\})$$

by instantiating X_2 to 1.

Example 32 (Adapting the Previous Example)**.** *What would happen if we changed the previous example and considered $\pi = (123456)(7891011)$? The first two steps would be exactly the same. In the third, we would have the equation $\pi^6 X_2 \approx^? X_2$, where $\pi^6 = (7891011)$ and we would then apply rule* **(Var)***:*

$$(\emptyset, \sigma_2\sigma_1, \{\pi^6 X_2 \approx^? X_2\}) \xRightarrow{(Var)} (\{7, 8, 9, 10, 11\}\#X_2, \sigma_2\sigma_1, \emptyset)$$

and return

$$(\{7, 8, 9, 10, 11\}\#X_2, \{X \mapsto *(+(X_2, \pi^2 X_2, \pi^4 X_2), +(\pi X_2, \pi^3 X_2, \pi^5 X_2))\}).$$

Again, notice that the particular solution:

$$(\emptyset, \{X \mapsto *(+(1,3,5), +(2,4,6))\})$$

can be obtained from

$$(\{7,8,9,10,11\}\#X_2, \{X \mapsto *(+(X_2, \pi^2 X_2, \pi^4 X_2), +(\pi X_2, \pi^3 X_2, \pi^5 X_2))\})$$

by instantiating X_2 *to* 1.

Indeed, recall that $1, 2, \ldots, 11$ *are syntax sugar for atoms* $a_1, \ldots, a_{11}$ *and for our instantiation of* X_2 *to* a_1 *to be valid we must guarantee that* $\emptyset \vdash a_i \# a_1$, *with* $i = 7, 8, 9, 10, 11$. *It is trivial that this holds.*

Example 33. *Consider* $\pi = (123456)(78)$ *and the fixpoint equation* $\pi X \approx^? X$. *Can our enumeration procedure generate the solution:*

$$(\emptyset, \{X \mapsto *(+(1,3,5,7), +(2,4,6,8))\})?$$

A more general solution than the one presented would be found by the following applications of the rules:

1. *First, we would apply* **(AC Func)**, *considering* $m = 2$ *and the permutation* $\psi = (12)$. *Let* $\sigma_1 = \{X \mapsto *(X_1, \pi X_1)\}$ *and notice that the triples before and after* **(AC Func)** *are:*
$$(\emptyset, id, \{\pi X \approx^? X\}) \xrightarrow{(AC)} (\emptyset, \sigma_1, \{\pi^2 X_1 \approx^? X_1\}).$$

2. *Then, we would apply* **(AC Func)**, *considering* $m = 4$ *and the permutation* $\psi = (123)(4)$. *Let* $\sigma_2' = \{X_1 \mapsto +(X_2, \pi^2 X_2, \pi^4 X_2, X_3)\}$ *and notice that the triples before and after* **(AC Func)** *are:*
$$(\emptyset, \sigma_1, \{\pi^2 X_1 \approx^? X_1\}) \xrightarrow{(AC)} (\emptyset, \sigma_1, \{(\pi^2)^3 X_2 \approx^? X_2, (\pi^2)^1 X_3 \approx^? X_3\}).$$

3. *We would eliminate equation* $\{(\pi^3)^2 X_2 \approx^? X_2\}$ *since* $(\pi^3)^2 = \pi^6 = Id$. *Finally, we would apply rule* **(Var)** *to handle* $\pi^2 X_3 \approx^? X_3$ *returning*
$$(\{1,2,3,4,5,6\}\#X_3, \{X \mapsto *(+(X_2, \pi^2 X_2, \pi^4 X_2, X_3), +(\pi X_2, \pi^3 X_2, \pi^5 X_2, \pi X_3))\}$$
Notice that the particular solution:
$$(\emptyset, \{X \mapsto *(+(1,3,5,7), +(2,4,6,8))\})?$$
can be obtained from
$$(\{1,2,3,4,5,6\}\#X_3, \{X \mapsto *(+(X_2, \pi^2 X_2, \pi^4 X_2, X_3), +(\pi X_2, \pi^3 X_2, \pi^5 X_2, \pi X_3))\}$$

by instantiating X_2 to 1 and X_3 to 7.

6.1.4 A Comparison With Fixpoint Equations in Nominal C-Unification

In Ayala-Rincón et al. [5], it is shown how we can solve fixpoint equations in nominal C-unification. The approach here described could also be adapted to nominal C-unification by incorporating two adaptations. First, we would need a rule for commutative functions. For this case, we could adapt and simplify rule `(AC Func)`, since commutative functions do not have a flattened form. This rule (see Section 6.1.2) would be modified to consider only permutations $\psi : \{1, 2\} \to \{1, 2\}$. Second, the rules described above only work when there is one fixpoint equation for a given variable X. What happens if we have to solve both $\pi_1 X \approx^? X$ and $\pi_2 X \approx^? X$? This is sketched in the next section.

In comparison with [5] our approach has the advantage of not generating some less general solutions that are generated by the technique in [5]. In contrast with [5] we do not include solutions (Γ, σ) that instantiate variables to terms with atoms, which are less general. Consider for instance the equational constraint $(a\ b)X \approx^? X$ and let $+$ be a commutative symbol. The technique in [5] generates as solution $(\emptyset, \{X \mapsto a + b\})$ and our non-deterministic procedure does not: it generate instead $(\emptyset, \{X \mapsto X_1 + (a\ b)X_1\})$, which is more general.

6.1.5 Handling More Than One Fixpoint Equation With The Same Variable

We believe we could adapt the approach taken by Ayala-Rincón et al. in [5] to handle cases like $\{\pi_1 X \approx^? X,\ \pi_2 X \approx^? X\}$ where we have two or more fixpoint equations with the same variable. In this approach, given equational constraints $\{\pi_1 X \approx^? X, \dots, \pi_n X \approx^? X\}$ we:

1. Compute solutions $(\Gamma_i, \{X \mapsto t_i\})$ to $\pi_i X \approx^? X$.
2. Find pairs (Γ', δ) that unify $\{t_1 \approx^? t_2, t_1 \approx^? t_3, \dots, t_1 \approx^? t_n\}$.
3. Find the minimal context Γ such that $\Gamma \vdash \delta\Gamma_i$.
4. Return solutions of the form $(\Gamma \cup \Gamma',\ \{X \mapsto \delta t_1\})$.

6.2 Termination of Nominal AC-Unification

6.2.1 The Loop in $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$

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

As an example, suppose that we are working under an empty context (i.e. $\Gamma = \emptyset$) and want to unify the equational constraint $f(X, W) \approx^? f(\pi X, \pi Y)$. The linear Diophantine equation associated with this problem $U_1 + U_2 = V_1 + V_2$, where variable U_1 is associated with argument X, variable U_2 is associated with argument W, variable V_1 is associated with argument πX and variable V_2 is associated with argument πY. A basis of solutions to this linear Diophantine equation is shown in Table 6.1.

Table 6.1: Solutions for Equation $U_1 + U_2 = V_1 + V_2$

$\mathbf{U_1}$	$\mathbf{U_2}$	$\mathbf{V_1}$	$\mathbf{V_2}$	$\mathbf{U_1 + U_2}$	$\mathbf{V_1 + V_2}$	**New Variables**
0	1	0	1	1	1	Z_1
0	1	1	0	1	1	W_1
1	0	0	1	1	1	Y_1
1	0	1	0	1	1	X_1

We choose the names of the new variables to be Z_1, W_1, Y_1 and X_1 deliberately to make the loop in nominal AC-unification clearer. Finally, we will branch into new equational constraints, using Table 6.1 to construct them. The algorithm bifurcates into 7 branches, shown below along with their corresponding equational constraints:

$$\mathit{branch}_1 = \{X \approx^? X_1, W \approx^? Z_1, \pi X \approx^? X_1, \pi Y \approx^? Z_1\}.$$
$$\mathit{branch}_2 = \{X \approx^? Y_1, W \approx^? W_1, \pi X \approx^? W_1, \pi Y \approx^? Y_1\}.$$
$$\mathit{branch}_3 = \{X \approx^? Y_1 + X_1, W \approx^? W_1, \pi X \approx^? W_1 + X_1, \pi Y \approx^? Y_1\}.$$
$$\mathit{branch}_4 = \{X \approx^? Y_1 + X_1, W \approx^? Z_1, \pi X \approx^? X_1, \pi Y \approx^? Z_1 + Y_1\}.$$
$$\mathit{branch}_5 = \{X \approx^? X_1, W \approx^? Z_1 + W_1, \pi X \approx^? W_1 + X_1, \pi Y \approx^? Z_1\}.$$
$$\mathit{branch}_6 = \{X \approx^? Y_1, W \approx^? Z_1 + W_1, \pi X \approx^? W_1, \pi Y \approx^? Z_1 + Y_1\}.$$
$$\mathit{branch}_7 = \{X \approx^? Y_1 + X_1, W \approx^? Z_1 + W_1, \pi X \approx^? W_1 + X_1, \pi Y \approx^? Z_1 + Y_1\}.$$

The next step is to instantiate moderated variables. We denote branch i by B_i, the substitution computed in this branch by σ_{Bi} and show the result after performing the instantiations. For brevity, when presenting σ_{Bi} we omit the instantiation of variables X_1, W_1, Y_1, Z_1 since they were not in the initial problem.

$$\begin{aligned}
&B1 - \{\pi X \approx^? X\}, \sigma_{B1} = \{W \mapsto \pi Y\}.\\
&B2 - \sigma_{B2} = \{W \mapsto \pi^2 Y, X \mapsto \pi Y\}.\\
&B3 - \{f(\pi^2 Y, \pi X_1) \approx^? f(W, X_1)\}, \sigma_{B3} = \{X \mapsto f(\pi Y, X_1)\}.\\
&B4 - \mathit{No\ solution.}\\
&B5 - \mathit{No\ solution.}\\
&B6 - \sigma_{B6} = \{W \mapsto f(Z_1, \pi X), Y \mapsto f(\pi^{-1} Z_1, \pi^{-1} X)\}.\\
&B7 - \{f(\pi Y_1, \pi X_1) \approx^? f(W_1, X_1)\},\\
&\qquad\qquad \sigma_{B7} = \{X \mapsto f(Y_1, X_1),\ W \mapsto f(Z_1, W_1), Y \mapsto f(\pi^{-1} Z_1, \pi^{-1} Y_1)\}.
\end{aligned} \tag{6.3}$$

Branches 3 and 7 are a renaming of the original problem

$$f(X, W) \approx^? f(\pi X, \pi Y).$$

Regarding Branch 3, notice that if we rewrite $\sigma_{B3} = \{X \mapsto f(\pi Y, X_1)\}$ as $\sigma'_{B3} = \{Y \mapsto \pi^{-1} Y_1, X \mapsto f(\pi Y, X_1),\}$, then the equational constraint of the mentioned branch is simply:

$$f(X_1, W_1) \approx^? f(\pi X_1, \pi Y_1).$$

Regarding Branch 7, it's even simpler to see the renaming, as the equational constraint is:

$$f(X_1, W_1) \approx^? f(\pi X_1, \pi Y_1).$$

This problem does not arise in first-order AC-unification because, in the corresponding first-order problem, we would not have two different permutations (id and π in this case) suspended on the same variable (X in this case). Instead, we would have the same variable X as an argument to both terms and eliminate it. Finally, this problem also does not arise in nominal AC-matching because X would be a protected variable, as it is in the right-hand side of the equational constraint. Hence, we would not compute the substitutions of branches 3 and 7, we would instead discard these branches.

6.2.2 Solving the Loop in $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$

Let k be the order of the permutation π. We will show that it is enough to branch into branches 3 or 7 at most $2k$ times. Since those branches are the only ones that cause us to loop, we can compute a finite set of triples (Γ, σ, FP) that is a complete set of solutions to the equational constraint $f(X, W) \approx^? f(\pi X, \pi Y)$.

If we denote X, W and Y by X_0, W_0 and Y_0 then, after i iterations of taking branches 3 or 7, the problem will be:

$$P_i = \{f(X_i, W_i) \approx^? f(\pi \cdot X_i, \pi \cdot Y_i)\},$$

and the substitutions of branches 3 and 7 after the $(i+1)$-th iteration will be:

$$\begin{aligned}
&\sigma_{B3}^{(i+1)} = \{X_i \mapsto f(Y_{i+1}, X_{i+1}), W_i \mapsto W_{i+1}, Y_i \mapsto \pi^{-1} Y_{i+1}\} \text{ and} \\
&\sigma_{B7}^{(i+1)} = \{X_i \mapsto f(Y_{i+1}, X_{i+1}), W_i \mapsto f(Z_{i+1}, W_{i+1}), Y_i \mapsto f(\pi^{-1} Z_{i+1}, \pi^{-1} Y_{i+1})\}.
\end{aligned}$$

The equivalent of Equations 6.3 after the $(i+1)$-th iteration is shown below:

$$\begin{aligned}
&B1 - \{\pi X_i \approx^? X_i\}, \sigma_{B1}^{(i+1)} = \{W_i \mapsto \pi Y_i\} \\
&B2 - \sigma_{B2}^{(i+1)} = \{W_i \mapsto \pi^2 Y_i, X_i \mapsto \pi Y_i\} \\
&B3 - \{f(\pi Y_{i+1}, \pi X_{i+1}) \approx^? f(W_{i+1}, X_{i+1})\}, \\
&\qquad \sigma_{B3}^{(i+1)} = \{X_i \mapsto f(Y_{i+1}, X_{i+1}), W_i \mapsto W_{i+1}, Y_i \mapsto \pi^{-1} Y_{i+1}\} \\
&B4 - \mathit{No\ solution} \\
&B5 - \mathit{No\ solution} \\
&B6 - \sigma_{B6}^{(i+1)} = \{W_i \mapsto f(Z_{i+1}, \pi X_i), Y \mapsto f(\pi^{-1} Z_{i+1}, \pi^{-1} X_i)\} \\
&B7 - \{f(\pi Y_{i+1}, \pi X_{i+1}) \approx^? f(W_{i+1}, X_{i+1})\}, \\
&\qquad \sigma_{B7}^{(i+1)} = \{X \mapsto f(Y_1, X_1),\ W \mapsto f(Z_1, W_1), Y \mapsto f(\pi^{-1} Z_1, \pi^{-1} Y_1)\}
\end{aligned} \tag{6.4}$$

The triples (Γ, σ, FP) are such that $\Gamma = \emptyset$ (see Equations 6.3); σ is of the form $\sigma_{By}^{(n+1)} \sigma_{Bx_n}^{(n)} \ldots \sigma_{Bx_1}^{(1)}$, where x_i is either 3 or 7 and y is different than 3 or 7; and FP is a set of fixpoint equations. We may have $n = 0$, i.e., σ is of the form $\sigma_{By}^{(1)}$ where y is different than 3 or 7. Notice that we have $n \leq 2k$, i.e, we take branches 3 or 7 at most $2k$ times.

Remark 35. *From now on, when it is clear the iteration we are referring, we may omit the superscript. For instance, we may omit the superscript (i) in $\sigma_{Bx_i}^{(i)}$, denoting this substitution simply as σ_{Bx_i}.*

The following notation will be used to prove that it is enough to take branches 3 and 7 at most $2k$ times (Theorem 41).

Notation 19. $[\pi_1, \ldots, \pi_n]X$. *From now on, we may use* $[\pi_1, \ldots, \pi_n]X$ *as a syntactic sugar for* $\pi_1 X, \ldots, \pi_n X$ *when specifying terms. For instance, we write* $f([\pi_1, \ldots, \pi_n]Y, Z)$ *to denote the term* $f(\pi_1 Y, \ldots, \pi_n Y, Z)$.

Notation 20 (Optional Argument Notation)**.** *We may use* Optional Argument Notation *to denote terms that are of a certain form. In this notation, the superscript* ^{o} *indicates that certain part of a term is optional. For instance,*

- *if a term is of the form* $f(Z^o, X, Y)$ *then the term is either* $f(X, Y)$ *or* $f(Z, X, Y)$.
- *if a term is of the form* $f(Z^o, X^o, Y)$ *then the term is either* Y, $f(Z, Y)$, $f(X, Y)$ *or* $f(Z, X, Y)$.

Finally, if a term is of the form $f([Id, \pi, \pi^2]Z^o, X, Y)$ *then either* all *the arguments* $Id\ Z, \pi Z, \pi^2 Z$ *are in the term or* none *of them are. Hence, the term is either* $f(X, Y)$ *or* $f([Id, \pi, \pi^2]Z, X, Y)$.

Theorem 41. *Let* (Δ, δ) *be a solution to* $f(X_0, W_0) \approx^? f(\pi X_0, \pi Y_0)$ *and* k *be the order of* π. *There exists a triple* (Γ, σ, FP) *such that:*

- σ *takes branches 3 or 7 at most* $2k$ *times.*
- $\sigma \leq \delta$.

PROOF SKETCH:

$\langle 1 \rangle 1$. If (Δ, δ) is an arbitrary solution to $f(X_0, W_0) \approx^? f(\pi X_0, \pi Y_0)$ then δ can be written as $\delta' \sigma_{By}^{(m+1)} \sigma_{Bx_m}^{(m)} \ldots \sigma_{Bx_1}^{(1)}$, where x_i is either 3 or 7 and y is different than 3 or 7. Notice that m may be arbitrarily large. The case where $m < 2k$ is trivial.

PROOF SKETCH: δ being written as $\delta' \sigma_{By}^{(m+1)} \sigma_{Bx_m}^{(m)} \ldots \sigma_{Bx_1}^{(1)}$ rests on the assumption that our branch generation step is complete. Although this has not been formally proved for nominal AC-unification, it has been done for nominal AC-matching and the case for nominal AC-unification should be identical.

$\langle 1 \rangle 2$. The form of the terms obtained when we apply $\sigma_{Bx_n} \ldots \sigma_{Bx_1}$ to X_0, Y_0 and W_0 are:

$$\begin{aligned}
X_0 &\mapsto f(\pi^{-1} Z_2^o, \ldots, [\pi^{-(n-1)}, \ldots, \pi^{-1}]Z_n^o, [\pi^{-(n-1)}, \ldots, Id]Y_n, X_n) \\
W_0 &\mapsto f(Z_1^o, Z_2^o, \ldots, Z_n^o, W_n) \\
Y_0 &\mapsto f(\pi^{-1} Z_1^o, \pi^{-2} Z_2^o, \ldots, \pi^{-n} Z_n^o, \pi^{-n} Y_n)
\end{aligned}$$

A similar formula holds for $\sigma_{Bx_m} \ldots \sigma_{Bx_1}$, simply replacing n by m.

The notation Z_i^o means that the term Z_i may be present or not, according to the branch σ_{Bx_i} that we took (3 or 7). If σ_{Bx_i} is branch 3, then Z_i is not present (see Equations 6.4). Alternatively, if σ_{Bx_i} is branch 7, then it is.

PROOF: Indeed notice that the form of the terms obtained when we apply $\sigma_{Bx_n} \dots \sigma_{Bx_1}$ to X_0 is:

$$\begin{aligned} X_0 &\mapsto f(Y_1, X_1) \\ &\mapsto f(\pi^{-1}Z_2^o, [\pi^{-1}, Id]Y_2, X_2) \\ &\mapsto f(\pi^{-1}Z_2^o, [\pi^{-2}, \pi^{-1}]Z_3^o, [\pi^{-2}, \pi^{-1}, Id]Y_3, X_3) \\ &\vdots \\ &\mapsto f(\pi^{-1}Z_2^o, \dots, [\pi^{-(n-1)}, \dots, \pi^{-1}]Z_n^o, [\pi^{-(n-1)}, \dots, Id]Y_n, X_n). \end{aligned}$$

The form of the terms obtained when we apply $\sigma_{Bx_n} \dots \sigma_{Bx_1}$ to W_0 is:

$$\begin{aligned} W_0 &\mapsto f(Z_1^o, W_1) \\ &\mapsto f(Z_1^o, Z_2^o, W_2) \\ &\vdots \\ &\mapsto f(Z_1^o, Z_2^o, \dots, Z_n^o, W_n). \end{aligned}$$

The form of the terms obtained when we apply $\sigma_{Bx_n} \dots \sigma_{Bx_1}$ to Y_0 is:

$$\begin{aligned} Y_0 &\mapsto f(\pi^{-1}Z_1^o, \pi^{-1}Y_1) \\ &\mapsto f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \pi^{-2}Y_2) \\ &\vdots \\ &\mapsto f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \dots, \pi^{-n}Z_n^o, \pi^{-n}Y_n). \end{aligned}$$

⟨1⟩3. The proof divides in three cases, according to whether By is $B1$, $B2$ or $B6$. Some preliminar steps are common to all the cases. LET: n be such that $k \leq n < 2k$ (the exact value of n changes according to our case). LET: σ^* be the substitution defined as $\sigma_n \dots \sigma_1$, where

$$\sigma_i = \begin{cases} \sigma_{B7}^{(i)}, & \text{if } i \leq k \text{ and } \exists j : j \equiv i \ (mod\ k) \text{ and } Z_j \in Args(\sigma_{Bx_m} \dots \sigma_{Bx_1} W_0) \\ \sigma_{B3}^{(i)}, & \text{otherwise.} \end{cases}$$

This means that we only take branch 7 in σ_i if there is some σ_{Bx_j}, with $j \equiv i \ (mod\ k)$ that took branch 7.

⟨1⟩4. PICK the triple $(\emptyset, \sigma_{B1}^{(n+1)}\sigma^*, FP_{B1}^{(n+1)})$, where $FP_{B1}^{(n+1)}$ is the set of fixpoint equations of branch 1 in the $n+1$ iteration.

⟨1⟩5. SUFFICES: to prove that exists λ (the form of λ depends whether By is $B1$, $B2$

or $B6$) such that $\lambda\sigma_{B1}^{(n+1)}\sigma^* = \sigma_{B1}^{(m+1)}\sigma_{Bx_m}^{(m)}\dots\sigma_{Bx_1}^{(1)}$. Then $\delta'\lambda\sigma_{B1}^{(n+1)}\sigma^* = \delta$, which proves that $\sigma_{B1}^{(n+1)}\sigma^* \le \delta$.

⟨1⟩6. CASE: $By = B1$. Then, $\sigma_{By}^{(m+1)} = \{W_m \mapsto \pi Y_m\}$ and $\sigma_{B1}^{(n+1)} = \{W_n \mapsto \pi Y_n\}$.

PROOF:

⟨2⟩1. PICK n such that $k \le n < 2k$ and $n \equiv m\ (mod\ k)$.

⟨2⟩2. LET: λ be defined as:

- For every $i < k$ let $j_1, \dots, j_l$ be all the indices that are equal to i modulo k such that $Z_{j_1}, \dots, Z_{j_l}$ appear in $Args(\sigma_{Bx_m}\dots\sigma_{Bx_1}W_0)$. Then,
$$\lambda Z_i = f(Z_{j_1}, \dots, Z_{j_l}).$$
- $\lambda Y_n = Y_m$.
- Let $Args$ be the list of arguments (counting repetitions) that are in
$$f(\pi^{-1}Z_2^o, \dots, [\pi^{-(m-1)}, \dots, \pi^{-1}]Z_m^o, [\pi^{-(m-1)}, \dots, Id]Y_m, X_m)$$
but not in
$$\lambda f(\pi^{-1}Z_2^o, \dots, [\pi^{-(n-1)}, \dots, \pi^{-1}]Z_n^o, [\pi^{-(n-1)}, \dots, Id]Y_n)$$
DEFINE: λX_n as $f(Args)$.

⟨2⟩3. For X_0 we have:
$$\begin{aligned}\lambda\sigma_{B1}^{(n+1)}\sigma^* X_0 &= \lambda\sigma_{B1}^{(n+1)} f(\pi^{-1}Z_2^o, \dots, [\pi^{-(n-1)}, \dots, \pi^{-1}]Z_n^o, [\pi^{-(n-1)}, \dots, Id]Y_n, X_n)\\ &= \lambda f(\pi^{-1}Z_2^o, \dots, [\pi^{-(n-1)}, \dots, \pi^{-1}]Z_n^o, [\pi^{-(n-1)}, \dots, Id]Y_n, X_n)\\ &= f(\pi^{-1}Z_2^o, \dots, [\pi^{-(m-1)}, \dots, \pi^{-1}]Z_m^o, [\pi^{-(m-1)}, \dots, Id]Y_m, X_m)\\ &= \sigma_{B1}^{(m+1)}\sigma_{Bx_m}^{(m)}\dots\sigma_{Bx_1}^{(1)}X_0.\end{aligned}$$
For W_0 we have:
$$\begin{aligned}\lambda\sigma_{B1}^{(n+1)}\sigma^* W_0 &= \lambda\sigma_{B1}^{(n+1)} f(Z_1^o, Z_2^o, \dots, Z_n^o, W_n)\\ &= \lambda f(Z_1^o, Z_2^o, \dots, Z_n^o, \pi Y_n)\\ &= f(Z_1^o, Z_2^o, \dots, Z_m^o, \pi Y_m)\\ &= \sigma_{B1}^{(m+1)} f(Z_1^o, Z_2^o, \dots, Z_m^o, W_m)\\ &= \sigma_{B1}^{(m+1)}\sigma_{Bx_m}^{(m)}\dots\sigma_{Bx_1}^{(1)}W_0.\end{aligned}$$
For Y_0 we have:
$$\begin{aligned}\lambda\sigma_{B1}^{(n+1)}\sigma^* Y_0 &= \lambda\sigma_{B1}^{(n+1)} f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \dots, \pi^{-n}Z_n^o, \pi^{-n}Y_n)\\ &= \lambda f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \dots, \pi^{-n}Z_n^o, \pi^{-n}Y_n)\\ &= f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \dots, \pi^{-m}Z_m^o, \pi^{-m}Y_m)\\ &= \sigma_{B1}^{(m+1)} f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \dots, \pi^{-m}Z_m^o, \pi^{-m}Y_m)\\ &= \sigma_{B1}^{(m+1)}\sigma_{Bx_m}^{(m)}\dots\sigma_{Bx_1}^{(1)}Y_0.\end{aligned}$$

Notice that in the computation of $\lambda\sigma^{(n+1)}_{B1}\sigma^*Y_0$ we used that $n \equiv m \pmod k$ to obtain $\pi^{-n}Y_m = \pi^{-m}Y_m$.

$\langle 1\rangle 7.$ CASE: $By = B2$. Then, $\sigma^{(m+1)}_{By} = \{W_m \mapsto \pi^2 Y_m, X_m \mapsto \pi Y_m\}$.

$\langle 2\rangle 1.$ PICK n such that $k \le n < 2k$ and $n \equiv m$ $(mod\ k)$.

$\langle 2\rangle 2.$ LET: λ be defined as:

- For every $i < k$ let $j_1, \dots, j_l$ be all the indices that are equal to i modulo k such that $Z_{j_1}, \dots, Z_{j_l}$ appear in $Args(\sigma_{Bx_m} \dots \sigma_{Bx_1} W_0)$. Then,
$$\lambda Z_i = f(Z_{j_1}, \dots, Z_{j_l}).$$
- $\lambda Y_n = \pi Y_m$.
- Let $Args$ be the list of arguments (counting repetitions) that are in
$$f(\pi^{-1}Z^o_2, \dots, [\pi^{-(m-1)}, \dots, \pi^{-1}]Z^o_m, [\pi^{-(m-1)}, \dots, Id]Y_m, \pi Y_m)$$
but not in
$$\lambda f(\pi^{-1}Z^o_2, \dots, [\pi^{-(n-1)}, \dots, \pi^{-1}]Z^o_n, [\pi^{-(n-1)}, \dots, Id]Y_n)$$
DEFINE: λX_n as $f(Args)$.

$\langle 2\rangle 3.$ Notice that $\sigma^{(n+1)}_{By} = \{W_n \mapsto \pi Y_n\}$. For X_0 we have:
$$\begin{aligned}
\lambda\sigma^{(n+1)}_{B1}\sigma^*X_0 &= \lambda\sigma^{(n+1)}_{B1} f(\pi^{-1}Z^o_2, \dots, [\pi^{-(n-1)}, \dots, \pi^{-1}]Z^o_n, [\pi^{-(n-1)}, \dots, Id]Y_n, X_n)\\
&= \lambda f(\pi^{-1}Z^o_2, \dots, [\pi^{-(n-1)}, \dots, \pi^{-1}]Z^o_n, [\pi^{-(n-1)}, \dots, Id]Y_n, X_n)\\
&= f(\pi^{-1}Z^o_2, \dots, [\pi^{-(m-1)}, \dots, \pi^{-1}]Z^o_m, [\pi^{-(m-1)}, \dots, Id]Y_m, \pi Y_m)\\
&= \sigma^{(m+1)}_{B2} f(\pi^{-1}Z^o_2, \dots, [\pi^{-(m-1)}, \dots, \pi^{-1}]Z^o_m, [\pi^{-(m-1)}, \dots, Id]Y_m, X_m)\\
&= \sigma^{(m+1)}_{B2}\sigma^{(m)}_{Bx_m} \dots \sigma^{(1)}_{Bx_1} X_0.
\end{aligned}$$
For W_0 we have:
$$\begin{aligned}
\lambda\sigma^{(n+1)}_{B1}\sigma^*W_0 &= \lambda\sigma^{(n+1)}_{B1} f(Z^o_1, Z^o_2, \dots, Z^o_n, W_n)\\
&= \lambda f(Z^o_1, Z^o_2, \dots, Z^o_n, \pi Y_n)\\
&= f(Z^o_1, Z^o_2, \dots, Z^o_m, \pi^2 Y_m)\\
&= \sigma^{(m+1)}_{B2} f(Z^o_1, Z^o_2, \dots, Z^o_m, W_m)\\
&= \sigma^{(m+1)}_{B2}\sigma^{(m)}_{Bx_m} \dots \sigma^{(1)}_{Bx_1} W_0.
\end{aligned}$$
For Y_0 we have:
$$\begin{aligned}
\lambda\sigma^{(n+1)}_{B1}\sigma^*Y_0 &= \lambda\sigma^{(n+1)}_{B1} f(\pi^{-1}Z^o_1, \pi^{-2}Z^o_2, \dots, \pi^{-n}Z^o_n, \pi^{-n}Y_n)\\
&= \lambda f(\pi^{-1}Z^o_1, \pi^{-2}Z^o_2, \dots, \pi^{-n}Z^o_n, \pi^{-n}Y_n)\\
&= f(\pi^{-1}Z^o_1, \pi^{-2}Z^o_2, \dots, \pi^{-m}Z^o_m, \pi^{-(n-1)}Y_m)\\
&= f(\pi^{-1}Z^o_1, \pi^{-2}Z^o_2, \dots, \pi^{-m}Z^o_m, \pi^{-m}Y_m)\\
&= \sigma^{(m+1)}_{B2} f(\pi^{-1}Z^o_1, \pi^{-2}Z^o_2, \dots, \pi^{-m}Z^o_m, \pi^{-m}Y_m)\\
&= \sigma^{(m+1)}_{B2}\sigma^{(m)}_{Bx_m} \dots \sigma^{(1)}_{Bx_1} Y_0.
\end{aligned}$$

$\langle 1\rangle 8.$ CASE: $By = B6$. Then, $\sigma^{(m+1)}_{By} = \{W_m \mapsto f(Z_{m+1}, \pi X_m), Y_m \mapsto f(\pi^{-1}Z_{m+1}, \pi^{-1}X_m)\}$

$\langle 2\rangle 1.$ PICK n such that $k \leq n < 2k$ and $n \equiv m-1 \ (mod\ k)$.

$\langle 2\rangle 2.$ LET: λ be defined as:

- For every $i < k$, let $j_1, \ldots, j_l$ be all the indices that are equal to i modulo k such that $Z_{j_1}, \ldots, Z_{j_l}$ appear in $Args(\sigma_{Bx_m} \ldots \sigma_{Bx_1} W_0)$. Then,
$$\lambda Z_i = f(Z_{j_1}, \ldots, Z_{j_l}).$$
- $\lambda Y_n = f(\pi^{-1} Z_{m+1}, X_m)$.
- Let $Args$ be the list of arguments (counting repetitions) that are in
$f(\pi^{-1}Z_2^o, \ldots, [\pi^{-(m-1)}, \ldots, \pi^{-1}]Z_m^o, [\pi^{-m}, \ldots, \pi^{-1}]Z_{m+1}, [\pi^{-m}, \ldots, Id]X_m)$
but not in
$$\lambda f(\pi^{-1}Z_2^o, \ldots, [\pi^{-(n-1)}, \ldots, \pi^{-1}]Z_n^o, [\pi^{-(n-1)}, \ldots, Id]Y_n)$$
DEFINE: λX_n as $f(Args)$.

$\langle 2\rangle 3.$ For X_0 we have:

$$\begin{aligned}
\lambda\sigma_{B1}^{(n+1)}\sigma^* X_0 &= \lambda\sigma_{B1}^{(n+1)} f(\pi^{-1}Z_2^o, \ldots, [\pi^{-(n-1)}, \ldots, \pi^{-1}]Z_n^o, [\pi^{-(n-1)}, \ldots, Id]Y_n, X_n) \\
&= \lambda f(\pi^{-1}Z_2^o, \ldots, [\pi^{-(n-1)}, \ldots, \pi^{-1}]Z_n^o, [\pi^{-(n-1)}, \ldots, Id]Y_n, X_n) \\
&= f(\pi^{-1}Z_2^o, \ldots, [\pi^{-(m-1)}, \ldots, \pi^{-1}]Z_m^o, [\pi^{-m}, \ldots, \pi^{-1}]Z_{m+1}, [\pi^{-m}, \ldots, Id]X_m) \\
&= \sigma_{B6}^{(m+1)} f(\pi^{-1}Z_2^o, \ldots, [\pi^{-(m-1)}, \ldots, \pi^{-1}]Z_m^o, [\pi^{-(m-1)}, \ldots, Id]Y_m, X_m) \\
&= \sigma_{B6}^{(m+1)} \sigma_{Bx_m}^{(m)} \ldots \sigma_{Bx_1}^{(1)} X_0.
\end{aligned}$$

For W_0 we have:

$$\begin{aligned}
\lambda\sigma_{B1}^{(n+1)}\sigma^* W_0 &= \lambda\sigma_{B1}^{(n+1)} f(Z_1^o, Z_2^o, \ldots, Z_n^o, W_n) \\
&= \lambda f(Z_1^o, Z_2^o, \ldots, Z_n^o, \pi Y_n) \\
&= f(Z_1^o, Z_2^o, \ldots, Z_m^o, Z_{m+1}, \pi X_{m+1}) \\
&= \sigma_{B6}^{(m+1)} f(Z_1^o, Z_2^o, \ldots, Z_m^o, W_m) \\
&= \sigma_{B6}^{(m+1)} \sigma_{Bx_m}^{(m)} \ldots \sigma_{Bx_1}^{(1)} W_0.
\end{aligned}$$

For Y_0 we have:

$$\begin{aligned}
\lambda\sigma_{B1}^{(n+1)}\sigma^* Y_0 &= \lambda\sigma_{B1}^{(n+1)} f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \ldots, \pi^{-n}Z_n^o, \pi^{-n}Y_n) \\
&= \lambda f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \ldots, \pi^{-n}Z_n^o, \pi^{-n}Y_n) \\
&= f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \ldots, \pi^{-m}Z_m^o, \pi^{-(n+1)}Z_{m+1}, \pi^{-(n+1)}X_{m+1}) \\
&= f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \ldots, \pi^{-m}Z_m^o, \pi^{-(m+1)}Z_{m+1}, \pi^{-(m+1)}X_{m+1}) \\
&= \sigma_{B6}^{(m+1)} f(\pi^{-1}Z_1^o, \pi^{-2}Z_2^o, \ldots, \pi^{-m}Z_m^o, \pi^{-m}Y_m) \\
&= \sigma_{B6}^{(m+1)} \sigma_{Bx_m}^{(m)} \ldots \sigma_{Bx_1}^{(1)} Y_0.
\end{aligned}$$

Remark 36 (Motivation for the Definition of λ in Theorem 41)**.** *To construct λ in the proof of Theorem 41 we proceeded in three steps, leaving the calculation of λX_n for last,* *since this was "the hardest to get it right".*

1. *Instatiate λZ_i introducing every variable Z_j such that $i \equiv j \pmod{k}$.*
2. *Calculate the value of λY_n by looking how we could make $\lambda \sigma_{B1}^{(n+1)} \sigma^* W_0$ equal to $\sigma_{By}^{(m+1)} \sigma_{Bx_m}^{(m)} \dots \sigma_{Bx_1}^{(1)} W_0$.*
3. *Calculate the relation between m and n by repeating the previous step with Y_0 instead of W_0.*
4. *Only when those "easy instantiations" had been performed we decided the value of λX_m, since this is the most complicated expression. We did it by looking at how we could make $\lambda \sigma_{B1}^{(n+1)} \sigma^* X_0$ equal to $\sigma_{By}^{(m+1)} \sigma_{Bx_m}^{(m)} \dots \sigma_{Bx_1}^{(1)} X_0$.*

A Python script to generate all solutions to $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$ was devised and more details about it are shown in Appendix A.

6.2.3 $f(2X_1, X_2, X_3) \approx^? f(2\pi \cdot X_2, Y_1)$

Section 6.2.2 shows that it is enough to loop a limited amount of times to generate a complete set of solutions for the equational constraint. In this section we investigate

$$f(2X_1, X_2, X_3) \approx^? f(2\pi X_2, Y_1).$$

This problem is an adaptation from Stickel's example described in Section 2.2.2, with only variable arguments and the same variable X_2 appearing in both terms (but with different permutations suspended on it).

To aid us in this investigation we used a Python script that calculates a basis of solution to a Diophantine equation and constructs the associated equation (similar to Table 2.2 and Equations 2.1 of Section 2.2.2). The script is available here.

The script correctly solves the equation[4] associated with

$$P_0 = \{f(2X_1, X_2, X_3) \approx^? f(2\pi X_2, Y_1)\}$$

[4]notice it is the same equation as the one in Section 2.2.2

and the equational constraints in one of the branches are:

$$\begin{aligned}
X_1 &\approx^? f(Z_3, Z_4)\\
X_2 &\approx^? f(Z_1, 2Z_5, Z_6)\\
X_3 &\approx^? f(Z_2, Z_6, 2Z_7)\\
\pi X_2 &\approx^? f(Z_3, Z_5, Z_6, Z_7)\\
Y_2 &\approx^? f(Z_1, Z_2, 2Z_4)
\end{aligned}$$

After we perform the instantiations, a substitution σ_1 is computed and a new set of equational constraints needs to be solved:

$$\begin{aligned}
P_1 &= \{f(\pi Z_1, 2\pi Z_5, \pi Z_6) \approx^? f(Z_3, Z_5, Z_6, Z_7)\},\\
\sigma_0 &= \{X_1 \mapsto f(Z_3, Z_4), X_2 \mapsto f(Z_1, 2Z_5, Z_6), X_3 \mapsto f(Z_2, Z_6, 2Z_7), Y_2 \mapsto f(Z_1, Z_2, 2Z_4)\}.
\end{aligned}$$

Running the Python script again to solve the associated Diophantine equation will result in the following equational constraints in one of the branches:

$$\begin{aligned}
\pi Z_1 &\approx^? f(Z_8, Z_9, Z_{10}, Z_{11})\\
\pi Z_5 &\approx^? f(Z_{16}, Z_{17}, Z_{18}, Z_{19}, Z_{20}, Z_{21}, Z_{22}, Z_{23}, Z_{24}, Z_{25})\\
\pi Z_6 &\approx^? f(Z_{12}, Z_{13}, Z_{14}, Z_{15})\\
Z_3 &\approx^? f(Z_8, Z_{12}, 2Z_{16}, Z_{17}, Z_{19}, Z_{22})\\
Z_5 &\approx^? f(Z_9, Z_{13}, Z_{17}, 2Z_{18}, Z_{20}, Z_{23})\\
Z_6 &\approx^? f(Z_{10}, Z_{14}, Z_{19}, Z_{20}, 2Z_{21}, Z_{24})\\
Z_7 &\approx^? f(Z_{11}, Z_{15}, Z_{22}, Z_{23}, Z_{24}, 2Z_{25})
\end{aligned}$$

After we perform the instantiations, the substitution σ_2 computed is:

$$\begin{aligned}
\sigma_2 = \{&Z_1 \mapsto \pi^{-1} \cdot f(Z_8, Z_9, Z_{10}, Z_{11}),\\
&Z_5 \mapsto \pi^{-1} \cdot f(Z_{16}, Z_{17}, Z_{18}, Z_{19}, Z_{20}, Z_{21}, Z_{22}, Z_{23}, Z_{24}, Z_{25})\\
&Z_6 \mapsto \pi^{-1} \cdot f(Z_{12}, Z_{13}, Z_{14}, Z_{15})\\
&Z_3 \mapsto f(Z_8, Z_{12}, 2Z_{16}, Z_{17}, Z_{19}, Z_{22})\\
&Z_7 \mapsto f(Z_{11}, Z_{15}, Z_{22}, Z_{23}, Z_{24}, 2Z_{25}).\}
\end{aligned}$$

The new set of equational constraints P_2 is:

$$P_2 = \{\pi^{-1} \cdot f(Z_{16}, Z_{17}, Z_{18}, Z_{19}, Z_{20}, Z_{21}, Z_{22}, Z_{23}, Z_{24}, Z_{25}) \approx^? f(Z_9, Z_{13}, Z_{17}, 2Z_{18}, Z_{20}, Z_{23}),$$
$$\pi^{-1} \cdot f(Z_{12}, Z_{13}, Z_{14}, Z_{15}) \approx^? f(Z_{10}, Z_{14}, Z_{19}, Z_{20}, 2Z_{21}, Z_{24})\}.$$

From this calculations it is not clear whether we can rewrite the problem to get into a loop or whether the expressions necessarily keep getting more and more complicated. Further investigation is necessary and we could start by improving the Python script. The script could be improved if, in addition to solving the Diophantine equation it also computed the substitution (in terms of the original variables X_1, X_2, X_3, Y_1) and listed the new set of equational constraints that must be solved.

6.2.4 Additional Considerations

What if we try include equational constraints such as $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$ in the output returned by our to-be-devised nominal AC-unification algorithm? More precisely, what if we include in the output of the algorithm equational constraints of the form

$$s \equiv f(s_1, \ldots, s_m) \approx^? f(t_1, \ldots, t_n) \equiv t$$

when one of the arguments of s is a moderated variable $\pi \cdot X$ and one of the arguments of t is a moderated variable $\pi' \cdot X$, i.e. when the same variable appears on both sides, but with different permutations suspended on it?

This is not ideal, because sometimes equational constraints such as the one described will be unsolvable. For instance, let 0 and 1 be constants and notice that

$$f(X, 0) \approx^? f(\pi X, 1)$$

has no solution, since no matter how we instantiate X, the number of occurrences of 0 in the left-hand side and in the right-hand side will never be the same.

Another point: could we try some sort of combinatorial argument to solve $f(X, W) \approx^? f(\pi X, \pi Y)$? Perhaps by finding all solutions to $\{X \approx^? \pi X,\ W \approx^? \pi Y\}$ and also all the solutions to $\{X \approx^? \pi Y, W \approx^? \pi X\}$? Unfortunately, this approach is not complete. Consider $\pi = (a\ b)$ and let

$$\sigma = \{X \mapsto f(a, c), W \mapsto f(b, d), Y \mapsto f(b, d)\}.$$

The issue is that the arguments of σX (a and c) are partially present in $\sigma\pi X = f(b, c)$ (c is present, a is not) and partially present in $\sigma\pi Y = f(a, d)$. The same happens for the arguments b and d of σW: b is present in $\sigma\pi X$ but not in $\sigma\pi Y$, while d is present in $\sigma\pi Y$ but not in $\sigma\pi X$.

6.3 Nominal AC-Unification Via AC-Unification of Higher-Order Patterns?

A promising way to obtain a nominal AC-unification would be to explore the connection between higher-order pattern unification and nominal unification described in Cheney [28] and Levy and Villaret [54] and use the work that has been done by Boudet and Contejean [22] in AC-unification of higher-order patterns.

λ-terms are built using the grammar:

$$t \;::=\; x \;|\; c \;|\; \lambda x.t \;|\; t_1\; t_2$$

where x is a variable, c is a constant, $\lambda x.t$ is a lambda abstraction and $t_1\ t_2$ is a function application. We use syntactic sugar and represent terms of the form $(\ldots((a\ t_1)\ t_2)\ldots t_n)$ as $a(t_1, t_2, \ldots, t_n)$, where a is a constant or a variable. Additionally, terms such as $\lambda x_1.\ldots.\lambda x_n.t$ may be denoted simply as $\lambda x_1 \ldots x_n.t$ or even more succinctly as $\lambda\vec{x}.t$, where $\vec{x}$ is shorthand notation for $x_1 \ldots x_n$.

A possible way to start investigating the connection between nominal AC-unification and higher-order pattern unification would be by looking at how problems such as $\pi \cdot X \approx^? X$ and $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$ would be "translated" to higher-order pattern AC-unification (as described in [54]), solved in higher-order pattern AC-unification (as described in [22]) and how the solutions would be "translated back" to the nominal setting (as described in [28]). The mentioned translations described in [54] and [28] would need to be extended to handle AC function symbols.

6.3.1 From Nominal to Higher-Order Pattern

Levy and Villaret [54] show that nominal unification can be reduced to higher-order pattern unification. The main idea of [54] is to translate atoms into bound variables, and moderated variables into free variables that receive as arguments a list of bound variables they can capture. This idea is a bit surprising at first, as atoms in nominal represent object level variables and may or may not be bound, while suspended variables represent meta level variables, are not bound by abstractions and are instantiated by substitutions (something that does not happen for atoms).

The translation described in [54] depends on ordering the atoms that appear in the unification problem into a list L. In order to translate nominal terms into higher-order patterns we have Definition 43.

Definition 43 (Translating Nominal Terms to Higher-Order Patterns)**.** *Given a context ∇ and a list of atoms L we define a translation function $[\![\ ]\!]$ from nominal terms into higher-order patterns inductively as :*

- $[\![a]\!]_{L,\nabla} = a$
- $[\![f(t_1, \ldots, t_n)]\!]_{L,\nabla} = f([\![t_1]\!]_{L,\nabla}, \ldots, [\![t_n]\!]_{L,\nabla})$
- $[\![\ [a]t\ ]\!]_{L,\nabla} = \lambda a.[\![t]\!]_{L,\nabla}$
- $[\![\pi \cdot X]\!]_{L,\nabla} = X([\![\pi \cdot a_1]\!]_{L,\nabla}, \ldots, [\![\pi \cdot a_n]\!]_{L,\nabla})$*, where* $(a_1, \ldots, a_n)$ *is a sublist of atoms of* L *defined by* $(a_1, \ldots, a_n) = (a \in L \mid a\#X \notin \nabla)$*.*

We believe the rule for AC function symbols, should be analogous with the one for syntactic function applications:

$$[\![f^{AC}(t_1, \ldots, t_n)]\!]_{L,\nabla} = f^{AC}([\![t_1]\!]_{L,\nabla}, \ldots, [\![t_n]\!]_{L,\nabla}).$$

Once we have a translation function from nominal terms to higher-order patterns, we can extend it to translate a set of nominal equational constraints to a higher-order pattern unification problem, as shown in Definition 44.

Definition 44. *Let $L = (a_1, \ldots, a_n)$ be a list containing all the atoms of a set of equational constraints P. The translation function defined in P given L is:*

$$[\![P]\!]_L = \{\lambda a_1 \ldots a_n.[\![t]\!]_{L,\emptyset} \approx^? \lambda a_1 \ldots a_n.[\![s]\!]_{L,\emptyset} \mid t \approx^? s \in P\}$$

Finally, we do not have to worry about freshness constraints of the form $a\#^? t$ (see [54]), as we can (and should in order for the mentioned translation to work) substitute them by equational constraints of the form $[a][b]t \approx^? [b][b]t$, where b is some new atom.

Example 34. *We illustrate how the translation would work for $\pi X \approx^? X$ and $f(X, W) \approx^? f(\pi X, \pi Y)$ when $\pi = (a\ b)$.*

- *The translation of $(a\ b)X \approx^? X$ is*

$$\lambda ab.X(a, b) \approx^? \lambda ab.X.$$

- *The translation of $f(X, W) \approx^? f((a\ b)X, (a\ b)Y)$ is*

$$\lambda ab.f(X, W) \approx^? \lambda ab.f(X(a, b),\ Y(a, b)).$$

6.3.2 AC-Unification of Higher-Order Patterns

Boudet and Contejean [22] presented an algorithm for AC-unification of higher-order patterns that relies on Boudet et al. [24] first-order AC-unification algorithm. Equations such as $\lambda ab.X(a,b) \approx^? \lambda ab.X(b,a)$ are called flexible-flexible equations with the same head variable on both sides, and are kept by the algorithm (see [22, 65]). Indeed, it has been proved that those equation are always solvable, however they may not have a finite complete set of unifiers, as shown in Example 35.

Example 35 (AC-Unification of Patterns is Nullary). *This example was adapted from [65]. Consider the equation $\lambda ab.X(a,b) \approx^? \lambda ab.X(b,a)$ and let $+$ be an AC function symbol in our signature. The substitutions*

$$\sigma_m = \{X \mapsto \lambda ab.G_m(Y_1(a,b) + Y_1(b,a), \ldots, Y_m(a,b) + Y_m(b,a))\}$$

for $m = 1, 2, \ldots$ are all AC-unifiers. It is possible to prove that every solution of $\lambda ab.X(a,b) \approx^?$ $\lambda ab.X(b,a)$ is an instance of some σ_m and that σ_{m+1} is strictly more general than σ_m.

Notice that if we employ the translation described in [53] to $\pi X \approx^? X$, where $\pi = (a\ b)$ we get a flexible-flexible equation with the same head variable on both sides: $\lambda ab.X(a,b) \approx^? \lambda ab.X$. This seems to suggest that the corresponding problem in nominal is nullary and reinforces that an enumeration procedure is the right approach.

When the algorithm of [24] deals with AC function symbols, it generates a system of linear Diophantine equations for one single equation, a different approach than the one of Stickel. Before discussing the bounds on this system of equations, we consider the equation

$$\lambda abc.2X(a,b,c) + X(b,c,a) \approx^? \lambda abc.2Y(a,b,c)$$

as an example.

A solution σ to this equation may introduce terms $t(a,b,c)$ that do not depend on the order of the arguments, i.e. $t(a,b,c) = t(b,c,a)$ and terms $t(a,b,c)$ that depend on the order of the arguments, i.e. $t(a,b,c) \neq t(b,c,a)$. We analyse these two cases.

If σ instantiates X to something containing the term $t(a,b,c)$, where $t(a,b,c)$ does not depend on the order of arguments then, on one hand, this term is introduced 2α times by $\sigma 2X(a,b,c)$ and α times by $\sigma X(b,c,a)$. On the other hand $t(a,b,c)$ must be introduced 2β times by $\sigma 2Y(a,b,c)$, where α and β are solutions to the linear Diophantine equation $3m = 2n$. This is a linear Diophantine equation whose set of minimal solutions is simply $\{(2,3)\}$. We associate a new variable Z_1 with this solution of the Diophantine system, as shown in Table 6.2.

Table 6.2: Solutions for $3m = 2n$.

m	**n**	**New Variable**
2	3	Z_1

Finally, let θ be a substitution such that

$$\theta Z_1(a,b,c) = \theta Z_1(b,c,a) = \theta Z_1(c,a,b).$$

Notice that

$$\{X \mapsto \lambda abc.\theta 2Z_1(a,b,c),\ Y \mapsto \lambda abc.\theta 3Z_1(a,b,c)\}$$

is a solution to the original problem.

Now for the case of σ instantiating X to something containing the term $t(a,b,c)$, where $t(a,b,c)$ depends on the order of arguments. Let α_1, α_2 and α_3 be the number of times $\sigma X(a,b,c)$ introduces $t(a,b,c)$, $t(b,c,a)$ and $t(c,a,b)$ respectively. Similarly, let β_1, β_2, β_3 be the number of times $\sigma Y(a,b,c)$ introduces $t(a,b,c)$, $t(b,c,a)$ and $t(c,a,b)$. Then, $(\alpha_1,\alpha_2,\alpha_3,\beta_1,\beta_2,\beta_3)$ are solutions to the system of linear Diophantine equations:

$$\begin{aligned} 2n_1 + n_3 &= 2m_1 \\ 2n_2 + n_1 &= 2m_2 \\ 2n_3 + n_2 &= 2m_3 \end{aligned} \tag{6.5}$$

The basis of solution for this system, along with the new variables associated to each solution, is shown on Table 6.5.

Table 6.3: Solutions for Equations 6.5.

$\mathbf{n_1}$	$\mathbf{n_2}$	$\mathbf{n_3}$	$\mathbf{m_1}$	$\mathbf{m_2}$	$\mathbf{m_3}$	**New Variables**
2	0	0	2	1	0	Z_2
0	2	0	0	2	1	Z_3
0	0	2	1	0	2	Z_4

We could combine the results expressed in Tables 6.2 and 6.3 to express solutions of the original problem. For instance,

$$\begin{gathered} \{X \mapsto \lambda abc.\theta 2Z_1(a,b,c) + 2Z_2(a,b,c) + 2Z_3(b,c,a) + 2Z_4(c,a,b) \\ Y \mapsto \lambda abc.\theta 3Z_1(a,b,c) + 2Z_2(a,b,c) + Z_2(b,c,a) + 2Z_3(b,c,a) \\ + Z_3(c,a,b) + Z_4(a,b,c) + 2Z_4(c,a,b)\} \end{gathered}$$

where we are again making the assumption that:

$$\theta Z_1(a, b, c) = \theta Z_1(b, c, a) = \theta Z_1(c, a, b).$$

As discussed for first-order AC-unification, we may omit some of the new variables, as long as the "old" variables X and Y are not mapped on an "empty" term and obtain other solutions.

Remark 37 (Variable Abstraction and Restriction to Terms of the Form $X(a_1, \ldots, a_n)$)**.** *The considerations so far have been focused exclusively on terms of the form $X(a_1, \ldots, a_n)$, where $a_1, \ldots, a_n$ are variables. This is not a restriction, as Boudet et al. [22] uses variable abstraction to guarantee that unification problems of the form $X(\ldots, t_1, \ldots)$ where t_1 is a function application do not occur.*

As illustrated in the previous example, the algorithm employed in the AC-part is different than the one from Stickel's, obtaining a system of linear Diophantine equations from only one equational constraint. Termination of the procedure relies on a bound in the number of new variables introduced. Let n be the number of coefficients in the AC equational constraint. Let Π be the subgroup of permutations of all bound variables in the problem. It has been proved that the number of new variables introduced is bound by $n \times |\Pi|$. For instance, the system corresponding to the higher-order pattern equational constraint $\lambda ab.f(X, W) \approx^? \lambda ab.f(X(a, b),\ Y(a, b))$ would be bound by $4 \times 2 = 8$ new variables.

6.3.3 From Higher-Order Patterns to Nominal

Cheney [28] established the opposite reduction than the one described in [54]: that full higher-order pattern unification can be reduced to nominal unification. This reduction employs nominal patterns, a variant of nominal terms closer to higher-order patterns that has a concretion operation (see [28] for more details). More precisely, Cheney [28] first shows that higher-order pattern unification can be reduced to nominal pattern unification and then goes on to show that nominal pattern unification can be reduced to nominal unification.

Chapter 7

Related Work

Syntactic unification seems[1] to have appeared first in the work of Herbrand [42], although Robinson [66] was the first who gave an algorithm accompanied by a proof of termination and correctness. Since Robinson's algorithm was exponential, it was later independently refined by others [19, 43]. In particular, Paterson and Wegman [63] discovered a linear unification algorithm that uses directed acyclic graphs (DAGs) to represent a unification problem, while Martelli and Montanari [56] gave an efficient unification algorithm which represents the unification problem as a set of equations.

Stickel [72] was the first to solve unification in the presence of AC-function symbols. He showed how the problem is connected to finding nonnegative integral solutions to linear equations and proved that his algorithm was sound, complete and terminating for a subclass of the general case [72, 73]. However, Stickel's proof of termination did not apply to the general case and almost a decade after the introduction of this algorithm, Fages discovered the flaw and proposed a measure fixing the termination proof for the general case [39, 40]. Since then, investigations on solving AC-unification efficiently and on the complexity of AC-unification were carried out.

Regarding the complexity of AC-unification, Benanav et al. [20] showed that the decision problem for AC-matching is NP-complete, and the decision problem for AC-unification is NP-hard. In addition, Kapur and Narendran [48] showed that the complexity of computing a complete set of AC-unifiers is double-exponential.

Both AC- and C-unification problems are of finitary type, but the complexity of computing a complete set of unifiers for the former problem is double-exponential, while for the latter one, it is "only" exponential as shown by Kapur and Narendran [48]. Indeed, to build complete sets of C-unifiers, only simple swapping-argument-combinations need to be considered to instantiate variables. However, to build complete sets of AC-unifiers,

[1]the information on this paragraph was taken from Baader and Nipkow seminal book "Term Rewriting and All That" [15]. See it for more interesting bibliographic notes on the subject.

all possible associations and permutations of arguments should be considered, which is precisely expressed by Stickel's method based on solving Diophantine equations.

Regarding solving AC-unification efficiently, Boudet et al. [24] proposed an AC-unification algorithm that explores constraints more efficiently than the standard algorithm. Further, Boudet [21] described and compared an implementation of this algorithm to previous ones. Also, Adi and Kirchner [2] implemented an AC-unification algorithm, proposed benchmarks and showed that their algorithm improves over previous ones in time and space.

Regarding formalisations, in 2004, Contejean [31] gave the first certified AC-matching algorithm in Coq. Additionally, in 2008, Meßner et al. [57] gave a formally verified solver for homogeneous linear Diophantine equations in Isabelle/HOL. As we shall see, the problem of AC-unification is connected to solving linear Diophantine equations. However, no formalisation of AC-unification was available until 2022, when we proved termination, soundness and completeness of Stickel's AC-unification algorithm [9] using the proof assistant PVS [62].

On the matter of nominal unification, as mentioned before, Urban et. al [75] gave the first nominal unification algorithm in 2004. It is a rule-based algorithm that was formalised terminating, correct and complete in Isabelle/HOL [74,75]. Research continued in the direction of making algorithms improvements to solve this problem, with works from Levy and Villaret [53], Calvès [25] and Calvès and Fernández [26, 27]. Another step ahead in nominal unification was taken in 2016, when Schmidt-Schauß et al. [69] presented a nominal unification approach for higher-order expressions with recursive `let`. Furthermore, one application of nominal unification in software systems is α-Prolog [29]: "a logic programming language with built-in names, fresh name generation, name binding, and unification up to α-equivalence (that is, consistent renaming of bound names)[2]".

There have been previous works on nominal equational theories using the PVS proof assistant. In [11], Ayala-Rincón et. al presented a nominal syntactic unification algorithm specified as a functional program and verified it in the proof assistant PVS[3]. Enriching nominal unification with equational reasoning started with developing rule-based techniques for commutative operators in Coq (see Ayala-Rincón et al. [4] or de Carvalho Segundo [33]) and then specifying functional algorithms in PVS (see Ayala-Rincón et al. [12]). An interesting difference between nominal unification and nominal C-unification was found: when expressing solutions as pairs consisting of a freshness context and substitutions, nominal unification is finitary whereas nominal C-unification is not. This compelling difference is based on the fact that the correct approach to solving fixpoint equations of the form $\pi \cdot X \approx^? \pi' \cdot X$ in nominal unification is not complete in nomi-

[2]quote taken from https://homepages.inf.ed.ac.uk/jcheney/programs/aprolog/

[3]This is also described in Oliveira [61].

nal C-unification (see [5] on how to generate every solution to this equation). Finally, in Ayala-Rincón et al. [6] a formalisation of nominal α-equivalence with A, C, and AC functions is given.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

We presented and discussed our formalisations of nominal C-unification, first-order AC-unification and nominal AC-matching that were done using the PVS proof assistant and are part of PVS' main repository of formalisations: NASALib. In each one of the three formalisations we delved into the files that compose the formalisation, detailing their structure, hierarchy and size.

Regarding nominal C-unification, we extended the nominal C-unification algorithm of Ayala-Rincón et al. [12] to also handle matching and equality-checking and proved that our extension is terminating, sound and complete. This was done by adding the parameter $\mathscr{X}$ of protected variables, i.e, variables that cannot be instantiated. Let P be the set of equational constraints we must unify. By setting $\mathscr{X}$ to $\emptyset$, $Vars(lhs(P))$ or $Vars(P)$ one can use our generalised algorithm for unification, matching or α-equality checking, respectively. Furthermore, we used the PVSio feature to test the correctness of a Python manual implementation of the algorithm.

Regarding first-order AC-unification we have given the first formalisation of first-order AC-unification by specifying and verifying Stickel's seminal AC-unification algorithm. The proof of termination used an intricate lexicographic measure based on Fages' termination proof. Proving completeness is done by first using an "additional hypothesis" $\delta \subseteq V$ and an arbitrary set V as the parameter of the algorithm that corresponds to the variables in the problem (Lemma 26); and then using the notion of a renamed inputs (Definition 35) to obtain completeness (Theorem 30) without this extra hypothesis and with $V = Vars(t, s)$, where t and s are the terms we want to unify.

Regarding nominal AC-matching we have proposed the first nominal AC-matching algorithm and formalised it to be terminating, sound and complete. We did it by extending the first-order AC-unification formalisation to the nominal setting and using the parameter

$\mathcal{X}$ for protected variables. The condition $\mathcal{X} \subseteq Vars(rhs(P))$, where P is the set of equational constraints, in the input of our algorithm means that it cannot be used for nominal AC-unification, although the algorithm can be used both for AC-matching and AC-equality-checking. In contrast to first-order AC-unification, in nominal AC-matching it is possible to prove that all the variables introduced "in the combinatorial part of the AC-step" (by SOLVEAC) are immediately instantiated (by INSTANTIATESTEP). This made the proof of termination of the nominal AC-matching much easier than its first-order AC-unification counterpart.

Finally, we presented our work in progress towards nominal AC-unification. The two main issues are solving fixpoint equations $\pi \cdot X \approx^? X$ and proving termination. Concerning solving fixpoint equations we provided a non-deterministic enumeration procedure and exemplified how it can compute interesting solutions. In comparison to the approach of Ayala-Rincón et al. [5] for fixpoint equations in nominal C-unification, the non-deterministic procedure here described has the advantage of not generating some less general solutions. Concerning termination we showed how the problem $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$ gives rise to a loop in some branches and why it is enough to take these branches a limited amount of times, where this limit depend on the order k of the permutation π. We were unable to show whether this behaviour happens in other problems such as $f(2X_1, X_2, X_3) \approx^? f(2\pi \cdot X_2, Y_1)$.

8.2 Future Work

Although the most pressing future work is solving the open theoretical question of termination in nominal AC-unification, there is room for work in first-order AC-unification, nominal AC-matching and other areas of equational reasoning. Regarding first-order AC-unification, one possible path of future work is using our formalisation as a basis to formalise more efficient algorithms. This can be done by generating a basis of solution to a linear Diophantine equation (instead of a spanning set that we currently generate) or by using a smaller bound (see [30]) when calculating solutions to a linear Diophantine equation or by formalising more efficient algorithms for first-order AC-unification that rely on representing terms as directed acyclic graphs DAGs (see [21]). A second valid path of future work is adapting the formalisation to leverage the PVSio feature of PVS in order to test implementations of PVS.

A different possible path of future work is formalising more efficient nominal AC-matching algorithms. Since the nominal AC-matching formalisation is based on the first-order AC-unification formalisation, improvements in the efficiency of the first-order AC-unification algorithm could be adapted to the nominal setting to get a more efficient nominal AC-matching algorithm. An alternative approach would be formalising a first-order AC-matching algorithm (for instance [31]) and then adapting it to the nominal setting. This could be more efficient than obtaining nominal AC-matching via first-order AC-unification, but would probably require more effort.

Another path of future work is investigating unification in the presence of multiple equational theories. For instance: how would one define an algorithm for unification in the presence of syntactic, commutative and associative-commutative function symbols? An approach would be to formalise the combination of equational unification algorithms. As a starting point, there have been works on this topic in first-order by Baader [16] and Schmidt-Schauß [68] . Formally verifying those works in a proof assistant or extending these theoretical results to the nominal setting are both interesting future work endeavors. We note that Boudet [23] has extended those results to higher-order patterns, and perhaps the lessons in [23] could be useful when considering the combination of equational unification algorithms in the nominal setting.

Finally, the most pressing future work is obtaining a nominal AC-unification algorithm and then formalising it to be sound and complete. There are two main open questions, about fixpoint equations $\pi \cdot X \approx^? X$ and termination. Regarding fixpoint equations, the non-deterministic enumeration procedure presented for the task should be proved sound and complete. Regarding termination we should be able to determine in which cases we get a loop such as the one described for $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$ and whether there always exists a bound in the number of times we need to take branches that loop. A promising approach would be to investigate the connection between higher-order pattern AC-unification and nominal AC-unification (see [28, 54]) since Boudet et al. [22] already devised an algorithm for AC-unification of higher-order patterns. Once devised, a nominal AC-unification algorithm would have applications in logic programming languages that employ the nominal paradigm, such as α-Prolog [29].

An alternative path of future work is formalising established theoretical results that are important in equational reasoning. One could consider formalising nominal unification modulo other equational theories, such as those that include distributivity, neutral element or idempotency. Formalising nominal anti-unification [18] and nominal disunification [10] are other interesting paths of future work.

Bibliography

[1] Abadi, M., Cardelli, L., Curien, P., and Lévy, J. Explicit Substitutions. In *Conference Record of the 17th Annual (ACM) Symposium on Principles of Programming Languages* (1990), F. E. Allen, Ed., ACM Press, pp. 31–46. 2

[2] Adi, M., and Kirchner, C. AC-Unification Race: The System Solving Approach, Implementation and Benchmarks. *J. of Sym. Computation 14*, 1 (1992), 51–70. 139

[3] Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., Ferreira-Silva, G., and Nantes-Sobrinho, D. Formalising Nominal C-Unification Generalised with Protected Variables. *Math. Struct. Comput. Sci. 31*, 3 (2021), 286–311. 3, 13, 25, 30, 31, 32, 34, 40, 90, 116

[4] Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., and Nantes-Sobrinho, D. Nominal C-Unification. In *Logic-Based Program Synthesis and Transformation - 27th International Symposium, (LOPSTR)* (2017), vol. 10855 of *Lecture Notes in Computer Science*, Springer, pp. 235–251. 21, 31, 139

[5] Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., and Nantes-Sobrinho, D. On Solving Nominal Fixpoint Equations. In *Frontiers of Combining Systems - 11th International Symposium, (FroCoS)* (2017), vol. 10483 of *Lecture Notes in Computer Science*, Springer, pp. 209–226. 20, 121, 140, 142

[6] Ayala-Rincón, M., de Carvalho Segundo, W., Fernández, M., Nantes-Sobrinho, D., and Oliveira, A. C. R. A Formalisation of Nominal α-Equivalence with A, C, and AC Function Symbols. *Theor. Comput. Sci. 781* (2019), 3–23. 21, 41, 42, 140

[7] Ayala-Rincón, M., Fernández, M., Ferreira-Silva, G., Kutsia, T., and Nantes-Sobrinho, D. Certified First-Order AC-Unification and Applications. *Submitted to J. of Autom. Reasoning* (2023). 3, 47

[8] Ayala-Rincón, M., Fernández, M., Ferreira-Silva, G., Kutsia, T., and Nantes-Sobrinho, D. Nominal AC-Matching. In *Intelligent Computer Mathematics - 16th International Conference, (CICM)* (2023), vol. 14101 of *Lecture Notes in Computer Science*, Springer, pp. 53–68. 4, 13, 21, 25, 94, 112

[9] Ayala-Rincón, M., Fernández, M., Ferreira-Silva, G., and Nantes-Sobrinho, D. A Certified Algorithm for AC-Unification. In *7th International Conference on Formal Structures for Computation and Deduction, (FSCD)* (Dagstuhl,

2022), vol. 228 of *LIPIcs*, Leibniz-Zentrum für Informatik, pp. 8:1–8:21. 3, 10, 25, 47, 97, 139

[10] Ayala-Rincón, M., Fernández, M., Nantes-Sobrinho, D., and Vale, D. On Solving Nominal Disunification Constraints. In *Proceedings of the 14th Workshop on Logical and Semantic Frameworks with Applications, (LSFA)* (2019), vol. 348 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 3–22. 143

[11] Ayala-Rincón, M., Fernández, M., and Oliveira, A. C. R. *Completeness in PVS of a Nominal Unification Algorithm. ENTCS 323* (2016), 57–74. 25, 30, 31, 32, 43, 96, 104, 139

[12] Ayala-Rincón, M., Fernández, M., Silva, G. F., and Nantes-Sobrinho, D. A Certified Functional Nominal C-Unification Algorithm. In *Logic-Based Program Synthesis and Transformation - 29th International Symposium, (LOPSTR)* (2019), vol. 12042 of *Lecture Notes in Computer Science*, Springer, pp. 123–138. 24, 31, 32, 34, 36, 37, 43, 139, 141

[13] Ayala-Rincón, M., and Ferreira-Silva, G. Why We Need Structured Proofs in Mathematics. In *Workshop on Natural Formal Mathematics - NatFoM* (2023). 21

[14] Baader, F. The Theory of Idempotent Semigroups is of Unification Type Zero. *J. of Autom. Reasoning 2*, 3 (1986), 283–286. 9

[15] Baader, F., and Nipkow, T. *Term Rewriting and All That.* Cambridge University Press, 1998. 1, 138

[16] Baader, F., and Schulz, K. U. Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures. *J. of Sym. Computation 21*, 2 (1996), 211–243. 143

[17] Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al. *The Coq proof assistant reference manual: Version 6.1*, 1997. 23

[18] Baumgartner, A., Kutsia, T., Levy, J., and Villaret, M. Nominal Anti-Unification. In *Proceedings of the 28th International Workshop on Unification, (UNIF)* (2014), pp. 62–68. 143

[19] Baxter, L. D. *The Complexity of Unification.* PhD thesis, University of Waterloo, Ontario, Canada, 1976. 138

[20] Benanav, D., Kapur, D., and Narendran, P. Complexity of Matching Problems. *J. of Sym. Computation 3*, 1/2 (1987), 203–216. 9, 138

[21] Boudet, A. Competing for the AC-Unification Race. *J. of Autom. Reasoning 11*, 2 (1993), 185–212. 92, 139, 142

[22] Boudet, A., and Contejean, E. AC-Unification of Higher-Order Patterns. In *3rd International Conference on Principles and Practice of Constraint Programming (CP)* (1997), vol. 1330 of *LNCS*, Springer, pp. 267–281. 133, 135, 137, 143

[23] Boudet, A., and Contejean, E. Combining Pattern E-Unification Algorithms. In *Rewriting Techniques and Applications, 12th International Conference, (RTA)* (2001), vol. 2051 of *LNCS*, Springer, pp. 63–76. 143

[24] Boudet, A., Contejean, E., and Devie, H. A New AC Unification Algorithm with an Algorithm for Solving Systems of Diophantine Equations. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science (LICS)* (Washington, 1990), IEEE Computer Society, pp. 289–299. 135, 139

[25] Calvès, C. *Complexity and Implementation of Nominal Algorithms.* PhD Thesis, King's College London, 2010. 139

[26] Calvès, C., and Fernández, M. A Polynomial Nominal Unification Algorithm. *Theor. Comput. Sci. 403*, 2-3 (2008), 285–306. 139

[27] Calvès, C., and Fernández, M. Matching and alpha-equivalence check for nominal terms. *J. Comput. Syst. Sci. 76*, 5 (2010), 283–301. 21, 41, 42, 139

[28] Cheney, J. Relating Nominal and Higher-order Pattern Unification. In *Proc. of the 19th international workshop on Unification, (UNIF)* (2005), pp. 104–119. 133, 137, 143

[29] Cheney, J., and Urban, C. alpha-Prolog: A Logic Programming Language with Names, Binding and α-Equivalence. In *Logic Programming, 20th International Conference, (ICLP)* (2004), vol. 3132 of *LNCS*, Springer, pp. 269–283. 139, 143

[30] Clausen, M., and Fortenbacher, A. Efficient Solution of Linear Diophantine Equations. *J. of Sym. Computation 8*, 1-2 (1989), 201–216. 50, 92, 142

[31] Contejean, E. A Certified AC Matching Algorithm. In *Rewriting Techniques and Applications, 15th International Conference, (RTA)* (Berlin, Heidelberg, 2004), vol. 3091 of *LNCS*, Springer, pp. 70–84. 10, 139, 143

[32] Crow, J., Owre, S., Rushby, J., Shankar, N., and Stringer-Calvert, D. Evaluating, testing, and animating PVS specifications. Tech. rep., Computer Science Laboratory, SRI International, 2001. 25

[33] de Carvalho Segundo, W. *Nominal Equational Problems Modulo Associativity, Commutativity and Associativity-Commutativity.* PhD thesis, Universidade de Brasília, 2019. 139

[34] de Moura, L., Kong, S., Avigad, J., Van Doorn, F., and von Raumer, J. The Lean Theorem Prover (system description). In *International Conference on Automated Deduction* (2015), Springer, pp. 378–388. 23

[35] Dummit, D. S., and Foote, R. M. *Abstract Algebra.* Wiley Hoboken, 2004. 113

[36] Dutle, A., Muñoz, C. A., Narkawicz, A., and Butler, R. W. Software validation via model animation. In *Tests and Proofs - 9th International Conference, (TAP)* (2015), vol. 9154 of *LNCS*, Springer, pp. 92–108. 25

[37] Eker, S. Associative-Commutative Rewriting on Large Terms. In *Proc. of the 14th International Conference on Rewriting Techniques and Applications, (RTA)* (2003), vol. 2706 of *LNCS*, Springer, pp. 14–29. 2

[38] Eker, S. Associative unification in Maude. *J. Log. Algebraic Methods Program. 126* (2022), 100747. 9

[39] Fages, F. Associative-Commutative Unification. In *Proceedings 7th International Conference on Automated Deduction (CADE)* (Heidelberg, 1984), vol. 170 of *LNCS*, Springer, pp. 194–208. 3, 9, 138

[40] Fages, F. Associative-Commutative Unification. *J. of Sym. Computation 3*, 3 (1987), 257–275. 3, 9, 13, 51, 59, 138

[41] Fernández, M., and Gabbay, M. *Nominal Rewriting. Information and Computation 205*, 6 (2007), 917–965. 2, 41

[42] Herbrand, J. *Recherces sur la théorie de la démonstracion.* PhD thesis, University of Paris, 1930. 138

[43] Huet, G. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω.* PhD thesis, Université Paris VII, 1976. 138

[44] Jouannaud, J., and Kirchner, C. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. In *Computational Logic - Essays in Honor of Alan Robinson* (1991), The MIT Press, pp. 257–321. 1

[45] Kamareddine, F., and Ríos, A. A Lambda-Calculus à la de Bruijn with Explicit Substitutions. In *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, (PLILP)* (1995), M. V. Hermenegildo and S. D. Swierstra, Eds., vol. 982 of *Lecture Notes in Computer Science*, Springer, pp. 45–62. 2

[46] Kapur, D., and Narendran, P. Matching, unification and complexity. *SIGSAM Bull. 21*, 4 (1987), 6–9. 9

[47] Kapur, D., and Narendran, P. Complexity of Unification Problems with Associative-Commutative Operators. *J. of Autom. Reasoning 9*, 2 (1992), 261–288. 9

[48] Kapur, D., and Narendran, P. Double-exponential Complexity of Computing a Complete Set of AC-Unifiers. In *Proceedings 7th Annual Symposium on Logic in Computer Science (LICS)* (Washington, 1992), IEEE Computer Society, pp. 11–21. 138

[49] Klíma, O. Unification Modulo Associativity and Idempotency Is NP-complete. In *Mathematical Foundations of Computer Science, 27th International Symposium, (MFCS)* (2002), vol. 2420 of *Lecture Notes in Computer Science*, Springer, pp. 423–432. 9

[50] Kumar, R., and Norrish, M. (Nominal) Unification by Recursive Descent with Triangular Substitutions. In *Interactive Theorem Proving, (ITP)* (2010), vol. 6172 of *Lecture Notes in Computer Science*, Springer, pp. 51–66. 7

[51] Lamport, L. How to Write a Proof. *The American mathematical monthly 102*, 7 (1995), 600–608. 21, 73

[52] Lamport, L. How to Write a 21st Century Proof. *Journal of fixed point theory and applications 11*, 1 (2012), 43–63. 21, 22, 23, 73

[53] Levy, J., and Villaret, M. An Efficient Nominal Unification Algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, (RTA)* (2010), vol. 6 of *LIPIcs*, pp. 209–226. 21, 135, 139

[54] Levy, J., and Villaret, M. Nominal Unification from a Higher-Order Perspective. *ACM Trans. Comput. Log. 13*, 2 (2012), 10:1–10:31. 133, 134, 137, 143

[55] Makanin, G. S. The Problem of Solvability of Equations in a Free Semigroup. *Matematicheskii Sbornik 145*, 2 (1977), 147–236. 9

[56] Martelli, A., and Montanari, U. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2 (1982), 258–282. 9, 138

[57] Messner, F., Parsert, J., Schöpf, J., and Sternagel, C. A Formally Verified Solver for Homogeneous Linear Diophantine Equations. In *Interactive Theorem Proving - 9th International Conference, (ITP)* (Cham, 2018), vol. 10895 of *LNCS*, Springer, pp. 441–458. 139

[58] Muñoz, C. Rapid prototyping in PVS. Tech. Rep. NASA/CR-2003-212418, NIA-2003-03, NASA Langley Research Center (NIA), 2003. 24

[59] Nadathur, G. The Suspension Notation for Lambda Terms and its Use in Metalanguage Implementations. In *9th Workshop on Logic, Language, Information and Computation, (WoLLIC)* (2002), vol. 67 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 35–48. 2

[60] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL: A Proof Assistant For Higher-Order Logic*, vol. 2283. Springer Science & Business Media, 2002. 23

[61] Oliveira, A. C. R. *Unificação, confluência e tipos com interseção para sistemas de reescrita nominal.* PhD thesis, Universidade de Brasília, 2016. 139

[62] Owre, S., Rushby, J., and Shankar, N. PVS: A Prototype Verification System. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction* (Berlin, Heidelberg, 1992), vol. 607 of *LNCS*, Springer, pp. 748–752. 23, 139

[63] Paterson, M., and Wegman, M. N. Linear Unification. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing* (1976), ACM, pp. 181–186. 9, 138

[64] Pitts, A. M. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013. 2

[65] Qian, Z., and Wang, K. Modular AC Unification of Higher-Order Patterns. In *Constraints in Computational Logics - 1st International Conference, (CCL)* (1994), vol. 845 of *Lecture Notes in Computer Science*, Springer, pp. 105–120. 135

[66] Robinson, J. A. A machine-oriented logic based on the resolution principle. *J. of the ACM 12*, 1 (1965), 23–41. 9, 138

[67] Schmidt-Schauss, M. Unification under Associativity and Idempotence is of Type Nullary. *J. of Autom. Reasoning 2*, 3 (1986), 277–281. 9

[68] Schmidt-Schauss, M. Unification in a Combination of Arbitrary Disjoint Equational Theories. *J. of Sym. Computation 8*, 1/2 (1989), 51–99. 143

[69] Schmidt-Schauss, M., Kutsia, T., Levy, J., and Villaret, M. Nominal Unification of Higher Order Expressions with Recursive Let. In *Logic-Based Program Synthesis and Transformation - 26th International Symposium, (LOPSTR)* (2016), vol. 10184 of *Lecture Notes in Computer Science*, Springer, pp. 328–344. 139

[70] Shankar, N. Abstraction Engineering with the Prototype Verification System (PVS). Talk given for the Topos Institute Colloquium. Link: `https://www.youtube.com/watch?v=MHf07no09KA`, 2013. 23

[71] Silva, G. F. Why We Need Structured Proofs, 2020. Blog post available at `https://medium.com/@gabrielferreirasilva/why-we-need-structured-proofs-in-mathematics-34a3034f2f90`. 21

[72] Stickel, M. E. A Complete Unification Algorithm for Associative-Commutative Functions. In *Advanced Papers of the Fourth International Joint Conference on Artificial Intelligence* (1975), pp. 71–76. 3, 9, 12, 50, 138

[73] Stickel, M. E. A Unification Algorithm for Associative-Commutative Functions. *J. of the ACM 28*, 3 (1981), 423–434. 3, 9, 10, 49, 92, 138

[74] Urban, C. Nominal Unification Revisited. In *Proceedings 24th International Workshop on Unification, (UNIF)* (2010), vol. 42 of *EPTCS*, pp. 1–11. 139

[75] Urban, C., Pitts, A. M., and Gabbay, M. Nominal Unification. *Theor. Comput. Sci. 323*, 1-3 (2004), 473–497. 3, 21, 31, 90, 116, 139

Appendix A

Generating All Solutions To $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$

We made a Python script to output all the solutions of $f(X, W) \approx^? f(\pi \cdot X, \pi \cdot Y)$. The results below generate all the solutions computed by taking branches 3 or 7 at most 3 times. We choose this bound since we believe it is enough to "give a feel" for the solutions computed while still not being too large, but the script could be easily adapted for other values other than 3. The output of the script, first in a non-verbose manner and then in a verbose manner, is shown below:

The solution of path [1] is: $(\emptyset, \sigma_{B1}, \pi X_0 \approx^? X_0)$
The solution of path [2] is: $(\emptyset, \sigma_{B2}, \emptyset)$
The solution of path [6] is: $(\emptyset, \sigma_{B6}, \emptyset)$
The solution of path [3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}, \pi X_1 \approx^? X_1)$
The solution of path [3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}, \emptyset)$
The solution of path [3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}, \emptyset)$
The solution of path [7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}, \pi X_1 \approx^? X_1)$
The solution of path [7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}, \emptyset)$
The solution of path [7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}, \emptyset)$
The solution of path [3, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B3}, \pi X_2 \approx^? X_2)$
The solution of path [3, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B3}, \emptyset)$
The solution of path [3, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B3}, \emptyset)$
The solution of path [3, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B3}, \pi X_2 \approx^? X_2)$
The solution of path [3, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B3}, \emptyset)$
The solution of path [3, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B3}, \emptyset)$
The solution of path [7, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B7}, \pi X_2 \approx^? X_2)$
The solution of path [7, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B7}, \emptyset)$
The solution of path [7, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B7}, \emptyset)$

The solution of path [7, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B7}, \pi X_2 \approx^? X_2)$
The solution of path [7, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B7}, \emptyset)$
The solution of path [7, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B7}, \emptyset)$
The solution of path [3, 3, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B3}\sigma_{B3}, \pi X_3 \approx^? X_3)$
The solution of path [3, 3, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B3}\sigma_{B3}, \emptyset)$
The solution of path [3, 3, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B3}\sigma_{B3}, \emptyset)$
The solution of path [3, 3, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B3}\sigma_{B3}, \pi X_3 \approx^? X_3)$
The solution of path [3, 3, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B3}\sigma_{B3}, \emptyset)$
The solution of path [3, 3, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B3}\sigma_{B3}, \emptyset)$
The solution of path [3, 7, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B7}\sigma_{B3}, \pi X_3 \approx^? X_3)$
The solution of path [3, 7, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B7}\sigma_{B3}, \emptyset)$
The solution of path [3, 7, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B7}\sigma_{B3}, \emptyset)$
The solution of path [3, 7, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B7}\sigma_{B3}, \pi X_3 \approx^? X_3)$
The solution of path [3, 7, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B7}\sigma_{B3}, \emptyset)$
The solution of path [3, 7, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B7}\sigma_{B3}, \emptyset)$
The solution of path [7, 3, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B3}\sigma_{B7}, \pi X_3 \approx^? X_3)$
The solution of path [7, 3, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B3}\sigma_{B7}, \emptyset)$
The solution of path [7, 3, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B3}\sigma_{B7}, \emptyset)$
The solution of path [7, 3, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B3}\sigma_{B7}, \pi X_3 \approx^? X_3)$
The solution of path [7, 3, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B3}\sigma_{B7}, \emptyset)$
The solution of path [7, 3, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B3}\sigma_{B7}, \emptyset)$
The solution of path [7, 7, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B7}\sigma_{B7}, \pi X_3 \approx^? X_3)$
The solution of path [7, 7, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B7}\sigma_{B7}, \emptyset)$
The solution of path [7, 7, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B7}\sigma_{B7}, \emptyset)$
The solution of path [7, 7, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B7}\sigma_{B7}, \pi X_3 \approx^? X_3)$
The solution of path [7, 7, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B7}\sigma_{B7}, \emptyset)$
The solution of path [7, 7, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B7}\sigma_{B7}, \emptyset)$
The solution of path [1] is: $(\emptyset, \sigma_{B1}, \pi X_0 \approx^? X_0)$ where σ_{B1}:

$$X_0 \mapsto Id * X_0$$
$$Y_0 \mapsto Id * Y_0$$
$$W_0 \mapsto \pi^1 * Y_0$$

The solution of path [2] is: $(\emptyset, \sigma_{B2}, \emptyset)$ where σ_{B2}:

$$\begin{aligned} X_0 &\mapsto \pi^1 * Y_0 \\ Y_0 &\mapsto Id * Y_0 \\ W_0 &\mapsto \pi^2 * Y_0 \end{aligned}$$

The solution of path [6] is: $(\emptyset, \sigma_{B6}, \emptyset)$ where σ_{B6}:

$$\begin{aligned} X_0 &\mapsto Id * X_0 \\ Y_0 &\mapsto f(\pi^{-1} Z_1, \pi^{-1} X_0) \\ W_0 &\mapsto f(Id^1 Z_1, \pi^1 X_0) \end{aligned}$$

The solution of path [3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}, \pi X_1 \approx^? X_1)$ where $\sigma_{B1}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f(Id^1 Y_1, Id^1 X_1) \\ Y_0 &\mapsto \pi^{-1} * Y_1 \\ W_0 &\mapsto \pi^1 * Y_1 \end{aligned}$$

The solution of path [3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}, \emptyset)$ where $\sigma_{B2}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f([Id^1, \pi^1] Y_1) \\ Y_0 &\mapsto \pi^{-1} * Y_1 \\ W_0 &\mapsto \pi^2 * Y_1 \end{aligned}$$

The solution of path [3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}, \emptyset)$ where $\sigma_{B6}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1} Z_2, [\pi^{-1}, Id^1] X_1) \\ Y_0 &\mapsto f(\pi^{-2} Z_2, \pi^{-2} X_1) \\ W_0 &\mapsto f(Id^1 Z_2, \pi^1 X_1) \end{aligned}$$

The solution of path [7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}, \pi X_1 \approx^? X_1)$ where $\sigma_{B1}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f(Id^1 Y_1, Id^1 X_1) \\ Y_0 &\mapsto f(\pi^{-1} Z_1, \pi^{-1} Y_1) \\ W_0 &\mapsto f(\pi^1 Y_1, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}, \emptyset)$ where $\sigma_{B2}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f([Id^1, \pi^1]Y_1) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-1}Y_1) \\ W_0 &\mapsto f(\pi^2 Y_1, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}, \emptyset)$ where $\sigma_{B6}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-1}, Id^1]X_1) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-2}X_1) \\ W_0 &\mapsto f(Id^1 Z_2, \pi^1 X_1, Id^1 Z_1) \end{aligned}$$

The solution of path [3, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B3}, \pi X_2 \approx^? X_2)$ where $\sigma_{B1}\sigma_{B3}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-1}, Id^1]Y_2, Id^1 X_2) \\ Y_0 &\mapsto \pi^{-2} * Y_2 \\ W_0 &\mapsto \pi^1 * Y_2 \end{aligned}$$

The solution of path [3, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B3}, \emptyset)$ where $\sigma_{B2}\sigma_{B3}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-1}, Id^1, \pi^1]Y_2) \\ Y_0 &\mapsto \pi^{-2} * Y_2 \\ W_0 &\mapsto \pi^2 * Y_2 \end{aligned}$$

The solution of path [3, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B3}, \emptyset)$ where $\sigma_{B6}\sigma_{B3}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1]X_2) \\ Y_0 &\mapsto f(\pi^{-3}Z_3, \pi^{-3}X_2) \\ W_0 &\mapsto f(Id^1 Z_3, \pi^1 X_2) \end{aligned}$$

The solution of path [3, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B3}, \pi X_2 \approx^? X_2)$ where $\sigma_{B1}\sigma_{B7}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-1}, Id^1]Y_2, Id^1 X_2) \\ Y_0 &\mapsto f(\pi^{-2}Z_2, \pi^{-2}Y_2) \\ W_0 &\mapsto f(\pi^1 Y_2, Id^1 Z_2) \end{aligned}$$

The solution of path [3, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B3}, \emptyset)$ where $\sigma_{B2}\sigma_{B7}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-1}, Id^1, \pi^1]Y_2) \\ Y_0 &\mapsto f(\pi^{-2}Z_2, \pi^{-2}Y_2) \\ W_0 &\mapsto f(\pi^2 Y_2, Id^1 Z_2) \end{aligned}$$

The solution of path [3, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B3}, \emptyset)$ where $\sigma_{B6}\sigma_{B7}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1]X_2) \\ Y_0 &\mapsto f(\pi^{-2}Z_2, \pi^{-3}Z_3, \pi^{-3}X_2) \\ W_0 &\mapsto f(Id^1 Z_3, \pi^1 X_2, Id^1 Z_2) \end{aligned}$$

The solution of path [7, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B7}, \pi X_2 \approx^? X_2)$ where $\sigma_{B1}\sigma_{B3}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-1}, Id^1]Y_2, Id^1 X_2) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Y_2) \\ W_0 &\mapsto f(\pi^1 Y_2, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B7}, \emptyset)$ where $\sigma_{B2}\sigma_{B3}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-1}, Id^1, \pi^1]Y_2) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Y_2) \\ W_0 &\mapsto f(\pi^2 Y_2, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B7}, \emptyset)$ where $\sigma_{B6}\sigma_{B3}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1]X_2) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-3}Z_3, \pi^{-3}X_2) \\ W_0 &\mapsto f(Id^1 Z_3, \pi^1 X_2, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B7}, \pi X_2 \approx^? X_2)$ where $\sigma_{B1}\sigma_{B7}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-1}, Id^1]Y_2, Id^1 X_2) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-2}Y_2) \\ W_0 &\mapsto f(\pi^1 Y_2, Id^1 Z_2, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B7}, \emptyset)$ where $\sigma_{B2}\sigma_{B7}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-1}, Id^1, \pi^1]Y_2) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-2}Y_2) \\ W_0 &\mapsto f(\pi^2 Y_2, Id^1 Z_2, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B7}, \emptyset)$ where $\sigma_{B6}\sigma_{B7}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1]X_2) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-3}Z_3, \pi^{-3}X_2) \\ W_0 &\mapsto f(Id^1 Z_3, \pi^1 X_2, Id^1 Z_2, Id^1 Z_1) \end{aligned}$$

The solution of path [3, 3, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B3}\sigma_{B3}, \pi X_3 \approx^? X_3)$ where $\sigma_{B1}\sigma_{B3}\sigma_{B3}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-2}, \pi^{-1}, Id^1]Y_3, Id^1 X_3) \\ Y_0 &\mapsto \pi^{-3} * Y_3 \\ W_0 &\mapsto \pi^1 * Y_3 \end{aligned}$$

The solution of path [3, 3, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B3}\sigma_{B3}, \emptyset)$ where $\sigma_{B2}\sigma_{B3}\sigma_{B3}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-2}, \pi^{-1}, Id^1, \pi^1]Y_3) \\ Y_0 &\mapsto \pi^{-3} * Y_3 \\ W_0 &\mapsto \pi^2 * Y_3 \end{aligned}$$

The solution of path [3, 3, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B3}\sigma_{B3}, \emptyset)$ where $\sigma_{B6}\sigma_{B3}\sigma_{B3}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-3}, \pi^{-2}, \pi^{-1}]Z_4, [\pi^{-3}, \pi^{-2}, \pi^{-1}, Id^1]X_3) \\ Y_0 &\mapsto f(\pi^{-4}Z_4, \pi^{-4}X_3) \\ W_0 &\mapsto f(Id^1 Z_4, \pi^1 X_3) \end{aligned}$$

The solution of path [3, 3, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B3}\sigma_{B3}, \pi X_3 \approx^? X_3)$ where $\sigma_{B1}\sigma_{B7}\sigma_{B3}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1]Y_3, Id^1 X_3) \\ Y_0 &\mapsto f(\pi^{-3}Z_3, \pi^{-3}Y_3) \\ W_0 &\mapsto f(\pi^1 Y_3, Id^1 Z_3) \end{aligned}$$

The solution of path [3, 3, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B3}\sigma_{B3}, \emptyset)$ where $\sigma_{B2}\sigma_{B7}\sigma_{B3}\sigma_{B3}$:

$$\begin{aligned}
X_0 &\mapsto f([\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1, \pi^1]Y_3)\\
Y_0 &\mapsto f(\pi^{-3}Z_3, \pi^{-3}Y_3)\\
W_0 &\mapsto f(\pi^2 Y_3, Id^1 Z_3)
\end{aligned}$$

The solution of path [3, 3, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B3}\sigma_{B3}, \emptyset)$ where $\sigma_{B6}\sigma_{B7}\sigma_{B3}\sigma_{B3}$:

$$\begin{aligned}
X_0 &\mapsto f([\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-3}, \pi^{-2}, \pi^{-1}]Z_4, [\pi^{-3}, \pi^{-2}, \pi^{-1}, Id^1]X_3)\\
Y_0 &\mapsto f(\pi^{-3}Z_3, \pi^{-4}Z_4, \pi^{-4}X_3)\\
W_0 &\mapsto f(Id^1 Z_4, \pi^1 X_3, Id^1 Z_3)
\end{aligned}$$

The solution of path [3, 7, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B7}\sigma_{B3}, \pi X_3 \approx^? X_3)$ where $\sigma_{B1}\sigma_{B3}\sigma_{B7}\sigma_{B3}$:

$$\begin{aligned}
X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}, Id^1]Y_3, Id^1 X_3)\\
Y_0 &\mapsto f(\pi^{-2}Z_2, \pi^{-3}Y_3)\\
W_0 &\mapsto f(\pi^1 Y_3, Id^1 Z_2)
\end{aligned}$$

The solution of path [3, 7, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B7}\sigma_{B3}, \emptyset)$ where $\sigma_{B2}\sigma_{B3}\sigma_{B7}\sigma_{B3}$:

$$\begin{aligned}
X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}, Id^1, \pi^1]Y_3)\\
Y_0 &\mapsto f(\pi^{-2}Z_2, \pi^{-3}Y_3)\\
W_0 &\mapsto f(\pi^2 Y_3, Id^1 Z_2)
\end{aligned}$$

The solution of path [3, 7, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B7}\sigma_{B3}, \emptyset)$ where $\sigma_{B6}\sigma_{B3}\sigma_{B7}\sigma_{B3}$:

$$\begin{aligned}
X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-3}, \pi^{-2}, \pi^{-1}]Z_4, [\pi^{-3}, \pi^{-2}, \pi^{-1}, Id^1]X_3)\\
Y_0 &\mapsto f(\pi^{-2}Z_2, \pi^{-4}Z_4, \pi^{-4}X_3)\\
W_0 &\mapsto f(Id^1 Z_4, \pi^1 X_3, Id^1 Z_2)
\end{aligned}$$

The solution of path [3, 7, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B7}\sigma_{B3}, \pi X_3 \approx^? X_3)$ where $\sigma_{B1}\sigma_{B7}\sigma_{B7}\sigma_{B3}$:

$$\begin{aligned}
X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1]Y_3, Id^1 X_3)\\
Y_0 &\mapsto f(\pi^{-2}Z_2, \pi^{-3}Z_3, \pi^{-3}Y_3)\\
W_0 &\mapsto f(\pi^1 Y_3, Id^1 Z_3, Id^1 Z_2)
\end{aligned}$$

The solution of path [3, 7, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B7}\sigma_{B3}, \emptyset)$ where $\sigma_{B2}\sigma_{B7}\sigma_{B7}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1, \pi^1]Y_3) \\ Y_0 &\mapsto f(\pi^{-2}Z_2, \pi^{-3}Z_3, \pi^{-3}Y_3) \\ W_0 &\mapsto f(\pi^2 Y_3, Id^1 Z_3, Id^1 Z_2) \end{aligned}$$

The solution of path [3, 7, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B7}\sigma_{B3}, \emptyset)$ where $\sigma_{B6}\sigma_{B7}\sigma_{B7}\sigma_{B3}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-3}, \pi^{-2}, \pi^{-1}]Z_4, [\pi^{-3}, \pi^{-2}, \pi^{-1}, Id^1]X_3) \\ Y_0 &\mapsto f(\pi^{-2}Z_2, \pi^{-3}Z_3, \pi^{-4}Z_4, \pi^{-4}X_3) \\ W_0 &\mapsto f(Id^1 Z_4, \pi^1 X_3, Id^1 Z_3, Id^1 Z_2) \end{aligned}$$

The solution of path [7, 3, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B3}\sigma_{B7}, \pi X_3 \approx^? X_3)$ where $\sigma_{B1}\sigma_{B3}\sigma_{B3}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-2}, \pi^{-1}, Id^1]Y_3, Id^1 X_3) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-3}Y_3) \\ W_0 &\mapsto f(\pi^1 Y_3, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 3, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B3}\sigma_{B7}, \emptyset)$ where $\sigma_{B2}\sigma_{B3}\sigma_{B3}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-2}, \pi^{-1}, Id^1, \pi^1]Y_3) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-3}Y_3) \\ W_0 &\mapsto f(\pi^2 Y_3, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 3, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B3}\sigma_{B7}, \emptyset)$ where $\sigma_{B6}\sigma_{B3}\sigma_{B3}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-3}, \pi^{-2}, \pi^{-1}]Z_4, [\pi^{-3}, \pi^{-2}, \pi^{-1}, Id^1]X_3) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-4}Z_4, \pi^{-4}X_3) \\ W_0 &\mapsto f(Id^1 Z_4, \pi^1 X_3, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 3, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B3}\sigma_{B7}, \pi X_3 \approx^? X_3)$ where $\sigma_{B1}\sigma_{B7}\sigma_{B3}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f([\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1]Y_3, Id^1 X_3) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-3}Z_3, \pi^{-3}Y_3) \\ W_0 &\mapsto f(\pi^1 Y_3, Id^1 Z_3, Id^1 Z_1) \end{aligned}$$

The solution of path [7, 3, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B3}\sigma_{B7}, \emptyset)$ where $\sigma_{B2}\sigma_{B7}\sigma_{B3}\sigma_{B7}$:

$$\begin{aligned}
X_0 &\mapsto f([\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1, \pi^1]Y_3) \\
Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-3}Z_3, \pi^{-3}Y_3) \\
W_0 &\mapsto f(\pi^2 Y_3, Id^1 Z_3, Id^1 Z_1)
\end{aligned}$$

The solution of path [7, 3, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B3}\sigma_{B7}, \emptyset)$ where $\sigma_{B6}\sigma_{B7}\sigma_{B3}\sigma_{B7}$:

$$\begin{aligned}
X_0 &\mapsto f([\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-3}, \pi^{-2}, \pi^{-1}]Z_4, [\pi^{-3}, \pi^{-2}, \pi^{-1}, Id^1]X_3) \\
Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-3}Z_3, \pi^{-4}Z_4, \pi^{-4}X_3) \\
W_0 &\mapsto f(Id^1 Z_4, \pi^1 X_3, Id^1 Z_3, Id^1 Z_1)
\end{aligned}$$

The solution of path [7, 7, 3, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B3}\sigma_{B7}\sigma_{B7}, \pi X_3 \approx^? X_3)$ where $\sigma_{B1}\sigma_{B3}\sigma_{B7}\sigma_{B7}$:

$$\begin{aligned}
X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}, Id^1]Y_3, Id^1 X_3) \\
Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-3}Y_3) \\
W_0 &\mapsto f(\pi^1 Y_3, Id^1 Z_2, Id^1 Z_1)
\end{aligned}$$

The solution of path [7, 7, 3, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B3}\sigma_{B7}\sigma_{B7}, \emptyset)$ where $\sigma_{B2}\sigma_{B3}\sigma_{B7}\sigma_{B7}$:

$$\begin{aligned}
X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}, Id^1, \pi^1]Y_3) \\
Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-3}Y_3) \\
W_0 &\mapsto f(\pi^2 Y_3, Id^1 Z_2, Id^1 Z_1)
\end{aligned}$$

The solution of path [7, 7, 3, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B3}\sigma_{B7}\sigma_{B7}, \emptyset)$ where $\sigma_{B6}\sigma_{B3}\sigma_{B7}\sigma_{B7}$:

$$\begin{aligned}
X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-3}, \pi^{-2}, \pi^{-1}]Z_4, [\pi^{-3}, \pi^{-2}, \pi^{-1}, Id^1]X_3) \\
Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-4}Z_4, \pi^{-4}X_3) \\
W_0 &\mapsto f(Id^1 Z_4, \pi^1 X_3, Id^1 Z_2, Id^1 Z_1)
\end{aligned}$$

The solution of path [7, 7, 7, 1] is: $(\emptyset, \sigma_{B1}\sigma_{B7}\sigma_{B7}\sigma_{B7}, \pi X_3 \approx^? X_3)$ where $\sigma_{B1}\sigma_{B7}\sigma_{B7}\sigma_{B7}$:

$$\begin{aligned}
X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1]Y_3, Id^1 X_3) \\
Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-3}Z_3, \pi^{-3}Y_3) \\
W_0 &\mapsto f(\pi^1 Y_3, Id^1 Z_3, Id^1 Z_2, Id^1 Z_1)
\end{aligned}$$

The solution of path [7, 7, 7, 2] is: $(\emptyset, \sigma_{B2}\sigma_{B7}\sigma_{B7}\sigma_{B7}, \emptyset)$ where $\sigma_{B2}\sigma_{B7}\sigma_{B7}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-2}, \pi^{-1}, Id^1, \pi^1]Y_3) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-3}Z_3, \pi^{-3}Y_3) \\ W_0 &\mapsto f(\pi^2Y_3, Id^1Z_3, Id^1Z_2, Id^1Z_1) \end{aligned}$$

The solution of path [7, 7, 7, 6] is: $(\emptyset, \sigma_{B6}\sigma_{B7}\sigma_{B7}\sigma_{B7}, \emptyset)$ where $\sigma_{B6}\sigma_{B7}\sigma_{B7}\sigma_{B7}$:

$$\begin{aligned} X_0 &\mapsto f(\pi^{-1}Z_2, [\pi^{-2}, \pi^{-1}]Z_3, [\pi^{-3}, \pi^{-2}, \pi^{-1}]Z_4, [\pi^{-3}, \pi^{-2}, \pi^{-1}, Id^1]X_3) \\ Y_0 &\mapsto f(\pi^{-1}Z_1, \pi^{-2}Z_2, \pi^{-3}Z_3, \pi^{-4}Z_4, \pi^{-4}X_3) \\ W_0 &\mapsto f(Id^1Z_4, \pi^1X_3, Id^1Z_3, Id^1Z_2, Id^1Z_1) \end{aligned}$$