Certified First-Order AC-Unification and Applications

Mauricio Ayala-Rincón[1,2], Maribel Fernández[3], Gabriel Ferreira Silva[1], Temur Kutsia[4], Daniele Nantes-Sobrinho[2,5]

[1]Department of Informatics, University of Brasília, Brasília, Brazil.
[2]Department of Mathematics, University of Brasília, Brasília, Brazil.
[3]Department of Informatics, King's College London, London, UK.
[4]Department of Informatics, Johannes Kepler University, Linz, Austria.
[5]Department of Computing, Imperial College, London, UK.

Contributing authors: ayala@unb.br; maribel.fernandez@kcl.ac.uk; gabrielfsilva1995@gmail.com; kutsia@risc.jku.at; dnantess@ic.ac.uk;

Abstract

AC-unification, i.e., unification modulo Associativity and Commutativity axioms is a key component in rewrite-based programming languages and theorem provers. We have used the PVS proof assistant to specify Stickel's pioneering AC-unification algorithm and proved it to be terminating (using an elaborate lexicographic measure based on Fages' termination proof), sound, and complete. We give a detailed account of the formalisation, including descriptions 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 formalisation has been used to obtain a certified nominal AC-matching algorithm. Also, it could serve as a basis to specify a nominal AC-unification algorithm once this open theoretical problem is solved.

Keywords: AC-Unification, PVS, Certified Algorithms, Formal Methods, Interactive Theorem Proving

1 Introduction

Given terms s and t, syntactic matching is the problem of finding a substitution σ such that $\sigma s = t$, and syntactic unification is the problem of finding a substitution σ such that $\sigma s = \sigma t$. 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 $\sigma s \approx_E \sigma t$ [18]. Similarly, E-matching is the problem of finding a substitution σ such that $\sigma s \approx_E t$.

Unification has practical applications in computer science and mathematics. It is used, for instance, in resolution-based theorem provers, interpreters of logic programming languages such as Prolog, confluence tests based on critical pairs, type-inference procedures, and so on [8]. Since associative and commutative operators are frequently used in programming languages and theorem provers, tools to support reasoning modulo **A**ssociativity and **C**ommutativity axioms are often required. The problem of AC-unification has been widely studied in this context (see [8, 29]).

Stickel [28] was the first to solve unification in the presence of AC-function symbols. He showed how the problem is connected to finding non-negative integral solutions to linear equations and proved that his algorithm was sound, complete, and terminating for a subclass of the general case [28, 29]. 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 to fix the termination proof for the general case [15, 16]. Since then, investigations on solving AC-unification efficiently, on the complexity of AC-unification, and on formalising unification modulo equational theories have been carried out.

Regarding the complexity of AC-unification, Benanav et al. [9] showed that the decision problem for AC-matching is NP-complete and the decision problem for AC-unification is NP-hard. 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 [19]. 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 unifiers, 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. [11] proposed an AC-unification algorithm that explores constraints more efficiently than the standard algorithm. Further, Boudet [10] described and compared an implementation of this algorithm to previous ones. Also, Adi and Kirchner [1] implemented an AC-unification algorithm, proposed benchmarks, and showed that their algorithm improves over previous ones in time and space. An efficient AC-unification algorithm [14] is in use in the programming language Maude.

Regarding formalisations, Ayala-Rincón et al. [3] formalised nominal α-equivalence for associative, commutative and associative-commutative function symbols. This work was done in the nominal setting (see [27]), which encompasses first-order AC-equivalence, but did not consider the AC-unification problem. A formalisation of nominal C-unification, which can also handle nominal C-matching, is also available [4].

In 2004, Contejean [13] gave the first certified AC-matching algorithm in Coq. Additionally, Meßner et al. [23] gave a formally verified solver in Isabelle/HOL for homogeneous linear Diophantine equations, a problem closely related to AC-unification. However, no formalisation of AC-unification was available until recently, when termination, soundness, and completeness of Stickel's AC-unification algorithm [6] was proved using the proof assistant PVS [25].

This paper is an extended and improved version of [6]. We extend [6] by presenting the main lemmas required for the proof of completeness and giving detailed proof for the most complicated one. Additionally, we give a more detailed account of the formalisation: each file is described in depth; a diagram showing the hierarchy of the files is presented; and there is a discussion on the grammar (of terms) we adopted and its consequences. Moreover, the applications of a first-order AC-unification algorithm are explored in more detail: we elaborate on how our simple algorithm can be used as a basis to formalise more efficient algorithms, or to test implemented AC-unification algorithms, or to formalise a nominal AC-unification algorithm (once this open theoretical question is solved), and briefly summarise how the first-order AC-unification algorithm was used to obtain a nominal AC-matching algorithm (see [5]). The most important distinction between this work and [6] is that the proof of completeness presented in [6] contained an unnecessary hypothesis. In this paper, we show why removing this hypothesis from the proof of completeness is non-trivial and how we removed the mentioned hypothesis, honing the proof of completeness.

There is already an extensive nominal unification library in PVS, which we would like to enrich in future work (see Section 9) with a nominal AC-unification algorithm. Additionally, PVS has expressive logic, useful features such as subtyping and effective proof automation. When deciding which AC-unification algorithm to formalise, we looked for concise and well-established algorithms, which led us to select Stickel's algorithm, using Fages' proof of termination. We applied minor modifications to Stickel's AC-unification algorithm in order to avoid mutual recursion (PVS does not allow mutual recursion directly, although this can be emulated using PVS higher-order features, see [26]) and to ease the formalisation.

The paper is organised as follows. Section 2 gives the necessary background, while Section 3 discusses examples of first-order AC-unification. Then, Section 4 explains the certified algorithm. Section 5 discusses how we proved the algorithm's termination, motivating the used lexicographic measure. After that, Section 6 explains the proofs of soundness and completeness, showing how we improved the proof of completeness. Section 7 gives additional information about the PVS formalisation and Section 8 describes applications of our formalisation. Finally, Section 9 concludes the paper. Appendix A gives more details on the proof of termination. We include cyan-coloured hyperlinks (using icon) to specific points of interest of the PVS formalisation, which is available as the "`nominal`" library, part of the PVS NASAlib (NASA PVS Library of Formal Developments).

2 Background

From now on, 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 a set of 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. In general, we represent the constants using the initial lowercase letters of the alphabet. X 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 in future work. 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 $f\langle\langle a, b\rangle, c\rangle$ and its arguments are a, b and c.

Remark 1 (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 6.3.3.*

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 7.1), we have restricted the terms in the unification problem that our algorithm receives to well-formed terms. Excluding pairs is natural since they are 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 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 ($Vars$). *$Vars(t)$ denotes the set of variables occurring in a term t. Similarly, $Vars(P)$ denotes the set of variables occurring in a unification problem 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 6.3.1).

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 introduced 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:*

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

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* 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 substitution as a simultaneous application of nuclear substitutions, although both are correct. The way we defined substitution here is closer to triangular substitutions [20]. 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 $\textsc{append}(\sigma, \delta)$.

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.*

We now define AC-unification 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 all i from 1 to n. When σ is a unifier for P we say that σ* unifies *P.*

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 $\sigma' X \approx \delta\sigma X$, for all variables $X \in V$. In this case, we write $\sigma \leq_V \sigma'$. When V is the set of all variables, we write $\sigma \leq \sigma'$.

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 a 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.*

3 Examples of AC-Unification

3.1 What Makes AC-unification Hard

Let f be an associative-commutative function symbol. Finding a complete set of unifiers for $\{f(X_1, X_2) \approx^? f(Y, a)\}$ 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 second term arguments. Indeed, this strategy would give us only $\sigma_1 = \{X_1 \mapsto Y, X_2 \mapsto a\}$ and $\sigma_2 = \{X_2 \mapsto Y, X_1 \mapsto Y\}$ as solutions, missing for example the substitution $\sigma_3 = \{X_1 \mapsto f(a, W), Y \mapsto f(X_2, W)\}$. The solution σ_3 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 2. *To guarantee the completeness of AC-matching, it is enough to explore all possible pairings of the first term's arguments with the second term's arguments. As the example above shows, this is not enough for AC-Unification. Evidence of the difficulty of AC-unification is that it took eighteen years to obtain the first formalisation of AC-unification (see [6]) despite the fact that Contejean formalised AC-matching in 2004, leaving a formalisation of AC-unification as future work (see [13]).*

3.2 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 [29]) of how we would solve

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

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

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

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

The second step is to connect 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 will soon see, the 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 in Table 1.

Table 1 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 1 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:

$$\begin{aligned} X_1 &= Z_6 + Z_7 \\ X_2 &= Z_2 + Z_4 + 2Z_5 \\ X_3 &= Z_1 + 2Z_3 + Z_4 \\ Y_1 &= Z_3 + Z_4 + Z_5 + Z_7 \\ Y_2 &= Z_1 + Z_2 + 2Z_6. \end{aligned} \tag{1}$$

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, we are left with 69 cases [28]. 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.

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 (1), becomes:

$$P = \{X_1 \approx^? Z_6, X_2 \approx^? Z_4, X_3 \approx^? f(Z_1, Z_4), Y_1 \approx^? Z_4, Y_2 \approx^? f(Z_1, Z_6, Z_6)\}. \quad (2)$$

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:

$$\begin{aligned}
\sigma_1 &= \{Y \mapsto f(b, b), Z \mapsto f(a, X, X)\}. \\
\sigma_2 &= \{Y \mapsto f(Z_2, b, b), Z \mapsto f(a, Z_2, X, X)\}. \\
\sigma_3 &= \{X \mapsto b, Z \mapsto f(a, Y)\}. \\
\sigma_4 &= \{X \mapsto f(Z_6, b), Z \mapsto f(a, Y, Z_6, Z_6)\}.
\end{aligned} \quad (3)$$

Remark 3. *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). 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 4 (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`*.*

3.3 Avoiding Infinite Loops

It is necessary to compose the substeps of solving AC-unification equations with some strategy, as the following example (adapted from [16]) shows.

Example 1 (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:*

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

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:

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

One way of solving the first equation is to decompose it into $\{X_1 \approx^? X_3, X_2 \approx^? X_4\}$, which gets 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 algorithm, we always instantiate the variables that we can before tackling AC-unification pairs.

4 Algorithm

For readability, we present the pseudocode of the algorithms instead of the actual PVS code. We have formalised Algorithm 1 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 5. *In the PVS code notation, this means that the initial call is done with parameters $P = cons((t, s), \text{NIL})$, $\sigma = \text{NIL}$, and $V = Vars(t, s)$.*

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 CHOOSE (line 3), which returns a pair (t, s) and a unification problem P_1, such that $P = \{t \approx^? s\} \cup P_1$. Note that problems are specified as lists in the formalisation, but here we simplify their presentation by using sets. 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. For clarity, Algorithm 1 is presented in OCaml style **match-with** pseudocode, although the actual PVS specification uses an **if-else if-else** structure.

Remark 6. *The algorithm does not check the arity consistency of the input.*

Algorithm 1 Algorithm to Solve an AC-Unification Problem P

1: **procedure** ACUNIF(P, σ, V)
2: **if** nil?(P) **then** $cons(\sigma, \text{NIL})$
3: **else let** $((t, s), P_1) = \text{CHOOSE}(P)$ **in**
4:   **match** t and s **with**
5:    "a" and "a" $\longrightarrow$ ACUNIF(P_1, σ, V)
6:    | "$\langle\rangle$" and "$\langle\rangle$" $\longrightarrow$ ACUNIF(P_1, σ, V)
7:    | "X" and "X" $\longrightarrow$ ACUNIF(P_1, σ, V)
8:    | "X" and "s" such that X not in s $\longrightarrow$
9:    **let** $\sigma_1 = \{X \mapsto s\}$ **in** ACUNIF$(\sigma_1 P_1, \sigma_1 \sigma, V)$
10:    | "t" such that X not in t and "X" $\longrightarrow$
11:    **let** $\sigma_1 = \{X \mapsto t\}$ **in** ACUNIF$(\sigma_1 P_1, \sigma_1 \sigma, V)$
12:    | "$f\ t_1$" and "$f\ s_1$" $\longrightarrow$
13:    **let** $(P_2, \mathit{flag}) = \text{DECOMPOSE}(t_1, s_1)$ **in**
14:    **if** *flag* **then** ACUNIF$(P_2 \cup P_1, \sigma, V)$
15:    **else** NIL
16:    | "$f^{AC}\ t_1$" and "$f^{AC}\ s_1$" $\longrightarrow$
17:    **let** $\mathit{InputLst} = \text{APPLYACSTEP}(P, \text{NIL}, \sigma, V)$,
18:     $\mathit{LstResults} = \text{MAP}(\text{ACUNIF}, \mathit{InputLst})$ **in**
19:    FLATTEN$(\mathit{LstResults})$
20:    | _ $\longrightarrow$ NIL

4.1 Function CHOOSE

The function CHOOSE selects a unification pair from the input problem, avoiding AC-unification pairs if possible. This means that we will only enter on the **case** of line 16 of ACUNIF (see Algorithm 1) 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 needed (the AC-part is the computationally heaviest). Also, it is not a significant deviation from Stickel's algorithm [29].

4.2 Function DECOMPOSE

Suppose the function DECOMPOSE receives two terms t and s and these terms are both pairs. In that case, 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 (NIL, *False*). In Algorithm 1, 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 2. *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$(X, \langle c, d\rangle) = (NIL,\ \mathit{False})$.

4.3 The AC-part of the Algorithm

The AC-part of Algorithm 1 relies on function APPLYACSTEP (Section 4.3.4), which depends on two functions: SOLVEAC (Section 4.3.1) and INSTANTIATESTEP (Section 4.3.3). Since there are multiple possibilities for simplifying each AC-unification pair, APPLYACSTEP will return a list (*InputLst* in Algorithm 1), where each entry of the list corresponds to a branch Algorithm 1 will explore (line 17). 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 18 of Algorithm 1). 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.

4.3.1 Function SOLVEAC

The function SOLVEAC does what was illustrated in the example of Section 3.2. 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 [28]) 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 1 of the Example in Section 3.2 is represented in our code as the matrix D (see Equation 4). 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 [12]) instead of a spanning set would certainly improve the algorithm's efficiency.

$$D = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{4}$$

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 3.2, we mentioned that we should observe two constraints:

- 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.
- an "original variable" that does not represent a variable term cannot be paired with an AC-function application.

As noted by Fages in [16], 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 7 (New Variables Introduced by solveAC)**.** *As mentioned in Remark 1, 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 $max(V)+1, max(V)+2, max(V)+3, \ldots$*

4.3.2 Common Structure of Unification Problems Returned by solveAC

Suppose function solveAC receives the terms u and v as input, 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)).
3. A term s_i is an AC-function headed by f only if the corresponding term t_i is a variable.

4.3.3 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 4.3.2), the variables on the right-hand side are the new variables; in contrast, the variables on the left-hand side are variables that were in the problem before calling solveAC. Indeed, as shown in Example 1, 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 1 from happening, Algorithm 1 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.

4.3.4 Function applyACStep

Function applyACStep relies on functions solveAC and instantiateStep, and is called by Algorithm 1 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.

5 Proving Termination

5.1 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 11, 13, 15 and 16, respectively. Table 2 shows which components do not increase (represented by $\leq$) and which components strictly decrease (represented by $<$) for each recursive call that Algorithm 1 makes.

Definition 11 ($V_{NAC}(P)$). *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 3. *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 12 ($Subterms(P)$). *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:

- $Subterms(a) = \{a\}$.
- $Subterms(Y) = \{Y\}$.
- $Subterms(\langle\rangle) = \{\langle\rangle\}$.
- $Subterms(\langle t_1, t_2\rangle) = Subterms(t_1) \cup Subterms(t_2)$.

- $\mathit{Subterms}(f\ t_1) = \{f\ t_1\} \cup \mathit{Subterms}(t_1)$.
- $\mathit{Subterms}(f^{AC} t_1) = \bigcup_{t_i \in \mathit{Args}(f^{AC} t_1)} \mathit{Subterms}(t_i) \cup \{f^{AC} t_1\}$.

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

Remark 8 (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*

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

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

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

Definition 13 ($V_{>1}(P)$)**.** *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 $\mathit{Subterms}(P)$. The informal meaning is that if $X \in V_{>1}(P)$, then X is an argument to at least two different function symbols.*

Example 4. *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 15.

Definition 14 (Proper Subterms)**.** *If t is not a pair, we define the proper subterms of t, denoted as $\mathit{PSubterms}(t)$ as:*

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

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

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

Definition 15 (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 5. *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 16 (Size of a Unification Problem)**.** *We define the size of a term t recursively as follows:*

- $\mathit{size}(a) = 1$.
- $\mathit{size}(Y) = 1$.

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

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

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

Remark 9 ($s \in AS(t) \implies size(s) < size(t)$)**.** *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 2 Decrease of the components of the lexicographic measure.

Recursive Call	$\|\mathbf{V}_{NAC}(\mathbf{P})\|$	$\|\mathbf{V}_{>1}(\mathbf{P})\|$	$\|AS(\mathbf{P})\|$	$size(\mathbf{P})$
lines 9, 11	$<$			
lines 5, 6, 7, 14	$\leq$	$\leq$	$\leq$	$<$
case 1 - line 18	$\leq$	$<$		
case 2 - line 18	$\leq$	$\leq$	$<$	
case 3 - line 18	$\leq$	$\leq$	$\leq$	$<$

5.2 Proof Sketch for Termination

5.2.1 Non AC Cases

To prove the termination of syntactic unification, we can use a lexicographic measure lex_s consisting of two components: $lex_s = (|\mathit{Vars}(P)|, size(P))$, where $\mathit{Vars}(P)$ 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 1 except AC (line 18) is similar to the proof of termination of syntactic unification, with two caveats.

First, we need to use $|V_{NAC}(P)|$ instead of $|\mathit{Vars}(P)|$ to avoid taking into account the variables that are arguments of the AC-function terms introduced by solveAC (see Section 4.3.2). 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 5, 6, 7, 14), 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.

5.2.2 The AC-case

Our proof of termination for the AC-case uses the components $|V_{>1}(P)|$ and $|AS(P)|$, proposed in [16]. 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 4.3.2, 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 9), 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 6. *In the example of Section 3.2, 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 7 shows.

Example 7 (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 7 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 8.

Example 8 (A case where INSTANTIATESTEP does not increase $|AS|$). *If we change slightly the problem from Example 7 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 were instantiated by INSTANTIATESTEP, it would be instantiated to an AC-function headed by f (see Section 4.3.2) 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 5.1.

The situation described is similar when our unification problem P has multiple equations. Let's say $P = \{t_1 \approx^? s_1, \dots, t_n \approx^? s_n\}$. 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. Finally, the full structured proof of termination for function APPLYACSTEP is shown in Appendix A.

6 Proving Soundness and Completeness

6.1 Nice Inputs

As mentioned, to unify terms t and s we use Algorithm 1 with $P = \{t \approx^? s\}$, $\sigma = id$ and $V = \mathit{Vars}((t, s))$. However, since the parameters of ACUnif may change between the recursive calls, we cannot directly prove soundness (Corollary 5) by induction. We must prove the more general Theorem 4, 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 17.

Definition 17 (Nice input). *Given an input (P, σ, V), we say that this input is nice if:*

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

6.2 Soundness

As mentioned, once we prove Theorem 4, then soundness (Corollary 5) is obtained immediately. To prove Theorem 4, we used Theorem 2 and Theorem 3. Finally, to establish Theorem 2 (soundness of applyACStep), we used Theorem 1 (soundness of solveAC).

Theorem 1 (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 2 (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 $\mathit{dom}(\sigma) \subseteq V$ and that $\mathit{dom}(\sigma) \cap (\mathit{Vars}(P_1) \cup \mathit{Vars}(P_2)) = \emptyset$. Then δ unifies P_1.*

Remark 10. *Hypotheses $\mathit{dom}(\sigma) \subseteq V$ and $\mathit{dom}(\sigma) \cap (\mathit{Vars}(P_1) \cup \mathit{Vars}(P_2)) = \emptyset$ of Theorem 2 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 3 (Soundness of Variable Instantiation). *Suppose that (P, σ, V) is a nice input, $\sigma_1 = \{X \mapsto t\}$, $P = \{X \approx^? t\} \cup P_1$, $X \notin \mathit{Vars}(t)$ 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 4 (Soundness for Nice Inputs). *Let (P, σ, V) be a nice input, and $\delta \in$ ACUnif(P, σ, V). Then, δ unifies P.*

Theorem 4 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 choose (see Algorithm 1). There are two interesting cases. The first case is in lines 17-19 when we only have AC-unification pairs (in that case, we used the soundness of applyACStep, i.e. Theorem 2). The second case happens when we instantiate a variable (lines 8 and 10) and is solved by using Theorem 3.

Corollary 5 (Soundness of ACUNIF). *If $\delta \in$ ACUNIF$(\{t \approx^? s\}, id, Vars((t, s)))$ then δ unifies $t \approx^? s$.*

6.3 Completeness

6.3.1 A Structured Proof of Completeness of SOLVEAC

Theorem 6 is completeness for SOLVEAC. Recalling the structure of a unification problem obtained after APPLYACSTEP (Section 4.3.2), we see that the hypothesis $\delta \subseteq V$ of Theorem 6 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 6 guarantees that the substitution γ will only impact the new variables introduced by SOLVEAC, since $dom(\gamma) \subseteq V_1 - V$. Regarding P_1, γ will only impact the right-hand side of P_1.

We give a structured proof (*à la* Leslie Lamport [21, 22]) of the completeness of SOLVEAC (Theorem 6). 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 6 (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:

$\langle 1 \rangle$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).

$\langle 1 \rangle$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:

$$a_1 X_1 + \ldots + a_m X_m = b_1 Y_1 + \ldots + b_n Y_n.$$

Let $|t|_A$ be the number of times the term A (or some term 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.

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

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

$\langle 1 \rangle$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 $\overrightarrow{Z_1}, \ldots, \overrightarrow{Z_l}$, we have:

$$\begin{aligned} \overrightarrow{n_{A_1}} &= c_{11}\overrightarrow{Z_1} + \ldots + c_{1l}\overrightarrow{Z_l} \\ &\vdots \\ \overrightarrow{n_{A_k}} &= c_{k1}\overrightarrow{Z_1} + \ldots + c_{kl}\overrightarrow{Z_l}. \end{aligned} \tag{5}$$

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

⟨1⟩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:

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

⟨1⟩8. Every column of D_1 has at least one coefficient different than zero.
PROOF:
⟨2⟩1. Let's prove for the arbitrary column j. Recall that the j-th term of the vector $(t_1, \ldots, t_m, s_1, \ldots s_n)$ is associated with column j of D_1. Let's denote by t_j this term.
⟨2⟩2. There exists an A_i such that $|\delta t_j|_{A_i} > 0$.
⟨2⟩3. Analysing the j-th component of i-th equality in Equation 5, 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.

⟨1⟩9. Define γ such that

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

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:

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

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

⟨2⟩3. The variables in $im(\gamma)$ are the variables in $A_1, \ldots, A_k$. These are the variables occurring in δt (see Step ⟨1⟩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 $im(\gamma) \subseteq V_1$.

⟨1⟩10. $\gamma\delta$ unifies P_1.

PROOF:

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

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

PROOF:

⟨3⟩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 $\gamma\delta t_i = \delta t_i$.

⟨3⟩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 $\gamma\delta t'_i = \gamma t'_i$.

⟨2⟩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.

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

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

⟨2⟩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 4.3.1, specially the part about DIOMATRIX2ACSOL . Therefore,

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

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

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

PROOF:

⟨2⟩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 4.3.1.

⟨2⟩2. As proved in Step ⟨1⟩8, D_1 satisfies the first constraint: every column has one coefficient greater than 0.

⟨2⟩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:

⟨3⟩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.

⟨3⟩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.

⟨3⟩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.

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

6.3.2 Completeness of APPLYACSTEP

Theorem 7 is completeness for APPLYACSTEP.

Theorem 7 (Completeness of APPLYACSTEP). *Suppose that δ unifies $P_1 \cup P_2$, that P_1 consists of only AC-unification pairs, that $\delta \subseteq V$, that $\sigma \leq \delta$ and that $(P_1 \cup P_2, \sigma, V)$ 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 $\sigma' \leq \gamma\delta$.*

6.3.3 Completeness of ACUNIF

Lemma 10 states the completeness of Algorithm 1 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 9.

Lemma 8 (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 9 (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 $\gamma \in \text{ACUNIF}(P, \sigma, V)$ such that $\gamma \leq_V \delta$.*

Lemma 9 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 CHOOSE (see Algorithm 1). There are two interesting cases. The first case is in lines 17-19 when we only have AC-unification pairs (in that case, we used the completeness of APPLYACSTEP, i.e. Lemma 7). The second case happens when we instantiate a variable (lines 8 and 10) and is solved by using Lemma 8.

To see the need for hypothesis $\sigma \leq \delta$ in Lemma 9, 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 10 (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 $\gamma \in \text{ACUNIF}(\{t \approx^? s\}, id, V)$ such that $\gamma \leq_V \delta$.*

In the proof of Lemma 10, the hypothesis $\delta \subseteq V$ is a technicality that was put 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 3) computed in the example of Section 3.2 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 10. The key step to prove completeness of ACUnif (an improvement of Lemma 10 where $V = Vars(t,s)$ 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 18). The number of items in this definition may seem excessive, but they were all used in our proof, as will be explained in Remark 12.

Definition 18 (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. $Vars(im(\rho)) \subseteq V'$.
7. *If* $X \in Vars(im(\rho))$ *and* $X \notin dom(\rho)$ *then* $X \notin V$

Example 9. *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 11 (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 18 could have been "Variant Inputs".

We can state Theorem 13 with this definition. The proof of Theorem 13 is done by induction, and the hardest cases are when we instantiate a variable (Lemma 11) and, inside the function APPLYACSTEP, when we call SOLVEAC (Lemma 12). We give a structured proof (*à la* Leslie Lamport) of the mentioned lemmas below.

Lemma 11 (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:

⟨1⟩1. First we prove that $\sigma_1'\rho =_V \rho\sigma_1$.

PROOF:

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

⟨2⟩2. CASE: $Z = X$. Then both sides are equal to ρt.

⟨2⟩3. CASE: $Z \neq X$.

PROOF:

⟨3⟩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 $\rho Z \neq \rho X$.

⟨3⟩2. Suppose by contradiction that $\rho Z = \rho X$.

⟨3⟩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 12 (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* SOLVEAC*, 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* SOLVEAC*(t, s, V, f) and V_1' the new set of variables to avoid after we call* SOLVEAC*$(\rho t, \rho s, V', f)$. Let P_c' be a unification problem in* SOLVEAC$(\rho t, \rho s, V', f)$*. Then, there exists P_c in* SOLVEAC(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: $Z_1', \ldots, Z_l'$ 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 $Z_1, \ldots, Z_l$. Notice that

$$V_1 = V \cup \{Z_1, \ldots, Z_l\}$$
$$V_1' = V' \cup \{Z_1', \ldots, Z_l'\}.$$

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

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

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 $\rho^* = \{Z_1 \mapsto Z_1', \ldots, Z_l \mapsto Z_l'\}$. Then, the renaming ρ_1 is defined in our formalisation as $\rho_1 = $ APPEND(ρ, ρ^*). This way of constructing ρ_1 only works since $dom(\rho) \subseteq V$ (Item 5 of the Definition of Renamed Inputs) and that $\{Z_1', \ldots, Z_l'\} \cap V = \emptyset$.

$\langle 1\rangle 3$. 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:

$\langle 2\rangle 1$. 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 $Z_1, \ldots, Z_l$ in P_c and correspondingly $Z'_1, \ldots, Z'_l$ 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 $u' = \rho_1 u$.

$\langle 2\rangle 2$. LET: $t_1, \ldots, t_m$ be the arguments of t and $s_1, \ldots, s_n$ 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 $t_1, \ldots, t_m, s_1, \ldots, s_n$. Similarly, the terms in the left-hand side of every unification problem returned by SOLVEAC$(\rho t, \rho s, V, f)$ will be respectively $\rho t_1, \ldots, \rho t_m, \rho s_1, \ldots, \rho s_n$. 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 $u' = \rho_1 u$.

$\langle 1\rangle 4$. 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 $\langle 1\rangle 3$, $P'_c = \rho_1 P_c$.

$\langle 1\rangle 5$. 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).

$\langle 1\rangle 6$. 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 $max(V_1) \leq max(V'_1)$.

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

PROOF: This follows from $\psi \subseteq V$ (the Definition of Renamed Inputs in our hypothesis) and $V \subseteq V_1$.

$\langle 1\rangle 8$. 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 $V_1 = V \cup \{Z_1, \dots, Z_l\}$ the result follows.

$\langle 1 \rangle 9$. Item 6 of the Definition of Renamed Inputs holds: $Vars(im(\rho_1)) \subseteq V_1'$.
Proof: We have

$$Vars(im(\rho_1)) \subseteq Vars(im(\rho)) \cup \{Z_1', \dots, Z_l'\}.$$

Since $Vars(im(\rho)) \subseteq V'$ (Item 6 of the Definition of Renamed Inputs in our hypothesis) and $V_1' = V' \cup \{Z_1', \dots, Z_l'\}$ the result follows.

$\langle 1 \rangle 10$. 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:
$\langle 2 \rangle 1$. Case: $max(V) = max(V')$.
Proof:
$\langle 3 \rangle 1$. $Z_i = Z_i'$ for every $1 \leq i \leq l$ and therefore $\rho_1 = \rho$.
$\langle 3 \rangle 2$. We have $X \in im(\rho)$ and $X \notin dom(\rho)$. Hence, by Item 7 of the Definition of Renamed Inputs, $X \notin V$.
$\langle 3 \rangle 3$. Since $V_1 = V \cup \{Z_1, \dots, Z_l\}$, all there is to prove is that $X \notin \{Z_1, \dots, Z_l\}$. Due to Step $\langle 3 \rangle 1$, it suffices to prove that $X \notin \{Z_1', \dots, Z_l'\}$.
$\langle 3 \rangle 4$. Suppose by contradiction that $X \in \{Z_1', \dots, Z_l'\}$. Then, $X \notin V'$. However, this contradicts the fact that $X \in im(\rho)$, by Item 6 of the Definition of Renamed Inputs.

$\langle 2 \rangle 2$. Case: $max(V) < max(V')$.
Proof:
$\langle 3 \rangle 1$. We have

$$\begin{aligned} dom(\rho_1) &= dom(\rho) \cup \{Z_1, \dots, Z_l\} \\ im(\rho_1) &= im(\rho) \cup \{Z_1', \dots, Z_l'\} \\ V_1 &= V \cup \{Z_1, \dots, Z_l\}. \end{aligned}$$

$\langle 3 \rangle 2$. 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 $X \in \{Z_1, \dots, Z_l\}$. This, however, contradicts the fact that $X \notin dom(\rho_1)$.
$\langle 3 \rangle 3$. Case: $X \notin im(\rho)$. Then, $X \in \{Z_1', \dots, Z_l'\}$. We have $|X| > max(V') > max(V)$ and hence $X \notin V$. Additionally, $X \notin \{Z_1, \dots, Z_l\}$ because otherwise we would have $X \in dom(\rho_1)$. Hence, we get that $X \notin V_1$.

With Lemmas 11 and 12 it is possible to prove Theorem 13, shown below.

Theorem 13 (Correctness of Renamed Inputs). *Let (P, σ, V) and (P', σ', V') be renamed inputs fixing ψ and suppose $\gamma' \in$ ACUnif(P', σ', V'). Then, there exist a renaming ρ and a substitution $\gamma \in$ ACUnif(P, σ, 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: $nil?(P')$.
Then, ACUNIF(P', σ', V') returns and we have $\gamma' = \sigma'$. Due to Item 1 of the Definition of Renamed Inputs, $P = \rho P' = \emptyset$ and hence ACUNIF(P, σ, 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 $((t', s'), P'_1) =$ CHOOSE(P'). The proof is divided into cases according to the structure of t and s, as Algorithm 1.

⟨1⟩4. CASE: (s' matches X) and (X not in t').

⟨2⟩1. Then,

$$\begin{aligned} \text{ACUNIF}(P', \sigma', V') &= \text{ACUNIF}(\sigma'_1 P'_1, \sigma'_1\sigma', V') \\ \text{ACUNIF}(P, \sigma, V) &= \text{ACUNIF}(\sigma_1 P_1, \sigma_1\sigma, V). \end{aligned}$$

⟨2⟩2. By Lemma 11, $(\sigma_1 P_1, \sigma_1\sigma, V)$ and $(\sigma'_1 P'_1, \sigma'_1\sigma', V')$ are renamed inputs fixing χ and therefore we can apply the induction hypothesis and conclude.

⟨1⟩5. CASE: $t' \approx^? s'$ is an AC-unification pair.

⟨2⟩1. Since $\gamma' \in$ ACUNIF(P', σ', V') there will be

$$(P'_*, \sigma'_*, V'_*) \in \text{APPLYACSTEP}(P', \sigma', V')$$

such that $\gamma' \in$ ACUNIF(P'_*, σ'_*, V'_*).

⟨2⟩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 APPLYACSTEP 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 12. Finally, since function INSTANTIATESTEP only performs variable instantiation, the corresponding Lemma is proved in the same manner as Lemma 11.

⟨2⟩3. Hence, we apply the induction hypothesis and conclude.

⟨1⟩6. The case when (t' matches X) and (X not in s') is similar to Step ⟨1⟩4. The remaining cases are straightforward.

Remark 12 (Necessity of Every Item in Definition of Renamed Inputs)**.** *Items 1 and 2 of the Definition of Renamed Inputs (Definition 18) are used in the main proof of Theorem 13. Theorem 13 relies on Lemmas 11 and 12 and we needed to add Items 3 through 7 in Definition 18 to prove those Lemmas, as explained next. Notice that Items 4 and 7 were used in Lemma 11 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 18 and hence should be included in the definition. Finally, in Lemma 12, 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 18.*

Finally, Theorem 13 is used along with Lemma 10 to prove the completeness of ACUnif (Theorem 14).

Theorem 14 (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:

$\langle 1 \rangle 1$. Let: $V = Vars(t,s)$ and $V' = V \cup dom(\delta) \cup Vars(im(\delta))$. By Theorem 10 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 $\delta =_{V'} \delta_1 \gamma'$

$\langle 1 \rangle 2$. 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 $\gamma' =_V \rho\gamma$.

$\langle 1 \rangle 3$. $\delta =_{V'} \delta_1 \gamma' =_V = \delta_1 \rho \gamma$. Therefore, $\gamma \leq_V \delta$.

Remark 13 (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 14 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.*

7 More Information on the PVS Formalisation

The first order AC-formalisation here described is in NASALib, the main repository for the PVS proof assistant. It is part of the `nominal` library, as it was used to formalise nominal AC-matching (see Section 8.3 for a brief comment and [27] for more details of the nominal paradigm). The functions specified in PVS and the statement of the theorems can be found in files `.pvs`, while the proofs of the theorems can be found in the `.prf` files. The PVS theories and their descriptions are shown below:

- `top_first_order_AC_unification` - High Level description of the first-order AC-unification formalisation.
- `unification_alg` - Function ACUnif (Algorithm 1) 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, choose 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 1 shows the dependency diagram for the PVS theories that compose our formalisation. An arrow going from `theoryA` to `theoryB` means that `theoryA` uses definitions and lemmas from `theoryB`. Besides the first-order AC-unification formalisation, there are other 3 formalisations in the nominal library, which we represent in the picture as orange ellipses. As shown in Figure 1, some of them use theories that are also used by the first-order formalisation.

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.

Example 10 (Automatically and Manually discharged 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* $\text{CONS}((X,a), \text{CONS}((Y,b), \text{NIL}))$*. 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.

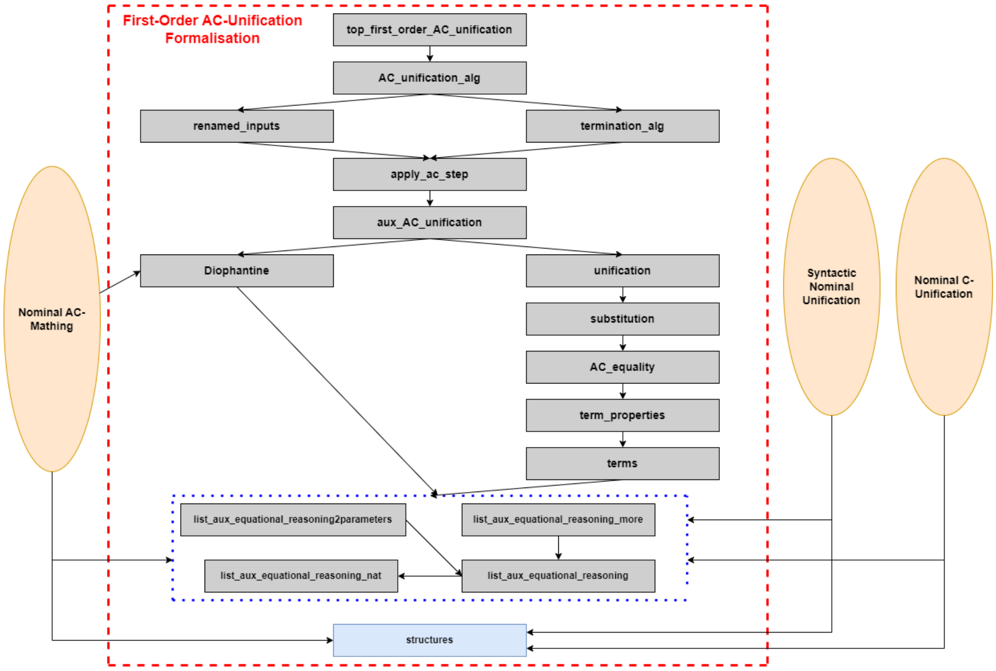

Fig. 1 PVS Formalisation of First-Order AC-Unification in the Nominal Library of Nasalib.

*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 3. 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 under the name `terms` in Table 3.

Table 3 Main Information on the Theories of Our 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%

7.1 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 11) and is used to obtain a list of arguments of an AC-function headed by f.

Example 11. *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 to make it easier to eventually adapt the formalisation to the nominal setting (previous papers in the subject, such as Nominal Unification [30] by Urban et al. and Nominal C-unification [2] by Ayala-Rincón 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 12, 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 1 received this unification problem P, it would return NIL (line 20). 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 12. *Let f be an AC-function symbol. Consider the terms $t \equiv ff\langle a,b\rangle$ 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 6) and is the reason why we imposed that a well-formed term t is not a pair.

Example 13. *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.
- $\langle a,b\rangle$.

7.2 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 15) is given below.

Definition 19 ($size_2$). *We define the $size_2$ of a term t recursively as follows:*

- $size_2(a) = 1$
- $size_2(Y) = 1$

- $size_2(\langle\rangle) = 1$
- $size_2(\langle t_1, t_2\rangle) = size_2(t_1) + size_2(t_2)$
- $size_2(f t_1) = 1 + size_2(t_1)$
- $size_2(f^{AC} t_1) = \sum_{t_i \in Args_f(f^{AC} t_1)} size_2(t_i)$

Theorem 15. *If $t \approx s$ then $size_2(t) = size_2(s)$.*

Theorem 15 is used to prove that if $X \in Vars(s)$ 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 s does not contain the variable X and we are in **case** of lines 8-9.

8 Applications

In this section, we discuss three applications of our certified AC-unification algorithm. First, it can be used as a first step to formalise more efficient first-order AC-unification algorithms. Second, it may be used to test the completeness of implemented first-order AC-unification algorithms. Finally, it was used to formalise a nominal AC-matching algorithm, which could serve as a basis to study nominal AC-unification. We describe each one of these applications in Sections 8.1, 8.2 and 8.3.

8.1 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 [29], but we can adapt the formalisation to use a smaller bound, such as the one mentioned by Clausen and Fortenbacher [12]. Finally, a third way to be more efficient when solving the mentioned Diophantine is to use the graph approach also described in [12].

There are efficient algorithms for AC-unification that rely on using directed acyclic graphs (DAGs) to represent terms (e.g., Boudet's [10]) and hence a different path would be to adapt our formalisation to formalise those algorithms. The dependency diagram of Figure 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 that are proved by induction on terms. In practice, 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 5.1, which is effectively done in file `termination_alg.pvs` (one of the hardest parts of our formalisation, see Table 3). 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.

8.2 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. According to Muñoz and Butler [24], 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.

For instance, the function `divides` is used 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 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
```

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.

8.3 Nominal AC-Matching Towards Nominal AC-Unification

Nominal syntax is an extension of first-order syntax that allows us to smoothly represent systems with binding operators. The first-order AC-unification formalisation presented in this paper has been used to obtain the first nominal AC-matching algorithm [5], which could be directly used to define a nominal rewriting algorithm modulo AC [17].

Going from first-order AC-unification to nominal AC-matching required stating and proving new lemmas and "reusing" old ones. This "reuse" of lemmas from the first-order AC-unification formalisation is not automatic, but interactive. The main issue is that a reasonable amount of proofs are done by induction on the structure of terms, and since the grammar of terms changes (it is extended with nominal abstractions, suspensions, etc.), modifications are required. As an example, consider a typical proof by induction on the structure of a nominal term t. The parts of the proof where t is also in the first-order grammar (t is a constant, t is a function application, t is an AC function application) can be reused. The parts of the proof where t is not in first-order grammar (t is an abstraction, t is a suspension) must be completed manually.

Finally, nominal AC-unification is currently an open theoretical problem, as a direct application of Stickel's method in the nominal setting gives rise to "cyclicities" in the generation of AC-unification problems [5]. If this open question can be solved (perhaps by cutting unnecessary derivations) the nominal AC-matching formalisation could be used as a starting point to formalise nominal AC-unification once the theoretical problems are solved.

9 Conclusion

We described a formalisation of Stickel's pioneering AC-unification algorithm [28, 29] in the PVS proof assistant, improving and extending the formalisation described in [6]. We proved the termination, soundness, and completeness of the algorithm. Our proof of termination is based on the work of Fages (see [15, 16]). However, since mutual recursion is not straightforward in PVS, we adapted the algorithm to receive as input an AC-unification problem P, instead of only two terms t and s. This introduces an additional complication in the proof of termination (it is not enough to call function solveAC and instantiateStep only once) as described in Section 5.2.2. In comparison to [6], we give more details about the grammar of terms, the hierarchy of the formalisation, the proof of completeness, and possible applications of our formalisation. The main lemmas for soundness and completeness are described and the proof of the most complicated one (Completeness of solveAC) is given in detail. Notably, we show here how removing the hypothesis $\delta \subseteq V$ from the theorem of completeness given in [6] is possible although not trivial, using the idea of renamed inputs.

The grammar of terms used in the formalisation was chosen based on previous works [7, 30] in the nominal setting, to make it easier to formalise results in nominal equational reasoning, such as matching and unification. This grammar of terms has some drawbacks as pointed out in Sections 7.1 and 6.3.1 and those were handled by restricting ourselves to well-formed terms.

As described in Section 8, we see three immediate possible paths of future work: formalising more efficient algorithms for first-order AC-unification, testing implementations of first-order AC-unification algorithms, or discovering a terminating, correct, and complete nominal AC-unification algorithm and formalising it. Other possible paths of future work are formalising unification/matching algorithms modulo different equational theories and formalising a more efficient nominal AC-matching algorithm.

Supplementary information. If your article has accompanying supplementary file/s please state so here.

Authors reporting data from electrophoretic gels and blots should supply the full unprocessed scans for key as part of their Supplementary information. This may be requested by the editorial team/s if it is missing.

Please refer to Journal-level guidance for any specific requirements.

Acknowledgments. This work was supported by the Brazilian agency CNPq, Grant Universal 409003/21-2, and the Brazilian Federal District Research Foundation FAPDF, Grant DE 00193-00001175/2021-11. A CNPq productivity RG 313290/21-0 partially supported the first author.

Declarations

9.1 Funding

A CAPES Ph.D. scholarship supported the third author.

9.2 Competing Interests

The authors have no competing interests to declare that are relevant to the content of this article.

9.3 Ethics Approval

Not applicable.

9.4 Consent to Participate

Not applicable.

9.5 Consent for Publication

Not applicable.

9.6 Availability of data and Materials

Not applicable.

9.7 Code Availability

Code is available at: https://github.com/nasa/pvslib/tree/master/nominal.

9.8 Author's Contribution

All authors contributed to the study's conception and design. All authors read and approved the final manuscript.

References

[1] Mohamed Adi and Claude Kirchner. AC-Unification Race: The System Solving Approach, Implementation and Benchmarks. *J. of Sym. Computation*, 14(1):51–70, 1992.

[2] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Gabriel Ferreira-Silva, and Daniele Nantes-Sobrinho. Formalising Nominal C-Unification Generalised with Protected Variables. *Math. Struct. Comput. Sci.*, 31(3):286–311, 2021.

[3] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Daniele Nantes-Sobrinho, and Ana Cristina Rocha Oliveira. A Formalisation of Nominal α-Equivalence with A, C, and AC Function Symbols. *Theor. Comput. Sci.*, 781:3–23, 2019.

[4] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. Formalising Nominal C-Unification Generalised with Protected Variables. *Math. Struct. Comput. Sci.*, 31(3):286–311, 2021.

[5] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira-Silva, Temur Kutsia, and Daniele Nantes-Sobrinho. Nominal AC-Matching. In *16th International Conference on Intelligent Computer Mathematics CICM*, volume 14101 of *LNCS*, pages 53–68, Cham, 2023. Springer.

[6] Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira-Silva, and Daniele Nantes-Sobrinho. A Certified Algorithm for AC-Unification. In *7th International Conference on Formal Structures for Computation and Deduction, FSCD*, volume 228 of *LIPIcs*, pages 8:1–8:21, Dagstuhl, 2022. Leibniz-Zentrum für Informatik.

[7] Mauricio Ayala-Rincón, Maribel Fernández, and Ana Cristina Rocha Oliveira. *Completeness in PVS of a Nominal Unification Algorithm*. *ENTCS*, 323:57–74, 2016.

[8] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998.

[9] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of Matching Problems. *J. of Sym. Computation*, 3(1/2):203–216, 1987.

[10] Alexandre Boudet. Competing for the AC-Unification Race. *J. of Autom. Reasoning*, 11(2):185–212, 1993.

[11] Alexandre Boudet, Evelyne Contejean, and Hervé Devie. A New AC Unification Algorithm with an Algorithm for Solving Systems of Diophantine Equations. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90)*, pages 289–299, Washington, 1990. IEEE Computer Society.

[12] Michael Clausen and Albrecht Fortenbacher. Efficient Solution of Linear Diophantine Equations. *J. of Sym. Computation*, 8(1-2):201–216, 1989.

[13] Evelyne Contejean. A Certified AC Matching Algorithm. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *LNCS*, pages 70–84, Berlin, Heidelberg, 2004. Springer.

[14] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. Equational Unification and Matching, and Symbolic Reachability Analysis in Maude 3.2 (System Description). In *11th International Joint Conference on Automated Reasoning IJCAR*, volume 13385 of *LNCS*, pages 529–540, Cham, 2022. Springer.

[15] François Fages. Associative-Commutative Unification. In *Proceedings 7th International Conference on Automated Deduction CADE*, volume 170 of *LNCS*, pages 194–208, Heidelberg, 1984. Springer.

[16] François Fages. Associative-Commutative Unification. *J. of Sym. Computation*, 3(3):257–275, 1987.

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

[18] Jean-Pierre Jouannaud and Claude Kirchner. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 257–321, Cambridge, MA, 1991. The MIT Press.

[19] Deepak Kapur and Paliath Narendran. Double-exponential Complexity of Computing a Complete Set of AC-Unifiers. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS)*, pages 11–21, Washington, 1992. IEEE Computer Society.

[20] Ramana Kumar and Michael Norrish. (Nominal) Unification by Recursive Descent with Triangular Substitutions. In *First International Conference on Interactive Theorem Proving ITP*, volume 6172 of *LNCS*, pages 51–66, Berlin,

Heidelberg, 2010. Springer.

[21] Leslie Lamport. How to write a proof. *The American mathematical monthly*, 102(7):600–608, 1995.

[22] Leslie Lamport. How to Write a 21st Century Proof. *Journal of fixed point theory and applications*, 11(1):43–63, 2012.

[23] Florian Meßner, Julian Parsert, Jonas Schöpf, and Christian Sternagel. A Formally Verified Solver for Homogeneous Linear Diophantine Equations. In *Interactive Theorem Proving - 9th International Conference, ITP*, volume 10895 of *LNCS*, pages 441–458, Cham, 2018. Springer.

[24] César A Muñoz and Ricky Butler. Rapid prototyping in PVS. Technical Report NASA/CR-2003-212418, NIA-2003-03, NASA Langley Research Center (NIA), 2003.

[25] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction*, volume 607 of *LNCS*, pages 748–752, Berlin, Heidelberg, 1992. Springer.

[26] Sam Owre, Natarajan Shankar, John Rushby, and David Stringer-Calvert. PVS Language Reference. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2000.

[27] Andrew M Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 2013.

[28] Mark E. Stickel. A Complete Unification Algorithm for Associative-Commutative Functions. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, pages 71–76, 1975.

[29] Mark E. Stickel. A Unification Algorithm for Associative-Commutative Functions. *J. of the ACM*, 28(3):423–434, 1981.

[30] Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.

Appendix A A Structured Proof of Termination for applyACStep

The proof of termination (Theorem 25) is based on Lemmas 17, 18, 23 and 24. Before presenting the mentioned results and its proofs, we first introduce some prior notation.

A.1 Notation for the Proof of Termination

Algorithm 1 calls applyACStep with input $(P, \textsc{nil}, \sigma, V)$. 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:

$$P_i \xrightarrow{\textsc{solveAC}} P_i^* \xrightarrow{\textsc{instantiateStep}} P_{i+1}$$

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 $\textsc{solveAC}(t_i, s_i, V_i, f_i)$.

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 14. *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 9. *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 10. *We denote by $NVS(t)$ the set of non-variable subterms of P.*
Remark 14 (Signature of INSTANTIATESTEP)**.** *Function* 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 $P_1 = P$, $P_2 =$ NIL and $\sigma =$ 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 11. *Denote by $[\![\text{INSTANTIATESTEP}(P_1, P_2, \sigma)]\!]_n$ the n-th component ($n = 1, 2, 3$) of the triple $(P', \delta, bool)$ returned by* INSTANTIATESTEP(P_1, P_2, σ).

A.2 Auxiliary Lemmas

Lemma 16. $(P', \sigma', V') \in$ APPLYACSTEP(P^A, P^B, σ, V) *if and only if* $(P', \sigma'', V') \in$ APPLYACSTEP$(P^A, P^B, \text{NIL}, V)$*, where $\sigma' = \sigma'' \circ \sigma$.*
Lemma 17 (V_{NAC} 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*

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

⟨1⟩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 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 NIL anymore, but we can fix that by using Lemma 16 to prove that if $(P_n, \sigma_{0n}, V') \in$ APPLYACSTEP$(P_1^A, P_1^B, \sigma_{01}, V_n)$ then there is $(P_n, \sigma_{1n}, V_n) \in$ APPLYACSTEP$(P_1^A, P_1^B, \text{NIL}, V1)$ such that $\sigma_{0n} = \sigma_{1n} \circ \sigma_{01}$. A similar reasoning happens when we prove Lemmas 18, 23.

⟨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 18 ($V_{>1}$ 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 exist 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 exist 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 $t_1 \in Subterms(P_1)$, there exists some t_1' in P_1 such that $t_1 \in Subterms(t_1')$. This t_1' can be written as σt_3, with t_3 in P_0^*. Hence, $t_1 \in Subterms(\sigma t_3)$.

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

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

⟨2⟩4. CASE: $t_1 \in Subterms(im(\sigma))$. If Y is an argument of a term t_1 in $Subterms(im(\sigma))$, then there exists a term t_4 (same symbol as t_1) in $Subterms(P^C)$ 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.

$\langle 2 \rangle 5$. 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 $\langle 2 \rangle 4$ apply.

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

$\langle 1 \rangle 6$. 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:

$\langle 2 \rangle 1$. 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. $\sigma Y = Y$.

$\langle 2 \rangle 2$. 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.

$\langle 1 \rangle 7$. 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:

$\langle 2 \rangle 1$. CASE: $t_2 \sim_{fsym} t$. Then, $s_2 \not\sim_{fsym} t$ and, by Step $\langle 1 \rangle 6$, $s_2 \in \mathit{Subterms}(P_0)$. Since $Y \in \mathit{Args}(s_2)$, this implies $Y \in \mathit{Vars}(P_0)$. From that and the fact that $Y \in \mathit{Vars}(t_2)$ we get that $t_2 \in \mathit{Subterms}(P_0)$. Hence, we have that $Y \in V_{>1}(P_0)$ and therefore $\psi(Y) \in \psi(V_{>1}(P_0))$.

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

$\langle 1 \rangle 8$. CASE: $X = Y$ and $W \neq Y$.

PROOF:

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

$\langle 2 \rangle 2$. 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 4.3.3), which contradicts the fact that $\sigma Y = Y$ (see Step $\langle 1 \rangle 6$).

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

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

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

$\langle 2 \rangle 5$. If $t_2 \sim_{fsym} t_3$ then $s_2 \not\sim_{fsym} t_3$. Then, $s_2 \in \mathit{Subterms}(P_0)$ (Fact from $\langle 1 \rangle 6$). Since $W \in \mathit{Args}(s_2)$ this means that $W \in \mathit{Vars}(P_0)$. Together with Step $\langle 2 \rangle 1$, 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.

$\langle 2 \rangle 6$. 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 $\psi W = \sigma W = Y \in \psi(V_{>1}(P_0))$.

$\langle 2 \rangle 7$. 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.

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

$\langle 1 \rangle 10$. CASE: $X \neq Y$ and $W \neq Y$.

$\langle 2 \rangle 1$. $\sigma X = Y$ 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.

$\langle 2 \rangle 2$. 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 $\langle 1 \rangle 6$).

$\langle 2 \rangle 3$. 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 19 (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 $t_1 \in Subterms(t)$ and $X \in Args(t_1)$ such that $\sigma X = t_s$ and if t_s is an AC function symbol, then $t_1 \not\approx_{fsym} t_s$.*

Lemma 20. *Let $\sigma = [\![\textsc{instantiateStep}(P, \textsc{nil}, \textsc{nil})]\!]_2$. 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 20). It let us prove Lemma 22, which is used in Lemma 23.

Definition 20 (Nice Unification Problem with respect to f)**.** *Let P be a unification problem, f be a function symbol and $\sigma = [\![\textsc{instantiateStep}(P, \textsc{nil}, \textsc{nil})]\!]_2$. Suppose that for every function term $t \in Subterms(P)$, if there is a variable $X \in Args(t)$ 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 21 (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 15. *Recall that the first time we call* $\textsc{applyACStep}$ *we have $P_0 = P_u$ and $P_s = \textsc{nil}$.*

Lemma 22 (*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 23 (*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 18, 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 $AS(P_1) \subseteq \sigma_{01}(AS(P_0))$.

⟨1⟩4. From now until the remaining of the proof, we denote σ_{01} by σ and ψ_{01} by ψ.

⟨1⟩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 19 we have 3 possibilities:

1. $t_{1s} \in \sigma(AS(t_2))$.
2. $t_{1s} \in AS(im(\sigma))$
3. There is $t_3 \in Subterms(t_2)$ and $X \in Args(t_3)$ 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 $t' \in Subterms(P_0^C)$ that is not an AC-function term headed by f and a variable X such that $X \in Args(t')$, $\sigma X = t_3$ 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 $t' \in Subterms(lhs(P_0^C))$ and therefore $X \in Subterms(lhs(P_0^C))$. This, along with the fact that $X \in dom(\sigma)$, let us conclude that $X \in Args(t) \cup Args(s)$. Suppose without loss of generality that $X \in Args(t)$. 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 $t_3 \in Subterms(P_n)$ such that $X \in Vars(t_3)$. However, every term in P_n can be written as $\sigma_{0n}t_4$, where $t_4 \in Subterms(P_0)$. Hence we would get $X \in Vars(\sigma_{0n}t_4)$. This cannot happen because $X \in dom(\sigma_{0n})$ 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 $t_2 \in rhs(P_0^C)$ and therefore $AS(t_2) = \emptyset$, which contradicts the fact that $t_{1s} \in \sigma AS(t_2)$.

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

PROOF:

⟨2⟩1. LET: $P^A = rhs(P_0^C)$ and $P^B = lhs(P_0^C)$. We can apply Lemma 22 and obtain that $t_{1s} \in \sigma AS(P^A) \cup \sigma NVS(P^B)$.

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

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

⟨1⟩9. CASE: There is $t_3 \in Subterms(t_2)$ and $X \in Args(t_3)$ 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. $t_{1s} \in im(\sigma)$, 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 $t_4 \in rhs(P_0^C)$.

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

⟨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 $t_4 \in rhs(P_0^C)$, 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 $X \in dom(\sigma)$.

⟨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, $t_3 \in Subterms(P_0)$ and therefore $X \in Subterms(P_0)$. Since $X \in P_0^C$, we conclude that $X \in lhs(P_0^C)$.

⟨3⟩3. $X \in Args(t) \cup Args(s)$. Suppose without loss of generality that $X \in Args(t)$. 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 $X \in dom(\sigma_{0n})$. If we had $X \in V_{>1}(P_n)$ there would be some term $t_5 \in Subterms(P_n)$ such that $X \in Vars(t_5)$. There exists some term $t_6 \in Subterms(P_0)$ such that $t_5 = \sigma_{0n}t_6$. Hence, $X \in Vars(\sigma_{0n}t_6)$. This however, contradicts the fact that $X \in dom(\sigma_{0n})$ 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 24 (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 23, 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 21, 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 $\sigma_{0n}(AS(P_0)) \not\subseteq AS(P_n)$. 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.

A.3 Termination of applyACStep

Theorem 25 (Termination of applyACStep)**.** *Suppose that Algorithm 1 is called with the nice input (P, σ, V) and enters the branch of* applyACStep *(lines 16-19). 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 16 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 17 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 18 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: $P_n = \text{NIL}$. Then $|AS(P_n)| = 0 \leq AS(P)$ and

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

since P is not null.

⟨2⟩2. Case: $P_n \neq \text{NIL}$. Then by Lemma 24 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

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