A Certified Algorithm for AC-Unification

Mauricio Ayala-Rincón
Departments of Computer Science and Mathematics, Universidade de Brasília, Brazil

Maribel Fernández
Department of Informatics, King's College London, U.K.

Gabriel Ferreira Silva
Department of Computer Science, Universidade de Brasília, Brazil

Daniele Nantes Sobrinho
Department of Mathematics, Universidade de Brasília, Brazil

Abstract

Implementing unification modulo Associativity and Commutativity (AC) axioms is crucial in rewrite-based programming and theorem provers. We modify Stickel's seminal AC-unification algorithm to avoid mutual recursion and formalise it in the PVS proof assistant. More precisely, we prove the adjusted algorithm's termination, soundness, and completeness. To do this, we adapted Fages' termination proof, providing a unique elaborated measure that guarantees termination of the modified AC-unification algorithm. This development (to the best of our knowledge) provides the first fully formalised AC-unification algorithm.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Equational logic and rewriting

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

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material Source code available at `nominal.cic.unb.br/ac_unification_first_order.zip`

1 Introduction

Syntactic unification is the problem of, given terms s and t, 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 $\sigma s \approx_E \sigma t$ [15].

Unification has practical applications in mathematics and computer science. It is used, for instance, in interpreters of logic programming languages such as Prolog, in resolution-based theorem provers, in confluence tests based on critical pairs, and so on [5]. 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 [22, 5]).

Related Work. Unification in the presence of AC-function symbols was first solved by Stickel [22]. He showed how the problem is connected to finding nonnegative integral solutions to linear equations and proved that his algorithm was terminating, sound, and complete for a subclass of the general case [22, 23]. However, Stickel's proof of termination did not apply to the general case: almost a decade after the introduction of this algorithm, Fages proposed a measure fixing the termination proof for the general case [12, 13]. Since then, investigations on solving AC-unification efficiently, on the complexity of AC-unification, and on formalising unification modulo equational theories were carried out.

Regarding solving AC-unification efficiently, Boudet et al. [8] proposed an AC-unification algorithm that explores constraints more efficiently than the standard algorithm. Further, Boudet [7] 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.

Regarding the complexity of AC-unification, Benanav et al. [6] showed that the decision problem for AC-matching is NP-complete, and the decision problem for AC-unification is NP-hard. In addition, Kapur and Narendran [16] showed that the complexity of computing a complete set of AC-unifiers is double-exponential.

As far as we know, there are no formalisations of AC-unification algorithms. Nevertheless, there are formalisations of related algorithms, and some preliminary work has been done. Ayala-Rincón et al. [2] formalised nominal α-equivalence for associative, commutative and associative-commutative function symbols. That work is in the nominal setting (see [21]), which encompasses first-order AC-equivalence. In 2004, Contejean [11] gave a certified AC-matching algorithm in Coq. AC-matching is an easier problem (see Remark 8) related with AC-unification, where we must find a substitution σ such that $\sigma s \approx_{AC} t$. A formalisation of nominal C-unification, which can also handle nominal C-matching, is also available [3]. Additionally, Meßner et al. [18] gave a formally verified solver for homogeneous linear Diophantine equations in Isabelle/HOL. As we shall see, the problem of AC-unification is connected to solving linear Diophantine equations.

It is well-known that although both C- and AC-unification problems are of finitary type, the complexity of computing a complete set of unifiers for the former problem is exponential, while for the latter one, it is double-exponential [16]. Indeed, to build minimal complete sets of C-unifiers, only simple swapping-argument-combinations need to be considered to instantiate variables. However, to build minimal complete sets of AC-unifiers, all possible associations and permutations of arguments should be considered, which is precisely expressed by Stickel's method based on solving Diophantine equations.

Contribution and Applications. In this work, we give the first (as far as we know) formalisation of termination, soundness and completeness of an algorithm for AC-unification. We formalised Stickel's algorithm for AC-unification using the proof assistant PVS [19]. We chose PVS since we want, as future work (see Section 5), to enrich the nominal unification library that already exists in PVS with a nominal AC-unification algorithm.

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 apply 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 [20]) and to ease the formalisation.

Our formalisation could be used as a starting point to prove the correctness of more efficient algorithms. For instance, when we solve the linear Diophantine equations necessary for AC-unification, we do it until a certain bound is reached, proved sufficient by Stickel [22]. One possible way to sharpen our formalisation is to use a smaller bound, such as the one mentioned by Clausen and Fortenbacher [10]. Another possible way to improve the efficiency of the algorithm is to solve the mentioned Diophantine equations more efficiently, using the graph approach, also described in [10]. Adapting our formalisation to algorithms that use directed acyclic graphs (DAGs) to represent terms (e.g., Boudet's [7]) would imply a reformulation of almost all subtheories of the formalisation due to their dependency on terms. But such a reformulation would be possible and faster than starting from scratch as discussed in Remark 34, Appendix B.

Organisation. Section 2 gives the necessary background; Section 3 explains the modification of Stickel's algorithm; Section 4 discusses the most interesting points of the formalisation; finally, Section 5 concludes and discusses possible paths of future work. The appendices provide further details about the algorithms, the PVS code and the proofs. In addition to the appendices, we include cyan-coloured hyperlinks to specific points of interest of the PVS formalisation.

2 Background and Example

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 $\mathcal{X}$ be a set of variables. The set $T(\Sigma, \mathcal{X})$ is generated by the grammar:*

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

where a denotes a constant, 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. 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.

▶ **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 Appendix C), 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.

▶ **Notation 1** (Flattened form of AC-functions). *When convenient, we may denote in this paper an AC-function in flattened form. For instance, the term $f^{AC}\langle f^{AC}\langle a, b\rangle, f^{AC}\langle c, d\rangle\rangle$ may be denoted simply as $f^{AC}(a, b, c, d)$. In our formalisation (for instance in function $Args_f$), when we manipulate an AC-function term t we are more interested in its arguments than in how they were encoded using pairs.*

▶ **Notation 2** ($Vars$). *We denote the set of variables of a term t by $Vars(t)$. Similarly, we denote the set of variables that occur in a unification problem P as $Vars(P)$.*

A substitution σ is a function from variables to terms, such that $\sigma X \neq X$ only for a finite set of variables, called the domain of σ and denoted as $dom(\sigma)$. The image of σ is then defined as $im(\sigma) = \{\sigma X \mid X \in dom(\sigma)\}$. A well-formed substitution only instantiates variables to well-formed terms. In the proofs of soundness and completeness of the algorithm, we restrict ourselves to well-formed substitutions. Let V be a set of variables. If $dom(\sigma) \subseteq V$ and $Vars(im(\sigma)) \subseteq V$ we write $\sigma \subseteq V$. In our PVS code, substitutions are represented by a list, where each entry of the list is called a nuclear substitution and is of the form $\{X \to t\}$.

▶ **Definition 3** (Nuclear substitution action on terms). *A nuclear substitution $\{X \to s\}$ acts over a term by induction as shown below:*

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

▶ **Definition 4** (Substitution acting on terms). *Since a substitution σ is a list of nuclear substitutions, the action of a substitution is defined as:*

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

▶ Remark 5. 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 $\text{APPEND}(\sigma, \delta)$.

We now define AC-unification problems (Definition 6), unifiers and a complete set of unifiers (Definition 7).

▶ **Definition 6** (AC-Unification problem). *An AC-unification problem is a finite set of equations $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$. The left-hand side of the unification problem P is defined as $\{t_1, \ldots, t_n\}$ while the right-hand side is defined as $\{s_1, \ldots, s_n\}$.*

▶ **Notation 3** (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.*

▶ **Definition 7** (AC-unifiers). *Let P be a unification problem $\{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$. An AC-unifier or solution of P is a substitution σ such that $\sigma t_i \approx \sigma s_i$ for every i from 1 to n.*

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 $\sigma \leq_V \sigma'$. If this happens for every variable, we write $\sigma \leq \sigma'$.*

With the notion of more general substitution, we can define a complete set $\mathcal{C}$ of unifiers of P as a set that satisfies two conditions: each $\sigma \in \mathcal{C}$ is an AC-unifier of P; and for every δ that unifies P, there is $\sigma \in \mathcal{C}$ such that $\sigma \leq \delta$.

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 $\{\sigma t_1 \approx^? \sigma s_1, \ldots, \sigma t_n \approx^? \sigma s_n\}$.

2.1 What Makes AC-unification Hard

Let f be an associative-commutative function symbol. Finding a complete set of unifiers for $\{f(X_1, X_2) \approx^? f(a, Y)\}$ is not as easy as it appears at first sight, since it is not enough to simply compare the arguments of the first term with the arguments of the second term. Indeed, this strategy would give us only $\sigma_1 = \{X_1 \to a, Y \to X_2\}$ and $\sigma_2 = \{X_2 \to a, Y \to X_1\}$ as solutions, missing for example the substitution $\sigma_3 = \{X_1 \to f\langle a, W\rangle, Y \to f\langle X_2, W\rangle\}$. This solution would be missed because the arguments of $\sigma_3 Y = f\langle X_2, W\rangle$ are partially contained in $\sigma_3 X_1 = f\langle a, W\rangle$ and partially contained in $\sigma_3 X_2 = X_2$.

▶ Remark 8. In contrast to AC-unification, for AC-matching, it is enough for completeness to explore all possible pairings of the arguments of the first term with the arguments of the second term. Evidence of the difficulty of AC-unification is the fact that, although Contejean (see [11]) formalised AC-matching in 2004, there is no formalisation of AC-unification yet.

2.2 An Example

Before presenting the pseudocode for the algorithm we formalised, we give a higher-level example (taken from [22]) of how we would solve $\{f(X, X, Y, a, b, c) \approx^? f(b, b, b, c, Z)\}$.

The first step is to eliminate common arguments in the terms that we are trying to unify. The problem is now $\{f(X, X, Y, a) \approx^? f(b, b, Z)\}$. The second step is to associate our unification problem with a linear Diophantine equation, where each argument of our terms corresponds to one variable in the equation (this process is called variable abstraction) and the coefficient of this variable in the equation is the number of occurrences of the argument. In our case, the linear Diophantine equation obtained is: $2X_1 + X_2 + X_3 = 2Y_1 + Y_2$ (variable X_1 was associated with argument X, variable X_2 with the argument Y and so on).

The third step is to generate a basis of solutions to the equation and associate a new variable (the Z_is) to each solution. The result is shown on Table 1.

Table 1 Solutions for the equation $2X_1 + X_2 + X_3 = 2Y_1 + Y_2$.

X_1	X_2	X_3	Y_1	Y_2	New Vars.
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):

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

In order to explore all possible solutions, we must consider whether we will include or not each solution on 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. Thus, we are left with only 69 cases to consider.

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 Equations (1), becomes:

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

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 terms 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 will be:

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

▶ Remark 9 (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`.

Algorithm 1 Algorithm to Solve an AC-Unification Problem P

1: **procedure** ACUNIF(P, σ, V)
2:   **if** nil?(P) **then return** $cons(\sigma, \textsc{nil})$
3:   **else let** $((t, s), P_1) = \textsc{choose}(P)$ **in**
4:     **if** (s matches X) and (X not in t) **then**
5:       $\sigma_1 = \{X \to t\}$
6:       **return** ACUNIF$(\sigma_1 P_1, \textsc{append}(\sigma_1, \sigma), V)$
7:     **else**
8:       **if** t matches a **then**
9:         **if** s matches a **then return** ACUNIF(P_1, σ, V)
10:         **else return** NIL
11:       **else if** t matches X **then**
12:         **if** X not in s **then**
13:           $\sigma_1 = \{X \to s\}$
14:           **return** ACUNIF$(\sigma_1 P_1, \textsc{append}(\sigma_1, \sigma), V)$
15:         **else if** s matches X **then return** ACUNIF(P_1, σ, V)
16:         **else return** NIL
17:       **else if** t matches $\langle\rangle$ **then**
18:         **if** s matches $\langle\rangle$ **then return** ACUNIF(P_1, σ, V)
19:         **else return** NIL
20:       **else if** t matches $f\ t_1$ **then**
21:         **if** s matches $f\ s_1$ **then**
22:           $(P_2, bool) = \textsc{decompose}(t_1, s_1)$
23:           **if** $bool$ **then return** ACUNIF$(\textsc{append}(P_2, P_1), \sigma, V)$
24:           **else return** NIL
25:         **else return** NIL
26:       **else**
27:         **if** s matches $f^{AC}\ s_1$ **then**
28:           $InputLst = \textsc{applyACStep}(P, \textsc{nil}, \sigma, V)$
29:           $LstResults = \textsc{map}(\text{ACUnif}, InputLst)$
30:           **return** FLATTEN $(LstResults)$
31:         **else return** NIL

3 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 output is a list of substitutions, where each substitution δ in this list is an AC-unifier of P. The first call to the algorithm, in order to unify two terms t and s, is done with $P = cons((t, s), nil)$, $\sigma = nil$ (because we have not computed any substitution yet) 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), that returns a pair (t, s) and a unification problem P_1, such that $P = \{t \approx^? s\} \cup P_1$. The algorithm will try to simplify our unification problem P by simplifying $\{t \approx^? s\}$, and it does that by seeing what the form of t and s is.

3.1 The Functions choose and decompose

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 **if** of line 27 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 was chosen to aid us in the proof of termination. However, it also makes the algorithm more efficient since it guarantees that we only enter on the AC-part of the algorithm when we need it (the AC-part is the computationally heaviest).

If the function DECOMPOSE receives two terms t and s and these terms are both pairs, it recursively tries to decompose them, returning a tuple $(P, bool)$, where P is a unification problem and $bool$ 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 $bool = True$. If one of the terms is a pair and the other is not, the function returns $(\text{NIL}, False)$. 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 10.** Below we give examples of function DECOMPOSE.

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

3.2 The AC-part of the Algorithm

The AC-part of Algorithm 1 relies on function APPLYACSTEP (Section 3.2.4), which depends on two functions: SOLVEAC (Section 3.2.1) and INSTANTIATESTEP (Section 3.2.3). 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 28). 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 29 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.

3.2.1 Function solveAC

The function SOLVEAC does what was illustrated in the example of Section 2.2. While APPLYACSTEP or ACUNIF take as part of the input the whole unification problem, SOLVEAC takes only two terms t and s. It assumes that both terms are headed by the same AC-function symbol f. It also receives as input the set of variables V that are/were in the problem (since SOLVEAC will introduce new variables, we must know the ones that are/were already in use).

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

To generate the basis of solutions for the linear Diophantine equation, it suffices to calculate all the solutions until an upper bound, computed by 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 [22]) 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 2.2 is represented in our code as the matrix D. 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.

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

To explore all possible cases, we must decide whether or not we will include each solution. In our code, this translates to considering submatrices of D by eliminating some rows. In the example of Section 2.2, we mentioned that we should observe two constraints:

- no "original variable" (the variables $X_1, \dots, X_m, Y_1, \dots, Y_n$ associated with the arguments of t and s) should receive the value 0. In terms of D, it means every column has at least one coefficient different than zero.
- an original variable, which does not represent a variable term, cannot be paired with an AC-function application. In terms of D, it means that 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. Notice that this is the submatrix associated with a solution including only the rows 4 and 6 (of the variables Z_4, Z_6).

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

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 $SubmatrixLst$.

3.2.2 Common Structure of Unification Problems Returned by solveAC

Suppose function SOLVEAC receives as input the terms u and v, both headed by the same AC-function symbol f. Let $u_1, \dots, u_m$ be the different arguments of u and let $v_1, \dots, v_n$ be the different arguments of v, after eliminating the common arguments of u and v. If $P_1 = \{t_1 \approx^? s_1, \dots, t_k \approx^? s_k\}$ 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 $t_1, \dots, t_k$) 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 (2)).
3. A term s_i is an AC-function headed by f only if the corresponding term t_i is a variable.

3.2.3 Function instantiateStep

After the application of function SOLVEAC, we instantiate the variables that we can by calling function INSTANTIATESTEP. Indeed, for the proof of termination, it is necessary to compose the substeps of the algorithm with some strategy, as the following example (adapted from [13]) shows.

▶ **Example 11** (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. Applying function SOLVEAC to try to unify $f(X, Y)$ with $f(U, V)$ we obtain as one of the branches the unification problem $\{X \approx^? f(X_1, X_2), Y \approx^? f(X_3, X_4), U \approx^? f(X_1, X_3), V \approx^? f(X_2, X_4)\}$. We can solve this branch by instantiating X, Y, U and V. After these instantiations, we have to unify the remaining two equations: $\{f(X_1, X_2) \approx^? f(X_3, X_4), f(X_1, X_3) \approx^? f(X_2, X_4)\}$. Solving the first equation, one branch obtained is $\{X_1 \approx^? X_3, X_2 \approx^? X_4\}$, which get us back to $P' = \{f(X_1, X_3) \approx^? f(X_2, X_4), X_1 \approx^? X_3, X_2 \approx^? X_4\}$, which is essentially the same unification problem we started with.

This infinite loop in our example would not have happened if we had instantiated $\{X \to Y\}$ and $\{U \to V\}$ in the beginning. To prevent this from happening, Algorithm 1 only handles AC-unification pairs when there are no equations $s \approx^? t$ of other type left, and as soon as we apply the function SOLVEAC we immediately instantiate the variables that we can by calling function INSTANTIATESTEP.

3.2.4 Function applyACStep

Function APPLYACSTEP relies on functions SOLVEAC and INSTANTIATESTEP, and is called by Algorithm 1 when all the equations $s \approx^? t \in P$ 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 in 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 $P_2 = nil$ and as the function goes recursively calling itself, P_1 diminishes while P_2 increases.

4 Interesting Points on the Formalisation

4.1 Avoiding Mutual Recursion

When specifying Stickel's algorithm, we tried to follow closely the pseudocode presented in [13] (the papers [22, 23] give a higher-level description of the algorithm). In [13] there is a function UNIAC used to unify terms t and s and a function UNICOMPOUND used to unify a list of terms $(t_1, \ldots, t_n)$ with a list of terms $(s_1, \ldots, s_n)$. These functions are mutually recursive, i.e. UNIAC calls UNICOMPOUND and vice-versa, something not allowed in PVS[1] [20].

We have adapted the algorithm to use only one main function, which receives a unification problem P and operates (except for the AC-part of the algorithm, see Section 3.2) by simplifying one of the equations $\{t \approx^? s\}$ of P. The main modification is that the lexicographic measure we use (adapted from [13]) would not diminish if in the AC-part of the unification problem we had simplified only one of the equations $\{t \approx^? s\}$ of P (see the discussion in Section 4.3.2).

4.2 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 12, 16, 19 and 21, 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 12** ($V_{NAC}(P)$). *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 13.** Let f be an AC-function symbol and let 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 14** ($Subterms(P)$). *The subterms of a unification problem P are given as: $Subterms(P) = \bigcup_{t \in P} Subterms(t)$, where the notion of subterms of a term t excludes all pairs and is defined recursively as follows:*

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

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

▶ Remark 15 (Subterms of AC and non-AC functions). The definition of subterms for non-AC functions cannot be used for AC functions, as the following counterexample shows. Let f be an AC-function symbol and consider the term $t = f\langle f\langle a, b\rangle, f\langle c, d\rangle\rangle$. Then $Subterms(t) = \{t, a, b, c, d\}$. However, if we had used the definition of subterms for non-AC functions, we would obtain $Subterms(t) = \{t, f\langle a, b\rangle, f\langle c, d\rangle, a, b, c, d\}$.

▶ **Definition 16** ($V_{>1}(P)$). *We denote by $V_{>1}(P)$ the set of variables that are arguments of (at least) two terms t and s such that t and s are headed by different function symbols and t and s are in $Subterms(P)$. The informal meaning is that if $X \in V_{>1}(P)$ then X is an argument to at least two different function symbols.*

▶ **Example 17.** Let f be an AC-function symbol and let 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 $V_{>1}(P) = \{Y\}$.

[1] Despite this restriction, since PVS has higher-order logic foundations, mutual recursion can be emulated, as usual, using functional parameters. However, this would imply a treatment of such parameter functions that restricts their domains according to the chosen measure.

We define proper subterms in order to define admissible subterms in Definition 19.

▶ **Definition 18** (Proper Subterms). *If t is not a pair, we define the proper subterms of t, denoted as $PSubterms(t)$ as: $PSubterms(t) = \{s \mid s \in Subterms(t) \text{ and } s \neq t\}$. We define the proper subterm of a pair $\langle t_1, t_2 \rangle$ as:*

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

▶ **Definition 19** (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 20.** 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 21** (Size of a Unification Problem). *We define the size of a term t recursively as follows:*

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

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

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

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

Table 2 Decrease of the components of the lexicographic measure.

Recursive Call	$\lvert V_{NAC}(P)\rvert$	$\lvert V_{>1}(P)\rvert$	$\lvert AS(P)\rvert$	$size(P)$
line 6, 14	$<$			
lines 9, 15, 18, 23	$\leq$	$\leq$	$\leq$	$<$
case 1 - line 29	$\leq$	$<$		
case 2 - line 29	$\leq$	$\leq$	$<$	
case 3 - line 29	$\leq$	$\leq$	$\leq$	$<$

4.3 Proof Sketch for Termination

4.3.1 Non AC Cases

To prove termination of syntactic unification, we can use a lexicographic measure lex_s consisting of two components: $lex_s = (|Vars(P)|, size(P))$, where $Vars(P)$ is the set of variables in the unification problem. We adapted this idea to our proof of termination, by using $|V_{NAC}(P)|$ as our first component and $size(P)$ as the fourth. The proof of termination for all the cases of Algorithm 1 except AC (line 29) is similar to the proof of termination of syntactic unification, with two caveats.

First, we need to use $|V_{NAC}(P)|$ instead of $|Vars(P)|$ to avoid taking into account the variables that are arguments of the AC-function terms introduced by solveAC (see Section 3.2.2). We would still have to take into account the variable terms introduced by solveAC, but those are instantiated by function instantiateStep and therefore eliminated from the problem.

Second, in some of the recursive calls (lines 9, 15, 18, 23) 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.

4.3.2 The AC-case

Our proof of termination for the AC-case uses the components $|V_{>1}(P)|$ and $|AS(P)|$, proposed in [13]. 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 3.2.2, 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 in 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 greatest size (if there is a tie, pick any term with greatest size). Then, u is an argument of either t or s and therefore $u \in AS(P)$. We also have $u \notin AS(P_1)$: otherwise there would be a term u' in P_1 such that $u \in AS(u')$, which would mean that the size of u' is greater than u (see Remark 22), contradicting our hypothesis that no term in P_1 has size greater than u. Combining the fact that $AS(P_1) \subseteq AS(P)$ and the fact that there is a term u with $u \in AS(P)$ and $u \notin AS(P_1)$ we obtain that $|AS(P_1)| < |AS(P)|$.

► **Example 23.** In the example of Section 2.2, $P = \{f(X, X, Y, a) \approx^? f(b, b, Z)\}$ and 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 24 shows.

► **Example 24** (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 24 to make it so that X only appears as 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 25.

▶ **Example 25** (A case where INSTANTIATESTEP does not increase $|AS|$)**.** If we change slightly the problem from Example 24 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 $AS(P_1') = AS(P_2') = \{a, c\}$.

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

Consider an arbitrary variable term X on the left-hand side of P_1. If X was instantiated by INSTANTIATESTEP, it would be instantiated to an AC-function headed by f (see Section 3.2.2) and therefore would only contribute in increasing $|AS(P_2)|$ in relation with $|AS(P_1)|$ if it also occurred as an argument to a function term (let's call it t^*) headed by a different symbol than f (let's say g). Since X is in the left-hand side of P_1 this means that it was an argument of t or s in P (suppose t, without loss of generality) and remember that both t and s are headed by the same symbol f. Then X is an argument of t^* and t and therefore, by definition, $X \in V_{>1}(P)$. However X was instantiated by INSTANTIATESTEP and therefore it is not in $V_{>1}(P_2)$. The new variables introduced by SOLVEAC will not make any difference in favour of $|V_{>1}(P_2)|$: when they occur as arguments of function terms, the terms are always headed by the same symbol f. Therefore $|V_{>1}(P)| > |V_{>1}(P_2)|$. Accordingly, to fix our problem we include the measure $|V_{>1}(P)|$ before $|AS(P)|$, obtaining the lexicographic measure described in Section 4.2.

The situation described is similar when our unification problem P has more than one equation. Let's say $P = \{t_1 \approx^? s_1, \dots, t_n \approx^? s_n\}$. The only difference is that it is not enough to call function SOLVEAC and then function INSTANTIATESTEP in only the first equation $t_1 \approx^? s_1$: we need to call function APPLYACSTEP and simplify every equation $t_i \approx^? s_i$.

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

4.4 Soundness and Completeness

As mentioned, to unify terms t and s we use Algorithm 1 with $P = cons((t, s), nil)$, $\sigma =$ NIL and $V = Vars((t, s))$. However, since the parameters of ACUNIF may change in between the recursive calls, we cannot prove soundness (Corollary 28) directly by induction. We must prove the more general Theorem 27, 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 26.

▶ **Definition 26** (Nice input)**.** *Given an input (P, σ, V), we say that this input is nice if:*

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

▶ **Theorem 27** (Soundness for nice inputs). *Let (P, σ, V) be a nice input, and $\delta \in ACU\textsc{nif}(P, \sigma, V)$. Then, δ unifies P.*

▶ **Corollary 28** (Soundness of ACUNIF). *If $\delta \in ACU\textsc{nif}(cons((t, s), \textsc{nil}), \textsc{nil}, Vars((t, s)))$ then δ unifies $t \approx^? s$.*

Proving completeness of Algorithm 1 boils down to proving Corollary 30 and similarly to the soundness case, this is proved immediately once we prove Theorem 29. In the proof of completeness, the hypothesis $\delta \subseteq V$ is simply a technicality that was put only in order to guarantee that the new variables introduced by the algorithm do not clash with the variables in $dom(\delta)$ or in the terms in $im(\delta)$ and could be replaced by a different mechanism that guarantees that the variables introduced by the AC-part of ACUNIF are indeed new. As an example, let's go back to the substitutions (see Equation 3) computed in the example of Section 2.2 and notice that the set of variables in the original problem is $V = \{X, Y, Z\}$. If $\delta = \{X \mapsto f(Z_2, a, b), Z \to f(a, Y, Z_2, a, Z_2, a), Z_4 \to c\}$ 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 $\sigma_4 = \{X \to f(Z_6, b), Z \to f(a, Y, Z_6, Z_6)\}$ that we computed is still more general than δ (restricted to the variables in V). Indeed, if we take $\delta_1 = \{Z_6 \to f(Z_2, a)\}$ then $\delta W = \delta_1 \sigma_4 W$ for all variables $W \in V$.

▶ **Theorem 29** (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 $\gamma \in ACU\textsc{nif}(P, \sigma, V)$ such that $\gamma \leq_V \delta$.*

▶ **Corollary 30** (Completeness of ACUNIF). *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 $\gamma \in ACU\textsc{nif}(cons((t, s), nil), nil, V)$ such that $\gamma \leq_V \delta$.*

4.5 More Information About the PVS Formalisation

The functions coded in PVS and the statement of the theorems can be found on files `.pvs`, while the proofs of the theorems can be found in the `.prf` files. The PVS theory `unification_alg` contains function ACUNIF and the theorems of soundness and completeness; `termination_alg` has the definitions and lemmas needed to prove termination; `apply_ac_step` contains function APPLYACSTEP and lemmas about its properties; `aux_unification` contains auxiliary functions such as SOLVEAC and INSTANTIATESTEP and lemmas about their properties. The PVS theories `diophantine`, `unification`, `substitution`, `equality` and `terms` contain, respectively, definitions and properties about solving linear Diophantine equations, unification, substitutions, equality modulo AC and terms. Finally `list` is a set of parametric theories that define generic functions that operate on lists, not strictly connected to unification.

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 discharge most of the TCCs. In our code, most TCCs were related to the termination of functions and PVS was able to prove almost all of them automatically. The number of theorems and TCCs proved for each theory, along with the approximate size of each theory and their percentage of the total size is shown in Table 3.

▶ Remark 31 (Limitation). When we solve a linear Diophantine equation we do not calculate a basis of solutions to the equation. Instead, we generate all solutions until the upper bound given by CALCULATEUPPERBOUND, i.e., we generate a spanning set (which is not necessarily linearly independent) of solutions. This was done in order to ease the specification of DIOSOLVER and the formalisation of its properties. Modifying DIOSOLVER to calculate a basis of solutions instead of a spanning set should improve the efficiency of the algorithm.

Table 3 Main Information on the Theories of Our Formalisation.

Theory	Theorems	TCCs	Size (.pvs)	Size (.prf)	Size (%)
`unification_alg`	9	18	5KB	1.4MB	4%
`termination_alg`	80	35	21KB	11.0MB	30%
`apply_ac_step`	23	12	13KB	9.0MB	25%
`aux_unification`	179	54	52KB	7.2MB	20%
`Diophantine`	73	44	23KB	1.1MB	3%
`unification`	75	14	19KB	0.8MB	2%
`substitution`	108	16	19KB	1.7MB	5%
`equality`	67	18	12KB	1.1MB	2%
`terms`	129	47	27KB	0.9MB	2%
`list`	251	109	52KB	2.5MB	6%
Total	994	367	243KB	36.7MB	100%

5 Conclusions and Future Work

We have specified Stickel's algorithm [22, 23] for AC-unification in the proof assistant PVS and proved it terminating, sound and complete. Our proof of termination was based on the work of Fages [12, 13]. Since mutual recursion is not straightforward in PVS, we adapted the algorithm to solve an AC-unification problem P, instead of only two terms t and s. This introduces some complications in the proof of termination, which we addressed in Section 4.3.2. We have discussed the most interesting points of our formalisation, such as the motivation for the lexicographic measure needed to prove termination.

We envision three possible paths of future work. First, we could extend this first-order algorithm to the nominal setting. A nominal AC-unification algorithm could be used in a logic programming language that employs the nominal setting such as α-Prolog [9] or in nominal rewriting [14] and narrowing [4] modulo AC. A second possible path is to use this formalisation as a basis to formalise more efficient algorithms, as discussed in the introduction and in Remark 31. Finally, although PVS does not support code extraction to a programming language such as Haskell or Ocaml, it has the PVSIO feature, which lets us execute a verified algorithm inside the PVS environment and provides input and output operators. Therefore, another possible path is using PVSIO to test existing (or to be developed) implementations of AC-unification.

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. doi:10.1016/0747-7171(92)90025-Y.

2 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. doi:10.1016/j.tcs.2019.02.020.

3 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. doi:10.1017/S0960129521000050.

4 Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal Narrowing. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016*, page 11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.FSCD.2016.11.

5 Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

6 Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of Matching Problems. *J. of Sym. Computation*, 3(1/2):203–216, 1987. doi:10.1007/3-540-15976-2_22.

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

8 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. IEEE Computer Society, 1990. doi:10.1109/LICS.1990.113755.

9 James Cheney and Christian Urban. alpha-Prolog: A Logic Programming Language with Names, Binding and α-Equivalence. In *Logic Programming, 20th International Conference, ICLP 2004*, volume 3132 of *LNCS*, pages 269–283. Springer, 2004. doi:10.1007/978-3-540-27775-0_19.

10 Michael Clausen and Albrecht Fortenbacher. Efficient Solution of Linear Diophantine Equations. *J. of Sym. Computation*, 8(1-2):201–216, 1989. doi:10.1016/S0747-7171(89)80025-2.

11 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. Springer, 2004. doi:10.1007/978-3-540-25979-4_5.

12 François Fages. Associative-Commutative Unification. In *7th International Conference on Automated Deduction, Napa*, volume 170 of *LNCS*, pages 194–208. Springer, 1984. doi:10.1007/978-0-387-34768-4_12.

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

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

15 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. The MIT Press, 1991.

16 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 '92)*, pages 11–21. IEEE Computer Society, 1992. doi:10.1109/LICS.1992.185515.

17 Leslie Lamport. How to Write a 21st Century Proof. *Journal of fixed point theory and applications*, 11(1):43–63, 2012. doi:10.1007/s11784-012-0071-6.

18 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 2018*, volume 10895 of *LNCS*, pages 441–458. Springer, 2018. doi:10.1007/978-3-319-94821-8_26.

19 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. Springer, 1992. doi:10.1007/3-540-55602-8_217.

20 Sam Owre, Natarajan Shankar, John Rushby, and David Stringer-Calvert. PVS Language Reference. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2000. URL: https://pvs.csl.sri.com/doc/pvs-language-reference.pdf.

21 Andrew M Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

22 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. URL: http://ijcai.org/Proceedings/75/Papers/011.pdf.

23 Mark E. Stickel. A Unification Algorithm for Associative-Commutative Functions. *J. of the ACM*, 28(3):423–434, 1981. doi:10.1145/322261.322262.

A Pseudocode for instantiateStep

A.1 Pseudocode for instantiateStep

Algorithm 2 is the pseudocode for INSTANTIATESTEP. It receives as input a unification problem P_1 (the part of our unification problem which we have not yet inspected), a unification problem P_2 (the part of our unification problem we have already inspected) and σ, the substitution computed so far. Therefore, the first call to this function in order to instantiate the unification problem P is with $P_1 = P$, $P_2 = nil$ and $\sigma = nil$. The algorithm returns a triple, where the first component is the remaining unification problem; the second component is the substitution computed by this step; and the third component is a Boolean to indicate if we found an equation $t \approx^? s$ which is not unifiable (in this case the Boolean is $True$) or not (in this case the Boolean is $False$). The only kind of equations that INSTANTIATESTEP identifies as not unifiable are those where one of the terms is a variable, and the other term is a non-variable term that contains this variable. The algorithm works by progressively inspecting every equation $s \approx^? t \in P_1$ and deciding whether:

- One of the terms is a variable and we can instantiate (lines 5-10).
- Both terms are the same variable and we can eliminate this equation from the problem (lines 11-12).
- The terms are impossible to unify (lines 13-14).
- Neither term is a variable, and so we do not act on this equation (lines 15-16).

A.2 Pseudocode for applyACStep

▶ Remark 32. In function APPLYACSTEP, we eliminate equations $u \approx^? v$ from our unification problem if $u \approx v$ (line 4). This was done because if we called function SOLVEAC in line 10 of Algorithm 3 passing as parameter two equal terms (modulo AC), the value returned would be $PLst =$ NIL. APPLYACSTEP would interpret that as meaning that the unification pair had no solution (when actually every substitution σ is a solution to $\{u \approx^? v\}$) and also return NIL. To prevent this corner case, we eliminate those trivial equations from our unification problem before calling SOLVEAC. In our code, the function EQUAL? tests equality (modulo AC) between terms t and s, returning $True$ if the terms are equal and $False$ otherwise.

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

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

Algorithm 2 Algorithm that instantiates when possible

```
procedure INSTANTIATESTEP(P1, P2, σ)
    if nil?(P1) then return (P2, σ, False)
    else
        let (t, s) = car(P1), P1' = cdr(P1) in
        if (s matches X) and (X not in t) then
            σ1 = {X → t}
            return INSTANTIATESTEP(σ1 P1', σ1 P2, APPEND(σ1, σ))
        else if (t matches X) and (X not in s) then
            σ1 = {X → s}
            return INSTANTIATESTEP(σ1 P1', σ1 P2, APPEND(σ1, σ))
        else if (t matches X) and (X matches s) then
            return INSTANTIATESTEP(P1', P2, σ)
        else if ((t matches X) and (X in s)) or ((s matches X) and (X in t)) then
            return (nil, σ, True)    ▷ the terms t and s are impossible to unify
        else
            return INSTANTIATESTEP(P1', cons((t, s), P2), σ)    ▷ we skip the equation
```

If t is not equal (modulo AC) to s, we call function SOLVEAC. This function will return a list of unification problems $PLst$ (line 7). Next we apply the function INSTANTIATESTEP to every problem P in $PLst$, obtaining a list $ACInstLst$ (lines 8-9), where each entry is a pair (P', δ). P' is the unification problem after we instantiate the variables and δ is the substitution computed by this function. It may happen that INSTANTIATESTEP "discovers" that a unification problem is actually unsolvable (this is communicated to APPLYACSTEP via the Boolean value that is part of the output of INSTANTIATESTEP) and in this case this problem is not included in $ACInstLst$.

We check if $ACInstLst$ is null (in this case there are no solutions to the first AC-unification pair, and therefore there are no solutions to the problem) and return NIL if it is. If $ACInstLst$ is not null (lines 12-16), there will be branches to explore. Given an entry (P', δ) of $ACInstLst$, the part of the unification problem to which we must call functions SOLVEAC and INSTANTIATESTEP is now $\delta cdr(P_1)$ and the part of the unification problem we have already explored is $\text{APPEND}(P', \delta P_2)$. The substitution computed so far is $\text{APPEND}(\delta, \sigma)$. We take care to update the set of variables that are/were in the problem to include the new variables introduced by SOLVEAC (in Algorithm 3 we change V to V'). In short, we make an input list $InputLst$ of all the branches we need to explore and each entry (P', δ) of $ACInstLst$ gives rise to an entry $(\delta cdr(P_1), \text{APPEND}(P', \delta P_2), \text{APPEND}(\delta, \sigma), V')$ in $InputLst$.

Finally, APPLYACSTEP calls itself recursively taking as argument every input in $InputLst$. This is done by calling MAP(APPLYACSTEP, $InputLst$) and the output is flattened using function FLATTEN.

B PVS Dependency File Diagram

Figure 1 shows the dependency diagram for the PVS theories that compose our formalisation. An arrow going from `theoryA` to `theoryB` means that `theoryA` imports definitions and lemmas from `theoryB`.

▶ Remark 33. The theory `terms` has its definitions and lemmas in the file `terms.pvs` and

Algorithm 3 Algorithm for APPLYACSTEP

procedure APPLYACSTEP(P_1, P_2, σ, V)
  if nil?(P_1) **then return** $cons((P_2, \sigma, V), \text{NIL})$
  else let $(t, s) = car(P_1)$ **in**
    if $t \approx s$ **then return** APPLYACSTEP $(cdr(P_1), P_2, \sigma, V)$
    else
      ▷ assuming t and s are headed by the same function symbol f
      $PLst = \text{SOLVEAC}(t, s, f, V)$
      ▷ Call INSTANTIATESTEP in every P in $PLst$ obtaining a list $ACInstLst$,
      ▷ where each entry in this list is a pair (P', δ).
      if nil?$(ACInstLst)$ **then return** NIL
      else
        ▷ make an input list $InputLst$ of all the branches we need to explore.
        ▷ For each (P', δ) in $ACInstLst$, the quadruple in $InputLst$ will be
        ▷ $(\delta cdr(P_1), \text{APPEND}(P', \delta P_2), \text{APPEND}(\delta, \sigma), V')$ to APPLYACSTEP
        ▷ recursively explore all the branches
        return FLATTEN(MAP(APPLYACSTEP, $InputLst$))

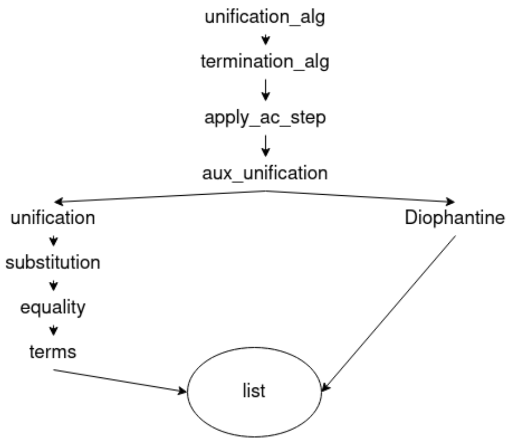

Figure 1 Dependency Diagram for PVS Theories.

the proofs of the lemmas in the file `terms.prf`. The same happens for all the theories mentioned in this diagram, except `list`. In our diagram, `list` represents a set of parametric theories that define generic functions (not strictly connected to unification) that operate on lists. The theories in `list` are `list_nat_theory`, `list_theory`, `list_theory2`, `map_theory` and `more_list_theory_props`. However, since the specifics of each theory in `list` is not significant to our formalisation, we grouped them together in our diagram.

▶ Remark 34 (Adapting the Formalisation to More Efficient Algorithms). The dependency diagram of Figure 1 hints on why adapting our formalisation to prove correctness of algorithms that represents 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 changes in files that depend on `terms.pvs`, specially 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`. Both adaptations should be faster than starting from scratch.

To further illustrate the additional work of changing the term representation in comparison to solving the linear Diophantine equations more efficiently, let's consider the proof of termination of ACUnif, described in Section 4.2, 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.

C 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 35) and is used to obtain a list of arguments of an AC-function headed by f.

▶ **Example 35.** Some examples to illustrate the behaviour of $Args_f$.

- $Args_f(a) = (a)$.
- $Args_f(Y) = (Y)$.
- $Args_f(\langle a, \langle b, c\rangle\rangle) = (a, b, c)$.
- $Args_f(f\langle c, b\rangle) = (c, b)$.
- $Args_f(f\ f\langle c, b\rangle) = (c, b)$.
- $Args_f(g\langle c, b\rangle) = (g\langle c, b\rangle)$.

As mentioned before, terms were defined as shown in Definition 1 in order to make it easier to eventually adapt the formalisation to the nominal setting. 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 has to do with AC-functions that receive only one argument, something allowed in the grammar of terms. Let f be an AC-function symbol and consider Example 36, 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 16). In the definition of well-formed terms, we avoid this problem by requiring that for every AC-function application $f^{AC}s$ that is a subterm of a well-formed term t does not receive only one argument.

▶ **Example 36.** 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 (this proof is in Appendix E) and is the reason why we imposed that a well-formed term t is not a pair.

▶ **Example 37.** Let f be an AC-function symbol and g be a syntactic function symbol. The following terms are well-formed terms:

- $f\langle a, \langle b, c\rangle\rangle$.
- $f\ f\langle a, \langle b, c\rangle\rangle$.
- a.
- $g(Y)$.

The following terms are not well-formed terms:

- fX.
- $\langle a, b\rangle$.

C.1 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 that 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 $size_2$, that has this property (Theorem 39) is given below.

▶ **Definition 38** ($size_2$). *We define the $size_2$ of a term t recursively as follows:*

- $size_2(a) = 1$
- $size_2(Y) = 1$
- $size_2(\langle\rangle) = 1$
- $size_2(\langle t_1, t_2\rangle) = size_2(t_1) + size_2(t_2)$
- $size_2(ft_1) = 1 + size_2(t_1)$
- $size_2(f^{AC}t_1) = \sum_{t_i \in Args_f(f^{AC}t_1)} size_2(t_i)$

▶ **Theorem 39.** *If $t \approx s$ then $size_2(t) = size_2(s)$.*

Theorem 39 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 we do not enter the `else` of line 16.

D Main Lemmas of the Formalisation

In this section we state the main lemmas of our formalisation (concerning functions ACUnif, applyACStep and solveAC) and make brief comments about them.

D.1 Main Lemmas for solveAC

Theorem 40 is used in the proof of soundness of applyACStep while Theorem 41 is used in the proof of completeness of applyACStep. Along with the proof of termination, the completeness of solveAC was one of the hardest part of the formalisation. We give a proof of completeness of solveAC in Appendix E.

▶ **Theorem 40** (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 41** (Completeness of solveAC). *Suppose that t and s are AC-function applications headed by the same symbol f, that t and s are not equal modulo AC, that δ unifies $\{t \approx^? s\}$, that $\delta \subseteq V$ 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 $Vars(im(\gamma)) \subseteq V_1$.*

Recalling the structure of a unification problem obtained after APPLYACSTEP (Section 3.2.2), we see that our hypothesis that $\delta \subseteq V$ means that 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). The theorem guarantees that the substitution γ will only impact the new variables introduced by SOLVEAC, since $dom(\gamma) \subseteq V_1 - V$. In terms of P_1, this means that γ will only impact the right-hand side of P_1.

D.2 Main Lemmas for applyACStep

The main Theorems related with APPLYACSTEP are Theorems 42 and 43, which were used, respectively, in the proof of soundness and completeness of ACUNIF.

▶ **Theorem 42** (Soundness of APPLYACSTEP). *Suppose that $(P', \sigma', V') \in$ APPLYACSTEP(P_1, P_2, σ, V), that δ unifies P', that $\exists \sigma_1 : \delta = \sigma_1\sigma'$, that $dom(\sigma) \subseteq V$ and that $dom(\sigma) \cap (Vars(P_1) \cup Vars(P_2)) = \emptyset$. Then δ unifies P_1.*

Notice that hypothesis $dom(\sigma) \subseteq V$ and that $dom(\sigma) \cap (Vars(P_1) \cup Vars(P_2)) = \emptyset$ of Theorem 42 are immediately satisfied if we have that (APPEND$(P_1, P_2), \sigma, V)$ is a nice input.

▶ **Theorem 43** (Completeness of APPLYACSTEP). *Suppose that δ unifies APPEND(P_1, P_2), that P_1 consists of only AC-unification pairs, that $\delta \subseteq V$, that $\sigma \leq \delta$ and that (APPEND(P_1, 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', $dom(\gamma) \subseteq V' - V$, $img(\gamma) \subseteq V'$ and $\sigma' \leq \gamma\delta$.*

D.3 Main lemmas for ACUnif

The theorems of soundness (Corollary 28) follows immediately from Theorem 27 and the theorem of completeness (Corollary 30) follows immediately from Theorem 29.

▶ **Theorem 44.** *Suppose that (P, σ, V) is a nice input, $\sigma_1 = \{X \mapsto t\}$, $P = \{X \approx^? t\} \cup P_1$, $X \notin Vars(t)$ and $\delta \in$ ACUNIF$(\sigma_1 P_1, \sigma_1\sigma, V)$. If δ unifies $\sigma_1 P_1$, then δ unifies $\{X \approx^? t\}$ and δ unifies P_1.*

▶ **Theorem 27** (Soundness for nice inputs). *Let (P, σ, V) be a nice input, and $\delta \in$ ACUNIF(P, σ, V). Then, δ unifies P.*

Theorem 27 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 difficult cases. The first case is in lines 28-30, when we only have AC-unification pairs (in that case we used the soundness of APPLYACSTEP, i.e. Theorem 42). The second case happens when we instantiate a variable (lines 5-6 and 13-14) and is solved by using Theorem 44.

▶ **Corollary 28** (Soundness of ACUNIF). *If $\delta \in$ ACUNIF$(cons((t, s), \text{NIL}), \text{NIL}, Vars((t, s)))$ then δ unifies $t \approx^? s$.*

This corollary follows from Theorem 27 once we notice that $(cons((t, s), \text{NIL}), \text{NIL}, Vars((t, s)))$ is a nice input.

▶ **Theorem 45.** *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 P_1.*

▶ **Theorem 29** (Completeness for nice inputs). *Let (P, σ, V) be a nice input, δ unifies P, $\sigma \leq \delta$, and $\delta \subseteq V$. Then, there is a substitution $\gamma \in$ ACUNIF(P, σ, V) such that $\gamma \leq_V \delta$.*

Theorem 29 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 difficult cases. The first case is in lines 28-30, when we only have AC-unification pairs (in that case we used the completeness of APPLYACSTEP, i.e. Theorem 43). The second case happens when we instantiate a variable (lines 5-6 and 13-14) and is solved by using Theorem 45.
To see the need for the hypothesis that $\sigma \leq \delta$ 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.

▶ **Corollary 30** (Completeness of ACUNIF). *Let V be a set of variables such that $\delta \subseteq V$ and $Vars((t, s)) \subseteq V$. If δ unifies $t \approx^? s$, then ACUNIF computes a substitution more general than δ, i.e., there is a substitution $\gamma \in$ ACUNIF$(cons((t, s), nil), nil, V)$ such that $\gamma \leq_V \delta$.*

Recall that in our code, the identity substitution is represented by NIL. This theorem follows from 29 once we notice that $(cons((t, s), \text{NIL}), \text{NIL}, Vars((t, s)))$ is a nice input and that for every δ we always have NIL $\leq \delta$.

E A Structured Proof of Completeness of solveAC

We give a structured proof (à la Leslie Lamport [17]) of the completeness of SOLVEAC (Theorem 41). 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 41** (Completeness of SOLVEAC). *Suppose that t and s are AC-function applications headed by the same symbol f, that t and s are not equal modulo AC, that δ unifies $\{t \approx^? s\}$, that $\delta \subseteq V$ 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 you eliminate 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 eliminates the common arguments, the output of SOLVEAC(t, s, V, f) is the same of 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 $|A|_t$ be the number of times the term A (or some term that is equal to A modulo AC) appears in the list of arguments of t, i.e. in $Args_f(t)$. Let $Args(\delta t) = \{A_1, \ldots, A_k\}$ be the set of all the different arguments (modulo AC) of δt.

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

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

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

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

⟨1⟩5. Let $\overrightarrow{n_{A_i}}$ be the vector $(|A_i|_{\delta t_1}, \ldots, |A_i|_{\delta t_m}, |A_i|_{\delta s_1}, \ldots, |A_i|_{\delta s_n})$. 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:

$$\overrightarrow{n_{A_1}} = c'_{11}\overrightarrow{Z'_1} + \ldots + c'_{1l'}\overrightarrow{Z'_{l'}}$$

$$\vdots$$

$$\overrightarrow{n_{A_k}} = c'_{k1}\overrightarrow{Z'_1} + \ldots + c'_{kl'}\overrightarrow{Z'_{l'}}.$$

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:

$$\begin{aligned} \overrightarrow{n_{A_1}} &= c_{11}\overrightarrow{Z_1} + \ldots + c_{1l}\overrightarrow{Z_l} \\ &\vdots \\ \overrightarrow{n_{A_k}} &= c_{k1}\overrightarrow{Z_1} + \ldots + c_{kl}\overrightarrow{Z_l}. \end{aligned} \tag{4}$$

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 $|A_i|_{\delta t_j} > 0$.

⟨2⟩3. Analysing the j-th component of i-th equality in Equation 4, we have $|A_i|_{\delta t_j} = c_{i1}z_{1j} + \ldots + c_{il}z_{lj}$. 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:

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

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

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

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

PROOF:

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

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

PROOF:

$\langle 3\rangle 1$. On one hand, since $Vars(\delta t_i) \subseteq (Vars(im(\delta)) \cup Vars(t_i)) \subseteq V$ and $dom(\gamma) \cap V = \emptyset$ we have $\gamma\delta t_i = \delta t_i$.

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

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

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 and substitutions).

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

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

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

we analyse the i-th entrie of this vectorial equality and conclude that $|u|_{\delta t_i} = |A_j|_{\delta t_i} = c_{j1}z_{1i} + \ldots + c_{jl}z_{li}$.

$\langle 2\rangle 6$. Recall that Z_1 will appears z_{1i} times in $Args_f(t'_i)$, Z_2 will appear z_{2i} times in $Args_f(t'_i)$ and so on - see Section 3.2.1, specially the part about DIOMATRIX2ACSOL . Therefore,

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

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

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

PROOF:

$\langle 2\rangle 1$. All that is left to prove is that EXTRACTSUBMATRICES does not discard the matrix D_1. It is enough to show that D_1 satisfies the two constraints mentioned in Section 3.2.1.

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

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

PROOF:

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

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

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

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