A Certified Sound Algorithm for AC-Unification

Anonymous Author(s)

Abstract

Unification modulo **A**ssociativity and **C**ommutativity (AC) axioms is a crucial notion in rewrite-based programming languages and theorem provers. In this paper, we modify Stickel's seminal AC-unification algorithm to avoid mutual recursion, and formalise it in the PVS proof assistant. More precisely, we show that the adjusted algorithm terminates and is sound. To do this, Fages' termination proof of Stickel's algorithm is adapted, 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 formalised AC-unification algorithm for which soundness has been proved.

Keywords: AC-unification, PVS, Certified algorithms

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 $\sigma s \approx_E \sigma t$.

Unification has practical applications in mathematics and computer science. It is used, for instance, in interpreters of the programming language Prolog, in confluence tests based on critical pairs and so on [BN98]. 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 [Sti75, BN98]).

1.1 Related Work

Unification in the presence of AC-function symbols was first solved by Stickel [Sti75]. He showed how the problem is connected to finding nonnegative integral solutions to linear equations and gave a proof that his algorithm was terminating, sound and complete for a subclass of the general case [Sti75, Sti81]. However, the proof of termination given by Stickel for the general case was incorrect. Almost a decade after the introduction of this algorithm, Fages discovered the flaw and proposed a measure fixing the proof of termination for the general case [Fag84, Fag87]. Below we discuss previous work on solving AC-unification efficiently, on the complexity of AC-unification and on formalising unification modulo equational theories.

Regarding solving AC-unification efficiently: Boudet et al. [BCD90] proposed an efficient AC-unification algorithm, which explores some constraints more efficiently than the standard algorithm; and Adi and Kirchner [AK92] implemented an AC-unification algorithm, proposed benchmarks and showed that their algorithm improves over the previous ones in time and space.

Regarding the complexity of AC-unification, Benanav et al. [BKN87] 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 [KN92] showed that the complexity of computing a complete set of AC-unifiers is double-exponential.

Although, as far as we know, there is no formalisation of AC-unification, there are formalisations of related algorithms and some preliminary work has been done. Regarding formalisation of related problems, Sternagel and Thiemann [ST20] defined a generalised version of Knuth-Bendix orders and formalised several properties such as strong normalisation and the subterm property. Ayala-Rincón et al. [AdCSF+19] formalised nominal α-equivalence for associative, commutative and associative-commutative function symbols. Their work is in the nominal setting (see [Pit13]), which encompasses first-order AC-equivalence.

Some preliminary works left as possible future work the formalisation of AC-unification. In 2004 Contejean [Con04] gave a certified AC-matching algorithm in Coq. AC-matching is an easier problem (see Remark 4) related with AC-unification, where we must find a substitution σ such that $\sigma s \approx_{AC} t$. Additionally, Meßner et al. [MPSS18] 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 this type of equation. Finally, Ayala-Rincón et al. [ARFS19] presented work in progress in formalising nominal AC-unification.

1.2 Contribution

In this work we give the first (as far as we know) formalisation of termination and soundness of a known complete

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

algorithm for AC-unification. This leaves only the proof of completeness for us to have a full formalisation of AC-unification.

We formalised Stickel's algorithm for AC-unification using the proof assistant PVS [ORS92]. When deciding which AC-unification algorithm to formalise, we looked for algorithms that are concise and well established. This led us to the decision of formalising Stickel's algorithm, using Fages' proof of termination.

PVS is an interactive higher-order proof assistant. We chose PVS since we want, as future work, to enrich the nominal unification library that already exists in PVS with a nominal AC-unification algorithm. Some PVS features made the formalisation easier. For instance, by using subtyping predicates we can guarantee that the input to a function satisfies desired properties. Other examples include some automated proofs of termination for simple cases once you give a measure function that decreases at each recursive call, the availability of functional programming features, such as the MAP function, and the possibility of specifying and formalising parametric theories. We made small modifications in 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 [OSRSC99]) and to ease the formalisation.

1.3 Organisation

The paper is organised as follows: in Section 2 we give the necessary background in unification and examples; in Section 3 we present and explain the modification of Stickel's algorithm; in Section 4 we comment on the most interesting points of the formalisation; finally, in Section 5 we conclude the paper and discuss possible paths of future work.

2 Background and Example

From now on, we omit the subscript and denote that two terms s and t are equal modulo AC as $t \approx s$.

Definition 2.1 (Terms)**.** Let Σ be a signature with function symbols and AC-function symbols. Let X be a set of variables. The set $T(\Sigma, X)$ is generated according to the grammar:

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

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

Terms were defined as shown in Definition 2.1 in order to make it easier to eventually adapt the formalisation to the nominal setting, in future work.

Remark 1 (Pairs)**.** *Pairs can be 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).*

Remark 2. *In Definition 2.1 we imposed that a function application $f\,t$ take exactly one term t as argument, which is not a limitation since this term t can be a pair. For instance, the term $f(a, b, c)$ can be represented as $f\langle\langle a, b\rangle, c\rangle$.*

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 $f^{AC}(a, b, c, d)$.*

Notation 2 ($Vars(t)$)**.** *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 $im(\sigma) = \{\sigma X \mid X \in dom(\sigma)\}$.

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 $\{X \to t\}$. The action of a nuclear substitution and a substitution over terms are shown in Definitions 2.2 and 2.3 respectively.

Definition 2.2 (Nuclear substitution action on terms)**.** A nuclear substitution $\{X \to s\}$ acts over a term by induction as shown below:

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

Definition 2.3 (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 3. *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 APPEND(σ, δ).*

We now define AC-unification problems (Definition 2.4), unifiers and a complete set of unifiers (Definition 2.5).

Definition 2.4 (AC-Unification problem)**.** An AC-unification problem is a finite set $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$. 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 2.5 (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' if there is a substitution δ such that $\sigma X \approx \delta\sigma' X$, for all variables X. In this case we say that $\sigma \leq \sigma'$.

With the notion of more general substitution, we can define a complete set of unifiers C of P as a set that satisfies two conditions:

- Each $\sigma \in C$ is an AC-unifier of P.
- For every δ that unifies P, there is a $\sigma \in C$ 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 4. *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 [Con04]) 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 of how we would solve

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

This example is taken from [Sti75].

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

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

We then 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).

Next, we 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 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 can 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 correspondent 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), for this case becomes:

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

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 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 of the unification problems that we will explore will be unsolvable and discarded later, such 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:

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

Remark 5 (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 Algorithm

For readability, in this paper we present the pseudocode of the algorithms, instead of the actual PVS code. We have formalised Algorithm 1 to be terminating and sound. 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 of the algorithm 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 (lines 2-3), 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 5), 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 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 **if** of line 50 of ACUNIF (see Algorithm 1 - Second Part) 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).

Algorithm 1 - First part - algorithm to solve an AC-unification problem P

```
1:  procedure ACUnif(P, σ, V)
2:      if nil?(P) then
3:          return cons(σ, nil)
4:      else
5:          ((t, s), P1) = choose(P)
6:          if (s matches X) and (X not in t) then
7:              σ1 = {X → t}
8:              σ' = append(σ1, σ)
9:              P' = σ1 P1
10:             return ACUnif(P', σ', V)
11:         else
12:             if t matches a then
13:                 if s matches a then
14:                     return ACUnif(P1, σ, V)
15:                 else
16:                     return nil
17:                 end if
18:             else if t matches X then
19:                 if X not in s then
20:                     σ1 = {X → s}
21:                     σ' = append(σ1, σ)
22:                     P' = σ1 P1
23:                     return ACUnif(P', σ', V)
24:                 else if s matches X then
25:                     return ACUnif(P1, σ, V)
26:                 else
27:                     return nil
28:                 end if
29:             else if t matches ⟨⟩ then
30:                 if s matches ⟨⟩ then
31:                     return ACUnif(P1, σ, V)
32:                 else
33:                     return nil
34:                 end if
35:             else if t matches ⟨t1, t2⟩ then
36:                 if s matches ⟨s1, s2⟩ then
37:                     P' = cons((t1, s1), cons((t2, s2), P1))
38:                     return ACUnif(P', σ, V)
39:                 else
40:                     return nil
41:                 end if
```

3.2 The AC-part of the algorithm

The AC-part of Algorithm 1 relies on functions SIMPLIFY (Section 3.2.1) and APPLYACSTEP (Section 3.2.5). APPLYACSTEP depends on two functions: SOLVEAC (Section 3.2.2) and INSTANTIATESTEP (Section 3.2.4). Function SIMPLIFY is necessary in order to prevent a corner case inside function APPLYACSTEP, as explained in Remark 6.

Algorithm 1 - Second Part - Algorithm to solve an AC-unification problem P

```
42:            else if t matches f t1 then
43:                if s matches f s1 then
44:                    P' = cons((t1, s1), P1)
45:                    return ACUnif(P', σ, V)
46:                else
47:                    return nil
48:                end if
49:            else
50:                if s matches f^AC s1 then
51:                    P' = simplify (P)
52:                    InputLst = applyACStep(P', σ, V)
53:                    LstResults = map(ACUnif, InputLst)
54:                    return flatten (LstResults)
55:                else
56:                    return nil
57:                end if
58:            end if
59:        end if
60:    end if
61: end procedure
```

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 52). 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 54 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 simplify. When we enter on lines 51-54 of the algorithm, we first call function simplify that eliminates equations of the unification problem P of the form $u \approx^? v$ such that $u \approx v$.

3.2.2 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 the terms are AC-functions headed by the same 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 [Sti75]) is the maximum of m and n times the maximum of all the least common multiples (lcm) obtained by pairing one of the a_is with 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:

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

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.

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:

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

As noted by Fages in [Fag87], 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 the submatrix D_1 of D, where:

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

Notice that this is the submatrix associated with a solution including only the 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*.

3.2.3 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 $u_1, \ldots, u_m$ 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.
3. A term s_i is an AC-function headed by f only if the corresponding term t_i is a variable.

3.2.4 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 [Fag87]) shows.

Example 3.1 (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. Applying function solveAC to try to unify $f(X, Y)$ with $f(U, V)$ we obtain as one of the branches the unification problem

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

If we then compute the substitution

$$\sigma = \{X \to f(X_1, X_2), Y \to f(X_3, X_4), U \to f(X_1, X_3), V \to f(X_2, X_4)\}.$$

and apply σ on the remaining equations we have to unify we get

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

If we then solve 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 beggining. 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 apply function 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 7-15).
- Both terms are the same variable and we can eliminate this equation from the problem (lines 17-18).
- The terms are impossible to unify (lines 20-23).
- Neither term is a variable, and so we do not act on this equation (lines 25-28).

Algorithm 2 Algorithm that instantiates when possible

```
1:  procedure INSTANTIATESTEP(P_1, P_2, σ)
2:      if nil?(P_1) then
3:          return (P_2, σ, False)
4:      else
5:          (t, s) = car(P_1)
6:          P'_1 = cdr(P_1)
7:          if (s matches X) and (X not in t) then
8:              σ_1 = {X → t}
9:              σ' = APPEND(σ_1, σ)
10:             return INSTANTIATESTEP(σ_1 P'_1, σ_1 P_2, σ')
11:
12:         else if (t matches X) and (X not in s) then
13:             σ_1 = {X → s}
14:             σ' = APPEND(σ_1, σ)
15:             return INSTANTIATESTEP(σ_1 P'_1, σ_1 P_2, σ')
16:
17:         else if (t matches X) and (X matches s) then
18:             return INSTANTIATESTEP(P'_1, P_2, σ)
19:
20:         else if ((t matches X) and (X in s)) or
21:            ((s matches X) and (X in t))  then
22:             ▷ the terms t and s are impossible to unify
23:             return (nil, σ, True)
24:
25:         else
26:             ▷ we skip the equation
27:             P'_2 = cons((t, s), P_2)
28:             return INSTANTIATESTEP(P'_1, P'_2, σ)
29:         end if
30:     end if
31: end procedure
```

3.2.5 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.

Algorithm 3 gives the pseudocode (in a higher level than the ones before) for the algorithm. 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.

The first thing APPLYACSTEP does is check if P_1 is the null list. If it is (lines 2-3), 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 call function SOLVEAC. This function will return a list of unification problems $PLst$ (line 8).

Next we apply the function INSTANTIATESTEP to every problem P in $PLst$, obtaining a list $ACInstLst$ (lines 10-12), 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 17-26), 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 $cdr(P_1)$ and the part of the unification problem we have already explored is APPEND(P', P_2). The substitution computed so far is APPEND(δ, σ). 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 $(cdr(P_1), \text{APPEND}(P', P_2), \text{APPEND}(\delta, \sigma), V')$ 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.

Remark 6 (The need for SIMPLIFY). *In Algorithm 1, we called function SIMPLIFY in order to eliminate equations of the form $u \approx^? v$ when $u \approx v$ from our unification problem. This was done because if we called function SOLVEAC in line 8 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 call function SIMPLIFY before calling APPLYACSTEP.*

4 Interesting Points on the Formalisation

Section 4.1 describes how we adapted the specification of Stickel's algorithm of [Fag87] to avoid mutual recursion; Sections 4.2 defines the lexicographic measure for termination and Section 4.3 explains the most interesting points in the proof of termination. Section 4.4 explains the most interesting points for the proof of soundness. Finally, Section 4.6 gives more information about our formalisation.

Algorithm 3 Algorithm for applyACStep

```
1:  procedure applyACStep(P1, P2, σ, V)
2:      if nil?(P1) then
3:          return cons((P2, σ, V), nil)
4:      else
5:          (t, s) = car(P1)
6:          ▷ assuming t and s are headed by the same
7:          ▷ function symbol f
8:          PLst = solveAC(t, s, f, V)
9:
10:         ▷ Call instantiateStep in every P in PLst
11:         ▷ obtaining a list ACInstLst, where each entry
12:         ▷ in this list is a pair (P', δ).
13:
14:         if nil?(ACInstLst) then
15:             return nil
16:         else
17:             ▷ make an input list InputLst of all
18:             ▷ the branches we need to explore. For each
19:             ▷ (P', δ) in ACInstLst, the quadruple in
20:             ▷ InputLst will be
21:             ▷ (cdr(P1), append(P', P2), append(δ, σ), V')
22:             ▷ to applyACStep
23:
24:             ▷ recursively explore all the branches
25:             return
26:             Flatten(map(applyACStep, InputLst))
27:         end if
28:     end if
29: end procedure
```

4.1 Avoiding Mutual Recursion

When specifying Stickel's algorithm, we tried to follow closely the pseudocode presented in [Fag87] (the papers [Sti75, Sti81] give a higher-level description of the algorithm). In [Fag87] 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, which is something not allowed in PVS[1] [OSRSC99].

We have adapted the algorithm to use only one main function, which receives a unification problem P and operates (with the exception of 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 [Fag87]) 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).

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

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 4.1, 4.4, 4.7 and 4.9, respectively.

Definition 4.1 ($V_{NAC}(P)$)**.** 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 4.2. 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 4.3 ($Subterms(P)$)**.** 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:

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

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

Remark 7 (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 4.4 ($V_{>1}(P)$)**.** 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 4.5. 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 $V_{>1}(P) = \{Y\}$.

We define proper subterms in order to define admissible subterms in Definition 4.7.

Definition 4.6 (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 4.7 ($AS(P)$)**.** 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 4.8. Let

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

Then $AS(P) = \{h(c), c\}$.

Definition 4.9 ($size(P)$)**.** 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 8 ($s \in AS(t) \implies size(s) < size(t)$)**.** *Given a term t and a term s, such that $s \in AS(t)$, we have that s is a proper subterm of t and therefore the size of s is less than the size of t.*

4.2.1 Which Component Decreases in Each Recursive Call.

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.

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 10	$<$			
line 14	$\leq$	$\leq$	$\leq$	$<$
line 23	$<$			
line 25	$\leq$	$\leq$	$\leq$	$<$
line 31	$\leq$	$\leq$	$\leq$	$<$
line 38	$\leq$	$\leq$	$\leq$	$<$
line 45	$\leq$	$\leq$	$\leq$	$<$
case 1 - line 53	$\leq$	$<$		
case 2 - line 53	$\leq$	$\leq$	$<$	
case 3 - line 53	$\leq$	$\leq$	$\leq$	$<$

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 53) 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 3.2.3). 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 14, 25, 31, 38, 45) 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 [Fag87]. 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 3.2.3, the left-hand side of P_1 is $\{t_1, \ldots, t_m, s_1, \ldots, s_n\}$. Denote by $\{t'_1, \ldots, t'_m, s'_1, \ldots, s'_n\}$ the right-hand side of P_1, which means that $P_1 = \{t_1 \approx^? t'_1, \ldots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \ldots, s_n \approx^? s'_n\}$. This is what

motivated our definition of admissible subterms: every term t'_i of the right-hand side of P_1 will have $AS(t'_i) = \emptyset$. Therefore, $AS(P_1) \subseteq AS(P)$ always holds.

If we are also in a situation where at least one of the terms 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 8), 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 4.10. In the example of Section 2.2, the unification problem before applying SOLVEAC was:

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

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') 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 4.11 shows.

Example 4.11 (A case where INSTANTIATESTEP increases $|AS|$)**.** Let f and g be AC-function symbols. If we apply INSTANTIATESTEP in the problem

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

we would 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 4.11 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 4.12.

Example 4.12 (A case where INSTANTIATESTEP does not increase $|AS|$)**.** If we change slightly the problem from Example 4.11 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 $P_1 = \{t_1 \approx^? t'_1, \ldots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \ldots, s_n \approx^? s'_n\}$, 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 3.2.3) 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 $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$. 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

As mentioned, to unify terms t and s we use Algorithm 1 with $P = cons((t,s), nil)$, $\sigma = \text{NIL}$ and $V = Vars((t,s))$. Therefore, proving soundness of Algorithm 1 boils down to proving Corollary 4.13:

Corollary 4.13. *If $\delta \in ACUNIF(cons((t,s), \text{NIL}), \text{NIL}, Vars((t,s)))$ then $\delta t \approx \delta s$.*

Since the parameters of ACUNIF may change in between the recursive calls, we cannot prove Corollary 4.13 directly by induction. We must prove the more general Theorem 4.15, 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 4.14.

Definition 4.14 (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 4.15. *If $\delta \in ACUNIF(P, \sigma, V)$ and (P, σ, V) is a nice input then δ is a unifier to P.*

The proof is by induction on the lexicographic measure, according to the form of the terms t and s that were selected by CHOOSE. The hardest cases are the ones where we instantiate variables and the case where we call APPLYACSTEP. We give a structured proof (à la Leslie Lamport, see [Lam12]) of the case where we instantiate variables. 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. Additionally, we give a sketch of the proof for the case where we call APPLYACSTEP.

PROOF FOR THE INSTANTIATION OF VARIABLES:

$\langle 1\rangle 1$. We consider the case where s is a variable X that is not in t. In this case, we have $\sigma_1 = \{X \to t\}$ and therefore $\delta \in \text{ACUNIF}(\sigma_1 P_1, \sigma_1\sigma, V)$.

$\langle 1\rangle 2$. It is possible to prove if $\delta \in \text{ACUNIF}(P, \sigma, V)$, then there exists a substitution σ_2 such that $\delta = \sigma_2\sigma$. In our case, since $\delta \in \text{ACUNIF}(\sigma_1 P_1, \sigma_1\sigma, V)$ this means that there is σ_2 such that $\delta = \sigma_2\sigma_1\sigma$.

$\langle 1\rangle 3$. It is simple to check that $(\sigma_1 P_1, \sigma_1\sigma, V)$ is a nice input. In this case, we can apply the induction hypothesis and get that δ unifies $\sigma_1 P_1$. We have to prove that δ unifies $P = \{t \approx^? s\} \cup P_1$.

$\langle 1\rangle 4$. $\delta t \approx \delta s$.

PROOF:

$\langle 2\rangle 1$. Since $\delta = \sigma_2\sigma_1\sigma t$, we must prove that $\sigma_2\sigma_1\sigma t \approx \sigma_2\sigma_1\sigma s$.

$\langle 2\rangle 2$. Since t is in P and (σ, P, V) is a nice input, we have $Vars(t) \cap dom(\sigma) = \emptyset$. Therefore $\sigma t = t$. A similar reasoning works for s and let us conclude that $\sigma s = s$. Therefore, we must prove that $\sigma_2\sigma_1 t \approx \sigma_2\sigma_1 s$.

$\langle 2\rangle 3$. By the definition of σ_1, we get $\sigma_1 s = \sigma_1 X = t$. Since X is not in t we have $\sigma_1 t = t$.

$\langle 2\rangle 4$. Therefore, what we need to prove is $\sigma_2 t \approx \sigma_2 t$, which is straightforward.

$\langle 1\rangle 5$. δ unifies P_1.

PROOF:

$\langle 2\rangle 1$. It suffices to prove that $\delta t_i \approx \delta s_i$ for an arbitrary equation $\{t_i \approx^? s_i\} \in P$.

$\langle 2\rangle 2$. By Step $\langle 1\rangle 3$ we have that δ unifies $\sigma_1 P_1$. From this we can conclude that $\delta\sigma_1 t_i \approx \delta\sigma_1 s_i$.

$\langle 2\rangle 3$. Since $\delta = \sigma_2\sigma_1\sigma$, we get that $\sigma_2\sigma_1\sigma\sigma_1 t_i \approx \sigma_2\sigma_1\sigma\sigma_1 s_i$.

$\langle 2\rangle 4$. Since (P, σ, V) is a nice input we have $Vars(P) \cap dom(\sigma) = \emptyset$. We have $Vars(\sigma_1 t_i) \subseteq Vars(P)$, which let us conclude that $\sigma\sigma_1 t_i = \sigma_1 t_i$. The same reasoning let us conclude that $\sigma\sigma_1 s_i = \sigma_1 s_i$ and simplify our expression to $\sigma_2\sigma_1\sigma_1 t_i \approx \sigma_2\sigma_1\sigma_1 s_i$.

$\langle 2\rangle 5$. It can be prove that σ_1 is an idempotent substitution and therefore that $\sigma_2\sigma_1 t_i \approx \sigma_2\sigma_1 s_i$.

$\langle 2\rangle 6$. Finally, since $Vars(t_i) \cap dom(\sigma) = \emptyset$ we can rewrite t_i as σt_i. We can do the same for s_i. Then, what we have is $\sigma_2\sigma_1\sigma t_i \approx \sigma_2\sigma_1\sigma s_i$, which is simply $\delta t_i \approx \delta s_i$.

The soundness of the case where we call APPLYACSTEP relies on the soundness of functions SOLVEAC and INSTANTIATESTEP. The proof of soundness of INSTANTIATESTEP is similar to the proof we just gave where Algorithm 1 instantiates a variable, since all INSTANTIATESTEP does is successive instantiations of variables.

To sketch the proof of soundness of SOLVEAC let's go back to our example where SOLVEAC is called with inputs t and s, and returns $P_1 = \{t_1 \approx^? t'_1, \ldots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \ldots, s_n \approx^? s'_n\}$ where $t_1, \ldots, t_m$ are the arguments of t and $s_1, \ldots, s_n$ are the arguments of s. Then if δ unifies P_1:

$$\delta t_1 \approx \delta t'_1, \ldots, \delta t_m \approx \delta t'_m, \delta s_1 \approx \delta s'_1, \ldots, \delta s_m \approx \delta s'_m. \quad (3)$$

Due to the correctness of our process for solving the Diophantine equations we know that the number of times any variable appear on $\{t'_1, \ldots, t'_m\}$ weighted by the multiplicity of $t_1, \ldots, t_n$ in t is equal to the number of times this variable appear on $\{s'_1, \ldots, s'_n\}$ weighted by the multiplicity of $s_1, \ldots, s_n$ in s. This, along with Equation 3, allows us to conclude that $\delta t \approx \delta s$.

4.5 Additional comments on the formalisation

Remark 9 (Limitation). *When function* DIOSOLVER *returns a matrix with natural coefficients to represent the solutions of the linear Diophantine equation*

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

we do not calculate a basis of solutions to this 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 to this equation. This was done in order to ease the specification of* DIOSOLVER *and the formalisation of its properties.*

Remark 10 (Possible Pitfall). *Given $l = (t_1, \ldots, t_n)$, a list of terms, let us denote by σl the list $(\sigma t_1, \ldots, \sigma t_n)$. A frequently used function in our code is* ARGS*, used to extract the arguments of AC-functions. For instance, if $t = f(t_1, \ldots, t_n)$ then* ARGS(t) *returns the list $(t_1, \ldots, t_n)$. The output of* ARGS(σt) *is not simply* σARGS(t)*. To illustrate this, suppose that f is an AC-function symbol, and consider $t = f(X, Y)$ and $\sigma = \{X \to f(a, b), Y \to f(c, d)\}$. Then* σARGS$(t) = (\sigma X, \sigma Y) = (f(a, b), f(c, d))$*, while* ARGS$(\sigma t)$ = ARGS$(f(a, b, c, d)) = (a, b, c, d)$*. However, if we apply* ARGS *to every element in the list* σARGS(t) *and then flatten the result into a list of terms, we obtain the same result as* ARGS(σt)*, i.e.:*

$$\text{ARGS}(\sigma t) = \text{FLATTEN}(\text{MAP}(\text{ARGS}, \sigma\text{ARGS}(t))).$$

4.6 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` files and their descriptions are shown below:

- `unification_alg.pvs` - Function ACUNIF (Algorithm 1) and the theorem of soundness.
- `termination_alg.pvs` - Definitions and theorems necessary for proving termination.
- `apply_ac_step.pvs` - Function APPLYACSTEP and its properties.
- `aux_unification.pvs` - Auxiliary functions such as SOLVEAC and INSTANTIATESTEP and its properties.
- `diophantine.pvs` - Code to solve Diophantine equations.
- `unification.pvs` - Definition of a unification problem and basic properties.
- `substitution.pvs` - Properties about substitutions.
- `equality.pvs` - Properties about equality modulo AC.
- `terms.pvs` - Basic properties about terms.
- `list.pvs` - 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 most of them automatically. The number of theorems and TCCs proved for each file are shown in Table 3.

Table 3. Number of theorems and TCCs in each file.

File	Theorems	TCCs	Total
`unification_alg.pvs`	6	8	14
`termination_alg.pvs`	84	35	119
`apply_ac_step.pvs`	16	10	26
`aux_unification.pvs`	134	44	178
`diophantine.pvs`	20	25	45
`unification.pvs`	43	11	54
`substitution.pvs`	76	16	92
`equality.pvs`	36	12	48
`terms.pvs`	115	44	159
`list.pvs`	161	84	245
Total	691	289	**980**

5 Conclusions and Future Work

We have specified Stickel's algorithm [Sti75, Sti81] for AC-unification in the proof assistant PVS and proved it sound and terminating. Our proof of termination was based on the work of Fages [Fag84, Fag87]. Since mutual recursion is not straightforward in PVS, we adapted the algorithm to unify an AC-unification problem P, instead of only two terms t and s. This introduces complications in the proof of termination, which we show how to solve 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.

The immediate future work for this paper is proving that the specified algorithm is complete, i.e., that the list of substitutions returned by ACUNIF is a complete set of unifiers. This would give us the first (to the best of our knowledge) formalised sound and complete AC-unification algorithm. After that a possible path is to 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 ([CU04]) or in nominal rewriting and narrowing ([ARFNS16]) modulo AC.

References

[AdCSF+19] 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.

[AK92] Mohamed Adi and Claude Kirchner. AC-Unification Race: The System Solving Approach, Implementation and Benchmarks. *J. of Sym. Computation*, 14(1):51–70, 1992.

[ARFNS16] 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.

[ARFS19] Mauricio Ayala-Rincón, Maribel Fernández, and Gabriel Ferreira Silva. Formalising Nominal AC-Unification. *Proceedings of the Internation Workshop on Unification, UNIF*, 2019.

[BCD90] 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), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 289–299. IEEE Computer Society, 1990.

[BKN87] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of Matching Problems. *J. of Sym. Computation*, 3(1/2):203–216, 1987.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge UP, 1998.

[Con04] Evelyne Contejean. A Certified AC Matching Algorithm. In Vincent van Oostrom, editor, *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.

[CU04] 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, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *LNCS*, pages 269–283. Springer, 2004.

[Fag84] François Fages. Associative-Commutative Unification. In Robert E. Shostak, editor, *7th International Conference on Automated Deduction, Napa, California, USA, May 14-16, 1984, Proceedings*, volume 170 of *LNCS*, pages 194–208. Springer, 1984.

[Fag87] François Fages. Associative-Commutative Unification. *J. of Sym. Computation*, 3(3):257–275, 1987.

[KN92] 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), Santa Cruz, California, USA, June 22-25, 1992*, pages 11–21. IEEE Computer Society, 1992.

[Lam12] Leslie Lamport. How to write a 21 st century proof. *Journal of fixed point theory and applications*, 11(1):43–63, 2012.

[MPSS18] 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, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *LNCS*, pages 441–458. Springer, 2018.

[ORS92] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

[OSRSC99] Sam Owre, Natarajan Shankar, John Rushby, and David Stringer-Calvert. PVS Language Reference. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1(2):21, 1999.

[Pit13] Andrew M Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge UP, 2013.

[ST20] Christian Sternagel and René Thiemann. A Formalization of Knuth-Bendix Orders. *Arch. Formal Proofs*, 2020, 2020.

[Sti75] Mark E. Stickel. A Complete Unification Algorithm for Associative-Commutative Functions. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*, pages 71–76, 1975.

[Sti81] Mark Stickel. A Unification Algorithm for Associative-Commutative Functions. *J. of the ACM*, 28(3):423–434, 1981.