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.FSCD.2022.8

Related Version Extended version available at: https://www.mat.unb.br/ayala/

Supplementary Material Source code available through hyperlinks on the paper.

Funding Research supported by a FAP-DF (DE 00193.00001175/2021-11) and a CNPq (Universal 409003/2021-2) grant. First author partially funded by a CNPq productivity research grant 313290/2021-0. Fourth author partially funded by Edital DPI/DPG n. 03/2020.

1 Introduction

Syntactic unification is the problem of, given terms $s$ and $t$, finding a substitution $\sigma$ 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 $\sigma$ such that $\sigma s$ and $\sigma t$ 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 Associativity and Commutativity 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 [21]. 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 [21, 22]. 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 [20]), 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 σs ≈_{AC} t. A formalisation of nominal C-unification, which can also handle nominal C-matching, is also available [3]. Additionally, Meßner et al. [17] 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 [18]. 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 [19]) 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 36, 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 $\Sigma$ be a signature with function symbols and AC-function symbols. Let $X$ be a set of variables. The set $T(\Sigma, X)$ is generated by the grammar:

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

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 $f^{\text{AC}} t$ is an associative-commutative function application.

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

Definition 3 (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 1 (AC-Unification pairs). When $t$ and $s$ are both headed by the same AC-function symbol, we refer to the equation $t \approx s$ as an AC-unification pair.

To ease our formalisation (more details in the extended version), 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.
A Certified Algorithm for AC-Unification

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 \( \sigma \) such that \( \sigma t_i \approx \sigma s_i \) for every \( i \) from 1 to \( n \).

A substitution \( \sigma \) is more general (modulo AC) than a substitution \( \sigma' \) in a set of variables \( V \) if there is a substitution \( \delta \) such that \( \sigma' X \approx \delta \sigma X \), for all variables \( X \in V \). In this case we write \( \sigma \leq_V \sigma' \). When \( V \) is the set of all variables, we write \( \sigma \leq \sigma' \).

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

We represent an AC-unification problem \( P \) as a list in our PVS code, where each element of the list is a pair \((t_i, s_i)\) that represents an equation \( t_i \approx^? s_i \). Finally, given a unification problem \( P = \{ t_1 \approx^? s_1, \ldots, t_n \approx^? s_n \} \), we define \( \sigma P \) 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 \( \sigma_3 = \{ X_1 \to f(a, W), Y \to f(X_2, W) \} \). This solution would be missed because the arguments of \( \sigma_3 Y = f(X_2, W) \) are partially contained in \( \sigma_3 X_1 = f(a, W) \) and partially contained in \( \sigma_3 X_2 = X_2 \).

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

2.2 An Example

Before presenting the pseudocode for the algorithm we formalised, we give a higher-level example (taken from the very accessible [22]) of how we would solve \( \{ f(X, X, Y, a, b, c) \approx f(b, b, b, c, Z) \} \). In a high-level view, this technique converts an AC-unification problem into a linear Diophantine equation and uses a basis of solutions of the Diophantine equation to get a complete set of AC-unifiers to our original problem.

The first step is to eliminate common arguments in the terms that we are trying to unify. The problem 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 coefficient of variable \( X_1 \) is two, since argument \( X \) occurs twice in \( f(X, X, Y, a) \) and so on).

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Solutions for the equation ( 2X_1 + X_2 + X_3 = 2Y_1 + Y_2 ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_1 )</td>
<td>( X_2 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The third step is to generate a basis of solutions to the equation and associate a new variable (the \( Z_i \)s) to each solution. As we shall soon see, the unification problem \( \{ f(X, X, Y, a) \approx f(b, b, Z) \} \) may branch into (possibly) many unification problems and the new variables \( Z_i \)s will be the building blocks for the right-hand side of these unification problems. The result is shown on Table 1. Observing Table 1 we relate the “old variables”
(X is and Y is) with the “new variables” (Z is):

\[
\begin{align*}
X_1 &= Z_6 + Z_7 \\
X_2 &= Z_2 + Z_4 + 2Z_5 \\
X_3 &= Z_1 + 2Z_3 + Z_4 \\
Y_1 &= Z_3 + Z_4 + Z_5 + Z_7 \\
Y_2 &= Z_1 + Z_2 + 2Z_6.
\end{align*}
\]

(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. Eliminating the cases that do not respect this constraint, we are left with 69 cases.

For example, if we decide to include only the solutions represented by the variables \(Z_1, Z_3\) and \(Z_6\), the corresponding unification problem, according Equations (1), becomes:

\[
P = \{X_1 \approx^\sigma Z_6, X_2 \approx^\sigma Z_4, X_3 \approx^\sigma f(Z_1, Z_4), Y_1 \approx^\sigma Z_4, Y_2 \approx^\sigma f(Z_1, Z_6, Z_6)\}.
\]

(2)

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

Finally, we replace the variables \(X_1, X_2, X_3, Y_1, Y_2\) by the original arguments they substituted and proceed with the unification. Some unification problems that we will explore will be unsolvable and discarded later, as: \(\{X \approx^\sigma Z_6, Y \approx^\sigma Z_4, a \approx^\sigma Z_1, b \approx^\sigma f(Z_6, Z_6)\}\) (we cannot unify both \(a\) with \(Z_4\) and \(b\) with \(Z_4\) simultaneously). In the end, the solutions computed will be:

\[
\sigma_1 = \{Y \rightarrow f(b, b), Z \rightarrow f(a, X, X)\}, \quad \sigma_2 = \{Y \rightarrow f(Z_2, b, b), Z \rightarrow f(a, Z_2, X, X)\},
\]

\[
\sigma_3 = \{X \rightarrow b, Z \rightarrow f(a, Y)\}, \quad \sigma_4 = \{X \rightarrow f(Z_6, b), Z \rightarrow f(a, Y, Z_6, Z_6)\}.
\]

(3)

**Remark 9.** When using the technique described in this section to unify \(f(X, X, Y, a, b, c)\) with \(f(b, b, b, c, Z)\), we obtained unification problems that only contain the variables \(X_1, X_2, X_3, Y_1, Y_2\) or AC-functions whose arguments are all variables (for instance \(P\) in Equation 2). However, this does not mean that our technique cannot be applied to general AC-unification problems, since we eventually replace the variables \(X_1, X_2, X_3, Y_1, Y_2\) by their corresponding arguments \((X, Y, a, b, Z\) respectively) and proceed with unification.

**Remark 10 (Cases on AC1-Unification).** If we were considering AC1-unification, where our signature has an identity \(\text{id}\) 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 \(\text{id}\).

### 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 \(\sigma\) computed so far, and the variables \(V\) that are/were in the problem. The output is a list of
Algorithm 1 Algorithm to Solve an AC-Unification Problem \( P \).

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

substitutions, where each substitution \( \delta \) 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 \( P = \text{cons}((t, s), \text{NIL}) \), \( \sigma = \text{NIL} \) (because we have not computed any substitution yet) and \( V = \text{Vars}((t, s)) \).

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 \( \sigma \) computed so far. Otherwise the algorithm calls the auxiliary function \text{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.

\begin{itemize}
  \item \textbf{Remark 11.} The algorithm does not check arity consistency of the input.
\end{itemize}

3.1 The Functions \text{choose} and \text{decompose}

The function \text{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 aid us in the proof of termination; 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); and is not a significant deviation from Stickel’s algorithm [22].

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 \((P, \text{bool})\), where \( P \) is a unification problem and \( \text{bool} \) 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 \( \text{bool} = \text{True} \). If one of the terms is a pair and the other is not, the function returns \((\text{nil}, \text{False})\). In Algorithm 1, we call decompose \((t_1, s_1)\) when we encounter an equation of the form \( f t_1 \approx^? f s_1 \) 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 \( f t_1 \approx^? f s_1 \) 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 12. Below we give examples of function decompose.

- decompose\((\langle a, \langle b, c \rangle \rangle, \langle c, \langle X, Y \rangle \rangle)\) = \((\{ a \approx^? c, b \approx^? X, c \approx^? Y \}, \text{True})\)
- decompose\((a, Y)\) = \((\{ a \approx^? Y \}, \text{True})\)
- decompose\((X, \langle c, d \rangle)\) = \((\text{nil}, \text{False})\)

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 \((\text{InputLst} \text{ in Algorithm 1})\), 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 ease the formalisation we do not calculate a basis of solutions for the linear Diophantine equation, but a spanning set (which is not necessarily linearly independent). To generate this spanning set, it suffices to calculate all the solutions until an upper bound, computed by function calculateUpperBound. Given a linear Diophantine equation \( a_1 X_1 + \ldots + a_m X_m = b_1 Y_1 + \ldots + b_n Y_n \), our upper bound (taken from [21]) is the maximum of \( m \) and...
n times the maximum of all the least common multiples (lcm) obtained by pairing each one of the $a_i$s with each one of the $b_j$s. In other words, our upper bound is: $\max(m,n) \ast \max_{i,j}(\text{lcm}(a_i,b_j))$.

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

The function dioSolver receives as input the multiplicity of the arguments of $t$ and $s$ and the upper bound calculated by calculateUpperBound and calculates the spanning set of solutions, returning a matrix. For instance, the Table 1 of the Example in Section 2.2 would be represented in our code as the matrix $D$. Each row of $D$ is associated with one solution and thus with one of the new variables. Each column of $D$ is associated with one of the arguments of $t$ or $s$. Modifying dioSolver to calculate a basis of solutions (for instance, by using the method described in [10]) instead of a spanning set would certainly improve the efficiency of the algorithm.

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 $X_1, \ldots, X_m, Y_1, \ldots, Y_n$ 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 $P_1 = \{X \approx Z_6, Y \approx Z_6, a \approx Z_4, b \approx Z_4, Z \approx f(Z_6, Z_6)\}$ would be obtained from submatrix $D_1$.

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

Notice that this is the submatrix associated with a solution including only 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 $Z_i$s 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 \texttt{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 \( P = \{ t_1 \approx^? s_1, \ldots, t_k \approx^? s_k \} \) is one of the unification problems generated by function \texttt{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 \texttt{solveAC} and are either new variables \( Z_i \)'s or AC-functions headed by \( f \) whose arguments are all new variables \( Z_i \)'s (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 \texttt{solveAC}, we instantiate the variables that we can by calling function \texttt{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.

\[ \textbf{Example 13 (Looping forever).} \text{ Let } f \text{ be an AC-function symbol. Suppose we want to solve } P = \{ f(X,Y) \approx^? f(U,V), X \approx^? Y, U \approx^? V \} \text{ and instead of instantiating the variables as soon as we can, we decide to try solving the first equation. Applying function } \texttt{solveAC} \text{ to try to unify } f(X,Y) \text{ with } f(U,V) \text{ we obtain as one of the branches the unification problem } \{ X \approx^? f(X_1,X_2), Y \approx^? f(X_3,X_4), U \approx^? f(X_1,X_3), V \approx^? f(X_2,X_4) \} \text{. We can solve this branch by instantiating } X, Y, U, \text{ and } V. \text{ After these instantiations, we have to unify the remaining two equations: } \{ f(X_1,X_2) \approx^? f(X_3,X_4), f(X_1,X_3) \approx^? f(X_2,X_4) \}. \text{ Solving the first equation, one branch obtained is } \{ X_1 \approx^? X_3, X_2 \approx^? X_4 \}, \text{ which gets us back to } P = \{ f(X_1,X_3) \approx^? f(X_2,X_4), X_1 \approx^? X_3, X_2 \approx^? X_4 \}, \text{ which is essentially the same unification problem we started with.} \]

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 \texttt{solveAC} we immediately instantiate the variables that we can by calling function \texttt{instantiateStep}.

3.2.4 Function applyACStep

Function \texttt{applyACStep} relies on functions \texttt{solveAC} and \texttt{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 \texttt{solveAC} and \texttt{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 \( \sigma \), and the set of variables to avoid \( V \). \( P_1 \) and \( P_2 \) are, respectively, the subset of the unification problem for which functions \texttt{solveAC} and
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 [21, 22] 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\(^1\) [19].

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 to the algorithm makes. We have chosen a lexicographic measure with four components: \( \text{lex} = (|V_{\text{NAC}}(P)|, |V_{>1}(P)|, |\text{AS}(P)|, \text{size}(P)) \), where \( V_{\text{NAC}}(P) \), \( V_{>1}(P) \), \( \text{AS}(P) \), \( \text{size}(P) \) are given in Definitions 14, 18, 21 and 23, 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.

\[\text{Definition 14}\] \( (V_{\text{NAC}}(P)) \). We denote by \( V_{\text{NAC}}(P) \) the set of variables that occur in the problem \( P \) excluding those that only occur as arguments of AC-function symbols.

\[\text{Example 15}\]. 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 \( V_{\text{NAC}}(P) = \{ X, Y, Z \} \).

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

\[\text{Definition 16}\] \( (\text{Subterms}(P)) \). The subterms of a unification problem \( P \) are given as: \( \text{Subterms}(P) = \bigcup_{t \in P} \text{Subterms}(t) \), where the notion of subterms of a term \( t \) excludes all pairs and is defined recursively as follows:

- \( \text{Subterms}(a) = \{ a \} \)
- \( \text{Subterms}((t_1, t_2)) = \text{Subterms}(t_1) \cup \text{Subterms}(t_2) \)
- \( \text{Subterms}(Y) = \{ Y \} \)
- \( \text{Subterms}(f t_1) = \{ f t_1 \} \cup \text{Subterms}(t_1) \)
- \( \text{Subterms}(\emptyset) = \{ \emptyset \} \)
- \( \text{Subterms}(f^{AC} t_1) = \bigcup_{t_i \in \text{Args}(f^{AC} t_1)} \text{Subterms}(t_i) \cup \{ f^{AC} t_1 \} \)

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

---

\(^1\) 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.
Remark 17 (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 $t = f(f(a,b), f(c,d))$. 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($t$) = \{ $t, f(a,b), f(c,d), a, b, c, d$ \}.

Definition 18 ($V_{\geq 1}(P)$). We denote by $V_{\geq 1}(P)$ 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 $X \in V_{\geq 1}(P)$ then $X$ is an argument to at least two different function symbols.

Example 19. Let $f$ be an AC-function symbol and let $g$ be a standard function symbol. Let $P = \{ X \approx^7 a, g(X) \approx^7 h(Y), f(Y, W, h(Z)) \approx^7 f(c, W) \}$. In this case $V_{\geq 1}(P) = \{ Y \}$.

We define proper subterms in order to define admissible subterms in Definition 21.

Definition 20 (Proper Subterms). If $t$ is not a pair, we define the proper subterms of $t$, denoted as $PSubterms(t)$ as: $PSubterms(t) = \{ s \mid s \in Subterms(t) and s \neq t \}$. 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 21 (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 22. If $P = \{ a \approx^7 f(Z_1, Z_2), b \approx^7 Z_3, g(h(c), Z) \approx^7 Z_4 \}$ then $AS(P) = \{ h(c), c \}$.

Definition 23 (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 $P = \{ t_1 \approx^7 s_1, \ldots, t_n \approx^7 s_n \}$, the size of $P$ is defined as:

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

Remark 24 ($s \in AS(t) \implies size(s) < size(t)$). 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 | $|V_{N,AC}(P)|$ | $|V_{\geq 1}(P)|$ | $|AS(P)|$ | $size(P)$ |
|----------------|----------------|-----------------|----------|---------|
| line 6, 14     | $<$            | $\leq$          | $\leq$   | $<$     |
| lines 9, 15, 18, 23 | $\leq$       | $\leq$          | $\leq$   | $<$     |
| case 1 - line 29 | $\leq$        | $\leq$          | $\leq$   | $<$     |
| case 2 - line 29 | $\leq$        | $\leq$          | $\leq$   | $<$     |
| 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 24), 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 25. 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: $P_1 = \{ X \approx^? Z_0, Y \approx^? f(Z_5, Z_0), a \approx^? Z_1, b \approx^? Z_5, Z \approx^? f(Z_1, Z_6, Z_0) \}$, 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 \( t \approx^7 s \) in the unification problem \( P \) is to make sure that when we call \texttt{instantiateStep} in a unification problem \( P_1 \) and obtain as output a unification problem \( P_2 \) we maintain \( |\text{AS}(P_2)| \leq |\text{AS}(P_1)| \). However, this does not necessarily happen, as Example 26 shows.

\textbf{Example 26 (A case where \texttt{instantiateStep} increases \(|\text{AS}|\)}. Let \( f \) and \( g \) be AC-function symbols and \( P_1 = \{ X \approx^7 f(Z_1, Z_2), g(X, W) \approx^3 g(a, c) \} \). Calling \texttt{instantiateStep} with input \( P_1 \) we obtain \( P_2 = \{ g(f(Z_1, Z_2), W) \approx^7 g(a, c) \} \). In this case we have \( \text{AS}(P_1) = \{ a, c \} \) while \( \text{AS}(P_2) = \{ f(Z_1, Z_2), a, c \} \) and therefore \( |\text{AS}(P_2)| > |\text{AS}(P_1)| \).

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 26 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 27.

\textbf{Example 27 (A case where \texttt{instantiateStep} does not increase \(|\text{AS}|\)}. If we change slightly the problem from Example 26 to \( P_1' = \{ X \approx^7 f(Z_1, Z_2), f(X, W) \approx^3 g(a, c) \} \) and apply \texttt{instantiateStep} we would obtain: \( P_2' = \{ f(Z_1, Z_2, W) \approx^7 g(a, c) \} \), and we would have \( \text{AS}(P_1') = \text{AS}(P_2') = \{ a, c \} \).

Now, let’s go back to our original example of \( P = \{ t \approx^7 s \} \) and \( P_1 = \{ t_1 \approx^7 t'_1, \ldots, t_m \approx^7 t'_m, s_1 \approx^7 s'_1, \ldots, s_n \approx^7 s'_n \} \), and denote by \( P_2 \) the unification problem obtained by calling \texttt{instantiateStep} passing as input \( P_1 \). We will show that in the cases where \( |\text{AS}(P_2)| \) may be greater than \( |\text{AS}(P)| \) 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 \texttt{instantiateStep}, it would be instantiated to an AC-function headed by \( f \) (see Section 3.2.2) and therefore would only contribute in increasing \( |\text{AS}(P_2)| \) in relation with \( |\text{AS}(P_1)| \) 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 \texttt{instantiateStep} and therefore it is not in \( V_{>1}(P_2) \). The new variables introduced by \texttt{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 \( |\text{AS}(P)| \), 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 \( P = \{ t_1 \approx^7 s_1, \ldots, t_n \approx^7 s_n \} \). The only difference is that it is not enough to call function \texttt{solveAC} and then function \texttt{instantiateStep} in only the first equation \( t_1 \approx^7 s_1 \): we need to call function \texttt{applyACStep} and simplify every equation \( t_i \approx^7 s_i \).

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 \texttt{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 \texttt{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 \( P = \text{cons}(t, s, \text{nil}), \sigma = \text{NIL} \) and \( V = \text{Vars}(t, s) \). However, since the parameters of ACUnif may change in between the recursive calls, we cannot prove soundness (Corollary 30) directly by induction. We must prove the more general Theorem 29, with generic parameters for the unification problem \( P \), the substitution \( \sigma \) 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 \( P, \sigma \) and \( V \), some nice relations between them are preserved. These relations between the three components of the input are captured by Definition 28.

Definition 28 (Nice input). Given an input \((P, \sigma, V)\), we say that this input is nice if:
- \( \sigma \) is idempotent
- \( \text{Vars}(P) \cap \text{dom}(\sigma) = \emptyset \)
- \( \text{Vars}(P) \subseteq V \)

Theorem 29 (Soundness for nice inputs). Let \((P, \sigma, V)\) be a nice input, and \( \delta \in \text{ACUnif}(P, \sigma, V) \). Then, \( \delta \) unifies \( t \approx^? s \).

In the proof of completeness, the hypothesis \( \delta \subseteq V \) is simply a technicality that was put in order to guarantee that the new variables introduced by the algorithm do not clash with the variables in \( \text{dom}(\delta) \) or in the terms in \( \text{im}(\delta) \) 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 \( \delta = \{X \mapsto f(Z_2, a, b), Z \mapsto f(a, Y, Z_2, a, Z_2, a, Z_4 \rightarrow c)\} \) there is some overlap between the variables in \( \text{dom}(\delta) \) and in the terms in \( \text{im}(\delta) \) and the ones introduced by the algorithm, but the substitution \( \sigma_4 = \{X \mapsto f(Z_0, b), Z \mapsto f(a, Y, Z_0, Z_0)\} \) that we computed does not compute more general than \( \delta \) (restricted to the variables in \( V \)). Indeed, if we take \( \delta_1 = \{Z_6 \mapsto f(Z_2, a)\} \) then \( \delta W = \delta_1 \sigma_4 W \) for all variables \( W \in V \).

Corollary 30 (Soundness of ACUnif). If \( \delta \in \text{ACUnif}(\text{cons}(t, s, \text{nil}), \text{NIL}, \text{Nil}, \text{Vars}(t, s)) \) then \( \delta \) unifies \( t \approx^? s \).

Corollary 31 (Completeness for nice inputs). Let \((P, \sigma, V)\) be a nice input, \( \delta \) unifies \( P \), \( \sigma \subseteq \delta \), and \( \delta \subseteq V \). Then, there is a substitution \( \gamma \in \text{ACUnif}(P, \sigma, V) \) such that \( \gamma \leq_V \delta \).

Corollary 32 (Completeness of ACUnif). Let \( V \) be a set of variables such that \( \delta \subseteq V \) and \( \text{Vars}(t, s) \subseteq V \). If \( \delta \) unifies \( t \approx^? s \), then ACUnif computes a substitution more general than \( \delta \), i.e., there is a substitution \( \gamma \in \text{ACUnif}(\text{cons}(t, s, \text{nil}), \text{nil}, V) \) such that \( \gamma \leq_V \delta \).

Remark 33 (High-level description of how to remove hypothesis \( \delta \subseteq V \)). The key step to prove a variant of Corollary 32 with \( V = \text{Vars}(t, s) \) and without the hypothesis \( \delta \subseteq V \) is to prove that the substitutions computed when we call ACUnif with input \((P, \sigma, V)\) “differ only by a renaming” from the substitutions computed when we call ACUnif with input \((P, \sigma, V')\), where \( \delta \subseteq V' \). This cannot be proven by induction directly because if \( V \) and \( V' \) differ and ACUnif enters the AC-part, the new variables introduced for each input may “differ only by a renaming”, i.e. the first component of the two inputs, will also “differ only by a renaming”. Once ACUnif instantiates variables, it may happen that the substitutions
computed so far, i.e. the second component of the two inputs, will also “differ only by a renaming”. The solution is to prove by induction the more general statement that if the inputs \((P, \sigma, V)\) and \((P', \sigma', V')\) “differ only by a renaming” then the substitutions computed when we call \text{ACUnif} with \((P, \sigma, V)\) “differ only by a renaming” from the substitutions computed when we call \text{ACUnif} with \((P', \sigma', V')\).

4.5 More Information About the PVS Formalisation

The functions coded in PVS and the statement of the theorems can be found in files \(\text{.pvs}\), while the proofs of the theorems can be found in the \(\text{.prf}\) files. The PVS theory \text{unification_alg} contains function \text{ACUnif} and the theorems of soundness and completeness; \text{termination_alg} has the definitions and lemmas needed to prove termination; \text{apply_ac_step} contains function \text{APPLYACStep} and lemmas about its properties; \text{aux_unification} contains auxiliary functions such as \text{SOLVEAC} and \text{INSTANTIATESTEP} and lemmas about their properties. The PVS theories \text{diophantine, unification, substitution, equality} and \text{terms} contain, respectively, definitions and properties about solving linear Diophantine equations, unification, substitutions, equality modulo AC and terms. Finally \text{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.

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

5 Conclusions and Future Work

We have specified Stickel’s algorithm [21, 22] 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 [4] 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 Section 3.2.1. 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


Appendix A

Pseudocode for instantiateStep and applyACStep

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 $\sigma$, the substitution computed so far. Therefore, the first call to this function in order to instantiate the unification problem $P$ is with $P_1 = P$, $P_2 = \text{nil}$ and $\sigma = \text{nil}$. 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 $\text{True}$) or not (in this case the Boolean is $\text{False}$). 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).
Algorithm 2 Algorithm that instantiates when possible.

1: procedure instantiateStep($P_1, P_2, \sigma$)
2: if $\text{nil?}(P_1)$ then return $(P_2, \sigma, False)$
3: else
4: let $(t, s) = \text{car}(P_1), P'_1 = \text{cdr}(P_1)$ in
5: if $(s \text{ matches } X) \land (X \text{ not in } t)$ then
6: $\sigma_1 = \{X \mapsto t\}$
7: return instantiateStep($\sigma_1 P'_1, \sigma_1 P_2, \text{append}(\sigma_1, \sigma)$)
8: else if $(t \text{ matches } X) \land (X \text{ not in } s)$ then
9: $\sigma_1 = \{X \mapsto s\}$
10: return instantiateStep($\sigma_1 P'_1, \sigma_1 P_2, \text{append}(\sigma_1, \sigma)$)
11: else if $(t \text{ matches } X) \land (X \text{ matches } s)$ then
12: return instantiateStep($P'_1, P_2, \sigma$)
13: else if ($t \text{ matches } X) \land (X \text{ in } s)$ or ($s \text{ matches } X) \land (X \text{ in } t)$ then
14: return $(\text{nil, } \sigma, True)$ \(\triangleright\) the terms $t$ and $s$ are impossible to unify
15: else
16: return instantiateStep($P'_1, \text{cons}(t, s), P_2, \sigma$) \(\triangleright\) we skip the equation

- 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

\(\triangleright\) Remark 34. In function \text{applyACStep}, we eliminate equations $u \approx \gamma v$ from our unification problem if $u \approx v$ (line 4). This was done because if we called function \text{solveAC} in line 10 of Algorithm 3 passing as parameter two equal terms (modulo AC), the value returned would be $\text{PLst} = \text{nil}$. \text{applyACStep} would interpret that as meaning that the unification pair had no solution (when actually every substitution $\sigma$ is a solution to $\{u \approx \gamma v\}$) and also return NIL. To prevent this corner case, we eliminate those trivial equations from our unification problem before calling \text{solveAC}. In our code, the function \text{EQUAL?} tests equality (modulo AC) between terms $t$ and $s$, returning True if the terms are equal and False otherwise.

The first thing \text{applyACStep} does is check if $P_1$ is the null list. If it is (line 2), we have finished applying functions \text{solveAC} and \text{instantiateStep} and we return a list with only one element: $(P_2, \sigma, V)$.

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 \text{applyACStep} with $(\text{cdr}(P_1), P_2, \sigma, V)$.

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

We check if $\text{ACInstLst}$ 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 $\text{ACInstLst}$ is not null (lines 12-16), there will be branches to explore. Given
Algorithm 3 Algorithm for APPLYACSTEP.

1: procedure APPLYACSTEP(P₁, P₂, σ, V)
2:   if nil?(P₁) then return cons((P₂, σ, V), nil)
3:   else let (t, s) = car(P₁) in
4:     if t ≈ s then return APPLYACSTEP(cdr(P₁), P₂, σ, V)
5:     else
6:      ▷ assuming t and s are headed by the same function symbol f
7:       PLst = solveAC(t, σ, V)
8:      ▷ Call instantiateStep in every P in PLst obtaining a list ACInstLst,
9:      ▷ where each entry in this list is a pair (P', δ).
10:      if nil?(ACInstLst) then return nil
11:     else
12:      ▷ make an input list InputLst of all the branches we need to explore.
13:      ▷ For each (P', δ) in ACInstLst, the quadruple in InputLst will be
14:      ▷ (δcdr(P₁), append(P', δP₂), append(δ, σ), V') to APPLYACSTEP
15:      ▷ recursively explore all the branches
16:       return Flatten(map(APPLYACSTEP, InputLst))

an entry (P', δ) of ACInstLst, the part of the unification problem to which we must call
functions solveAC and instantiateStep is now δcdr(P₁) and the part of the unification
problem we have already explored is append(P', δP₂). 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 (δcdr(P₁), append(P', δP₂), append(δ, σ), 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

![Dependency Diagram for PVS Theories](image_url)
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 35.** The theory terms has its definitions and lemmas in the file terms.pvs and 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 36 (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:

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

we see that the procedure used to solve the linear Diophantine equations plays no role in this proof. In contrast to that, $V_{\text{NAC}}(P)$, $V_{>1}(P)$, $\text{AS}(P)$, $\text{size}(P)$ depend respectively on $V_{\text{NAC}}(t)$, Subterms$(t)$ and $\text{size}(t)$ 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.