Formalising AC-Unification

Gabriel Silva
XIV Summer Workshop on Mathematics (Universidade de Brasília)
Funded by a CAPES PhD scholarship
Advisor: Mauricio Ayala-Rincón, Co-Advisor: Maribel Fernández
https://gabriel951.github.io/

January 19, 2022

Joint work

This work was done in collaboration with:

Figure 1: Mauricio Ayala-Rincón

Figure 2: Maribel Fernández

Figure 3: Daniele Nantes

Overview

Unification

Unification is about "finding a way" to make two terms equal:

- $f(a, X)$ and $f(Y, b)$ can be made equal by "sending" X to b and Y to a, as they both become $f(a, b)$.

Unification has a lot of applications: logic programming, theorem proving, type inference and so on.

Unification Modulo AC

We consider the problem of AC-unification, i.e., unification in the presence of associative-commutative function symbols.

For instance, if f is an AC function symbol, then:

$$f(a, f(b, c)) \approx f(c, f(a, b)).$$

Related Work

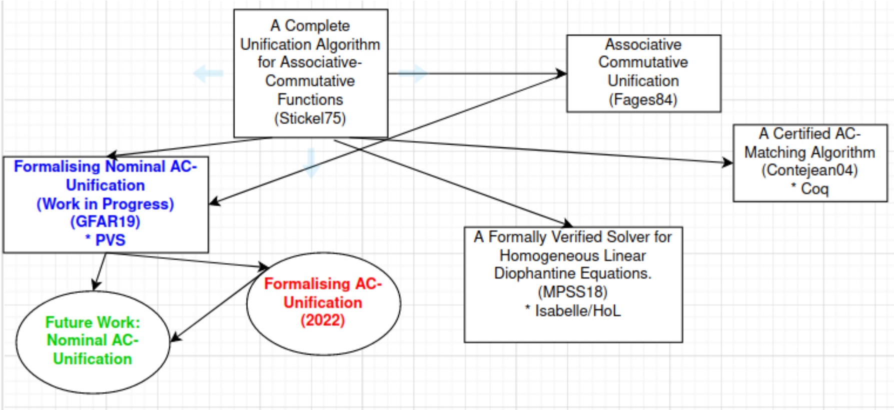

Figure 4: Main Related Work.

What We Have

An AC-unification algorithm, which we have specified in PVS and formalised it to be terminating, sound and complete.

The algorithm is recursive, calling itself on progressively simpler versions of the problem until it finishes.

In This Talk

- Briefly discuss the challenge in AC-unification.
- Present our approach to AC-unification (based on [1]).
- Comment interesting points in formalising termination and completeness.
- Discuss possible future work.

What is Tricky About AC? An Example

Let f be an AC function symbol.

The solutions that come to mind when unifying:

$$f(X, Y) \approx_? f(a, Z)$$

are: $\{X \to a, Y \to Z\}$ and $\{X \to Z, Y \to a\}$.

Are there other solutions?

Yes!

For instance, $\{X \to f(a, Z_1),\ Y \to Z_2,\ Z \to f(Z_1, Z_2)\}$ and $\{X \to Z_1,\ Y \to f(a, Z_2),\ Z \to f(Z_1, Z_2)\}$.

The AC-Step for AC-Unification

We explain via an example the `AC-Step` for AC-unification.

How do we generate a complete set of unifiers for:

$$f(X, X, Y, a, b, c) \approx_? f(b, b, b, c, Z).$$

Eliminate Common Arguments

1. Eliminate common arguments in the terms we are trying to unify.

Now we must unify $f(X, X, Y, a)$ with $f(b, b, Z)$.

Introducing a Linear Equation on $\mathbb{N}$

2. According to the number of times each argument appear in the terms, transform the unification problem into a linear equation on $\mathbb{N}$.

After this step, our equation is:

$$2X_1 + X_2 + X_3 = 2Y_1 + Y_2,$$

where variable X_1 corresponds to argument X, variable X_2 corresponds to argument Y and so on.

3. Generate a basis of solutions to the linear equation.

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	$2X_1 + X_2 + X_3$	$2Y_1 + Y_2$
0	0	1	0	1	1	1
0	1	0	0	1	1	1
0	0	2	1	0	2	2
0	1	1	1	0	2	2
0	2	0	1	0	2	2
1	0	0	0	2	2	2
1	0	0	1	0	2	2

4. Associate new variables with each solution.

Table 2: Solutions for the Equation $2X_1 + X_2 + X_3 = 2Y_1 + Y_2$

X_1	X_2	X_3	Y_1	Y_2	$2X_1 + X_2 + X_3$	$2Y_1 + Y_2$	**New Variables**
0	0	1	0	1	1	1	Z_1
0	1	0	0	1	1	1	Z_2
0	0	2	1	0	2	2	Z_3
0	1	1	1	0	2	2	Z_4
0	2	0	1	0	2	2	Z_5
1	0	0	0	2	2	2	Z_6
1	0	0	1	0	2	2	Z_7

5. Observing Table 2, relate the "old" variables and the "new" ones.

After this step, we obtain:

$$
\begin{aligned}
X_1 &\approx_? Z_6 + Z_7\\
X_2 &\approx_? Z_2 + Z_4 + 2Z_5\\
X_3 &\approx_? Z_1 + 2Z_3 + Z_4\\
Y_1 &\approx_? Z_3 + Z_4 + Z_5 + Z_7\\
Y_2 &\approx_? Z_1 + Z_2 + 2Z_6
\end{aligned}
$$

All the Possible Cases

6. Decide whether we will include (set to 1) or not (set to 0) every "new" variable. Observe that every "old" variable must be different than zero.

In our example, we have $2^7 = 128$ possibilities of including/excluding the variables $Z_1, \ldots, Z_7$, but after observing that X_1, X_2, X_3, Y_1, Y_2 cannot be set to zero, we have 69 cases.

7. Drop the cases where the variables that in fact represent constants or subterms headed by a different AC function symbol are assigned to more than one of the "new" variables.

For instance, the potential new unification problem

$$\{X_1 \approx_? Z_6, X_2 \approx_? Z_4, X_3 \approx_? f(Z_1, Z_4), \\ Y_1 \approx_? Z_4, Y_2 \approx_? f(Z_1, Z_6, Z_6)\}$$

should be discarded as the variable X_3, which represents the constant a, cannot unify with $f(Z_1, Z_4)$.

Dropping More Cases and Proceeding

8. Replace "old" variables by the original terms they substituted and proceed with the unification.

Some new unification problems may be unsolvable and **will be discarded later**. For instance:

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

Solutions For The Example

In our example, the solutions will be:

$$\left\{ \begin{array}{c} \sigma_1 = \{Y \to f(b,b), Z \to f(a,X,X)\} \\ \sigma_2 = \{Y \to f(Z_2,b,b), Z \to f(a,Z_2,X,X)\} \\ \sigma_3 = \{X \to b, Z \to f(a,Y)\} \\ \sigma_4 = \{X \to f(Z_6,b), Z \to f(a,Y,Z_6,Z_6)\} \end{array} \right\}$$

Suppose that $P = \{t \approx^? s\}$, where t and s are AC-functions, headed by a symbol f. Let $t_1, \ldots, t_m$ be the different arguments of t and let $s_1, \ldots, s_n$ be the different arguments of s, after we eliminate common arguments.

An arbitrary unification problem P_1 after the AC-Step is of the form $P_1 = \{t_1 \approx^? t'_1, \ldots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \ldots, s_n \approx^? s'_n\}$, where the terms in the right-hand side are either new variables Z_is or AC-functions headed by f whose arguments are all new variables Z_is.

Related Work

Our formalisation is based on the works of Stickel ([1]) and Fages ([2]).

1. Stickel, in 1975, presents the first AC-unification algorithm
2. Fages, in 1984, discovered an error in Stickel's proof of termination and presented a fix for it.

Termination - 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 we decide to solve the first equation. We obtain as one of the branches the unification problem

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

We then instantiate the variables that we can, obtaining:

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

If we then solve the first equation, one of the branches get us:

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

which is essentially the same unification problem we started with.

Avoiding Infinite Loops

How did we avoid looping forever?

Instantiate as early as possible, leave the AC-part last.

When specifying the algorithm, we tried to follow closely the pseudocode of Fages. However, in Fages work, there are two functions:

1. `uniAC` - used to unify terms t and s
2. `unicompound` - used to unify a list of terms $(t_1, \ldots, t_n)$ with $(s_1, \ldots, s_n)$

which are mutually recursive, something not allowed in PVS.

Our Approach

We adapted the algorithm to use only one function, which works in a unification problem P and operates (with the exception of the AC-part of the algorithm) by simplifying one of the equations $\{t \approx^? s\}$ of P.

Why is this a big deal?

The lexicographic measure we used to prove termination would not diminish if in the AC-part of the algorithm we simplified only one equation $\{t \approx^? s\}$ of P. (More about termination on the Appendix).

The Algorithm in a Nutshell

Choose an equation $t \approx^? s \in P$ that we will simplify. Heuristic: leave AC-equations last.

If $t \approx^? s$ is not an AC-equation, proceed as in syntactic unification.

If all that remains are AC-equations, pick the first AC-equation, apply `AC-Step` and instantiate the variables. Go to the second AC-equation, apply `AC-Step` and instantiate the variables. Proceed in this way until the last one.

Grammar of Terms

We aim at extending our formalisation to obtain a nominal AC-unification algorithm. Therefore, the grammar of terms we used is:

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

Pairs Encode Multiple Arguments

Pairs can be used to represent tuples with an arbitrary number of terms. Therefore, the term $f(t_1, t_2, t_3)$ can be represented in our grammar as $f\langle t_1, \langle t_2, t_3\rangle\rangle$.

A Problem with Pairs

Example

Let $P = \{f(X, X, Y, a) \approx^? f(b, b, Z)\}$. One of the unification problems obtained after `AC-Step` is:

$$P_1 = \{X \approx^? f(Z_6, Z_7), Y \approx^? Z_2, a \approx^? Z_1, b \approx^? Z_7, Z \approx^? f(Z_1, Z_2, 2Z_6)\}.$$

However, consider the substitutions σ and $\sigma_{awkward}$, defined as follow:

$$\sigma = \{X \mapsto f(Z_6, b), Z_2 \mapsto Y, Z_1 \mapsto a, Z_7 \mapsto b, Z \mapsto f(a, Y, Z_6, Z_6)\}$$

$$\sigma_{awkward} = \{X \mapsto \langle Z_6, b\rangle, Z_2 \mapsto Y, Z_1 \mapsto a, Z_7 \mapsto b, Z \mapsto f(a, Y, Z_6, Z_6)\}$$

Both σ and $\sigma_{awkward}$ unify P, but only σ unifies P_1.

Well-formed Terms

This motivated us to define well-formed terms and consider that a pair in itself is not a well-formed term:

1. $\langle a, b \rangle$ is not a well-formed term.
2. $f^{AC}\langle a, b \rangle$ is a well-formed term.

Well-formed Substitutions

We say that a substitution δ is well-formed if δX is a well-formed term, for every X.

In the previous Example, the substitution σ is well-formed, but the substitution $\sigma_{awkward}$ is not.

Restriction to Well-Formed Terms

In our algorithm and in the theorem of completeness, we only consider well-formed terms and substitutions.

Is this a meaningful restriction?

Grammar of Terms

No. In first-order AC-unification the following grammar for terms is also used:

$$t ::= a \mid X \mid f(t_1, \ldots, t_n)$$

Grammar of Nominal Terms

Will this restriction be meaningful when extending the algorithm to the nominal setting?

No. Pairs are used to encode a list of arguments and there are papers where pairs do not even appear in the grammar of nominal terms.

Grammar of Nominal Terms - Some Papers Do Not Use Pairs

In "Nominal Narrowing" (Ayala et. al) there are no pairs in the grammar of nominal terms.

In "α-Prolog: A Logic Programming Language With Names, Binding and α-Equivalence" (Cheney and Urban) neither.

Arguments of the Algorithm

The algorithm `ACUnif` is recursive 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.

Initial Call

The first call to the algorithm, in order to unify terms t and s is done with $P = cons((t, s), nil)$, $\sigma = nil$ and $Vars((t, s)) \subset V$.

Theorem (Soundness)

If $\delta \in \mathtt{ACUnif}(cons((t, s), nil), nil, Vars((t, s)))$ *then* δ *is a unifier to* P.

Theorem (Completeness)

If δ *unifies* $t \approx^? s$ *and* $\delta \subset V$ *and* $Vars((t, s)) \subset V$ *then there is a substitution* $\sigma \in \mathtt{ACUnif}(cons((t, s), nil), nil, V)$ *such that* $\sigma \leq_V \delta$.

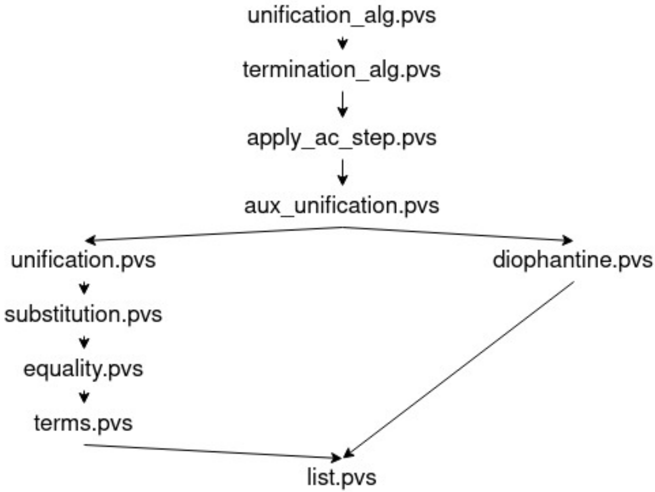

Figure 5: PVS Files Dependency Diagram

Amount of Theorems and TCCs Proved

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

File	Theorems	TCCs	Total
unification_alg.pvs	9	23	32
termination_alg.pvs	80	49	129
apply_ac_step.pvs	23	25	48
aux_unification.pvs	179	95	274
diophantine.pvs	73	63	136
unification.pvs	80	53	133
substitution.pvs	108	32	140
equality.pvs	67	53	120
terms.pvs	129	105	234
list.pvs	251	109	360
Total	999	607	**1606**

Size By File

Table 4: Size of .pvs and .prf files

File	.pvs	.prf	Percentage
unification_alg	5kB	1.4MB	4 %
termination_alg	21.6kB	11MB	30 %
apply_ac_step	13kB	9MB	25 %
aux_unification	52.2kB	7.2MB	20 %
diophantine	22.8kB	1.1MB	3 %
unification	18.8kB	867kB	2 %
substitution	18.9kB	1.7MB	5 %
equality	12.2kB	1.1MB	3 %
terms	26.3kB	959kB	2 %
list	52kB	1.8MB	5 %
Total	242.8kB	36.1MB	**100%**

Conclusion

- We specified Stickel's AC-unification algorithm in the proof assistant PVS and proved it terminating, sound and complete.
- We discussed how to solve equations of the form $t \approx^? s$ when t and s are AC-functions headed by the same symbol and the connection between this problem and solving Diophantine linear equations.
- We pointed some interesting issues that arose while proving termination and completeness.

Future Work

We envision three different paths for future work:

1. Coming back to our initial goal: adapting the algorithm to the nominal setting, which would give the first nominal AC-unification algorithm.
2. Use the formalisation as a basis to formalise more efficient first-order AC-unification algorithms (for instance the one in [3]).
3. Use the formalisation to extract verified code and test AC-unification implementations (for instance in Maude, see [4]) for correctness/completeness.

Thank You

Thank you! Any comments/suggestions/doubts?

Bibliography

[1] M. E. Stickel, "A unification algorithm for associative-commutative functions," *Journal of the ACM (JACM)*, vol. 28, no. 3, pp. 423–434, 1981.

[2] F. Fages, "Associative-commutative unification," *Journal of Symbolic Computation*, vol. 3, no. 3, pp. 257–275, 1987.

[3] M. Adi and C. Kirchner, "Ac-unification race: The system solving approach, implementation and benchmarks," *Journal of Symbolic Computation*, vol. 14, no. 1, pp. 51–70, 1992.

[4] M. Clavel, F. Durán, S. Eker, *et al.*, "Maude: Specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.

Pseudocode for the Algorithm

```
procedure ACUnif(P, σ, V)
    if nil?(P) then
        return cons(σ, nil)
    else
        ((t, s), P₁) = choose(P)
        if (s matches X) and (X not in t) then
            σ₁ = {X → t}
            σ' = append(σ₁, σ)
            P' = σ₁P₁
            return ACUnif(P', σ', V)
        else
            if t matches a then
                if s matches a then
                    return ACUnif(P₁, σ, V)
                else
                    return nil
                end if
```

Pseudocode for the Algorithm (cont.)

else if t matches X **then**
if X not in s **then**
$\sigma_1 = \{X \to s\}$
$\sigma' = \mathtt{append}(\sigma_1, \sigma)$
$P' = \sigma_1 P_1$
return $\mathtt{ACUnif}(P', \sigma', V)$
else if s matches X **then**
return $\mathtt{ACUnif}(P_1, \sigma, V)$
else
return `nil`
end if
else if t matches $\langle\rangle$ **then**
if s matches $\langle\rangle$ **then**
return $\mathtt{ACUnif}(P_1, \sigma, V)$
else
return `nil`
end if

Pseudocode for the Algorithm (cont.)

```
        else if t matches f t_1 then
            if s matches f s_1 then
                P' = cons((t_1, s_1), P_1)
                return ACUnif(P', σ, V)
            else
                return nil
            end if
        else
            if s matches f^AC s_1 then
                InputLst = applyACStep(P, σ, V)
                LstResults = map(ACUnif, InputLst)
                return flatten (LstResults)
            else
                return nil
            end if
        end if
    end if
end if
```

Pseudocode for the Algorithm (cont.)

end procedure

Termination - Notation

To explain the ideas to prove termination, we will consider the restricted case where $P = \{t \approx^? s\}$, and $t \equiv f(t_1, \ldots, t_m)$ and $s \equiv f(s_1, \ldots, s_n)$.

After we apply `AC-Step`, we will denote an arbitrary unification problem obtained as $P_1 = \{t_1 \approx^? t'_1, \ldots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \ldots, s_n \approx^? s'_n\}$. We will denote by P_2 the unification problem obtained from P_1 after you do the necessary instantiations.

What we need for termination

All the terms in the right-hand side of P_1 are new terms. After introducing all these new terms and possible making some instantiations, can we still find a lexicographic measure lex such that $lex(P_2) < lex(P)$?

Termination - First Idea

Idea: Define a set of admissible subterms (AS) of a term in a way that every term t_i' in the right-hand side of P_1 has $AS(t_i') = \emptyset$.

Admissible Subterms - Definition

We say that s is an admissible subterms of t if s is a proper subterm of t and s is not a variable.

The set of admissible subterms of a unification problem P is defined as:

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

Admissible Subterms - Example

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

We may have $|AS(P_1)| < |AS(P)|$

If at least one of the terms in the left-hand side of P_1 is not a variable, then $|AS(P_1)| < |AS(P)|$.

Example

In the previous example, the unification problem before the `AC-Step` was:

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

and we had $AS(P) = \{a, b\}$. After the `AC-Step`, 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 $AS(P_1) = \emptyset$.

When All The Arguments Are Variables

But what happens if all the arguments of t and s are variables?

Then, after the `AC-Step` we would instantiate all of them and the problem would be solved.

Can Instantiation Mess Up with $|AS|$?

All that is left in this simplified example where $P = \{t \approx^? s\}$ is to make sure that when we instantiate the variables in the unification problem P_1 and obtain as output a unification problem P_2 we maintain $|AS(P_2)| \leq |AS(P_1)|$.

Can we prove this?

Instantiation May Mess up with $|AS|$

Unfortunately not.

Example

Let f and g be AC-function symbols. If we instantiate the variables in

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

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 $|AS(P_2)| > |AS(P_1)|$.

If we changed the previous example 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 $|AS|$:

Example

If

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

and we instantiate the variables we would obtain:

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

where $AS(P_1') = AS(P_2') = \{a, c\}$.

Suppose that X is a variable in the left-hand side of P_1 and is instantiated to an AC-function headed by f. X would only contribute in increasing $|AS(P_2)|$ in relation to $|AS(P_1)|$ if X also occurred as an argument of a function term t^* headed by a different symbol than f.

Also, if X is in the left-hand side of P_1, then it is an argument of either t or s, both of which are functions headed by f.

Idea: X only contributes in increasing $|AS(P_2)|$ in relation to $|AS(P_1)|$ if X were "an argument to two different function symbols" in P. Since X was instantiated it is not "an argument to two different function symbols" in P_2.

$V_{>1}(P)$ - Definition

To capture the idea of a variable being "an argument to two different function symbols" in P we define $V_{>1}(P)$.

Definition

We denote by $V_{>1}(P)$ the set of variables that are arguments of (at least) two terms t and s in $Subterms(P)$ such that t and s are headed by different function symbols.

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

$V_{>1}(P)$ in a Nutshell

In the cases where $|AS(P_2)|$ may be greater than $|AS(P_1)|$, we necessarily have $|V_{>1}(P_2)| < |V_{>1}(P)|$.

Lexicographic Measure for the non-associative part

In syntactic unification, given a unification problem P a usual lexicographic measure for termination $(|Vars(P)|, size(P))$.

We needed to change $Vars(P)$ to $V_{NAC}(P)$, the variables that occur in the problem P excluding those that only occur as arguments of AC-function symbols.

$V_{NAC}(P)$ - Example

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

Motivation for changing $Vars(P)$ to $V_{NAC}(P)$

The AC-Step introduces new variables. By replacing $Vars(P)$ with $V_{NAC}(P)$, we exclude the new variables that only occur as arguments of AC-function symbols.

But in a problem like:

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

the new variable Z_1 does not occur only as an argument of AC-function symbols. Can variables like Z_1 potentially cause $|V_{NAC}(P_2)| > |V_{NAC}(P)|$?

Motivation for changing $Vars(P)$ to $V_{NAC}(P)$ (cont.)

No. Variables like Z_1 would be instantiated and we will always have $|V_{NAC}(P_2)| \leq |V_{NAC}(P)|$.

The lexicographic measure for termination is:

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

We always have $lex(P_2) < lex(P)$.