Proving Termination of AC-rewriting without Extended Rules

Cristina Borralleras[1] and Albert Rubio[2]

[1] Universitat de Vic, Spain
Email: cristina.borralleras@uvic.es
[2] Universitat Politècnica de Catalunya, Barcelona, SPAIN
Email: rubio@lsi.upc.es

Abstract. Polynomial interpretations and RPO-like orderings allow one to prove termination of Associative and Commutative (AC-)rewriting by only checking the rules of the given rewrite system without considering the so-called extended rules. However, these methods have important limitations as termination proving tools.

To overcome these limitations, more powerful methods like the dependency pairs method have been extended to the AC-case. Unfortunately, in order to ensure AC-termination, the extended rules, which, in general, are hard to prove, must be added to the rewrite system.

In this paper we present a new automatable ordering-based termination proving method for AC-rewriting which does not need to consider extended rules. Due to this, we can easily prove several non-trivial examples appearing in the literature that, to our knowledge, can be handled by no other automatic method.

1 Introduction

In programming, as well as in theorem proving, it is very common to have binary operators which satisfy the associative and commutative properties. However, such axioms cannot be treated as additional rules in the rewrite system (e.g., commutativity cannot be oriented by any well-founded ordering), and hence they require a special treatment. The most common approach to deal with AC symbols is rewriting *modulo* AC, that is, rewriting using matching modulo associativity and commutativity to detect the applicability of the rules. The following example (taken from [MU98]) describes addition ($+$) and multiplication ($*$) for natural numbers in binary notation defined by a constant $\#$, denoting the empty sequence of digits, and two unary postfixed functions $(\_)0$ and $(\_)1$, to add 0's and 1's to the right. For instance, 5 is written as $(((\#)1)0)1$.

Example 1. In this example both $+$ and $*$ are AC-symbols.

$$
\begin{aligned}
(\#)0 &\to \# \\
x + \# &\to x \\
x * \# &\to \#
\end{aligned}
$$

$$
\begin{aligned}
(x)0 + (y)0 &\to (x+y)0 \\
(x)0 + (y)1 &\to (x+y)1 \\
(x)1 + (y)1 &\to (x+y+(\#)1)0
\end{aligned}
$$

$$
\begin{aligned}
x * (y)0 &\to (x*y)0 \\
x * (y)1 &\to x + (x*y)0
\end{aligned}
$$

Due to the fact that rewriting is performed modulo the AC-axioms, proving termination becomes a more difficult task. In particular, the applied method must be AC-compatible, which roughly means that all terms in the same AC-equivalent class are treated in the same way. When using an ordering-based termination proof method, AC-compatibility means that if some term s is greater than some term t, then any term that is AC-equivalent to s is greater than any term that is AC-equivalent to t.

Many efforts have been made [BCL87,BP85,KSZ95,DP93,RN95,KS00,Rub02] in order to obtain AC-compatible simplification orderings (i.e., monotonic orderings including the subterm relation) by extending the methods used for standard rewriting.

These methods are, in general, well suited for automation. However, as for standard rewriting, such AC-compatible simplification orderings have important limitations as termination proving tools. For instance, there are many term rewrite systems (TRSs) that are terminating but are not contained in any simplification ordering, i.e., they are not *simply terminating*.

In [MU98,KT01,Urb01] the dependency pairs method [AG00], which can prove termination of TRSs that are not simply terminating, is adapted to deal with AC-rewriting, and in [GK01] it is adapted for rewriting modulo more general equational theories. All these AC-versions of the dependency pairs method need to consider the so-called *extended rules*. For the AC-case this means to add to the set of rules one extended rule $f(l, x) \to f(r, x)$, where x is a new variable, for every rule $l \to r$ with l headed by an AC-symbol f. In the previous example we have to add

$$
\begin{aligned}
x + \# + z &\to x + z \\
x * \# * z &\to \# * z
\end{aligned}
$$

$$
\begin{aligned}
(x)0 + (y)0 + z &\to (x+y)0 + z \\
(x)0 + (y)1 + z &\to (x+y)1 + z \\
(x)1 + (y)1 + z &\to (x+y+(\#)1)0 + z
\end{aligned}
$$

$$
\begin{aligned}
x * (y)0 * z &\to (x*y)0 * z \\
x * (y)1 * z &\to x + (x*y)0 * z
\end{aligned}
$$

These extended rules are needed to ensure monotonicity with respect to the AC-symbols. But adding them to the system usually makes the termination proof harder.

An alternative to the dependency pair method is the use of the monotonic semantic path ordering (MSPO; [BFR00]), which is a monotonic version of the *semantic path ordering* (SPO; [KL80]). The SPO generalizes path orderings like Dershowit's *recursive path ordering* (RPO; [Der82]) by replacing the use of a precedence by the use of any measure, defined by an *underlaying quasi-ordering*, involving the whole term and not only the head symbol.

The aim of this paper is to adapt the MSPO to deal with AC symbols, obtaining a fully monotonic AC-ordering which allows us to avoid the use of extended rules. Following the ideas used to adapt RPO to the AC-case in [Rub02], SPO has been adapted to obtain an associative and commutative SPO (ACSPO). After that, a monotonic version of ACSPO, called ACMSPO, is obtained. Even though the ACSPO and the ACRPO share the same structure, finding the appropriate formulation of the ordering was not an easy task. In order to ensure the properties of the ACSPO and the ACMSPO, some new conditions on the underlying quasi-ordering used inside SPO are needed. This conditions have to be tight, otherwise the ordering becomes too weak. Additionally, to ensure stability under substitutions it is necessary to *linearize* some variables of the terms, but, fortunately, we have shown that this is only required in a single case.

With ACMSPO we have been able to obtain a simple termination proof for the *ternary integral arithmetic* example used in [CMR97] for computing addition and multiplication of integers in balanced ternary notation, where there are three unary postfixed functions $(x)0$, $(x)1$ and $(x)j$ representing respectively $3x$, $3x+1$ and $3x-1$ (see also [Knu97], pages 207 and 208 for further details).

Example 2. Both $+$ and $*$ are AC-symbols.

$$
\begin{array}{rl}
(\#)0 &\to \# \\
x + \# &\to x \\
(x)0 + (y)0 &\to (x+y)0 \\
(x)0 + (y)1 &\to (x+y)1 \\
(x)0 + (y)j &\to (x+y)j \\
(x)1 + (y)j &\to (x+y)0 \\
(x)1 + (y)1 &\to (x+y+(\#)1)j \\
(x)j + (y)j &\to (x+y+(\#)j)1
\end{array}
\qquad
\begin{array}{rl}
x - y &\to x + opp(y) \\
opp(\#) &\to \# \\
opp((x)0) &\to (opp(x))0 \\
opp((x)1) &\to (opp(x))j \\
opp((x)j) &\to (opp(x))1 \\
x * \# &\to \# \\
x * (y)0 &\to (x*y)0 \\
x * (y)1 &\to x + (x*y)0 \\
x * (y)j &\to (x*y)0 + opp(x)
\end{array}
$$

Proving termination of this example is not trivial at all. In [CMR97] there is a quite complex ad-hoc hand-tailored proof of termination based on a lexicographic combination of three interpretations on the non-negative integers. In [MU98] an AC-version of the *dependency pairs method* [AG00] was used for proving termination of this example (together with its extended rules), but the use of some AC-marked symbols, which was necessary to deal with the example, turned out to be unsound.

As shown for MSPO [BR02], the ACMSPO method can be fully automated in a constraint solving based system (see Section 6 for details).

The paper is organized as follows. The next section is devoted to preliminaries on orderings and AC-rewriting. In Section 3, the AC semantic path ordering (ACSPO), on which we will build our method, is introduced. A monotonic version of the ACSPO is given in Section 4. In Section 5, ACMSPO is used to check the termination of several examples and in Section 6 it is shown how ACMSPO can be implemented as a fully automated termination proving tool, using a constraint framework similar to the one of the dependency pairs method. Finally, conclusions and further work are discussed in Section 7.

Due to the lack of space we have not included any proof of the properties of the orderings. They can all be found in [BR03].

2 AC-rewriting and termination

In the following we consider that $\mathcal{F}$ is a set of function symbols, $\mathcal{F}_{AC}$ the subset containing all the AC symbols, $\mathcal{X}$ a set of variables and $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of terms built from $\mathcal{F}$ and $\mathcal{X}$.

Rewriting modulo AC, or AC-rewriting, consists of rewriting over the equivalence classes defined by the associativity and commutativity axioms. Therefore, instead of syntactic equality, rewriting modulo AC considers $=_{AC}$, the congruence generated on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ by the associativity and commutativity axioms for the symbols in $\mathcal{F}_{AC}$.

Given a TRS $\mathcal{R}$, a term $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ rewrites to t with $\mathcal{R}$ modulo AC, denoted by $s \rightarrow_{\mathcal{R}/AC} t$ if $s =_{AC} s'$, $s'|_p = l\sigma$ for some rule $l \rightarrow r \in \mathcal{R}$, term s', position p and substitution σ, and $t =_{AC} s'[r\sigma]_p$.

Let s and t be arbitrary terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, let f be a function symbol in $\mathcal{F}$ and let σ be a substitution. A quasi-ordering $\succeq$ is a transitive reflexive relation. Its *strict part* $\succ$ is the strict ordering $\succeq \setminus \preceq$ (i.e, $s \succ t$ iff $s \succeq t$ and $s \not\preceq t$). Its *equivalence* $\sim$ is $\succeq \cap \preceq$. Note that $\succeq$ is the disjoint union of $\succ$ and $\sim$, and that if $=$ denotes syntactic equality then $\succ \cup =$ is a quasi-ordering whose strict part is $\succ$. A quasi-ordering $\succeq$ is *monotonic* if $s \succeq t$ implies $f(\ldots s \ldots) \succeq f(\ldots t \ldots)$, and *stable under substitutions* if $s \succeq t$ implies $s\sigma \succeq t\sigma$.

A (strict partial) ordering $\succ$ is a transitive irreflexive relation. It is *monotonic* if $s \succ t$ implies $f(\ldots s \ldots) \succ f(\ldots t \ldots)$, and *stable under substitutions* if $s \succ t$ implies $s\sigma \succ t\sigma$. Monotonic orderings that are stable under substitutions are called *rewrite orderings*. A *reduction ordering* is a rewrite ordering that is *well-founded*: there are no infinite sequences $t_1 \succ t_2 \succ \ldots$

As the rewrite relation is defined over AC-equivalence classes, any ordering $\succ$ used to prove termination of AC-rewriting has to be *AC-compatible*.

Definition 1. *A relation is said to be AC-compatible if it is compatible with the* $=_{AC}$ *relation. In particular* $\succ$ *is* AC-compatible *if* $s' =_{AC} s \succ t =_{AC} t'$ *implies* $s' \succ t'$*; and* $\succeq$ *is* AC-compatible *if* $s' =_{AC} s \succeq t =_{AC} t'$ *implies* $s' \succeq t'$.

Theorem 1. *Let $\succ$ be an AC-compatible reduction ordering and let $\mathcal{R}$ be a TRS. If $l \succ r$ for all rules $l \to r \in \mathcal{R}$ then $\mathcal{R}$ is terminating for rewriting modulo AC.*

The AC-multiset extension of an AC-compatible ordering is defined in the standard way but using AC-equality instead of syntactic equality.

A very common way to easily obtain AC-compatibility is to consider the terms flattened wrt. the AC-symbols. The flat form of t, denoted by $\overline{t}$, is the normal form of t wrt. the (infinite set of) rules

$f(x_1 \dots x_{i-1}, f(y_1 \dots y_k), x_{i+1} \dots x_n) \to f(x_1 \dots x_{i-1}, y_1 \dots y_k, x_{i+1} \dots x_n)$
for each $f \in \mathcal{F}_{AC}$. E.g., let $t = +(g(+(a,+(b,c))), +(d,+(e,f)))$ then $\overline{t} = +(g(+(a,b,c)), d, e, f)$. Note that due to the flattening we need to consider variable arities (greater than or equal to 2) for symbols in $\mathcal{F}_{AC}$.

It follows from an easy induction that $s =_{AC} t$ iff $\overline{s}$ and $\overline{t}$ are equal up to permutation of arguments for the AC-symbols. We also denote by $=_{AC}$ this equality up to permutation of arguments for the AC symbols, and by $==_{AC}$ the standard extension of $=_{AC}$ to multisets. From now on, all terms are assumed to be in their flattened form.

However, when considering flattened terms the orderings frequently loses monotonicity. For instance, with the *recursive path ordering* (RPO) if a symbol f is greater than a symbol g and f is an AC-symbol, we have that $f(a,a) \succ_{rpo} g(a)$ but $f(f(a,a),b) \not\succ_{rpo} f(g(a),b)$ as the flattened form of $f(f(a,a),b)$ is $f(a,a,b)$ and $f(g(a),b) \succ_{rpo} f(a,a,b)$. The usual way to recover monotonicity consists of giving a special treatment to the small symbols like g when they occur below a big AC-symbol like f.

Remark that if we have a term s in flattened form, when adding a context $f(\dots,s,\dots)$, the flattening rules need to be applied at top position of the arguments only. Thus, we define the top-flattening of a term s wrt. an AC-symbol f, denoted by $tf_f(s)$, as $tf_f(f(s_1,\dots,s_n)) = s_1,\dots,s_n$ and $tf_f(s) = s$ if $top(s) \neq f$. Then $f(\dots,tf_f(s),\dots)$ is the flattened form of $f(\dots,s,\dots)$ for any given flattened term s and flattened context $f(\dots[]\dots)$.

Finally, let us give some definitions, which only concerns the AC-symbols. A strict ordering $\succ$ is AC-monotonic if $s \succ t$ implies $f(\dots,tf_f(s),\dots) \succ f(\dots,tf_f(t),\dots)$ for any $f \in \mathcal{F}_{AC}$. A quasi-ordering $\succeq$ is AC-monotonic if $s \succeq t$ implies $f(\dots,tf_f(s),\dots) \succeq f(\dots,tf_f(t),\dots)$ for any $f \in \mathcal{F}_{AC}$ and it fulfils the AC-deletion property if for all $f \in \mathcal{F}_{AC}$, if $n > m$ and $1 \leq i_1 < \dots < i_m \leq n$ then $f(s_1,\dots,s_n) \succeq f(s_{i_1},\dots,s_{i_m})$.

2.1 ACRPO

We give a restricted version of the ACRPO in [Rub02], where the *precedence*, i.e., the well-founded quasi-ordering on the set of function symbols, is total and the arguments are always compared as multisets.

In the definition of ACRPO the main difference with RPO appears when comparing two terms headed by some AC-symbol f.

In this case, in order to ensure monotonicity, we have to give a different treatment to the arguments headed by symbols bigger than f and the arguments headed by symbols smaller that f. For this reason we need to define the following (multi)sets of terms.

Definition 2. *Let s be a term of the form $f(s_1, \ldots, s_n)$ with $f \in \mathcal{F}_{AC}$.*

- *The multiset of arguments of s headed by a big symbol, denoted by $BigHead(s)$, is defined as $\{s_i \mid 1 \leq i \leq n \wedge top(s_i) \succ_{\mathcal{F}} f\}$*
- *The multiset of arguments of s headed by a symbol not smaller than f, denoted by $NoSmallHead(s)$, is defined as $\{s_i \mid 1 \leq i \leq n \wedge f \not\succ_{\mathcal{F}} top(s_i)\}$*
- *The set of terms embedded in s through an argument headed by a small symbol, denoted by $EmbSmall(s)$, is defined as $\{f(s_1, \ldots, tf_f(v_j), \ldots, s_n) \mid s_i = h(v_1, \ldots, v_r) \wedge f \succ_{\mathcal{F}} h \wedge j \in \{1, \ldots, r\}\}$*

Note that the difference between $NoSmallHead(s)$ and $BigHead(s)$ is that the former includes the variables which are arguments of s and the latter does not. For AC-equivalent terms all these three (multi)sets coincide (modulo AC).

For instance, let $t = f(g(g(x,a),a), x, h(a), a)$ with $f \in \mathcal{F}_{AC}$, $h \succ_{\mathcal{F}} a \succ_{\mathcal{F}} f \succ_{\mathcal{F}} g$. Then $BigHead(t) = \{h(a), a\}$, $NoSmallHead(t) = \{x, h(a), a\}$ and $EmbSmall(t) = \{f(g(x,a), x, h(a), a), f(a, x, h(a), a)\}$.

Additionally, we need to count the number of arguments of a term t headed by an AC-symbol, but in order to preserve stability under substitutions we have to describe the counting by means of an expression with variables.

Definition 3. *Let s be a term of the form $f(s_1, \ldots, s_n)$ with $f \in \mathcal{F}_{AC}$. Then $\#(s)$ is an expression with variables on the positive integers, defined as $\#(f(s_1, \ldots, s_n)) = \#_v(s_1) + \ldots + \#_v(s_n)$, where $\#_v(x) = x$ and $\#_v(t) = 1$ if t is not a variable.*

For example, we have $\#(f(x, y, g(x))) = x+y+1$, which means the arguments coming from x (when applying a substitution and flattening), the ones coming from y plus 1. Then can compare the amount of arguments of two terms by checking $\#(f(x, y, g(x))) = x + y + 1 > x + y = \#(f(x, y))$, which is necessary to achieve stability under substitution. For AC-equivalent terms the counting coincides.

Definition 4. *Let $\succeq_{\mathcal{F}}$ be a well-founded total precedence and let s and t be terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Then $s = f(s_1 ... s_n) \succ_{ACRPO} t$ if and only if*

1. *$s_i \succeq_{ACRPO} t$ for some $i \in \{1...n\}$ or*
2. *$t = g(t_1...t_m)$, $f \succ_{\mathcal{F}} g$ and $s \succ_{ACRPO} t_i$ for all $i \in \{1...m\}$ or*
3. *$t = g(t_1...t_m)$, $f = g \notin \mathcal{F}_{AC}$ and $\{s_1, ..., s_n\} \succ\!\succ_{ACRPO} \{t_1, ..., t_n\}$ or*
4. *$t = g(t_1...t_m)$, $f = g \in \mathcal{F}_{AC}$ and $s' \succeq_{ACRPO} t$ for some $s' \in EmbSmall(s)$ or*
5. *$t = g(t_1...t_m)$, $f = g \in \mathcal{F}_{AC}$, $s \succ_{ACRPO} t'$ for all $t' \in EmbSmall(t)$, $NoSmallHead(s) \succeq\!\succeq_{ACRPO} NoSmallHead(t)$ and either*
   (a) $BigHead(s) \succ\!\succ_{ACRPO} BigHead(t)$ or
   (b) $\#(s) > \#(t)$ or
   (c) $\#(s) \geq \#(t)$ and $\{s_1, ..., s_n\} \succ\!\succ_{ACRPO} \{t_1, ..., t_m\}$

where $\succeq_{ACRPO}$ *is* $\succ_{ACRPO} \cup =_{AC}$ *and* $\succ\!\!\succ_{ACRPO}$ *is the AC-multiset extension of* $\succ_{ACRPO}$.

The condition $NoSmallHead(s) \;\underline{\succ\!\!\succ}\; NoSmallHead(t)$ ensures that every variable in t is taken care of by a variable in s or by an argument of s headed by a big symbol. Then, if by instantiation, some variable becomes a term headed by a big symbol, we know that some argument of the (instantiation) of s headed by a big symbol takes care of it.

The following examples show the behavior of the ordering when comparing terms headed by the same AC-symbol.

Example 3. Let $f \in \mathcal{F}_{AC}$ and $h, g, a, b \in \mathcal{F} \setminus \mathcal{F}_{AC}$, and take the precedence $h \succ_{\mathcal{F}} f \succ_{\mathcal{F}} g \succ_{\mathcal{F}} a \succ_{\mathcal{F}} b$. Then we have

1. $s = f(g(f(h(a), a)), a) \succ_{ACRPO} f(h(a), a, a) = t$ by case 4 since $f \succ_{\mathcal{F}} g$ implies $f(tf_f(f(h(a), a)), a) = f(h(a), a, a) \in EmbSmall(s)$.
2. $s = f(h(a), g(a)) \succ_{ACRPO} f(g(h(a)), a) = t$ by case 5a, since we have that $s \succ_{\mathcal{F}} f(h(a), a) \in EmbSmall(t)$ by case 4, $NoSmallHead(s) = \{h(a)\} \succeq\!\!\succeq_{ACRPO} \emptyset = NoSmallHead(t)$ and $BigHead(s) = \{h(a)\} \succ\!\!\succ_{ACRPO} \emptyset = BigHead(t)$.
3. $s = f(g(h(a)), a, a, a) \succ_{ACRPO} f(g(f(h(a), a)), a) = t$ by case 5b, since we have $\#(s) = 4 > 2 = \#(t)$, $NoSmallHead(s) = \emptyset = NoSmallHead(t)$, $BigHead(s) = \emptyset = BigHead(t)$ and, finally, $s \succ_{ACRPO} f(h(a), a, a) = t' \in EmbSmall(t)$ by applying first case 4 and then $s' = f(h(a), a, a, a) \succ_{ACRPO} f(h(a), a, a) = t'$ by case 5b, since $NoSmallHead(s') = BigHead(s') = \{h(a)\} = BigHead(t') = NoSmallHead(t')$, $EmbSmall(t') = \emptyset$ and $\#(s') = 4 > 3 = \#(t')$.
4. $s = f(h(a), a) \succ_{ACRPO} f(h(a), b) = t$, by case 5c, as $EmbSmall(t) = \emptyset$, $NoSmallHead(s) = BigHead(s) = \{h(a)\}$ and $NoSmallHead(t) = BigHead(t) = \{h(a)\}$, $\#(s) = 2 = \#(t)$ and $\{h(a), a\} \succ\!\!\succ_{ACRPO} \{h(a), b\}$.
5. $s = f(h(x, y), a, y) \succ_{ACRPO} f(g(h(x, y)), y) = t$ by case 5a, since we have that $NoSmallHead(s) = \{h(x, y), y\} \;\underline{\succ\!\!\succ}_{ACRPO}\; \{y\} = NoSmallHead(t)$, $BigHead(s) = \{h(x, y)\} \succ\!\!\succ_{ACRPO} \emptyset = BigHead(t)$ and finally, $s \succ_{ACRPO} f(h(x, y), y) = t' \in EmbSmall(t)$ by case 5b, since $NoSmallHead(s) = \{h(x, y), y\} = NoSmallHead(t')$, $BigHead(s) = \{h(x, y)\} = BigHead(t')$, $EmbSmall(t') = \emptyset$ and $\#(s) = y + 2 > y + 1 = \#(t')$.

2.2 AC-compatible polynomial interpretations

Polynomial interpretations can be used to prove AC-termination as well. To achieve AC-compatibility we need to define polynomial interpretations such that all AC-equivalent terms have the same interpretation. In this paper we will use polynomial interpretations as ingredient in our termination proofs.

Then, we can define a quasi-ordering and a strict ordering using this interpretations as follows:

- $s \succeq_{\mu} t$ iff $\mu(s) \geq \mu(t)$

– $s \succ_\mu t$ iff $\mu(s) > \mu(t)$

To ensure monotonicity of $\succeq_\mu$, we will only consider polynomial interpretations with non-negative integer coefficients.

Note that $\succ_\mu$ is included in $\succeq_\mu$ but does not coincide with the strict part of $\succeq_\mu$. The reason for making this distinction is that the given $\succ_\mu$ is stable under substitutions and the strict part of $\succeq_\mu$ is not, as shown in the example: if $\mu(f(x,y)) = x + y$, $\mu(g(x)) = x + 1$ and $\mu(h(x)) = 0$ then $\mu(g(f(z_1, z_2))) = z_1 + z_2 + 1$ and $\mu(g(z_1)) = z_1 + 1$, and hence, $\mu(g(f(z_1, z_2))) \geq \mu(g(z_1))$ and $\mu(g(z_1)) \not\geq \mu(g(f(z_1, z_2)))$, but replacing z_2 by $h(z_3)$ we have $\mu(g(f(z_1, h(z_3)))) = z_1 + 1 = \mu(g(z_1))$. Hence before the instantiation, $g(f(z_1, z_2))$ and $g(z_1)$ compare in the strict part of $\succeq_\mu$ and, after the instantiation, they do not. Finally, the defined strict ordering $\succ_\mu$ does not have this stability problem since $g(f(z_1, z_2)) \not\succ_\mu g(z_1)$.

The following lemma gives a sufficient condition to ensure AC-compatibility.

Lemma 1. *[BCL87] Let μ be a polynomial interpretation. If for all AC-symbol f we have that $\mu(f(x,y))$ is of the form $axy + b(x+y) + c$ with $b^2 = b + ac$ then $\succeq_\mu$ and $\succ_\mu$ are AC-compatible.*

For instance, if f and g are AC-symbols, the interpretation μ defined by $\mu(f(x,y)) = x + y + 1$ or by $\mu(f(x,y)) = 2 * x * y + x + y$ fulfills the aforemention conditions.

2.3 The SPO

To conclude the preliminaries we will recall briefly the definition of the semantic path ordering [KL80]. The SPO is a generalization of RPO where the comparison with the precedence is replaced by a comparison using an underlying quasi-ordering involving the whole term.

Here we will present a slightly modified version of SPO where instead of using only an underlying quasi-ordering we will use a compatible ordering pair which includes a quasi-ordering $\succeq_Q$ and a strict ordering $\succ_q$ which is compatible with $\succeq_Q$. This is done in order to use polynomial interpretations as underlying quasi-ordering, since as seen, the strict polynomial ordering we have defined does not coincide with the strict part of the given polynomial quasi-ordering, since the latter is not stable under substitutions (see Section 2.2).

Definition 5. *Let $\succeq_Q$ be a quasi-ordering and let $\succ_q$ be an ordering. Then $\langle \succeq_Q, \succ_q \rangle$ is a* compatible ordering pair *if*

- *$\succ_q$ is compatible with $\succeq_Q$, i.e., $s' \succeq_Q s \succ_q t \succeq_Q t'$ implies $s' \succ_q t'$.*
- *$\succ_q$ is well-founded.*
- *$\succ_q$ and $\succeq_Q$ are stable under substitutions.*

These compatible ordering pairs can be defined using polynomial interpretations or using standard term interpretations over some term ordering. Another possibility is to combine (lexicographically) a precedence with (quasi-)orderings obtained by using interpretations.

From a given compatible ordering pair we can build the SPO in a recursive way following a path ordering scheme.

Definition 6. *(The SPO). Let* $\langle \succeq_Q, \succ_q \rangle$ *be a compatible ordering pair.*

Let s *and* t *be terms in* $\mathcal{T}(\mathcal{F}, \mathcal{X})$*. Then* $s = f(s_1 ... s_n) \succ_{spo} t$ *if and only if*

1. $s_i \succeq_{spo} t$ *for some* $i \in \{1...n\}$ *or*
2. $t = g(t_1, \ldots, t_n)$, $s \succ_q t$ *and* $s \succ_{spo} t_i$ *for all* $i \in \{1...m\}$ *and*
3. $t = g(t_1, \ldots, t_n)$, $s \succeq_Q t$ *and* $\{s_1, ..., s_n\} \succ\!\succ_{spo} \{t_1, ..., t_m\}$

where $\succeq_{spo}$ *is* $\succ_{spo} \cup =$ *and* $\succ\!\succ_{spo}$ *is the multiset extension of* $\succ_{spo}$.

The semantic path ordering is a well-founded ordering which is stable under substitutions. Unfortunately, it is, in general, not monotonic.

3 The AC-Semantic Path Ordering

In this section we will define a AC-compatible version of SPO, based on the definition of ACRPO. In this case, to be able to adapt the ACRPO scheme, we need to restrict the family of underlying orderings used inside SPO, to those using first a precedence and later (lexicographically) some quasi-ordering comparing the whole terms. Additionally, in order to ensure stability under substitutions we need to impose some extra conditions on this quasi-ordering.

Definition 7. *Let* $\succeq_Q$ *be a quasi-ordering and let* $\succ_q$ *be an ordering. Then* $\langle \succeq_Q, \succ_q \rangle$ *is an* AC-compatible ordering pair *if*

- $\langle \succeq_Q, \succ_q \rangle$ *is a compatible ordering pair.*
- $\succ_q$ *and* $\succeq_Q$ *are AC-compatible.*
- $\succeq_Q$ *fulfils the AC-deletion property.*

In addition, we say that $\langle \succeq_Q, \succ_q \rangle$ *is AC-monotonic if both* $\succeq_Q$ *and* $\succ_q$ *are AC-monotonic.*

Finally we have two more conditions relating the precedence and $\succeq_Q$.

(R1) $\forall g \in \mathcal{F}$ such that $f \succ_{\mathcal{F}} g$ for some $f \in \mathcal{F}_{AC}$,
$g(x_1, \ldots, x_n) \succeq_Q x_i$ for all $i = 1 \ldots n$

(R2) $f(s_1, \ldots, x, \ldots, s_n) \succeq_Q g(t_1, \ldots, x, \ldots, t_m)$ implies
$f(s_1, \ldots, y, \ldots, s_n) \succeq_Q g(t_1, \ldots, y, \ldots, t_m)$ for all $f =_{\mathcal{F}} g \in \mathcal{F}_{AC}$.

Due to the unavoidable condition of being AC-compatible, all other conditions except (R1) are easily satisfied. Condition (R1) has an impact on those symbols that are smaller, in the precedence, than an AC-symbol.

However, the allowed classes of quasi-orderings turn out to be powerful enough to obtain simple proofs of termination for the non-trivial examples given in the introduction.

Additionally, again due to the stability under substitutions, we need to *linearize* some variables when comparing two terms headed by equivalent AC-symbols.

Definition 8. *Given two terms s and t such that $top(s) =_{\mathcal{F}} top(t) \in \mathcal{F}_{AC}$ and a set of variables L, the* linearization *of s and t wrt. L is defined as*

$$lin(s,t,L) = \begin{cases} <s,t> & \text{if } \nexists\ t_i = s_j \in L \\ \\ lin(s',t',L) & \text{if } t_i = s_j = v \in L \text{ where} \\ & w = new(v),\ s' = s[w]_j,\ t' = t[w]_i \end{cases}$$

Example 4. Let $f \in \mathcal{F}_{AC}$ and $h \succ_{\mathcal{F}} f \succ_{\mathcal{F}} g$. Given $s = f(h(x,y), g(z), z, y, z)$, $t = f(g(x), z, x, y, z)$ and $L = \{z\}$ we have that
$\langle \hat{s}, \hat{t} \rangle = lin(s,t,L) = \langle f(h(x,y), g(z), z_1, y, z_2), f(g(x), z_1, x, y, z_2) \rangle$.

We now give the definition of the ACSPO. For simplicity reasons, here we have only considered total precedences, but like for ACRPO this condition can be removed. On the other hand, since we consider $\succeq_{\mathcal{F}}$ as $\succ_{\mathcal{F}} \cup =_{\mathcal{F}}$ instead of $\succ_{\mathcal{F}}$ union syntactic equality then the flat form $\overline{t}$ of t has to be defined as the normal form of t wrt. the rules
$f(x_1 \dots x_{i-1}, g(y_1 \dots y_k), x_{i+1} \dots x_n) \to f(x_1 \dots x_{i-1}, y_1 \dots y_k, x_{i+1} \dots x_n)$
for each $f \in \mathcal{F}_{AC}$ and $g =_{\mathcal{F}} f$.

Definition 9. *(The ACSPO). Let $\succeq_{\mathcal{F}}$ be a total well-founded precedence such that for all $f, g \in \mathcal{F}$, $f =_{\mathcal{F}} g$ implies that either $f, g \in \mathcal{F} \setminus \mathcal{F}_{AC}$ or $f, g \in \mathcal{F}_{AC}$, and let $\langle \succeq_Q, \succ_q \rangle$ be an AC-monotonic and AC-compatible ordering pair, with $\succeq_Q$ fulfilling the above conditions (R1) and (R2).*

Let s, t be terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Then $s = f(s_1 ... s_n) \succ_{acspo} t$ if and only if

1. *$s_i \succeq_{acspo} t$ for some $i \in \{1...n\}$ or*
2. *$f \in \mathcal{F}_{AC}$ and $s' \succeq_{acspo} t$ for some $s' \in EmbSmall(s)$ or*
3. *$t = g(t_1 ... t_m)$ and either $f \succ_{\mathcal{F}} g$ or ($f =_{\mathcal{F}} g$ and $s \succ_q t$), and $s \succ_{acspo} t_i$ for all $i \in \{1...m\}$, or*
4. *$t = g(t_1 ... t_m)$, $f =_{\mathcal{F}} g \notin \mathcal{F}_{AC}$, $s \succeq_Q t$ and $\{s_1, ..., s_n\} \succ\!\!\succ_{acspo} \{t_1, ..., t_n\}$ or*
5. *$t = g(t_1 ... t_m)$, $f =_{\mathcal{F}} g \in \mathcal{F}_{AC}$, $s \succeq_Q t$, $\hat{s} \succ_{acspo} t'$ for all $t' \in EmbSmall(\hat{t})$, $NoSmallHead(s) \succeq\!\!\succeq_{acspo} NoSmallHead(t)$ and either*
   (a) $BigHead(s) \succ\!\!\succ_{acspo} BigHead(t)$ or
   (b) $\#(s) > \#(t)$ or
   (c) $\#(s) \geq \#(t)$ and $\{s_1, ..., s_n\} \succ\!\!\succ_{acspo} \{t_1, ..., t_m\}$

where $\succeq_{acspo}$ is $\succ_{acspo} \cup =_{AC}$, $\succ\!\!\succ_{acspo}$ is the AC-multiset extension of $\succ_{acspo}$ and $<\hat{s}, \hat{t}> = lin(s,t,L_{<s,t>})$ where $L_{<s,t>}$ is the set of variables $\{x \mid \exists\ t_i = x \in \mathcal{X}$ and $BigHead(s) \not\succ\!\!\succ_{acspo} BigHead(t) \cup \{x\}\}$

The definition for *NoSmallHead*, *BigHead* and *EmbSmall* are given in Section 2. Note that linearization is only needed in case 5, when checking recursively the terms in *EmbSmall*. Remark that in this case, we have also required $NoSmallHead(s) \succeq\!\!\succeq_{acspo} NoSmallHead(t)$, which ensures that for any variable x occurring as argument of t such that $BigHead(s) \not\succ\!\!\succ_{acspo} BigHead(t) \cup \{x\}$ there are, at least, as many occurrences of x as argument of s than as argument of t. Hence, after linearization all occurrences of variables in $L_{<s,t>}$ that are arguments in t have been renamed.

Theorem 2. *$\succ_{acspo}$ is a well-founded AC-compatible ordering which is stable under substitutions.*

4 The Monotonic AC-Semantic Path Ordering

Now we will obtain a monotonic version by using the same technique as for the MSPO in [BFR00] but considering AC-reduction triplets of the form $\langle \succeq_I, \succeq_Q, \succ_q \rangle$.

Definition 10. *A quasi-ordering $\succeq_Q$ is monotonic wrt. $\succeq_I$ if $s \succeq_I t$ implies $f(\ldots s \ldots) \succeq_Q f(\ldots t \ldots)$ for every $f \in \mathcal{F}$. Note that if $f \in \mathcal{F}_{AC}$ then we may need to apply some flattening.*

Then, $\langle \succeq_I, \succeq_Q, \succ_q \rangle$ is an AC-reduction triplet if $\langle \succeq_Q, \succ_q \rangle$ is an AC-monotonic AC-compatible ordering pair and $\succeq_I$ is an AC-compatible monotonic and stable under substitutions quasi-ordering and $\succeq_Q$ is monotonic wrt. $\succeq_I$.

Definition 11. *(The ACMSPO). Let $\succeq_{\mathcal{F}}$ be a precedence and $\langle \succeq_I, \succeq_Q, \succ_q \rangle$ be an AC-reduction triplet, where $\succeq_Q$ satisfies conditions (R1) and (R2).*

$$s \succ_{acmspo} t \quad iff \quad s \succeq_I t \text{ and } s \succ_{acspo} t$$

Note that $\succeq_{\mathcal{F}}$ must fulfil the same condition required in the definition of ACSPO.

Theorem 3. *$\succ_{acmspo}$ is an AC-compatible reduction ordering.*

This result implies that ACMSPO is a suitable method for proving AC-termination without the need of considering any extension rule. However, in order to use this method in practice we need to build actual adequate triplets $\langle \succeq_I, \succeq_Q, \succ_q \rangle$ fulfilling all requirements.

In most of the cases it is enough to define $\succeq_Q$ (respectively $\succ_q$) as the application of $\succeq_I$ (respectively $\succ_i$) on terms after applying a renaming of the non-AC head symbols (as in the dependency pairs method). Hence we have $s \succeq_Q t$ iff $N(s) \succeq_I N(t)$ (respectively $s \succ_q t$ iff $N(s) \succ_i N(t)$) for some renaming mapping applied only at top positions for non-AC symbols. In this case, the monotonicity of $\succeq_Q$ wrt. $\succeq_I$ is implied by the monotonicity of $\succeq_I$.

In all examples considered in this paper, even the renaming is not needed, and hence $\langle \succeq_Q, \succ_q \rangle$ and $\langle \succeq_I, \succ_i \rangle$ coincide.

In this paper, the pair $\langle \succeq_I, \succ_i \rangle$ is defined by a polynomial interpretation μ as seen in Section 2.2. The pair $\langle \succeq_I, \succ_i \rangle$ can also be obtained by using standard term interpretations (with straightforward conditions to ensure AC-compatibility and the other properties) over an AC-compatible reduction ordering on terms like ACRPO (in a similar way as done for the dependency pairs method and MSPO).

Furthermore, since $\langle \succeq_Q, \succ_q \rangle$ and $\langle \succeq_I, \succ_i \rangle$ coincide all conditions we have imposed on $\succeq_Q$ and $\succ_q$ should also be satisfied by $\succeq_I$ and $\succ_i$. Hence, we have to check that $\langle \succeq_I, \succ_i \rangle$ is an AC-monotonic AC-compatible ordering pair and $\succeq_I$, apart from monotonicity (which was already required), satisfies also R1 and R2.

The next lemma provides sufficient conditions for a polynomial interpretation to satisfy all required properties. A polynomial interpretations I is said to be *strictly positive* if no symbol is interpreted to 0 in I. The interpretation of a symbol f is said to be *fully argument dependent* if $I(f(x_1, \ldots, x_n))$ depends on all $x_1, \ldots, x_n$.

Lemma 2. *Given a precedence $\succeq_{\mathcal{F}}$ and a polynomial interpretation I with non-negative integer coefficients. The pair $\langle \succeq_I, \succ_i \rangle$ is an AC-monotonic AC-compatible ordering pair and $\succeq_I$ is monotonic and satisfies R1 and R2 if*

- *for all AC-symbol f we have that $I(f(x,y)) = axy + b(x+y) + c$ with*
  1. *$b^2 = b + ac$ and*
  2. *either I is strictly positive and $a \neq 0$ or $b \neq 0$.*
- *for all symbol g s.t. $f \succ_{\mathcal{F}} g$ for some AC-symbol f we have that either*
  1. *I is strictly positive and the interpretation of g is fully argument dependent, or*
  2. *$I(g(x_1, \ldots, x_n)) = P + b_1 x_1 + \ldots + b_n x_n$ where P is a polynomial and all $b_i \neq 0$.*

Similarly, we can impose some simple condition on term interpretations in order to satisfy all required properties.

5 Examples

In this section we will present several examples together with the ingredients, i.e., a precedence and a polynomial interpretation, required to show its termination by ACMSPO. For these examples, we have checked that all rules are included in ACMSPO (this is detailed in the Example 6). In this section, we assume that the ingredients are provided by the user and, hence, we only need to check the rules following the definition of ACMSPO (as usually done for path orderings). In the following section a way to automatically generate these ingredients is studied. Let us start with a very simple example.

Example 5. Both $+$ and $*$ are AC-symbols.

$$\begin{aligned} x + 0 &\to x \\ x + s(y) &\to s(x+y) \\ x * 0 &\to 0 \\ x * s(y) &\to x * y + x \end{aligned}$$

$$\begin{aligned} minus(x, 0) &\to x \\ minus(s(x), s(y)) &\to minus(x, y) \\ quot(0, s(y)) &\to 0 \\ quot(s(x), s(y)) &\to s(quot(minus(x,y), s(y))) \end{aligned}$$

This rules are included in ACMSPO taking as precedence $quot \succ_{\mathcal{F}} minus \succ_{\mathcal{F}} * \succ_{\mathcal{F}} + \succ_{\mathcal{F}} s$ and as $\langle \succeq_I, \succ_i \rangle = \langle \succeq_Q, \succ_q \rangle$ the polynomial interpretation $I(quot(x,y)) = x$, $I(minus(x,y)) = x$, $I(s(x)) = x+1$, $I(0) = 1$, $I(x+y) = x+y$ and $I(x * y) = x * y + x + y$.

The following example comes with the complete proof showing that all rules are included in ACMSPO with the given ingredients (i.e., a precedence and a polynomial interpretation).

Example 6. Binary arithmetic (given in the introduction). Let us recall the rules ($+$ and $*$ are the only AC-symbols).

$$(\#)0 \to \#$$
$$x + \# \to x$$
$$x * \# \to \#$$

$$(x)0 + (y)0 \to (x+y)0$$
$$(x)0 + (y)1 \to (x+y)1$$
$$(x)1 + (y)1 \to (x+y+(\#)1)0$$

$$x * (y)0 \to (x*y)0$$
$$x * (y)1 \to x + (x*y)0$$

Take as precedence $* \succ_{\mathcal{F}} + \succ_{\mathcal{F}} 1 \succ_{\mathcal{F}} 0 \succ_{\mathcal{F}} \#$ and as $\langle \succeq_I, \succ_i \rangle = \langle \succeq_Q, \succ_q \rangle$ the polynomial interpretation $I(\#) = 1$, $I((x)0) = x$, $I((x)1) = x+2$, $I(x+y) = x+y$ and $I(x*y) = x*y+x+y$.

Now, we will show how the definition of ACMSPO is used for checking the rules. We have to check that all rules $l \to r$ are included in ACMSPO, which, by definition, means that $l \succeq_I r$ and $l \succ_{acspo} r$.

$\diamond$ First we prove $l \succeq_I r$ for all $l \to r \in \mathcal{R}$:

$$I((\#)0) = 1 \geq 1 = I(\#)$$
$$I(x + \#) = x + 1 \geq x = I(x)$$
$$I(x * \#) = x * 1 + x + 1 \geq 1 = I(\#)$$
$$I((x)0 + (y)0) = x + y \geq x + y = I((x+y)0)$$
$$I((x)0 + (y)1) = x + y + 2 \geq x + y + 2 = I((x+y)1)$$
$$I((x)1 + (y)1) = x + 2 + y + 2 \geq x + y + 1 + 2 = I((x + y + (\#)1)0)$$
$$I(x * (y)0) = x * y + x + y \geq x * y + x + y = I((x*y)0)$$
$$I(x * (y)1) = x * (y+2) + x + y + 2 \geq x + x * y + x + y = I(x + (x*y)0)$$

$\diamond$ Now we show $l \succ_{acspo} r$ for all $l \to r \in \mathcal{R}$.

(1) $(\#)0 \succ_{acspo} \#$ by case 1,

(2) $x + \# \succ_{acspo} x$ by case 1,

(3) $x * \# \succ_{acspo} \#$ by case 1,

(4) $(x)0 + (y)0 \succ_{acspo} (x+y)0$ by case 3

$+ \succ_{\mathcal{F}} 0$ and $(x)0 + (y)0 \succ_{acspo} x + y$ by applying case 2 twice since

$(x)0 + y \in EmbSmall((x)0 + (y)0)$ and

$x + y \in EmbSmall((x)0 + y)$

(5) $(x)0 + (y)1 \succ_{acspo} (x + y)1$ by case 3
  $+ \succ_{\mathcal{F}} 1$ and $(x)0 + (y)1 \succ_{acspo} x + y$ by applying case 2 twice since
    $(x)0 + y \in EmbSmall((x)0 + (y)1)$ and
      $x + y \in EmbSmall((x)0 + y)$
(6) $(x)1 + (y)1 \succ_{acspo} (x + y + (\#)1)0$ by case 3
  $+ \succ_{\mathcal{F}} 0$ and $(x)1 + (y)1 \succ_{acspo} x + y + (\#)1$ by case 3:
    $+ =_{\mathcal{F}} +$ and $I((x)1 + (y)1) = x + 2 + y + 2 > x + y + 1 + 2 = I(x + y + (\#)1)$
    $(x)1 + (y)1 \succ_{acspo} x$ by applying case 1 twice
    $(x)1 + (y)1 \succ_{acspo} y$ by applying case 1 twice
    $(x)1 + (y)1 \succ_{acspo} (\#)1$ by case 3 :
      $+ \succ_{\mathcal{F}} 1$
      $(x)1 + (y)1 \succ_{acspo} \#$ by case 3 since $+ \succ_{\mathcal{F}} \#$
(7) $x * (y)0 \succ_{acspo} (x * y)0$ by case 3
  $* \succ_{\mathcal{F}} 0$ and $x * (y)0 \succ_{acspo} x * y$ by case 2 since $x * y \in EmbSmall(x * (y)0)$
(8) $x * (y)1 \succ_{acspo} x + (x * y)0$ by case 3
  $* \succ_{\mathcal{F}} +$ and $x * (y)1 \succ_{acspo} x$ by case 1
  $x * (y)1 \succ_{acspo} (x * y)0$ by case 3
    $* \succ_{\mathcal{F}} 0$
    $x * (y)1 \succ_{acspo} x * y$ by case 2 since $x * y \in EmbSmall(x * (y)1)$

We come back two the second example of the introduction. Here we only provide the ingredients for the proof. Note that the polynomial interpretation we are using is very simple, which shows that the use of a precedence first is crucial.

Example 7. Ternary integral arithmetic (given in the introduction in Example 2).

It is proved included in $\succ_{acmspo}$ as follows : take as precedence $* \succ_{\mathcal{F}} - \succ_{\mathcal{F}} + \succ_{\mathcal{F}} opp \succ_{\mathcal{F}} 1 =_{\mathcal{F}} j \succ_{\mathcal{F}} 0 \succ_{\mathcal{F}} \#$ and as $\langle \succeq_I, \succ_i \rangle = \langle \succeq_Q, \succ_q \rangle$ the polynomial interpretation $I(\#) = 0$, $I(opp(x)) = x$, $I((x)0) = x$, $I((x)1) = I((x)j) = x + 1$, $I(x - y) = I(x + y) = x + y$ and $I(x * y) = x * y + x + y$.

The last example does not include any AC-symbol, but it is proved terminating in combination with the rules of the ternary integral arithmetic of the previous example. These two TRS are used in [CMR97] to later describe the rational numbers.

Example 8. Ternary integral arithmetic (cont.)

$$
\begin{array}{rlrl}
 & & abs(x) \rightarrow & abs?(x, x, \#) \\
sign(x) \rightarrow & sign?(x, \#) & abs?((x)0, y, z) \rightarrow & abs?(x, y, z) \\
sign?(\#, x) \rightarrow & x & abs?((x)1, y, z) \rightarrow & abs?(x, y, (\#)1) \\
sign?((x)0, y) \rightarrow & sign?(x, y) & abs?((x)j, y, z) \rightarrow & abs?(x, y, (\#)j) \\
sign?((x)1, y) \rightarrow & sign?(x, (\#)1) & abs?(\#, x, \#) \rightarrow & \# \\
sign?((x)j, y) \rightarrow & sign?(x, (\#)j) & abs?(\#, x, (\#)1) \rightarrow & x \\
 & & abs?(\#, x, (\#)j) \rightarrow & opp(x)
\end{array}
$$

$$
\begin{aligned}
min(x,y) &\rightarrow min?(abs(y) - abs(x), x, y, \#)\\
min'(x,y) &\rightarrow min?(abs((y)1) - abs((x)1), x, y, \#)\\
min''(x,y) &\rightarrow min?(abs((y)j) - abs((x)j), x, y, \#)\\
min?((x)0, y, z, u) &\rightarrow min?(x, y, z, u)\\
min?((x)1, y, z, u) &\rightarrow min?(x, y, z, (\#)1)\\
min?((x)j, y, z, u) &\rightarrow min?(x, y, z, (\#)j)\\
min?(\#, x, y, \#) &\rightarrow x\\
min?(\#, x, y, (\#)1) &\rightarrow x\\
min?(\#, x, y, (\#)j) &\rightarrow y
\end{aligned}
$$

Let R_1 be the rules for the Example 7 and R_2 the rules given above. Then $R_1 \cup R_2$ is proved included in ACMSPO by taking as precedence $min =_{\mathcal{F}} min' =_{\mathcal{F}} min'' \succ_{\mathcal{F}} min? \succ_{\mathcal{F}} abs \succ_{\mathcal{F}} abs? \succ_{\mathcal{F}} * \succ_{\mathcal{F}} - \succ_{\mathcal{F}} + \succ_{\mathcal{F}} opp \succ_{\mathcal{F}} 1 =_{\mathcal{F}} j \succ_{\mathcal{F}} 0 \succ_{\mathcal{F}} \#$ and as $\langle \succeq_I, \succ_i \rangle = \langle \succeq_Q, \succ_q \rangle$ the polynomial interpretation $I(\#) = 0$, $I(opp(x)) = x$, $I((x)0) = x$, $I((x)1) = I((x)j) = x+1$, $I(x-y) = I(x+y) = x+y$, $I(x*y) = x*y+x+y$, $I(sign(x)) = 2x$, $I(sign?(x,y)) = 2x+y$, $I(abs(x)) = 2x$, $I(abs?(x,y,z)) = x+y$, $I(min(x,y)) = 3x+3y$, $I(min'(x,y)) = I(min''(x,y)) = 3x+3y+4$ and $I(min?(x,y,z,u)) = x+y+z$.

6 Constraints

Proving termination with ACMSPO can be translated into a constraint solving problem in the same way as it is done for MSPO in [BFR00] and implemented in a fully automated system called *Termptation* [BR02]. The idea is to extract a set of constraints from the application of the definition of ACMSPO to the set of rules. Hence, if we can solve the constraints then there is a proof of termination of the given TRS using ACMSPO.

First, we obtain set of constraints on $\succeq_I$ from the first condition of the definition of ACMSPO applied to all rules. Then, we have to automatically generate a precedence to be used in the application of ACSPO. Note that, although theoretically there are exponentially many possibilities, in practice it can be done efficiently (as for the non-AC case). After that, we obtain constraints on $\succeq_I$ and $\succ_i$ by applying the definition of ACSPO. For instance, for the Example 6, after guessing the precedence $* \succ_{\mathcal{F}} + \succ_{\mathcal{F}} 1 \succ_{\mathcal{F}} 0 \succ_{\mathcal{F}} \#$ and assuming, for simplicity reasons, that $\succeq_I$ (resp. $\succ_i$) and $\succeq_Q$ (resp. $\succ_q$) coincide, we have the following constraints.

The first set is coming from the first condition of the definition of ACMSPO:

$$
\begin{array}{lll}
(\#)0 \succeq_I \# & (x)0 + (y)0 \succeq_I (x+y)0 & x * (y)0 \succeq_I (x*y)0\\
x + \# \succeq_I x & (x)0 + (y)1 \succeq_I (x+y)1 & x * (y)1 \succeq_I x + (x*y)0\\
x * \# \succeq_I \# & (x)1 + (y)1 \succeq_I (x+y+(\#)1)0 &
\end{array}
$$

The second set is coming from the recursive application of the definition of ACSPO, by adding all conditions on $\succeq_I$ and $\succ_i$ (in fact, on $\succeq_Q$ and $\succ_q$) that we find in the chosen path of the definition of ACSPO (we have taken the same path as in the proof given in the example). In this example, since most of the comparisons are solved by subterm (case 1), AC-embedding (case 2) or precedence (case 3), we have only one constraint in this second set:

$$(x)1 + (y)1 \succ_i x + y + (\#)1$$

These constraints can be handled and solved by finding an appropriate polynomial interpretation (as done, for instance, in CiME system [CMMU00]). We are currently working on the implementation of this method inside the *Termptation* system.

7 Further work and conclusions

The ACMSPO is the first method not being a simplification ordering that can prove AC-termination without considering extended rules. This allow us to prove in a simple way several non-trivial examples of the literature.

In order to ensure monotonicity and stability under substitutions we have needed to impose some additional conditions on the underlying orderings used in ACSPO. These conditions can be relaxed by adding more checking in the ACSPO and some more linearizations, which restrict the applicability of the method. This trade off between the conditions imposed on the underlying orderings and the conditions required in the checking of ACSPO has to be further analyzed.

Another nice property of our method is that we strongly believe that it can be easily combined with methods based on extended rules. In fact, we can obtain a much simpler AC-compatible version of SPO, called EACSPO, which needs to consider extended rules. In this case, we only need to flatten terms and require the underlying quasi-ordering to be AC-compatible. The resulting method is very similar to the existing AC-versions of the dependency pairs method.

We are currently working on the combination both AC-extensions of SPO, namely ACSPO and EACSPO, with the aim of obtaining a method such that for every AC-symbol one can choose whether to add the extended rules or not. Then an AC-symbol is treated like in the ACSPO if it has no extended rules included, otherwise is treated like in EACSPO, which is almost like the other symbols. In this way one can get the best of both methods.

References

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[BCL87] Ahlem Ben-Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9:137–160, 1987.

[BFR00] C. Borralleras, M. Ferreira, and A. Rubio. Complete monotonic semantic path orderings. Proc. of *17th Int. Conf. on Automated Deduction*, LNAI 1831:346–364, 2000.

[BP85] L. Bachmair and D.A. Plaisted. Termination orderings for associative-commutative rewriting systems. *Journal of Symbolic Computation*, 1:329–349, 1985.

[BR01] C. Borralleras and A. Rubio. A Monotonic Higher-Order Semantic Path Ordering. Proc. of *8th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'01)* LNAI 2250:531–547, 2001.

[BR02] C. Borralleras and A. Rubio. TERMPTATION: TERMination Proof Techniques automATION. Available at `//www.lsi.upc.es/~albert/`.

[BR03] C. Borralleras and A. Rubio. Proving Termination of AC-rewriting without Extended Rules. Long version. Available at `//www.lsi.upc.es/~albert/`.

[CMMU00] E. Cotejean, C. Marché, B. Monate and X. Urban. CiME system. 2000. Available at `//cime.lri.fr/`.

[CMR97] E. Cotejean, C. Marché and L. Rabehasaina. Rewrite systems for natural, integral, and rational arithmetic. Proc. of *8th Int. Conf. on Rewriting Techniques and Applications, RTA'97*, LNCS 1232:98–112, 1997.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[DP93] Catherine Delor and Laurence Puel. Extension of the associative path ordering to a chain of associative commutative symbols. Proc. of *5th Int. Conf. on Rewriting Techniques and Applications RTA'93*, LNCS 690:389–404, 1993.

[GK01] J. Giesl and D. Kapur Dependency pairs for Equational Rewriting. Proc. of *12th Int. Conf. on Rewriting Techniques and Applications, RTA'01*, LNCS 2051:93–108, 2001.

[KL80] S. Kamin and J. J. Levy. Two generalizations of the recursive path ordering. Unpublished note, Dept. of Computer Science, Univ. of Illinois, Urbana, IL, 1980.

[Knu97] D. E. Knuth. The Art of Computer Programming. Vol. 2: Seminumerical Algorithms. Third Edition. Addison-Wesley, 1997.

[KS00] D. Kapur and G. Sivakumar. Proving Associative-Commutative Termination Using RPO-compatible Orderings. In *Proc. Automated Deduction in Classical and Non-Classical Logics*, LNAI 1761:40–62, 2000.

[KSZ95] D. Kapur, G. Sivakumar and H. Zhang. A Path Ordering for Proving Termination of AC Rewrite Systems. In *Journal of Automated Reasoning*, 14(2):293-316, 1995.

[KT01] K. Kusakari and Y. Toyama. On proving AC-termination by AC-dependency pairs. IEICE Transactions on Information and Systems, E84-D(5):604–612, 2001.

[MU98] Claude Marché and Xavier Urbain. Termination of associative-commutative rewriting by dependency pairs. Proc. of *9th Int. Conf. on Rewriting Techniques and Applications, RTA'98*, LNCS 1379:241–255, 1998.

[RN95] A. Rubio and R. Nieuwenhuis. A total AC-compatible ordering based on RPO *Theoretical Computer Science*, 142(2):209–227, 1995.

[Rub02] A. Rubio. A fully syntactic *AC*-RPO. *Information and Computation*, 178(2):515-533, 2002.

[Urb01] X. Urbain. Approche incrémentale des preuves automatiques de terminaison. PhD thesis, Université Paris-Sud, UFR Scientifique d'Orsay, 2001.