ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

M P A Sellink (Ed)

2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam 1997

Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, 25-26 September 1997

Compilation Techniques for Associative-Commutative Normalisation

P.-E. Moreau and H. Kirchner

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN: 3-540-76228-0

Compilation Techniques for Associative-Commutative Normalisation

Pierre-Etienne Moreau
LORIA & UHP

Hélène Kirchner
LORIA & CNRS

BP 101, 54602 Villers-les-Nancy CEDEX, FRANCE
<http://www.loria.fr/equipes/protheo>

Abstract

We consider the problem of term normalisation modulo associative-commutative (AC) theories and describe several techniques for compiling many-to-one AC matching and reduced term construction. The proposed method, illustrated on three examples, is based on compact bipartite graphs, and is designed for working very efficiently on specific classes of AC patterns. Our experimental results provide strong evidence that compilation of many-to-one AC normalisation is a useful technique for improving the performance of algebraic programming languages.

1 Introduction

Term rewriting techniques are used in software development, compiler generation, computer algebra, theorem provers, more recently in constraint solving [6]. Several programming languages (for example ASF+SDF [12], OBJ [7], or Maude [3]) use term rewriting as the execution engine of their programs.

Algebraic structures often involve axioms like associative-commutative (AC) properties of function symbols, that cannot be used as terminating rewrite rules, but instead can be handled implicitly by working with congruence classes of terms. For practical implementation purposes, representatives of these congruence classes are chosen, and the matching step of term rewriting is performed by special matching algorithms, specific to the equational theories in use. An important case in practice is the case of associative-commutative theories.

In this paper, we present several compilation techniques developed for an extension of the ELAN compiler [15] in order to handle AC function symbols. Experiments have shown speed-ups of the ELAN interpreter up to 2,000 times when rewriting with AC function symbols.

In section 2, we give the useful preliminary definitions and describe the term data structure used by the compiler. In section 3, we explain the fundamental ideas to compile leftmost-innermost normalisation, first without AC symbols. We then explain the additional difficulties brought by the AC case. In section 4, we propose several new techniques to compile AC normalisation in an efficient way. In section 5, we evaluate the performance of our compiler and compare it with other existing systems. Concluding remarks are given in section 6.

2 Preliminaries

2.1 Basic notions

We consider *terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ built from a given finite set $\mathcal{F}$ of function symbols and a set $\mathcal{X}$ of variables. $\mathcal{T}(\mathcal{F})$ is the set of ground terms (without variables). We denote the set of variables occurring in a term t by $\text{Var}(t)$.

Given a binary function symbol u , let AC be the set of associativity and commutativity axioms

$$u(x, u(y, z)) = u(u(x, y), z) \quad \text{and} \quad u(x, y) = u(y, x).$$

The notation u_{AC} emphasises that u is such an AC function symbol that satisfies these two axioms. $\mathcal{F}_\emptyset$ is the subset of $\mathcal{F}$ made of function symbols which are not associative-commutative, and are called *free* function symbols. A term is said to be *syntactic* if it does not contain any AC function symbol.

Positions in a term are represented as sequences of integers. The empty sequence ϵ denotes the top-most position. A term t is said to be *linear* if no variable occurs more than once.

A *substitution* is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, written $\sigma = \{y_1 \mapsto t_1, \dots, y_k \mapsto t_k\}$. It uniquely extends to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

A *conditional rewrite rule* is a triple of terms denoted $c \Rightarrow l \rightarrow r$ such that $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, c is a boolean term, and $\text{Var}(c) \cup \text{Var}(r) \subseteq \text{Var}(l)$. c is called the *condition*, l the *left-hand side* or *pattern* and r the *right-hand side*. A rewrite rule is said to be *syntactic* if the left-hand side is a syntactic term.

We assume the reader to be familiar with basic ideas of term rewriting and AC theories. Full definitions can be found for instance in [14, 11].

2.2 Canonical form

Canonical forms are terms that are representatives of their AC congruence class.

We allow AC function symbols to be varyadic. Identical arguments of an AC function symbol are grouped together. So any argument of an AC function symbol has a *multiplicity* which is a positive integer, denoted by a superscript.

Canonical form computation can be defined as a function $\mathcal{CF} : \mathcal{T}(\mathcal{F}, \mathcal{X}) \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$ that chooses, for each term t , a unique representative of its congruence class. In order to recursively compute canonical forms, we assume given a total ordering $>$ on the set of function symbols $\mathcal{F}$ and variables, and extend it to terms in canonical form as follows. For terms that are just variable or constant symbols, $\mathcal{CF}$ is the identity function and the total ordering is the given ordering $>$ on symbols. For terms in canonical form with different top symbols, the total ordering on canonical form is given by the ordering on their top symbols. The canonical form is obtained by flattening nested occurrences of the same AC function symbol, recursively computing the canonical forms and sorting the subterms, and replacing k identical subterms by a single instance of the subterm with multiplicity k . These canonical forms and term ordering were introduced in [10]. For more details, see [5].

Given an AC function symbol u_{AC} , consider for example $t = u_{AC}(u_{AC}(t_1, t_2), t_3, u_{AC}(t_4, t_5))$ where no t_i has u_{AC} as top symbol. Suppose that $\mathcal{CF}(t_1) = \mathcal{CF}(t_5)$, $\mathcal{CF}(t_3) = \mathcal{CF}(t_4)$ and $\mathcal{CF}(t_1) > \mathcal{CF}(t_2) > \mathcal{CF}(t_3)$. Then $\mathcal{CF}(t) = u_{AC}(\mathcal{CF}(t_1)^2, \mathcal{CF}(t_2), \mathcal{CF}(t_3)^2)$.

2.3 Term representation

The fundamental object in an algebraic programming language is the first-order term. In implementations, the representation of first-order terms is generally based on trees. An alternative representation of terms which is linear rather than tree-like has been proposed by Jim Christian [2]. Those *flatterms*, represented by a doubly-linked list data structure, yield in practice simpler and faster traversal algorithms than the conventional tree representation. In our compiler, AC function symbols may be varyadic and it must be possible to share subterms. The *flatterms* representation is not compatible with this last requirement because subterms stored in the doubly-linked list cannot be shared. The last constraint is that no time-overhead is wanted when no AC symbol is used.

To satisfy those constraints, two different tree based representations are used. Subterms of free function symbols are stored in an array, while subterms of AC function symbols are stored in a simply-linked-list. The representation of terms $f(g(b), a, g(b))$ and $u_{AC}(g(b), a, g(b))$ is illustrated in the two first pictures of the following figure:

This data structure can be easily extended to represent terms in canonical form. The third picture illustrates how multiple copies of equivalent terms are represented by storing their multiplicity in list-cells.

3 Compiling leftmost-innermost normalisation

In order to present the currently known compilation techniques, we first briefly introduce in this section the method used for compilation of leftmost-innermost normalisation in the case of syntactic rules in the ELAN compiler [15]. Then we sketch a few techniques used for AC normalisation and the related difficulties.

3.1 Normalisation with syntactic rules

The general idea of the ELAN compiler is to transform a given ELAN program (a conditional rewrite rule system) into a C program, which takes a ground input term s in internal representation (usually called *subject*) and computes a normal form of s .

Given a set of conditional rules, a normalisation step begins by selecting a rule that can be applied to the subject. A successful matching instantiates the variables of the left-hand side, and the condition is then evaluated. If the condition is evaluated to *false*, another rule is selected. Then the normalisation process proceeds by cutting the part of the input term corresponding to the instantiated left-hand side of the rewrite rule and finally constructing the reduced term by plugging the instantiated right-hand side of the rule. If no rule is applicable, a special function *match_fail* is called. This function returns the current subject as normal form. Several refinements of this general schema are possible and have been implemented in the ELAN compiler.

3.1.1 Many-to-one syntactic matching

The first possible improvement concerns the compilation of the matching algorithm. It consists in regrouping rules with the same top symbol on the left-hand side. For each functional symbol f of arity n , the collected rules are compiled into a C function fun_f of n arguments. This function when applied to the terms $t_1, \dots, t_n$, returns the reduced form of the term $f(t_1, \dots, t_n)$, according to the leftmost innermost normalisation strategy. All rewrite rules are known at compile time. Matching of their left-hand sides to an unknown input term can be pre-compiled using some existing algorithm [9, 8]. The resulting code gives the set of applicable rules represented in a very efficient manner by a bitset. Since we are interested in leftmost innermost evaluation, the construction of the reduced term can be compiled into nested calls of different fun_f functions.

3.1.2 Reusing parts of terms

Another improvement comes from the Berlin project OPAL [4]. The main idea is to reuse parts of left-hand sides of rules to construct the right-hand sides when applying a rewrite rule. However this technique requires to use tags or reference counters in order to indicate whether a subterm is shared by several terms. More details can be found in [15].

3.2 Associative-Commutative normalisation

Associative-commutative rewriting is a more complex mechanism and it is known since 1987 that AC matching is NP-complete [1]. Let us focus now on difficulties added by AC theories.

3.2.1 One-to-one general AC-matching

Given a ground subject s and a set of rules $r_1, \dots, r_n$, the first thing to do is to select a rule r_i and try to apply it at the top position of s . To check if a selected rule is applicable, a matching algorithm is needed. In the syntactic case (when all rules are syntactic), it is quite simple. Starting from the root of s and the left-hand side of r_i it consists in checking that function symbols from the *pattern* appear at the same position in s . Suppose that $f(x, g(h(a, y), z))$ is the pattern and $f(a, g(h(a, b), c))$ is the subject. There is a trivial substitution $\{x \mapsto a, y \mapsto b, z \mapsto c\}$ such that the pattern matches the subject.

For the AC case, the problem becomes more difficult, since different permutations of s have to be considered. For instance, in the previous example, if f is AC, the substitution $\{x \mapsto c, y \mapsto b, z \mapsto a\}$ is another solution to the matching problem. Given a ground subject $s = u_{AC}(s_1^{\alpha_1}, \dots, s_p^{\alpha_p})$, and a pattern $u_{AC}(p_1, \dots, p_n)$, bipartite graphs are usually used to represent the matching problems: an edge is built between p_j and s_i if p_j matches s_i .

Associative-commutative matching can be presented in four steps [5]:

1. convert pattern and subject to their canonical-forms;
2. decompose the matching problem into a hierarchy of bipartite graphs;
3. find a compatible set of solutions to the bipartite graph matching problems and construct a Diophantine equational system which encodes the constraints on the remaining unbound variables;
4. solve the Diophantine equational system to get a matching substitution.

This algorithm is quite general. Each time it tries to apply a rule, it has to build and solve the hierarchy of bipartite graphs and the Diophantine equational system. This approach is very expensive in memory management because many dynamic data structures have to be built and deleted.

The method described above is efficient when a complex matching problem has to be solved (many nested AC operators and a lot of non-linearity). In contrast to theorem proving, where rewrite steps may involve complex non-linear patterns and relatively small subjects, in algebraic programming, we usually have a lot of simple linear patterns and huge subjects.

3.2.2 Reduced term construction

If the selected rule can be applied to the subject, the only thing to do consists in transforming the subject. As in the syntactic case, this is done by replacing the current subject by the right-hand side of the selected rule in which all variables have been instantiated by the substitution. This substitution is usually built during the AC matching phase. But an additional difficulty comes with conditional rewrite rules. In conditional rewriting, conditions have to be normalised and compared to *true* before building the right-hand side. If a condition is evaluated to *false*, this means that the current substitution is not compatible with this condition. As long as there is a solution to the matching problem for which the condition is not satisfied, another solution has to be found and extracted from the matching problem. If no solution is remaining, the selected rule is no longer a good candidate and another one is selected. Conditional rewriting requires matching problems to be solved in a particular way: the first solution has to be found as fast as possible, and the others have to be found “on request”. It is not necessary to compute immediately all solutions to the matching problem.

4 New compilation techniques for AC normalisation

The method presented in section 3.2.1 is correct but quite inefficient in practice. The matching algorithm is too general for the usual cases, but it seems to be very difficult to get a better equivalent algorithm. The main drawbacks of the previous approach are:

- one-to-one matching is too slow in practice;
- too many data structures depending on the non-static subject size have to be managed during runtime;
- the reduced term construction phase is not sufficiently optimised and it is not connected with the matching phase.

In this section we propose some new techniques that give significant speed-ups on problems met in practice. Our solutions are based on the fact that all rewrite rules are known before the compilation phase. So we can do particular treatments at compile time such as many-to-one pre-compilation and static data structure constructions. In addition, our approach consists in designing particular algorithms for frequently occurring cases. We did an inspection of the most useful AC patterns used in ELAN, and extracted the pattern classes presented in 4.1. Specific compilation techniques have been developed for those classes of patterns. Matching problems involving other patterns can be solved with the general matching algorithm sketched in section 3.2.1.

4.1 Pattern classes

Compilation algorithms presented in this section only work on a subset of patterns in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Here, we describe by successive refinements several pattern classes that correspond to families of frequently used left-hand sides in rewrite rule programming.

Recall that $\mathcal{F}_\emptyset$ is the set of free function symbols, $\mathcal{F}_{AC}$ the set of AC function symbols and $\mathcal{X}$ the set of variables.

The pattern class C_0 contains flat terms in canonical form with one AC function symbol on top and two variables with multiplicity greater than 1 immediately under this AC function symbol.

Definition 1 *The pattern class C_0 consists of terms t in canonical form of the form $t = u_{AC}(x^\alpha, y^\beta)$ with $u_{AC} \in \mathcal{F}_{AC}$, $x, y \in \mathcal{X}$ with $x \neq y$, and α, β integers ≥ 1 .*

The pattern class C_1 contains linear terms in canonical form that have only one AC function symbol on top and at most one variable immediately under this AC function symbol.

Definition 2 *The pattern class C_1 consists of linear terms t in canonical form such that:*

- $t \in \mathcal{T}(\mathcal{F}_\emptyset, \mathcal{X}) \setminus \mathcal{X}$; or
- $t = u_{AC}(t_1, \dots, t_n)$, $u_{AC} \in \mathcal{F}_{AC}$ and $t_1, \dots, t_n \in \mathcal{T}(\mathcal{F}_\emptyset, \mathcal{X}) \setminus \mathcal{X}$; or
- $t = u_{AC}(x, t_1, \dots, t_n)$, $u_{AC} \in \mathcal{F}_{AC}$, $x \in \mathcal{X}$ and $t_1, \dots, t_n \in \mathcal{T}(\mathcal{F}_\emptyset, \mathcal{X}) \setminus \mathcal{X}$.

The third class contains terms from classes C_0 and C_1 , and terms built by composition with free function symbols.

Definition 3 *The pattern class C_2 is the smallest set of terms in canonical form that contains C_0 and C_1 , and such that $f(t_1, \dots, t_n)$ is in C_2 , if $f \in \mathcal{F}_\emptyset$, $t_1, \dots, t_n \in C_2$ and all the t_i have disjoint sets of variables.*

The fourth class extends the previous class to two levels of AC function symbols.

Definition 4 *The pattern class C_3 is the set of terms t in canonical form such that:*

- $t = u_{AC}(t_1, \dots, t_n)$ with $u_{AC} \in \mathcal{F}_{AC}$ and $t_1, \dots, t_n \in C_2$; or
- $t = u_{AC}(x, t_1, \dots, t_n)$ with $u_{AC} \in \mathcal{F}_{AC}$, $t_1, \dots, t_n \in C_2$, $x \in \mathcal{X}$ and x does not belong to the sets of variables of $t_1, \dots, t_n$.

Consider the term $v_{AC}(y, g(x_1, \dots, x_m))$ where $x_1, \dots, x_m$ are variables. It contains only one AC function symbol ' v_{AC} ' and one variable ' y ' immediately under v_{AC} . This term is in the class C_1 . By adding a free function symbol f of arity 3 at the top position, a term $u_{AC}(y_1, y_1, y_2)$ of the class C_0 , and a variable z , the term: $f(v_{AC}(y, g(s_1, \dots, s_m)), z, u_{AC}(y_1, y_1, y_2))$ becomes a member of the class C_2 . By adding a second level AC function symbol w_{AC} and a variable x , the term $w_{AC}(x, f(v_{AC}(y, g(s_1, \dots, s_m)), z, u_{AC}(y_1, y_1, y_2)))$ becomes a member of the class C_3 .

4.2 Many-to-one AC-matching

In order to minimise the number of matching attempts, the technique used in syntactic many-to-one matching can be adapted to the AC case.

Given a set of rewrite rules, the method consists in regrouping subterms with the same AC top function symbol. A many-to-one matching algorithm for this set of subterms is used to build a particular bipartite graph called *Compact Bipartite Graph* and described below. Given a subject, the purpose of a compact bipartite graph is to encode in only one data structure all matching problems relative to the given set of rewrite rules.

Instead of repeatedly selecting a rule, building a bipartite graph, solving it, selecting another rule, and so on... until conditions are evaluated to *true*, this compact data structure represents all bipartite graphs that the general algorithm has to build, in a more efficient way. The speedup comes from the fact that common syntactic subterms are matched only once, even if they appear in several rules.

The method presented here is related to discrimination nets [2] and AC discrimination nets [13]. AC discrimination nets do similar improvements without any limitation on pattern structure. However the automaton built to encode the discrimination net is not really compiled. It is stored in an abstract data structure that is interpreted to simulate the automaton execution. In the approach proposed here, the automaton is directly coded in an imperative language like C. No intermediate abstract data structure is built to represent the automaton. Although the two approaches are not easily comparable, AC discrimination nets seem to be a good choice to implement theorem provers whose pattern database can be modified during computations. On the contrary, our approach seems to be promising to implement algebraic programming languages in an efficient way.

4.3 Compact Bipartite Graph

Given a set of rewrite rules, syntactic subterms $p_1, \dots, p_n$ of their left-hand sides with the same AC top function symbol u_{AC} are grouped together and compiled into a function ($fun_{u_{AC}}$) that implements the many-to-one syntactic matching described in section 3.1.1. Given a subject $s = u_{AC}(s_1^{\alpha_1}, \dots, s_p^{\alpha_p})$, this function returns a bitset for each s_i which encodes possible matches. The j^{th} bit is set to 1 if p_j matches s_i . This information is used to build the compact bipartite graph: an edge is built between p_j and s_i if p_j matches s_i .

Let us illustrate this construction on an example. Given the two rewrite rules, $u_{AC}(f(a, x), f(y, g(b))) \rightarrow r_1$, and $u_{AC}(z, f(a, x'), g(a)) \rightarrow r_2$, an analysis of subterms with the same AC top function symbol u_{AC} gives three distinct non-variable subterms up to variable renaming: $p_1 = f(a, x)$, $p_2 = f(y, g(b))$ and $p_3 = g(a)$. Variable subterms (z in the previous example) are not involved in compact bipartite graph construction. They are instantiated in the reduced term construction phase described in 4.4. Let us consider the subject

$$s = u_{AC}(f(a, a)^{\alpha_1}, f(a, c)^{\alpha_2}, f(b, g(b))^{\alpha_3}, f(g(c), g(b))^{\alpha_4}, g(a)^{\alpha_5}).$$

After trying to match all subterms $f(a, a)$, $f(a, c)$, $f(b, g(b))$, $f(g(c), g(b))$ and $g(a)$ by p_1, p_2 and p_3 , the following compact bipartite graph is built:

The current compact bipartite graph is represented by three bitsets. The first one: 11000, means that the corresponding pattern p_1 matches only the two first subterms: $f(a, a)$ and $f(a, c)$, and similarly for the other ones.

In order to normalise the subject, a rule has to be selected, say the first one: $u_{AC}(f(a, x), f(y, g(b))) \rightarrow r_1$. The bipartite graph that should have been created by the general method can be easily constructed by extracting edges that are joining p_1 and p_2 to subject subterms. This is done by selecting two bitsets: 11000 and 00110, which gives the following “classic” bipartite graph:

This bipartite graph has four solutions: $\{x \mapsto a, y \mapsto b\}$, $\{x \mapsto a, y \mapsto g(c)\}$, $\{x \mapsto c, y \mapsto b\}$ and $\{x \mapsto c, y \mapsto g(c)\}$. If the selected rule has some conditions not compatible with those four substitutions, another rule has to be selected (here the second one): $u_{AC}(z, f(a, x), g(a)) \rightarrow r_2$.

The new associated bipartite graph is built with only two selected bitsets: 11000 and 00001.

Note that common syntactic subterms are matched only once, even if they appear in several rules. Moreover, there is no need to flatten the subject because it is supposed to be in canonical form by construction and we will see in 4.5 how to maintain this condition.

To handle patterns of class C_3 , a hierarchy of bipartite graphs has to be built. Thanks to the construction of the class C_3 , the hierarchy is limited to only two levels of bipartite graphs. For each edge of the graph that joins two AC function symbols, an AC subproblem is attached. This hierarchy is represented by a matrix of bipartite graphs where rows correspond to subject subterms, and columns correspond to patterns p_i . Before selecting another rule (from the compact bipartite graph), the two-level-hierarchy of bipartite graphs has to be solved.

4.4 Reduced term construction

Before constructing the reduced term, a substitution which encodes a solution to the matching problem has to be built. In the syntactic case, this substitution is easily built during the matching phase, since there is at most one solution, so static variables (of the implementation language) can be used to store instantiations of variables occurring in the pattern. In the AC case, the problem is more complex. There may be several different instantiations for each variable (as a result of permutations). Combinations of variables occurring in the two possible levels can produce a number of solutions exponential in the subject size. It is not realistic to store all those substitutions in the bipartite graph in order to use them for constructing the reduced term.

Furthermore, all this work is not necessary when the first selected rule is applied because all pre-computed substitutions are deleted and a lot of work is wasted. Our approach consists in building the substitution only when a solution is found. Each solution stores the selected rule and a list of subject subterms (s_i) matched by pattern subterms (p_i). All pattern subterms are known at compile time, positions of their variables are static and can be pre-computed. It is not difficult to find a corresponding position in a given subject subterm and to extract the variable instantiation.

In the last given example, $u_{AC}(z, f(a, x), g(a)) \rightarrow r_2$ is the selected rule and the first solution of the bipartite graph indicates that $f(a, x)$ matches $f(a, a)$. The static position of x (ε.2, second argument of $f(a, x)$) is sufficient to find its instantiation (a , second argument of $f(a, a)$).

The pattern structure restriction is such that only one variable may appear directly under each AC function symbol, except for patterns of the form $u_{AC}(x, y)$ for which a special algorithm is used. To instantiate a single variable, there is no need to construct a Diophantine equational system: all unmatched subject subterms are captured by this variable. For completeness of AC rewriting, extension variables have to be added. Those variables store the context and allow to do rewrite steps in subterms.

In the syntactic case, all variable instantiations are irreducible by construction. Nested function calls are such that before a match phase, each subterm is in normal form wrt. the rewrite rule system. This is no longer the case in AC rewriting: for instance, in our previous example, the variable z can be instantiated by $u_{AC}(f(a, c), f(b, g(b)), f(g(c), g(b)))$ which is reducible by the first rule $u_{AC}(f(a, x), f(y, g(b))) \rightarrow r_1$. So instantiations of variables that appear under an AC function symbol have to be normalised before using them to build the right-hand side.

Once a substitution is found, the right-hand side is built as it is done in the syntactic case: subterms of the left-hand side can be re-used to build the reduced term and non-linear right-hand sides share their common subterms.

4.5 Maintaining canonical forms

The compact bipartite graph construction (matching phase) supposes that both pattern and subject are in canonical form. Each subterm is also in canonical form and must be irreducible wrt. the rewrite rule system. A basic idea could be to flatten and compute the canonical form after each right-hand side construction, but the cost of this approach is very high. Note that all subterms that instantiate a variable are, by construction, in canonical form. It is not necessary to re-compute their canonical form more than once. Our approach consists in maintaining this canonical form during the reduced term construction. Whenever a new term t is added as a subterm of $s = u_{AC}(s_1^{\alpha_1}, \dots, s_n^{\alpha_n})$, if an equivalent subterm s_i already exists, its multiplicity is incremented, else, the subterm t (which is in normal form by construction) is inserted in the list $s_1^{\alpha_1}, \dots, s_n^{\alpha_n}$ at a position compatible with the chosen ordering. If t has the same AC top symbol u_{AC} , a flattening step is done and the two subterm lists are merged with a merge sort algorithm.

Definition 5 Let mcf be defined for two terms $s = u_{AC}(s_1^{\alpha_1}, \dots, s_n^{\alpha_n})$ and $t = v(t_1^{\beta_1}, \dots, t_m^{\beta_m})$ in canonical form, as follows:

- case $u_{AC} \neq v$ (s and t have different top symbol)
 - if there exists i in $[1..n]$ such that $s_i = t$ the multiplicity α_i is incremented:
 $\text{mcf}(u_{AC}(s_1^{\alpha_1}, \dots, s_n^{\alpha_n}), t) = u_{AC}(s_1^{\alpha_1}, \dots, s_i^{\alpha_i+1}, \dots, s_n^{\alpha_n})$
 - else, there exists i in $[1..n]$ such that $\forall j \leq i, s_j > t$ and $\forall j > i, t > s_j$:
 $\text{mcf}(u_{AC}(s_1^{\alpha_1}, \dots, s_n^{\alpha_n}), t) = u_{AC}(s_1^{\alpha_1}, \dots, s_i^{\alpha_i}, t, s_{i+1}^{\alpha_{i+1}}, \dots, s_n^{\alpha_n})$

- case $u_{AC} = v$ (s and t have same top symbol)

– $\text{mcf}(u_{AC}(s_1^{\alpha_1}, \dots, s_n^{\alpha_n}), t) = u_{AC}(u_1^{\gamma_1}, \dots, u_k^{\gamma_k})$ such that $(u_1^{\gamma_1}, \dots, u_k^{\gamma_k})$ is the merged sort (without multiple occurrences) of $(s_1^{\alpha_1}, \dots, s_n^{\alpha_n})$ and $(t_1^{\beta_1}, \dots, t_m^{\beta_m})$

From the definition of mcf , it is easy to get:

Proposition 1 *Let $s = u_{AC}(s_1^{\alpha_1}, \dots, s_n^{\alpha_n})$ and t be two terms in canonical form. The function mcf applied to s and t returns the canonical form of $u_{AC}(s_1^{\alpha_1}, \dots, s_n^{\alpha_n}, t)$.*

As a consequence, the bottom-up construction of a term, using the mcf function, ensures that the resulting term is in canonical form.

5 Experimental results

To illustrate the power of our compilation techniques, let us consider three examples that make heavy use of AC normalisation. Those examples have been compiled with the new ELAN compiler, which is implemented with approximately 6,000 lines of Java. A runtime library support has been implemented in C to support basic term and AC matching operations. This library contains more than 6,000 lines.

We compared our implementation with three AC rewriting systems: OBJ [7], Maude [3] and of course the ELAN interpreter. The number of applied rewrite rules is difficult to compare because it depends on the internal strategy chosen and on the method used for counting applied rules¹. Two different architectures were used to run benchmarks: a Pentium Pro 200 Linux (Intel) and a Sun Ultra-Enterprise Solaris 5.5.1 (Sparc).

5.1 A propositional prover

This first example implements the two basic AC operators *and*, *xor*, and four syntactic rules that transform *not*, *implies*, *or* and *if* functions into nested calls of *xor* and *and*. The rewrite system is defined by the following rules:

$\text{and}(x, \top)$	$\rightarrow x$
$\text{and}(x, \perp)$	$\rightarrow \perp$
$\text{and}(x, x)$	$\rightarrow x$
$\text{and}(x, \text{xor}(y, z))$	$\rightarrow \text{xor}(\text{and}(x, y), \text{and}(x, z))$
$\text{xor}(x, \perp)$	$\rightarrow x$
$\text{xor}(x, x)$	$\rightarrow \perp$
$\text{implies}(x, y)$	$\rightarrow \text{not}(\text{xor}(x, \text{and}(x, y)))$
$\text{not}(x)$	$\rightarrow \text{xor}(x, \top)$
$\text{or}(x, y)$	$\rightarrow \text{xor}(\text{and}(x, y), \text{xor}(x, y))$
$\text{if}(x, y)$	$\rightarrow \text{not}(\text{xor}(x, y))$

The computation consists in normalising a huge term and comparing the result to $\top$. Results are given in the following table:

Prop		Sparc	Intel
	rwr	sec	sec
Maude	12,280	-	0.57
OBJ	12,837	766	-
ELAN interpreter		>24h	-
ELAN compiler	12,853	0.41	0.45

¹In ELAN, the term construction (when the match fails) is not counted as an applied rule.

5.2 A score counting problem

The second example implements a score counting problem. Suppose you want to play darts. The dartboard is divided by wires into 20 equals segments, each numbered from 1 to 20 and contains two distinct centres: the 25 ring and the Bull's-eye (50). Some places count double (2, 4, ..., 40) or triple (3, 6, ..., 60). Rules are simple: to begin a play, you need to shoot the Bull's-eye or a double, then you have two darts. The problem consists in listing all possible scores.

The following rewrite rules describe the game rules:

<i>singles</i>	$\rightarrow$	$set(1) \vee \dots \vee set(20)$
<i>doubles</i>	$\rightarrow$	$singles * set(2)$
<i>triples</i>	$\rightarrow$	$singles * set(3)$
<i>turn</i>	$\rightarrow$	$singles \vee doubles \vee triples \vee set(0) \vee set(25) \vee set(50)$
<i>finish</i>	$\rightarrow$	$(set(50) \vee doubles) + turn + turn$

The rule $turn \rightarrow \dots$ means that with one dart, a single, a double or a triple can be hit. Bad players can miss the target (0 point) and lucky players can hit a centre (25 or 50 points). The problem consists in evaluating *finish*.

Rules for score counting are given in the following rewrite system where $\vee$, $+$ and $*$ are AC functions symbols. x , y , i and j are variables.

$x \vee \emptyset$	$\rightarrow$	x
$x \vee x$	$\rightarrow$	x
$x + \emptyset$	$\rightarrow$	x
$set(i) + set(j)$	$\rightarrow$	$set(i + j)$
$x + (y \vee set(i))$	$\rightarrow$	$(x + y) \vee (x + set(i))$
$x * \emptyset$	$\rightarrow$	x
$set(i) * set(j)$	$\rightarrow$	$set(i * j)$
$x * (y \vee set(i))$	$\rightarrow$	$(x * y) \vee (x * set(i))$

Results are given in the following table:

Dart		Sparc	Intel
	rwr	sec	sec
Maude	219,894	-	1.96
OBJ	2,435,206	4,877	-
ELAN interpreter		>24h	-
ELAN compiler	124,195	1.03	0.76

5.3 Bool 3

This last example implements computation in a 3-valued logic *bool3*. The rewrite system is defined by the following rules:

$x + 0$	$\rightarrow$	x
$x + x + x$	$\rightarrow$	0
$x * 0$	$\rightarrow$	0
$x * 1$	$\rightarrow$	x
$x * x * x$	$\rightarrow$	x
$(x + y) * z$	$\rightarrow$	$(x * z) + (y * z)$
2	$\rightarrow$	$1 + 1$
$and(x, y)$	$\rightarrow$	$(x * x * y * y) + (2 * x * x * y) + (2 * x * y * y) + (2 * x * y)$
$or(x, y)$	$\rightarrow$	$(2 * x * x * y * y) + (x * x * y) + (x * y * y) + (x * y) + (x + y)$
$not(x)$	$\rightarrow$	$(2 * x) + 1$

The problem consists in normalising $and(a_1, \dots, a_8)$ and $not(or(not(a_1), \dots, not(a_8)))$ to prove their equality.

Results are given in the following table:

Bool3		Sparc	Intel
	rwr	sec	sec
Maude	15,127	-	1.17
OBJ		>24h	-
ELAN interpreter		>24h	-
ELAN compiler	11,119	0.43	0.48

6 Concluding Remarks

In this paper, we have proposed a new approach to compile AC rewrite rule systems in an efficient way, thanks to the design of optimised algorithms to solve AC matching problems in the case of several specific classes of patterns.

We have introduced a new data structure, called Compact Bipartite Graph, and used it to compile many-to-one AC matching. A compact bipartite graph encodes, in only one data structure, all matching problems relative to a set of rules.

Many techniques implemented in the previous compiler [15] have been extended and re-used in the current implementation: an efficient syntactic many-to-one matching, shared term data structures, and the re-use of parts of the left-hand sides to construct the reduced term.

Several examples have been implemented and results are promising. We feel that the method can yet be improved by designing a new algorithm for matching, optimised for generating a single solution to matching problems.

Further work consists in extending presented techniques to other theories (A, ACI, AC1, AC0) and breaking the C_3 class restriction to extend algorithms to $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Acknowledgements

We sincerely thank Peter Borovanský, Claude Kirchner, Christophe Ringeissen and Laurent Vigneron for helpful discussions and comments.

References

- [1] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1 & 2):203–216, Apr. 1987.
- [2] J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, Feb. 1993.
- [3] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), Sept. 1996. Electronic Notes in Theoretical Computer Science.
- [4] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and implementation of an algebraic programming language. In J. Gutknecht, editor, *Programming Languages and System Architectures PLSA’94*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer-Verlag, Mar. 1994.
- [5] S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.
- [6] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, Mar. 1995.
- [7] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégreis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

- [8] A. Gräf. Left-to-right tree pattern matching. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 323–334. Springer-Verlag, Apr. 1991.
- [9] P. Graph. Term indexing. volume 1053 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
- [10] J.-M. Hullot. *Compilation de Formes Canoniques dans les Théories équationnelles*. Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France), 1980.
- [11] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.
- [12] P. Klint. The ASF+SDF Meta-environment User’s Guide. Technical report, CWI, 1993.
- [13] B. L., C. T., and R. I. V. Associative-commutative discrimination nets. *Lecture Notes in Computer Science*, 668:61–74, 1993.
- [14] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.
- [15] M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA’96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag.