Compilation of Associative-Commutative Normalisation with Strategies in ELAN (Full version)

Pierre-Etienne Moreau and Hélène Kirchner
CRIN-CNRS & INRIA-Lorraine
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France

Abstract

We address the problem of term normalisation modulo associative-commutative (AC) theories with strategies, and describe several techniques for compiling many-to-one AC matching and reduced term construction. The proposed matching method is based on the construction of compact bipartite graphs, and is designed for working very efficiently on specific classes of AC patterns. General patterns are handled through a program transformation process. To improve efficiency of term normalisation modulo AC, we propose a few solutions to share parts of terms thanks to a suitable term representation, avoid useless cases and failure in matching, take advantage of deterministic computations whenever possible, thanks to an eager matching algorithm easily combinable with the general one. Our experimental results in the system ELAN provide strong evidence that compilation of many-to-one AC normalisation using the combination of these several techniques is useful for improving the performance of algebraic programming languages.

1 Introduction

In the area of formal specifications, rewriting techniques have been developed for two main applications: prototyping algebraic specifications of user-defined data types and theorem proving related to program verification. Through various implementations and experiments in automated theorem proving, for instance with provers like RRL, REDUX, OTTER, REVEAL and EQP, it has been demonstrated that simplification of formulas by rewriting is indeed an effective way of pruning the search space. On the other hand, with the emergence of equationally specified abstract data types in the late 1970’s, term rewriting has gained popularity also as a bridge between programming language theory and program verification. Several specification languages or programming environments, such as LARCH, OBJ, ASF+SDF, RAP, MAUDE, ELAN, to cite a few, are using rewriting as their basic evaluation mechanism.

Mathematic and algebraic structures often involve 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. An important case in practice is the case of associative-commutative theories, but their implementation requires specific efficient techniques.

Our own interest in compilation of AC-normalisation comes from the development of the ELAN system [KKV95]. ELAN is an environment for prototyping and combining different deduction systems described using rewrite rules and strategies. Non-deterministic strategies
used for controlling rewriting is one of the main originalities of ELAN compared to other algebraic specification systems based on rewriting. An ELAN program is composed of a signature part describing operators with their types, a set of rules and a set of strategies. A strategy is a way to describe which computations the user is interested in, and specifies where a given rule should be applied in the term to be reduced. We describe informally here the evaluation mechanism and how it deals with rewrite rules and strategies.

In ELAN, rules are labelled conditional rewrite rules with local variable assignments

\[ [l]: l \rightarrow r \text{ if } v \text{ where } y := (S)u \]

where \( l \) is the label, \( l \) and \( r \) the respective left and right-hand sides, \( v \) the condition and \( y := (S)u \) a local assignment, giving to the local variable \( y \) the results of the strategy \( S \) applied to the term \( u \). Any sequence of \text{where} and \text{if} is allowed but their order is relevant for the evaluation.

For applying such a rule on a term \( t \), say at top position, first \( t \) is matched against \( t \), then the expressions introduced by \text{where} and \text{if} are instantiated with the matching substitution and evaluated in order. Instantiations of local variables (such as \( y \)) after \text{where} extend the matching substitution. When every condition is satisfied, the replacement by the instantiated right-hand side is performed.

Instantiations of local variables after \text{where} invoke ELAN built-in strategies. These strategies are regular expressions built on the alphabet of rule labels and a few primitive constructors “\( \cdot \)”, \text{dc, dk, id, fail, repeat, iterate} (others can be found in [BKK+97]). Two strategies can be concatenated by the symbol “\( ; \)”, i.e. the second strategy is applied on all results of the first one. Non-determinism is handled with two operators: \text{dc} standing for don’t-care-choose and \text{dk} standing for don’t-know-choose. For rewrite rules \( t_1, \ldots, t_n \), the strategy \( \text{dc}(t_1, \ldots, t_n) \) over these rules returns results of one non-failing and un-deterministically chosen rule among the \( t_i \)'s. On the contrary, for the strategy \( \text{dk}(t_1, \ldots, t_n) \), all possible results are computed and returned. This is implemented by backtracking on all rules \( t_1, \ldots, t_n \). \text{dc}(S_1, \ldots, S_n) \) over strategies \( S_i \) un-deterministically chooses one of the non-failing strategies \( S_i \), i.e. whose application gives a non-empty set of results, and those results are also results of \( \text{dc}(S_1, \ldots, S_n) \). If all sub-strategies fail, then it fails too. \( \text{dk}(S_1, \ldots, S_n) \) returns all results of all sub-strategies \( S_i \). The identity strategy \text{id} does not change a term, while the strategy \text{fail} always fails, and never gives any result. The strategy \text{repeat}(S) \) applies the strategy \( S \) on a term \( t \) until it fails and returns the last unfailing result \( S^h(t) \). Beyond these regular expressions, ELAN gives the possibility to the user to define recursive and parameterised strategies with rewrite rules [BKK96, BKK97a, BKK97].

A first ELAN compiler was designed and presented in [Vit96], but did not handle AC function symbols nor user-defined strategies. However remarkable performances obtained by this compiler were a strong encouragement to design an extension to handle AC theories and more ambitiously any theory which is a combination of associativity, commutativity, idempotency and/or identity element.

In Section 2, the basic definitions are given, known compilation techniques for standard normalisation are recalled, and AC-matching main ingredients are presented. In Section 3, we explain on which problems we concentrated our effort and provide contributions during the design of the ELAN compiler. Then Section 4 presents the different techniques that are used in our compiler, for the matching phase, the substitution and result term construction, the compilation of strategies. In Section 5, we evaluate the performance of our compiler and compare it with other existing systems. Future work is proposed in Section 6.
2 Preliminary concepts and techniques

2.1 Notations

\( \mathcal{T}(\mathcal{F}, \mathcal{X}) \) is the set of terms 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). The set of variables occurring in a term \( t \) is denoted by \( \text{Var}(t) \). If \( \text{Var}(t) \) is not empty, \( t \) is called an open term.

Given a binary function symbol \( f \), let \( \text{AC} \) be the set of associativity and commutativity axioms

\[
    f(x, f(y, z)) = f(f(x, y), z) \quad \text{and} \quad f(x, y) = f(y, x).
\]

The notation \( f_{\text{AC}} \) emphasises that \( f \) is such an AC function symbol that satisfies these two axioms. We write \( s =_{\text{AC}} t \) to indicate that the terms \( s \) and \( t \) are equivalent modulo associativity and commutativity. \( \mathcal{F}_g \) 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 contains only free function symbols and possibly variables. Positions in a term are represented as sequences of integers. The empty sequence \( e \) denotes the top-most position. The subterm of \( t \) at position \( \omega \) is denoted \( t_{\omega} \). 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 \( \sigma = \{ y_1 \mapsto t_1, \ldots, y_k \mapsto t_k \} \). It uniquely extends to an endomorphism of \( \mathcal{T}(\mathcal{F}, \mathcal{X}) \). A conditional rewrite rule is a triple of terms denoted \( l \to r \) if \( c \) 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. In ELAN, rules also have where conditions in which a variable \( y \) occurring in \( l, r \) or \( c \) may be instantiated, through a local affection where \( y := (S)u \), by respectively all results of evaluation of the strategy \( S \) applied to \( u \).

We assume the reader to be familiar with basic ideas of term rewriting and AC theories. Full definitions can be found for instance in [PS81, JK86].

2.2 Compilation techniques for rewriting

We started from the work presented in [Vit96] whose main points are recalled below.

Many-to-one normalisation. When all rules are syntactic, many-to-one normalisation consists first 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 \( \text{fun}_f \) of \( n \) arguments. This function when applied to the terms \( t_1, \ldots, t_n \), returns the reduced form of the term \( f(t_1, \ldots, t_n) \), according to the leftmost innermost normalisation strategy. Since all rewrite rules are known at compile time, matching their left-hand sides to an unknown input term can be pre-compiled using an existing algorithm such as [Gra96, Grä91]. 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 \( \text{fun}_f \) functions.

Reusing parts of terms. The main idea, coming from [DFG+94], is to reuse parts of left-hand sides of rules to construct the right-hand sides when applying a rewrite rule. 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 [Vit96].
Non-deterministic computations and strategies. Rewriting is non-deterministic in the sense that there may exist several reductions issued from an initial term and leading to different results. A strategy is a function which, when applied to an initial term, returns a set of possible results. The strategy fails if the set is empty. A strategy that returns at most one solution is said deterministic. Otherwise it is said non-deterministic.

When a strategy is applied to a term, a non-deterministic computation looks for all the resulting terms and this search is obviously implemented by backtracking and pruning some branches of the derivation tree. One main difficulty in compiling strategies is to handle search in non-deterministic computations, which leads to use set choice points and jump operations.

2.3 AC-matching

AC-matching has already been extensively studied, for instance in [Hul80, BKN87, KL91, LTV93, LM94, Eke95].

A term \( t \) is said to AC-match another term \( s \) if there exists a substitution \( \sigma \) such that \( teA Cs \).

For example, let us assume that \( f(x, f(g(a, y), z)) \) is the pattern and \( f(a, f(g(a, b), c)) \) is the subject. If none of the symbols is AC, there is a substitution \( \{ x \mapsto a, y \mapsto b, z \mapsto c \} \) such that the pattern matches the subject. If \( f \) is AC, the substitution \( \{ x \mapsto c, y \mapsto b, z \mapsto a \} \) is another solution to the matching problem.

AC-matching is NP-complete [BKN87] but can be done in polynomial time for linear terms. So it is usually done in two phases: first find an AC-match for linearised versions, and then check the consistency of each variable’s instances.

2.3.1 Flattened form

Usually AC function symbols are considered as vrayadic, which means that they accept any finite number of arguments. This allows to define flattened terms. Terms are flattened using the rewrite rule \( fAC(X, fAC(Y), Z) \rightarrow fAC(X, Y, Z) \) where \( fAC \) is AC, \( X, Y, Z \) are sequences of terms whose respective lengths \( |X|, |Y|, |Z| \) satisfy \( |X| + |Z| \geq 1 \) and \( |Y| \geq 2 \). Thanks to associativity and commutativity, it is possible to work modulo the permutation of arguments of an AC function symbol. The permutation congruence on flattened terms is denoted by \( \sim \). It is also often convenient to group together identical arguments. So any argument of an AC function symbol may be given a multiplicity which is a positive integer, denoted by a superscript. A flattened term is said almost linear if the term obtained by forgetting the multiplicities of subterms is linear. The AC-matching problem can essentially be reduced to ordinary matching up to permutation of AC-symbols, provided terms are flattened first.

2.3.2 Canonical form

Based on this flattened form, a canonical form can be defined for any term \( t \), as a representative \( CF(t) \) of its AC congruence class. In order to recursively compute canonical forms, we assume given a total ordering \( > \) on the set of function symbols \( F \) and variables, and extend it to terms in canonical form as follows. For terms that are just variable or constant symbols, \( CF \) 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 [Hul80]. For more details, see [Eke95].

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

2.3.3 Discrimination nets

The AC-pattern matching problem is the following: given a set of terms \( P = \{p_1, \ldots, p_n\} \) called patterns, and a term \( s \), called subject, find one (or more) patterns in \( P \) that AC-matches \( s \). Patterns and subject are assumed to be in canonical form. Efficient many-to-one matching algorithms (both in syntactic case and AC-theories) use matching trees called discrimination nets and the corresponding tree automata. In the case of AC theories, the decomposition of matching problems gives rise to a hierarchically structured collection of standard discrimination nets, called an AC-discrimination net [LT93].

AC-matching of \( P = \{p_1, \ldots, p_n\} \) to \( s \) is characterised by two conditions: first, a (non-AC) matching condition of the top layers \( P = \{p_1, \ldots, p_n\} \) (where \( p_i \) is obtained from \( p_i \) by removing subterms below the first AC-symbol in each branch) to \( s \). Second, several AC-matching problems corresponding to \( p_{1,\omega} \) and \( s_{\omega} \), where \( \omega \) is a position of an AC-function symbol in \( p_i \). This second step leads to bipartite graph matching problems and Diophantine equational systems to encode constraints on variables.

2.3.4 Bipartite graph matching

Assume that \( p_{1,\omega} = f_{AC}(t_1, \ldots, t_m), s_{\omega} = f_{AC}(s_1, \ldots, s_n) \), and for some \( k, 0 \leq k \leq m \), no \( t_1, \ldots, t_k \) is a variable, and all \( t_{k+1}, \ldots, t_m \) are variables. The associated bipartite graph is \( G = (V_1 \cup V_2, E) \) where \( V_1 = \{s_1, \ldots, s_n\} \), \( V_2 = \{t_1, \ldots, t_k\} \), and \( E \) consists of all pairs \((s_i, t_j)\) such that \( t_j \sigma \sim s_i \) for some substitution \( \sigma \). It can easily be seen that if (a) either \( n = m \) or \( n > m > k \), and if (b) there is a maximum bipartite matching [HK73, FM89] of size \( k \) in the bipartite graph \( G \), then \( f_{AC}(t_1, \ldots, t_m) \sigma \sim f_{AC}(s_1, \ldots, s_n) \).

2.3.5 Many-to-one AC-matching

A many-to-one AC-matching algorithm based on techniques presented in [LT93], can be sketched as a succession of different steps as follows. Given a set of patterns \( P \) and a subject \( s \),

1. linearise patterns;
2. convert patterns to their canonical forms;
3. compute the AC-discrimination nets associated to \( P \) and the corresponding matching automata;
4. build the hierarchy of bipartite graphs according to the given subject \( s \) in canonical form;
5. find a set of solutions to the hierarchy of bipartite graphs and construct a Diophantine equational system which encodes the constraints on the remaining unbound variables;
6. solve the Diophantine equational system to get a matching substitution;
7. check consistency of substitutions for non-linear variables.
The three first steps only depend on the set of patterns and thus can be done once for all at compile time for a given set of rewrite rules. The steps 4 to 7 are done at execution time for the given subject.

Starting from this quite general algorithm, our goal was to improve its efficiency by lowering the cost of some steps, such as traversing the levels of the hierarchy of discrimination nets, solving Diophantine equational systems, or checking consistency for non-linear variables. The idea is to apply these costly steps on specific patterns for which they can be designed efficiently, or simply skipped. Generality is retained by a pre-processing of the rewriting system, as explained in Section 4.3.

3 Our contributions

In our design of the ELAN compiler, we were guided by several ideas which led us to work on the following problems.

Term representation. First of all, we wanted to keep the same efficiency on programs with syntactic rules as the first ELAN compiler [Vit96]. In order to generalise some optimisations to AC theories, we had to design a particular term representation to share parts of terms (see section 4.1).

Classes of patterns. Another way to get an efficient execution of rewrite programs is to design specific techniques for frequently used patterns. A natural idea is to examine which patterns are the most usual ones in ELAN programs, handle these cases first and use a more powerful, but in general more expensive, method only when needed. This led us to design many-to-one AC-matching in Section 4.4 for a few classes of patterns described in Section 4.2. For the general case, two approaches were considered: either to reuse some existing AC-matching algorithm as [Eke95, CA95], or to transform ELAN programs so that they fall into the previous classes and combine our many-to-one approach with a general one-to-one algorithm [Eke95]. We chose the second one and give in Section 4.3 the transformation which is possible thanks to the where construction in ELAN rules (see section 4.3).

Conditional rules. An additional difficulty comes with conditional rewrite rules in AC theories. 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 AC matching problem for which the condition is not satisfied, another solution has to be found and extracted. If no solution is remaining, the selected rule is no longer a good candidate and another one is selected. Conditional rewriting requires AC 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 AC matching problem.

Deterministic computations. When rules are not conditional and used inside a deterministic strategy, only the first found match is needed to apply a rewrite step. Taking advantage of this remark to improve efficiency of matching leads to the design of an eager matching described in Section 4.4.
Avoid useless cases and failure. In a normalisation process using a rewrite program, a huge number of rewrite rules has to be considered, but a lot of matching attempts are failure. Avoiding unsuccessful search by detecting failure and useless construction as early as possible is a general philosophy of the design, which appears in the construction of substitutions (see Section 4.5) and of the result (see Section 4.6).

Strategies. ELAN provides in its strategy language a few primitives for combining rewrite rules: iteration, don't-know and don't-care choices. Compilation of strategies built on these primitives requires the use of set choice points and jump operations (see Section 4.7).

The difficulty of this work (and also its originality) was not so much to solve each technical point (for which some solutions were often already proposed in the literature) but rather to combine all these solutions together to achieve an efficient implementation.

4 Compilation techniques

The techniques adopted for the ELAN compiler are based on the fact that all rewrite rules are known at compilation time.

4.1 Term structure

A specific data structure for terms has been adopted, assuming that terms are maintained in some canonical form. In implementations of algebraic programming language, 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 [Chr93]. Those flatterms, represented by a doubly-linked list data structure, yield in practice simpler and faster traversal algorithms than the conventional tree representation. But in the flatterm representation, subterms stored in the doubly-linked list cannot be shared. In our compiler, 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. This data structure can be easily extended to represent terms in canonical form. The representation of terms $f(g(b), a, g(b))$ and $f_{AC}(g(b), a, g(b))$ is illustrated in the two first pictures of the following figure:

![Diagram of terms](image)

The third picture illustrates how multiple copies of equivalent terms are represented by storing their multiplicity in list-cells.
4.2 Classes of patterns

As already said, 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 some pattern classes. Specific compilation techniques have been developed for those classes of patterns. Solving matching problems involving other patterns is explained in the next section 4.3.

All terms in the pattern classes are assumed to be in canonical form and almost linear. The pattern classes $C_0, C_1, C_2$ contain respectively terms with no AC function symbol, at most one and at most two levels of AC function symbols.

**Definition 1** Let $\mathcal{F}_f$ be 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 $C_0$ consists of linear terms $t \in T(\mathcal{F}_f, \mathcal{X}) \setminus \mathcal{X}$ in canonical form.
- The pattern class $C_1$ is the smallest set of almost linear terms in canonical form that contains $C_0$, all terms $t$ of the form $t = f_{AC}(x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}, t_1, \ldots, t_n)$, with $f_{AC} \in \mathcal{F}_{AC}$, $0 \leq n, t_1, \ldots, t_n \in C_0, x_1, \ldots, x_m \in \mathcal{X}$, $\alpha_1, \ldots, \alpha_m \geq 0$, and all terms $t$ of the form $f(t_1, \ldots, t_n)$, with $f \in \mathcal{F}_f, t_1, \ldots, t_n \in C_1 \cup \mathcal{X}$.
- The pattern class $C_2$ is the smallest set of almost linear terms in canonical form that contains $C_1$, all terms $t$ of the form $t = f_{AC}(x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}, t_1, \ldots, t_n)$, with $f_{AC} \in \mathcal{F}_{AC}$, $0 \leq n, t_1, \ldots, t_n \in C_1, x_1, \ldots, x_m \in \mathcal{X}$, $\alpha_1, \ldots, \alpha_m \geq 0$, and all terms $t$ of the form $f(t_1, \ldots, t_n)$, with $f \in \mathcal{F}_f, t_1, \ldots, t_n \in C_2 \cup \mathcal{X}$.

For AC matching, four subclasses of patterns are distinguished in $C_1$ and $C_2$, corresponding to the number of variables immediately below an AC function symbol: for $i=1\ldots 2$,

- $C_i^0$ contains all terms of the form $t = f_{AC}(t_1, \ldots, t_n)$;
- $C_i^1$ contains all terms of the form $t = f_{AC}(x, t_1, \ldots, t_n)$;
- $C_i^2$ contains all terms of the form $t = f_{AC}(x_1^{\alpha_1}, x_2, t_1, \ldots, t_n)$, with $\alpha_1 \geq 1$;
- and $C_i^3$ contains all other terms of the general form $t = f_{AC}(x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}, t_1, \ldots, t_n)$.

Consider the term $f_{AC}(x_1, k[x_2, a])$ where $x_1, x_2$ are distinct variables. It contains only one AC function symbol $f_{AC}$ and one variable $x_2$ immediately under $f_{AC}$. This term is in the class $C_1$. By adding a free function symbol $l$ of arity 3 at the top position, a variable $x_3$, and a term $g_{AC}(x_3, x_5)$ of the class $C_1$, the term: $l(f_{AC}(x_1, k[x_2, a]), x_3, g_{AC}(x_3, x_5))$ is again in $C_1$. By adding a second level of AC function symbol $h_{AC}$ and a variable $x_6$, the term $h_{AC}(x_6, l(f_{AC}(x_1, k[x_2, a]), x_3, g_{AC}(x_3, x_5)))$ becomes a member of the class $C_2$.

It should be emphasised that for completeness of AC rewriting, extension variables have to be added. Those variables store the context and allow to do rewrite steps in subterms. Adding extension variables amounts to consider additional rule patterns of the form $f_{AC}(x, l_1, \ldots, l_n)$ for each rule with a left-hand side of the form $l = f_{AC}(l_1, \ldots, l_n)$. This clearly shows the usefulness of our pattern classes.

The AC-matching techniques described in the following Section 4.4 are restricted to the class of patterns with at most two levels of AC function symbols. In order to handle general patterns, we take advantage of the **where** construction in ELAN rewrite rules, as follows.
4.3 Handling other patterns

In the new version of ELAN, the local affectation mechanism has been extended for pattern assignments. Rules of the following form are allowed:

\[ [l] : l \rightarrow r \text{ if } v \text{ where } p := (S)u \]

where \( p \) is an open term not necessarily reduced to a single variable. If the open pattern \( p \) matches the result of the strategy \( S \) applied to the term \( u \), its variables are instantiated and extend the main matching substitution, as before. When computing such local affectations, only one-to-one matching problems occur. If the pattern \( p \) contains AC function symbols, a general one-to-one AC-matching procedure [Elke95] can be used. Thus, the term \( p \) is no longer restricted to be in \( C_2 \).

In order to handle rules whose patterns are not in \( C_2 \), a program transformation is applied. It transforms those rules into equivalent ones whose left-hand sides are in the class \( C_2 \). Extended local affectations are added to preserve the semantics.

- Let \( l = f_{AC}(x_1^{a_1}, \ldots, x_m^{a_m}, t_1, \ldots, t_k, t_{k+1}, \ldots, t_n) \) with \( x_1, \ldots, x_m \in \mathcal{X}, t_1, \ldots, t_k \in C_1 \) and \( t_{k+1}, \ldots, t_n \notin C_1 \), where \( k < n \). If \( l' = f_{AC}(x_1^{a_1}, \ldots, x_m^{a_m}, t_1, \ldots, t_k, y) \), the rule
  \[ l' \rightarrow r \text{ where } f_{AC}(t_{k+1}, \ldots, t_n) := ()y \]
  is equivalent to \( l \rightarrow r \). Let us remind that when computing the local assignment, \( y \) is instantiated by the matching substitution from \( l' \) to the ground subject.

- Let \( l = f(t_1, \ldots, t_n) \) with some \( t_i \notin C_2 \). Let \( \lambda \) be an abstraction function that replaces each maximal non-variable subterm of \( l \) which is not in \( C_2 \), say \( u_j \), by a new variable \( x_j \), for \( j = 1, \ldots, k \). Let consider \( l' = \lambda(f(t_1, \ldots, t_n)) \), and the rewrite rule
  \[ l' \rightarrow r \text{ where } u_1 := ()x_1 \]
  \[ \vdots \]
  \[ \text{where } u_k := ()x_k \]
  which is equivalent to \( l \rightarrow r \).

In the worst case, many-to-one AC-matching is not used and the program transformation builds a rewrite rule system where AC problems are solved with a one-to-one AC-matching procedure in \text{where} parts, helped by a full indexing for the topmost free function symbol layer. We get back a frequently implemented matching technique, used in MAUDE for instance.

4.4 Many-to-one AC matching

Our AC-matching algorithm is derived from the general algorithm presented in Section 2. The differences are:

1. Patterns are linearised according to the pattern restrictions, since a rule with a non-linear left-hand side can be systematically transformed into a conditional rule with a linear left-hand side and a condition of the form \( x = y \). This linearisation process is automatically performed in the front-end of the ELAN compiler.

2. In [Chr93, BCR93], 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. Here, the automaton is directly coded in C. No intermediate abstract data structure is built to represent the automaton.
3. Thanks to the restriction put on patterns, the hierarchy of bipartite graphs has at most two levels. Thus, the construction can be efficiently compiled with no recursion.

4. We use a compact representation of bipartite graphs, which encodes, in only one data structure, all matching problems relative to the given set of rewrite rules.

5. No Diophantine equational system is generated when there is at most one or two variables, with (restricted) multiplicity, under an AC function symbol in the patterns. Instantiating these variables is done at the substitution construction step in a very simple and efficient way.

The method presented here is related to discrimination nets [Chr93] and AC discrimination nets [LTV93]. AC discrimination nets do not have any limitation on pattern structure. 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. We now develop two original aspects of our AC matching algorithm, which are the construction of compact bipartite graphs and their adaptation to perform eager matching.

**Compact Bipartite Graph** Given a set of rewrite rules, the method consists in regrouping subterms with the same AC top function symbol, and to build for this set of subterms a particular bipartite graph called *Compact Bipartite Graph* described below. For a given subject, the compact bipartite graph encodes all matching problems relative to the given set of rewrite rules. Then, 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 subterms are matched only once, even if they appear in several rules.

A compact bipartite graph is built as follows. Given a set of rewrite rules, syntactic subterms \( p_1, \ldots, p_n \) of their left-hand sides with the same AC top function symbol \( f_{AC} \) are grouped together and compiled into a function \( \text{fun}_{\text{AC}}(x) \) that implements the many-to-one syntactic matching described in [Vit96]. Given a subject \( s = f_{AC}(s_1^{\alpha_1}, \ldots, s_t^{\alpha_t}) \), this function returns a bitset for each \( s_i \) which encodes possible matches. The \( f_{AC}^{\beta} \) 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 \( s_i \).

For example, given the two rules, \( f_{AC}(f(a, x), f(y, g(b))) \rightarrow r_1 \), and \( f_{AC}(z, f(a, x'), g(a)) \rightarrow r_2 \), an analysis of subterms with the same AC top function symbol \( f_{AC} \) 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 substitution construction phase described in section 4.5. Let us consider the subject

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

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, for instance the first one: $f_{AC}(f(a,x), f(y,g(b))) \rightarrow r_1$. 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:

\[
\begin{array}{ccc}
1 & 1 & 0 & 0 & 0 \\
& f(a,x) & f(y,g(b)) & g(a) \\
& f(a,a) & f(a,c) & f(b,g(b)) & f(g(c),g(b)) & g(a)
\end{array}
\]

To check if the selected rule can be applied, a maximum bipartite matching has to be found [HK73, FM89]. At this step, the subject subterm multiplicities have to be taken into account. For simplicity, in the following example, we consider multiplicities equal to 1. In this case, the bipartite graph has four solutions: \{x → a, y → b\}, \{x → a, y → g(c)\}, \{x → c, y → b\} and \{x → c, y → 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): $f_{AC}(z, f(a,x), g(a)) \rightarrow r_2$.

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

\[
\begin{array}{ccc}
1 & 1 & 0 & 0 & 0 \\
& f(a,x) & f(y,g(b)) & g(a) \\
& f(a,a) & f(a,c) & f(b,g(b)) & f(g(c),g(b)) & g(a)
\end{array}
\]

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 4.6 how to maintain this condition.

To handle patterns with two levels of AC function symbols, a two-level hierarchy of bipartite graphs has to be built. To each edge of the graph issued from an AC pattern, is attached the corresponding AC subproblem. This hierarchy is represented by a matrix of bipartite graphs where rows correspond to subject subterms, and columns correspond to

11
patterns $p_i$. Before selecting another rule (from the compact bipartite graph), the two-level-hierarchy of bipartite graphs has to be solved.

**Eager matching** An AC matching problem usually has more than one solution. But for applying an unconditional rule in a deterministic strategy, there is no need to compute a set of solutions: the first found match is used to apply the corresponding rule. These rules are called *eager rules*. This leads us to design a new *eager matching* algorithm which avoids building the whole compact bipartite graph before solving it. The idea consists in merging the building and the solving phases to minimise the number of matching attempts on subject subterms.

Let us consider the previous example. With the first method (called the main algorithm), five matching attempts were done to build the compact bipartite graph. Only after this building phase, bipartite graphs could be extracted and solved. However, to find the first suitable solution, it is sufficient to match only the three first subterms. The following (partial) compact bipartite graph is built:

$$
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0
\end{array}
$$

Fortunately, the bipartite graph associated to the rule $f_{AC}(f(a,x), f(y,g(b))) \rightarrow r_1$ is the same as the presented compact bipartite graph. It has two solutions: $\{x \mapsto a, y \mapsto b\}$ and $\{x \mapsto c, y \mapsto b\}$.

A naive approach could be to extract and solve bipartite graphs after each subterms matching attempts. To detect when the compact bipartite graph construction can be stopped, only a satisfiability test is needed to check whether a given bipartite graph has a solution, and this can be performed by a particular automata described in [BCR93]. Let us further remark that only bipartite graphs associated to an eager rule have to be extracted and checked; no check is necessary on a bipartite graph if no modification occurred since the last applied satisfiability test (i.e. no edge has been added). A bipartite graph is said to be modified if at least one edge is added. Let us remind that for each subterm $s_i$, an edge is built between $p_j$ and $s_i$ if $p_j$ matches $s_i$. For a given $s_i$, the set of $p_j$ that match $s_i$ defines bipartite graphs that have to be extracted and checked, namely those which are using a selected $p_j$. No check is needed on remaining bipartite graphs because no update occurred. It is not difficult to check that the eager algorithm cannot miss an applicable eager rule.

A relevant question to ask is how to combine the eager method with the main algorithm? In a compact bipartite graph, patterns of eager and non-eager rules may occur together. If no solution is found before the complete construction of the compact bipartite graph, no gain is brought by the eager algorithm. But the compact bipartite graph can be exploited as previously. In this case, only bipartite graphs associated to non-eager rules have to be extracted and solved.

In practice, eager matching usually reduces the number of matching attempts and there is only a small time overhead when no eager rule is applied. A few results of experiments with eager matching are shown in Section 5. Note however that eager matching is not compatible with ordered rewrite rules, since the eager rule chosen by the eager matching algorithm may not correspond to the first applicable rule in the set of rules ordered by the
programmer. Fortunately we do not assume in ELAN that the ordering in which rules appear in a program is relevant for the semantics.

4.5 Construction of substitutions

Before constructing the reduced term, a substitution which encodes a solution to the matching problem has to be built. At this stage, three problems can be addressed: how to instantiate the remaining variables in patterns? How to optimise the substitution construction? Are instances of variables in normal form?

**Variable instantiation.** Variables that occur in patterns just below an AC-function are not handled in the previously described phases of the compiler. This problem is delayed until the construction of substitutions. When only one or two distinct variables (with multiplicity) appear directly under each AC function symbol, their instantiation does not need to construct a Diophantine equational system. Several cases can be distinguished. Let $i = 1, 2$.

- For $f_{AC}(x_1, t_1, \ldots, t_n)$ in $C_i^1$, once $t_1, \ldots, t_n$ are matched, all the unmatched subject subterms are captured by $x$.

- For $f_{AC}(x_1^{a_1}, x_2, t_1, \ldots, t_n)$ in $C_i^2$, let us first consider the case where $a_1 = 1$. Then once $t_1, \ldots, t_n$ are matched, the remaining subject subterms are partitioned into two non-empty classes in all possible ways. One class is used to build the instance of $x_1$, the other for $x_2$.

  When $a_1 > 1$, once $t_1, \ldots, t_n$ are matched, one tries to find in all possible ways $a_1$ identical remaining subjects to match $x_1$ and then, all the remaining unmatched subject subterms are captured by $x_2$.

- For $f_{AC}(x_1^{a_1}, \ldots, x_m^{a_m}, t_1, \ldots, t_n)$ in $C_i^3$, once $t_1, \ldots, t_n$ are matched, a system of Diophantine equations is solved for computing instances of $x_1, \ldots, x_m$.

**Optimisation of the construction.** In the syntactic case, the matching 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.

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 of the bipartite graph is found. Each solution stores the selected rule and a list of subject subterms ($s_i$) matched by pattern subterms ($p_j$). All pattern subterms are known at compile time, positions of their variables are static and can be pre-computed. From the corresponding position in the subject subterm, the variable instantiation can be extracted.

In the last given example, $f_{AC}(z, f(a, x), g(a)) \rightarrow r_2$ 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 $x$ (e, 2, second argument of $f(a, x)$) is sufficient to find its instantiation: $a$ (second argument of $f(a, a)$).
Normalised substitutions. In the syntactic case, all variable instantiations are irreducible by construction. Nested function calls are such that before a matching phase, each subterm is in normal form w.r.t. 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 $f_{AC}(f(a, c), f(b, g(b)), f(g(c), g(b)))$ which is reducible by the first rule $f_{AC}(f(a, x), f(y, g(b))) \rightarrow r_1$. To ensure that instances of variables that appear immediately under an AC top function symbol are irreducible, they are normalised before using them to build the right-hand side. Moreover, if the considered rule has a non-linear right-hand side, this normalisation substitution step allows reducing the number of further rewrite steps: the irreducible form is computed only once. Without this optimisation, normalisation of identical terms frequently occurs even if a shared data structure is used because flattening can create multiple copies. Thus, in practice, the number of applied rules is significantly reduced.

4.6 Construction of the result

Once a substitution is found, the result term is built. As 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.

Sharing and reusing. It is usually possible to reuse parts of the left-hand side to construct the instantiated right-hand side. At least, instances of variables that occur in the left-hand side can be reused. More details can be found in [Vit96] for the syntactic case. In AC rewriting, when computing the instantiation of a single variable directly under an AC top function symbol, another optimisation is possible. It consists in reusing list-cells of the AC top symbol. Let the rule $f_{AC}(z, f(a, x), g(a)) \rightarrow r_2$ be applied on a subject $s = f_{AC}(f(a, x), g(a), t_1, \ldots, t_n)$ (in practice, $n$ may be greater than 1,000). If no reusing is done, the $f_{AC}$ top symbol is freed. In this case $f_{AC}$ has $n + 2$ subterms, so, $n + 2$ cells are freed. The instance of $z$ is then built: $n$ cells are allocated. Instead of freeing and reallocating $n$ cells to build the instantiation of $z$, the reusing technique only frees the two cells that corresponds to $f(a, x)$ and $g(a)$. Subterms $t_1, \ldots, t_n$ are indeed reused. This optimisation may be done only if the subject is not shared by two or more terms. This 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 [Vit96].

Maintaining canonical forms. The compact bipartite graph construction (and thus the matching phase) supposes that both pattern and subject are in canonical form. Instead of recomputing the canonical form after each right-hand side construction, one can maintain this canonical form during the reduced term construction. Whenever a new term $t$ is added as a subterm of $s = f_{AC}(s_1^{\alpha_1}, \ldots, s_n^{\alpha_n})$, 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 $s_1^{\alpha_1}, \ldots, s_n^{\alpha_n}$ at a position compatible with the chosen ordering. If $t$ has the same AC top symbol $f_{AC}$, a flattening step is done and the two subterm lists are merged with a merge sort algorithm. A precise algorithm is given in [MK97].

4.7 Compilation of strategies

Strategies provide an additional difficulty to the compilation of rewriting. The compiler transforms each strategy $S$ into a C function $\text{str} S$ with one argument. An application of the strategy $S$ to the term $t$ is compiled into a call to $\text{str} S$, whose argument is a pointer to $t$. When the strategy is deterministic, i.e. has at most one result, the function
returns a pointer to the resulting term. Otherwise, backtracking is needed to implement nondeterministic computations. For that, two functions are usually required: the first one, to create a choice point and save the execution environment; the second one, to backtrack to the last created choice point and restore the saved environment. Many languages that offer nondeterministic capabilities provide similar functions: for instance \textit{world+} and \textit{world-} in Claire [CL96], \textit{try} and \textit{retry} in WAM.

Following [Vit96], two flow control functions, \textit{setChoicePoint} and \textit{fail}, have been implemented in assembly language. \textit{setChoicePoint} sets a choice point, and the computation goes on. The \textit{fail} function performs a jump into the last call of \textit{setChoicePoint}. These functions can remind the pair of standard C functions \textit{setjmp} and \textit{longjmp}. However, the \textit{longjmp} can be used only in a function called from the function setting \textit{setjmp}. These two functions do not have such a limitation. Let us illustrate the use of \textit{setChoicePoint} and \textit{fail} on a few compilation schemes of non-deterministic computations in ELAN.

Any computation in ELAN begins with a call to \textit{setChoicePoint} to set an initial choice point and ends with a call to the \textit{fail} function. So coming back to the initial choice point with a failure means that the computation has terminated and all results have been returned.

- \texttt{ck(S_n);} this strategy tries all substrategies and returns all possible results. For each strategy, a choice point is created and the strategy is tried. If there is a failure, the choice point is removed and the next strategy is tried. If the strategy returns some results, the choice point is kept for further backtracking. When the control comes back to the initial choice point, all possible results have been returned.

- \texttt{iterate(S);} iterating a strategy has a simple compilation scheme. In an infinite loop, a choice point is created and an exit from the loop is done. The current term is returned. So applying \texttt{iterate(S)} on a term \(t\) returns \(t\) first (zero application of \(S\)). When coming back with a failure which means that the computation of the \(n\)-th iteration is finished, if the strategy \(S\) can be applied once on the result, a new choice point is created, and \(S(S^n)(t)\) is returned. Else the failure is propagated.

- \texttt{repeat(S);} \texttt{repeat} differs from \texttt{iterate} because only the last result is returned. Each time the strategy \(S\) is applied, the resulting term is saved in a special variable \texttt{lastTerm}. When a fail is executed, the saved term \texttt{lastTerm} is returned.

Compared to implementation of strategies described in [Vit96], in which reference counters were explicitly handled, memory management in the new compiler is done by the Boehm garbage collector [BW88]. This has considerably simplified reference counter handling in backtracking, and some of the compilation schemes. The time spent in the garbage collector is comparable to the overhead due to maintaining reference counters, and performance is sometimes even better.

5 Experimental results

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

- The \texttt{Prop} example implements the two basic AC operators \textit{and}, \textit{xor}, and four syntactic rules that transform \textit{not}, \textit{implies}, \textit{or} and \textit{iff} functions into nested calls of \textit{xor} and \textit{and}. The rewrite system is defined by the following rules:
\[\text{and}(x, \top) \rightarrow x\]
\[\text{and}(x, \bot) \rightarrow \bot\]
\[\text{and}(x, x) \rightarrow x\]
\[\text{and}(x, \text{xor}(y, z)) \rightarrow \text{xor}(\text{and}(x, y), \text{and}(x, z))\]
\[\text{implies}(x, y) \rightarrow \not(\text{xor}(x, \text{and}(x, y)))\]
\[\not(x) \rightarrow \text{xor}(x, \top)\]
\[\text{or}(x, y) \rightarrow \text{xor}(\text{and}(x, y), \text{xor}(x, y))\]
\[\text{iff}(x, y) \rightarrow \not(\text{xor}(x, y))\]

The \textbf{Bool3} example implements computation in a 3-valued logic. The rewrite system is defined by the following rules, where + and * are AC.

\[
\begin{align*}
  x + 0 & \rightarrow x \\
  x + x + x & \rightarrow 0 \\
  (x + y) * z & \rightarrow (x * z) + (y * z) \\
  \text{and}(x, y) & \rightarrow (x * x * y * y) + (2 * x * x * y) + (2 * x * y) \\
  \text{or}(x, y) & \rightarrow (2 * x * x * y * y) + (x * x * y) + (x * y) + (x + y) \\
  \not(x) & \rightarrow (2 * x) + 1 \\
  2 & \rightarrow 1 + 1
\end{align*}
\]

In \cite{CMR97}, a rewrite system modulo AC for natural arithmetic was presented: \textbf{Nat10}. This system contains 56 rules rooted by the AC symbol +, 11 rules rooted by the AC symbol *, and 82 syntactic rules. The authors conjecture in their paper that compilation techniques and many-to-one matching should improve their implementation.

We experimented AC normalisation of various terms using the two first rewrite systems which produce a large number of AC symbols, and we used the last rewrite system to compute the 16\textsuperscript{th} Fibonacci number: \textit{Fib}(16). Those three systems have been tested with OBJ, the ELAN interpreter and the new ELAN compiler on a Sun Ultra-Enterprise (Solaris) which gives approximately same results as a Pentium Pro 200 (Linux). Results are given in the following table:

<table>
<thead>
<tr>
<th></th>
<th>\textbf{Prop}</th>
<th>\textbf{Bool3}</th>
<th>\textbf{Nat10}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rwr</td>
<td>sec</td>
<td>rwr</td>
</tr>
<tr>
<td>OBJ</td>
<td>12,837</td>
<td>766</td>
<td>-</td>
</tr>
<tr>
<td>ELAN interpreter</td>
<td>-</td>
<td>&gt;24h</td>
<td>-</td>
</tr>
<tr>
<td>ELAN compiler</td>
<td>12,853</td>
<td>0.41</td>
<td>11,119</td>
</tr>
</tbody>
</table>

The last example was originally implemented in CiME which is rather a theorem prover than a programming environment. To compute \textit{Fib}(16), CiME applies 10,599 rules in 16,400 seconds.

Some experiments have been done with the eager matching algorithm in order to show how it minimises the number of matching attempts. The following array shows the number of matched subterms with and without the eager algorithm:

<table>
<thead>
<tr>
<th></th>
<th>\textbf{Main method}</th>
<th>\textbf{Eager algorithm}</th>
<th>\Delta</th>
<th>rwr/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Prop}</td>
<td>50,108</td>
<td>32,599</td>
<td>17,509</td>
<td>31,000</td>
</tr>
<tr>
<td>\textbf{Bool3}</td>
<td>44,861</td>
<td>8,050</td>
<td>36,811</td>
<td>26,000</td>
</tr>
</tbody>
</table>

In most experiments, the number of matching attempts decreases about 50% with the eager algorithm.
6 Conclusion

Experimental results show that the combination of techniques presented in this paper leads to a significant improvement of performance of AC-rewriting normalisation. A last characteristic of the implementation, not yet mentioned, is to provide modular compilation, in the sense that each function or strategy corresponds to one compiled code module. The idea is that when a function or a strategy is modified, only the corresponding module is recompiled. This is extremely important for compilation of large programs.

To conclude this paper, let us mention further work.

- **Compilation of built-in data structures.** The class of built-in data structures has to be extended to incorporate for instance useful data such as strings. Actually, this is just a matter of adapting the work done in the previous compiler.

- **User-defined strategies.** The strategy language of ELAN provides the predefined constructors for strategies briefly presented in this paper, but it also gives the possibility to the user to define his own strategies in a very flexible way using the same paradigm of rewriting [BKK97a, BK97]. Compilation techniques for this powerful strategy language are under development.

- **Combination of theories.** AC-theories are the most frequent ones in mathematical and algebraic structures, but a programmer may be interested in mixing in his specifications AC-function symbols with others that may be only associative, or commutative, or idempotent or with a unit. Although theoretical problems related to combination of matching algorithms have been already explored [Nie91, Rin96], providing an efficient matching algorithm for combination of such symbols is a challenging open problem first attacked in [Eke96].

Acknowledgements

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

References


