J. Functional Programming **11** (2): 207–251, March 2001. Printed in the United Kingdom
© 2001 Cambridge University Press

Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in associative-commutative theories

HÉLÈNE KIRCHNER and PIERRE-ETIENNE MOREAU
LORIA-CNRS & INRIA,
BP 239 54506 Vandœuvre-lès-Nancy Cedex, France
(*e-mail:* `Helene.Kirchner@loria.fr`, `moreau@loria.fr`)

Abstract

First-order languages based on rewrite rules share many features with functional languages, but one difference is that matching and rewriting can be made much more expressive and powerful by incorporating some built-in equational theories. To provide reasonable programming environments, compilation techniques for such languages based on rewriting have to be designed. This is the topic addressed in this paper. The proposed techniques are independent from the rewriting language, and may be useful to build a compiler for any system using rewriting modulo Associative and Commutative (AC) theories. An algorithm for many-to-one AC matching is presented, that works efficiently for a restricted class of patterns. Other patterns are transformed to fit into this class. A refined data structure, namely compact bipartite graph, allows encoding of all matching problems relative to a set of rewrite rules. A few optimisations concerning the construction of the substitution and of the reduced term are described. We also address the problem of non-determinism related to AC rewriting, and show how to handle it through the concept of strategies. We explain how an analysis of the determinism can be performed at compile time, and we illustrate the benefits of this analysis for the performance of the compiled evaluation process. Then we briefly introduce the ELAN system and its compiler, in order to give some experimental results and comparisons with other languages or rewrite engines.

Capsule Review

This paper describes the ELAN language and a compiler for it developed at Nancy. ELAN is relevant to both functional and logic programming, but also has facets closer to the term rewriting community, or even theorem proving. Thinking of an ELAN input as a program, it is a Haskell-like set of equations, called rules. Differences from traditional functional programming include the fact that ELAN's terms are not free trees, but may have AC (Associative and Commutative) constructors; that rewrite rules may be applied anywhere in a tree, not just at the top as in functional languages; and that a rule set need not be deterministic. A sophisticated matching algorithm, based on improving that by Bachmair *et al.* (1993), plays a crucial role in efficiently executing such term rewrite systems. To deal with non-determinism a rich language of "strategy annotations" has been devised to allow choice in a broad space of search and backtracking strategies.

1 Introduction

Rewrite rules are pairs of terms (the left- and right-hand sides) with variables that describe a transformation on given expressions. A rewrite rule may be guarded by a condition which is a boolean expression. A rewrite system is a set of rewrite rules. Rewrite rules are frequent in many areas of computer science, but languages based on rewriting are not so common. Let us cite, for instance, the first-order languages OBJ (Goguen & Winkler, 1988), ASF+SDF (Klint, 1993), Maude (Clavel *et al.*, 1996), Cafe-OBJ (Futatsugi & Nakagawa, 1997) and ELAN (Borovanský *et al.*, 1998b). In these languages, programs are sets of rewrite rules, called rewrite programs, and a query is an expression to evaluate according to these rules. Evaluation is performed by applying rewrite rules. Informally, rewriting an expression consists of selecting a rule whose left-hand side (also called a pattern) matches the current expression, or a sub-expression, computing a substitution that gives the values of rule variables, checking that the condition evaluates to true under this substitution, and applying it to the right-hand side of the selected rule to build the reduced term. In general, the evaluation may not terminate, or terminate with different results according to which rules are applied. So evaluation by rewriting is essentially non-deterministic, and backtracking is used to generate all results. In this paper, we address compilation techniques for languages based on rewriting.

First-order languages based on rewrite rules share many features with functional languages (Bird & Wadler, 1988), such as CAML (Cousineau & Mauny, 1998; Weis & Leroy, 1993), Clean (Brus *et al.*, 1987; Plasmeijer & van Eekelen, 1993), Erlang (Armstrong *et al.*, 1996; Armstrong, 1997), Gofer (Jones, 1994), Haskell (Jones, 1996; Peyton Jones, 1996) or ML (Cousineau *et al.*, 1985; Leroy & Mauny, 1993). They provide the same capability of writing specifications which can be actually executed, tested and debugged. Such a specification is the first prototype of the final program. Re-usability is encouraged through language features such as modules, polymorphism, algebraic types and predefined types. Both classes of languages share concepts like pattern matching (first-order versus higher-order), (tree or graph) rewriting, guards (or conditions), sometimes "where" blocks and "let" expressions. All these programs tend to be concise, easy to understand, and relatively easy to maintain because the code is short, clear, with no side effects or unforeseen interactions. They are strongly typed, eliminating a huge class of errors at compile time. In such languages, the programmer is relieved of the storage management burden. Store is allocated and initialised implicitly, and reclaimed by the garbage collector. Compared to imperative languages, programs are easier to design, write and maintain, but the language offers the programmer less control over the machine.

However, contrary to functional languages, λ-abstraction and higher-order matching are not used in first-order languages based on rewrite rules. Higher-order functions provide a powerful abstraction mechanism, since a function can be freely passed to other functions, returned as a result of a function, stored in a data structure and so on. This is not possible in most rewriting based languages. However, we later present the ELAN system, where the notion of strategy is indeed similar to an higher-order function applied with an explicit application operator. Moreover, although no λ-expression can be written in ELAN, there is an extension of the language, based on ρ-calculus (Cirstea & Kirchner, 1999), which provides a uniform integration of λ-calculus and first-order rewriting.

The loss of abstraction due to the first-order restriction is balanced by the ability to build equational theories in the matching and rewriting mechanism. To illustrate how expressivity and conciseness are gained with this approach, let us consider the formalisation of sorting a list of elements by applying only one rule:

$$L_1 \cdot x \cdot L_2 \cdot y \cdot L_3 \rightarrow L_1 \cdot y \cdot L_2 \cdot x \cdot L_3 \text{ if } x > y$$

where L_1, L_2, L_3 are variables of type $List[Element]$, x, y are variables of sort $Element$ and where the '$\cdot$' list concatenation operator is associative and has a unit element which is the empty list nil. Thanks to these built-in properties, the matching algorithm applied to the list $(3 \cdot 5 \cdot 1)$ will discover several solutions, among which for instance: $L_1 = nil, x = 3, L_2 = nil, y = 5, L_3 = (1)$ and $L_1 = (3), x = 5, L_2 = nil, y = 1, L_3 = nil$. Since the first match does not satisfy the condition $x > y$, the second has to be chosen to apply the rewrite rule and to get the list $(3 \cdot 1 \cdot 5)$. A second rule application yields the now irreducible result $(1 \cdot 3 \cdot 5)$.

Beyond lists, other data structures that can benefit from this kind of rewriting are sets and multisets, where the union operator is Associative and Commutative (AC for short). AC operators provide a high level of abstraction: the programmer may use set data structures without knowing how they are represented. Moreover, their implementation may be more efficient than a list implementation, simply because some optimisations are performed at compile time. To illustrate the practical interest of AC operators, let us consider the well known N-queens problem, that will be further developed in section 5. The problem consists of setting a queen on each row of the chessboard and assigning a unique column number (taken in the set of integers $\{1, \dots, N\}$) such that the queens do not attack each other. We represent a solution by a list of integers and we use a set to represent columns that may be assigned to a queen. As we will see in Example 3 on page 232, the program may be expressed in ELAN with three rules and a strategy. The first of these three rules is the following one, where i and S are variables of respective types $Integer$ and Set:

$$(i) \cup S \rightarrow [i, S]$$

The $\_ \cup \_$ operator (where $\_$ stands for a place-holder) is an AC infix operator, which allows expressing constructions such as $(1) \cup (2) \cup (3)$. The $[\_, \_]$ operator is a constructor used to represent pairs, for example. When this rule is applied on a given term, say $(1) \cup (2) \cup (3)$, the first result is $[1, (2) \cup (3)]$, when the variable i is instantiated to 1 (i.e. (i) matches (1)). Another possibility is to match i with 2, and in this case, the result is $[2, (1) \cup (3)]$. Thus, the application of an AC rule returns a multiset of results, and behaves like a *generator* over a set data structure. This feature will be used in Example 3 on page 232 to enumerate all possibilities to set a queen on a chessboard.

Let us explain in more detail the properties of first-order languages based on rewrite rules. The evaluation mechanism relies primarily on a matching algorithm that must be carefully designed. When programming with rewrite rules, it frequently happens that programs contain several rules whose left-hand sides begin with the same top function symbol. This often corresponds to a case definition of a function, as usual in functional programming languages. In this context, many-to-one pattern matching is quite relevant. The idea is to group all these rules together and search for a candidate rewrite rule in this group when the subject to be reduced begins with the same top function symbol.

When the program involves algebraic structures with axioms stating commutativity of function symbols, these axioms cannot be oriented as a terminating rewrite system. As mentioned above, they 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 and commutative theories. However, AC matching has a high computational complexity, as analysed by several authors (Benanav *et al.*, 1987; Hermann & Kolaitis, 1995). Moreover, since an AC matching problem may have several minimal solutions, if the first match which is found does not satisfy the condition of the rule, or if all possibilities are looked for, backtracking is used to get the next solutions. With this additional non-determinism, rewriting in such theories becomes computationally difficult and it is a real challenge to identify subclasses of programs for which it is possible to provide an efficient compiler for the language.

Although it is essential to have a good matching algorithm to get an efficient rewriting engine, matching is not the only operation involved in a normalisation process. To compute the reduced term, a global consideration of the whole process is crucial: all data structures and sub-algorithms have to be well designed to cooperate without introducing any bottleneck. Optimisations go through the careful combination of several algorithms and data structures. In this paper, we show how we have reused and improved existing techniques, and invented new ones, in order to build a compiler for AC rewriting.

As already mentioned, there are different sources of non-determinism in the evaluation process. If the rewrite system is not confluent, different normal forms, when they exist, may be considered as relevant results of the computation. In presence of AC function symbols, several matching solutions have to be considered and lead to different results. We explain in this paper how to use strategy constructors to handle sets of results, how to perform a determinism analysis at compile time, and the benefits of this analysis for the performance of the compiled evaluation process.

This paper is an extension of previous work (Moreau & Kirchner, 1998; Kirchner & Moreau, 1998). After a short introduction of notation and classical definitions in section 2, an algorithm for many-to-one AC matching is presented in section 3. This algorithm works efficiently for a restricted class of patterns, and other patterns are transformed to fit into this class. A refined compact bipartite graph data structure allows encoding all matching problems relative to a set of rewrite rules. A few optimisations concerning the construction of the substitution and of the reduced term are described in section 4. In section 5, we turn to the problem of non-determinism, and show how to handle it through the concept of strategies. All these sections are independent from the rewriting language, and the techniques described may be useful to build a compiler for any language based on non-deterministic rewriting in associative and commutative theories. Section 6 briefly introduces the ELAN system and its compiler, implementing the techniques described in this paper, in order to give some experimental results and comparisons with other languages or rewrite engines. The conclusion in section 7 points out a few directions where there seems to be yet some room for further improvements.

2 Preliminary concepts

We assume the reader familiar with basic definitions of term rewriting given in particular in Dershowitz & Jouannaud (1990) and Baader & Nipkow (1998), and associative commutative theories, handled for instance in Peterson & Stickel (1981) and Jouannaud & Kirchner (1986). We briefly recall or introduce notation for a few concepts that will be used along this paper.

$\mathscr{T}(\mathscr{F},\mathscr{X})$ is the set of *terms* built from a given finite set $\mathscr{F}$ of function symbols and a denumerable set $\mathscr{X}$ of variables, denoted $x, y, z, \ldots$ Positions in a term are represented as sequences of integers and denoted by the Greek letters ϵ, ν. The empty sequence ϵ denotes the position associated to the root, and so it is the position of the top symbol. The subterm of t at position ν is denoted $t_{|\nu}$. The replacement at position ν of the subterm $t_{|\nu}$ by t' is written $t[\nu \hookleftarrow t']$. The set of variables occurring in a term t is denoted by $\mathscr{V}ar(t)$. If $\mathscr{V}ar(t)$ is empty, t is called a *ground term*, and $\mathscr{T}(\mathscr{F})$ is the set of ground terms. A term t is said to be *linear* if no variable occurs more than once in t. A *substitution* is an assignment from a finite subset of $\mathscr{X}$ to $\mathscr{T}(\mathscr{F},\mathscr{X})$, written $\sigma = \{y_1 \mapsto t_1, \ldots, y_k \mapsto t_k\}$. The result of applying σ to a term t is denoted by $t\sigma$.

In the first part of this paper, we focus on theories and rewrite systems in which there is at least one binary function symbol F, that satisfies the following set AC of associativity and commutativity axioms:

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

Such symbols are called AC function symbols, and are always in uppercase in the following. On the other hand, $\mathscr{F}_\emptyset$ is the subset of $\mathscr{F}$ made of function symbols which are not AC, and are called *free* function symbols. In the following, we consider that a function symbol is either free or AC. A term is said to be *syntactic* if it contains only free function symbols. We write $s =_{AC} t$ to indicate that the two terms s and t are equivalent modulo associativity and commutativity.

Definition 1 *A rewrite rule is a pair of terms denoted $l \to r$ such that $l, r \in \mathscr{T}(\mathscr{F},\mathscr{X})$ and $\mathscr{V}ar(r) \subseteq \mathscr{V}ar(l)$. The term l is called the left-hand side or pattern and r is the right-hand side.*

To get a better control on the application of the rewrite rules, conditions can be added. In this paper, we consider an enriched notion of condition, called *matching condition*, as used, for instance, in ASF+SDF and ELAN.

Definition 2 *A* conditional rewrite rule *denoted $l \to r$ **where** $p := c$ is such that $l, r, p, c \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\mathcal{V}ar(p) \cap \mathcal{V}ar(l) = \emptyset$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p)$ and $\mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$. When the term p is just the boolean constant true, the condition is usually written **if** c.*

The notion of conditional rewrite rule can be generalised with a sequence of conditions, as in $l \to r$ **where** $p_1 := c_1 \ldots$ **where** $p_n := c_n$ where:

- $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
- $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$ and
- $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$.

A (conditional) rewrite rule is said to be *syntactic* if the left-hand side is a syntactic term.

To apply a syntactic rule $l \to r$ on a term t at some position v, one looks for a matching, i.e. a substitution σ satisfying $l\sigma = t_{|v}$. Note that t is always a ground term. The algorithm which provides the unique substitution σ, whenever it exists, is called *syntactic matching*. Once a substitution σ is found, the application of the rewrite rule consists of building the *reduced term* $t' = t[v \hookleftarrow r\sigma]$. Computing the normal form of a term t w.r.t. a rewrite system R consists of successively applying the rewrite rules of R, at any position, until no more applies. The existence and uniqueness of normal forms require the rewrite system R to be, respectively, terminating and confluent.

To apply a syntactic conditional rule $l \to r$ **where** $p := c$ on a term t, the satisfiability of the condition **where** $p := c$ has to be checked before building the reduced term. Let σ be the matching substitution from l to $t_{|v}$. Checking the matching condition **where** $p := c$ consists first of using the rewrite system R to compute a normal form c' of $c\sigma$, whenever it exists, and then verifying that p matches the ground term c'. If there exists a matching μ, such that $p\mu = c'$, the composed substitution $\sigma\mu$ is used to build the reduced term $t' = t[v \hookleftarrow r\sigma\mu]$. Otherwise, the application of the conditional rule fails. For usual boolean conditions of the form **if** c, μ is the identity when the normal form of c is *true*. In cases where $c\sigma$ has no normal form, the application of the rule does not terminate.

When the conditional rule is of the form $l \to r$ **where** $p_1 := c_1 \ldots$ **where** $p_n := c_n$, the matching substitution is successively composed with each matching μ_i from p_i to a normal form of $c_i\sigma\mu_1 \ldots \mu_{i-1}$, for $i = 1, \ldots, n$, when it exists. If one of these μ_i does not exist, the application of the conditional rule fails. If one of the $c_i\sigma\mu_1 \ldots \mu_{i-1}$ for $i = 1, \ldots, n$ has no normal form, the application of the rule does not terminate.

When the left-hand side of the (conditional) rule contains AC function symbols, AC matching is invoked from l to $t_{|v}$. The term l is said to AC match the term $t_{|v}$ if there exists a substitution σ such that $l\sigma =_{AC} t_{|v}$. In general, AC matching can return several solutions, which introduces a need for backtracking for conditional rules: as long as there is a solution to the AC matching problem for which the matching condition is not satisfied, another solution has to be extracted. Similarly, if the pattern p contains AC function symbols, an AC matching procedure is called. Only when all solutions have been tried unsuccessfully, the application of this conditional rule fails. When the rule contains a sequence of matching conditions, failing to satisfy the *i*th condition causes a backtracking to the previous one.

So in our case, 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 provided 'on request'. AC matching has already been extensively studied (Hullot, 1980; Benanav *et al.*, 1987; Kounalis & Lugiez, 1991; Bachmair *et al.*, 1993; Lugiez & Moysset, 1994; Eker, 1995). In this paper, we borrow from these works some techniques for the compilation of AC-matching, but we go further and address the more global problem of its integration in a normalisation procedure.

One of the first problems encountered for an efficient implementation is to choose an adequate term representation. It is well known that terms involving associative function symbols can be converted to a normal form by grouping associative operators to the right (or left). Equivalently, and more usefully for machine representation, we may flatten such terms by replacing nested occurrences of the same associative operator by a single variadic operator (i.e. an operator with a variable arity). The same may be done with AC operators. However, this does not give a unique normal form, since the commutativity axiom may be used to arbitrarily permute the arguments of a variadic operator. Nevertheless, we obtain a unique normal form by regarding the arguments of a variadic operator as a multiset of terms (since the same term may occur as an argument more than once). A canonical form (Hullot, 1980; Eker, 1995) corresponds to an implementation of this idea where a unique syntactic representation of this multiset of arguments is obtained by sorting and grouping. Canonical form computation can be seen as a function $\mathcal{CF}$ on $\mathcal{T}(\mathcal{F},\mathcal{X})$ that returns, for each term t, a unique representative of its congruence class. Let $>$ be a total ordering on the set of symbols $\mathcal{F} \cup \mathcal{X}$.[1] For terms that are just variables or constants, $\mathcal{CF}$ 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 is given by the ordering on their top symbols. This ordering on canonical forms is also denoted $>$. Considering that AC function symbols can be variadic, 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 α identical subterms by a single instance of the subterm with multiplicity α, denoted by t^{α}. A formal definition can be found in Eker (1995). An algorithm that performs a bottom-up computation of $\mathcal{CF}$ is given later on, in section 4.1 on page 228. At this stage, canonical form computation is easily understandable by an example. Consider the term $t = F(F(t_1,t_2), t_3, F(t_4,t_5))$ where no t_i has the AC function symbol F as top symbol. Assuming 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 $\mathcal{CF}(t) = F(\mathcal{CF}(t_1)^2, \mathcal{CF}(t_2), \mathcal{CF}(t_3)^2)$.

A term in canonical form is said to be *almost linear* if the term obtained by forgetting the multiplicities of variable subterms is linear. For instance, the term $t = F(x^3, y^2, g(a))$ is almost linear.

For a term t in canonical form, the *syntactic top layer* $\hat{t}$ is obtained from t

[1] Any ordering can be chosen but, once chosen, it is fixed, since it determines the uniqueness of representation.

by removing subterms below the first AC symbol in each branch and considering remaining AC symbols as constants. For instance the top layer of the term $f(g(a), F(f(a,x), f(y, g(b))))$ is $f(g(a), F)$, if $f, g, a, b \in \mathscr{F}_\emptyset$. Note that $f(g(a), F)$ is a syntactic term if we consider F as a constant. A formal definition of the top layer and a few properties can be found in Bachmair *et al.* (1993) and Moreau (1999).

Another theoretical difficulty related to AC rewriting is to ensure its completeness with respect to equality in congruence classes: given a rewrite system R, how to ensure that for any terms t and t', t and t' are equivalent in the theory defined by R and AC, if and only if t and t' AC rewrite, respectively, to terms u and u' such that $u =_{AC} u'$. The interested reader can refer to Jouannaud & Kirchner (1986), which provides an extensive study of this question. For the purpose of this paper, it is enough to know that to achieve this completeness property, we must already ensure the following property of coherence with AC congruence classes: if a term is reducible, any AC equivalent term is reducible too. To illustrate the problem and its solution, let us consider the associative-commutative union operator: $\cup$, and the rewrite rule that removes identical elements of a multiset:

$$x \cup y \to x \textbf{ if } x = y$$

When applied on $a \cup a$, the result is a, but when applied on $(a \cup b) \cup (a \cup c)$, the result is not $a \cup b \cup c$ because the rule cannot be applied directly to $(a \cup a)$. To perform the expected rewrite step on subterms of equivalent terms such as the subterm $(a \cup a)$ of $(a \cup a) \cup (b \cup c)$ in this example, a new rule with an extension variable z' has to be added:

$$z' \cup (x \cup y) \to z' \cup x \textbf{ if } x = y$$

The variable z' matches the context and allows us to perform rewrite steps in subterms. To obtain this coherence property of AC rewriting, some rules called extensions are automatically added. The extension of the rule $F(l_1, \ldots, l_n) \to r$, already put in flattened form, where F is an AC function symbol, is of the form $F(z', l_1, \ldots, l_n) \to F(z', r)$, where z' is called an extension variable. This well known technique was introduced by Peterson & Stickel (1981), and generalised by Jouannaud & Kirchner (1986), where a more detailed justification of these extensions can be found. Extensions are not needed for rules of the form $F(l_1, \ldots, l_n) \to r$, where F is AC, and where one of the $l_1, \ldots, l_n$ is a variable with multiplicity 1 which also does not occur in the condition of the rule. Indeed, this variable can capture the context as well as an extension variable. Note that an extension is always necessary for rules where $l_1, \ldots, l_n$ are non-variable subterms. In addition, from the implementation point of view, the reduction with a rule $F(l_1, \ldots, l_n) \to r$ can be simulated with $F(z', l_1, \ldots, l_n) \to F(z', r)$ only, by allowing the extension variable z' to be instantiated to an empty context.

Example 1 *To support intuition throughout this paper, we choose the following running example of two rewrite rules with the same AC top symbol F and whose right-hand sides are irrelevant. The first rule has a boolean condition* ***if*** *$z = x$, which is equivalent to the matching condition* ***where*** *true $:= (z = x)$, and we assume that the boolean* *function = is completely defined by rewrite rules.*

$$\begin{array}{lcll} F(z, f(a,x), g(a)) & \rightarrow & r_1 & \textbf{\textit{if}} \quad z = x \\ F(f(a,x), f(y, g(b))) & \rightarrow & r_2 & \end{array}$$

Their respective extensions are:

$$\begin{array}{lcll} F(z', z, f(a,x), g(a)) & \rightarrow & F(z', r_1) & \textbf{\textit{if}} \quad z = x \\ F(z', f(a,x), f(y, g(b))) & \rightarrow & F(z', r_2) & \end{array}$$

Both extensions are necessary and for reduction, only extensions are sufficient. In all following examples related to these four rules, we assume that x, y, z, z' are variables, f, g, a, b are free function symbols, and F, G are AC function symbols. We also assume that the following total order is used to compute canonical forms:

$$z' > x > y > z > f > g > a > b$$

3 Many-to-one AC matching

An AC matching problem is said to be *one-to-one* when only one pattern p and only one subject s are involved: the problem consists of finding substitutions σ such that $p\sigma =_{AC} s$. By extension, an AC many-to-one matching is the following problem: given a set of terms $P = \{p_1, \ldots, p_n\}$, called patterns, and a ground term s, called a 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 the syntactic case and in AC theories) are based on the general idea of factoring patterns to produce a matching automaton. The discrimination net approach (Gräf, 1991; McCune, 1992; Christian, 1993; Voronkov, 1995; Nedjah *et al.*, 1997; Moreau, 1999) is one of this kind: it is a variant of the data structure used to index dictionaries. Given a set of patterns, the idea is to partition terms based upon their structure. As presented by Christian (1993), a tree is formed and at each point where two terms have different symbols, a separate branch for each term is added.

Figure 1 shows a discrimination net associated to the set of patterns $P = \{f(a,x), f(y, g(b)), g(a)\}$. At the end of each path in the tree, there is a list of terms sharing the same structure. Since at this stage of the matching algorithm, we are dealing only with linearised terms, all variables are different and may be treated as a single wildcard symbol ω. A discrimination net is said to be deterministic when no backtracking is needed to compute the maximal set of terms that match a given subject. To build such a discrimination net, several algorithms exist and are fully detailed elsewhere (Gräf, 1991; Christian, 1993; Nedjah *et al.*, 1997; Moreau, 1999). In our implementation we use the algorithm presented by Moreau (1999), which is incremental and produces compact discrimination nets (i.e. with a reduced number of nodes).

Given a deterministic discrimination net D, a matching automaton A is associated: the nodes of D are the states of A, the root being the initial state and the leaves being the final states. On a given input term t, the automaton scans t in preorder and makes a transition $u \rightarrow v$ if D contains an edge (u, v) labelled by the current symbol of t or

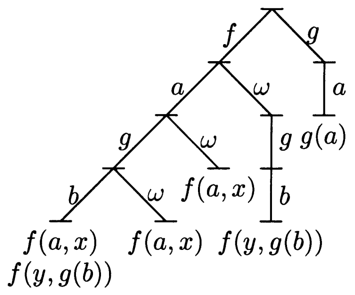


Fig. 1. Deterministic discrimination net associated to $f(a,x)$, $f(y,g(b))$ and $g(a)$.

by ω. In the latter case, the current subterm of t is skipped and used to build the substitution. Given the ground term $f(g(c),g(b))$, the matching automaton associated to the previous discrimination net recognises the pattern $f(y,g(b))$ (following the path $f \to \omega \to g \to b$) where y is instantiated to the subterm $g(c)$.

In the case of AC theories, the matching problems are decomposed according to the syntactic top layers and the different AC symbols occurring in the patterns. This decomposition gives rise to a hierarchically structured collection of standard discrimination nets, called an AC discrimination net (Bachmair *et al.*, 1993; Graf, 1996). Given a set of patterns $P_i = P = \{p_1, \ldots, p_n\}$, the construction of such a structure is performed in four steps:

1. computation of the syntactic top layer $\hat{P}_i = \{\hat{p_{i_1}}, \ldots, \hat{p_{i_n}}\}$;
2. construction of the matching automaton A_i associated to $\hat{P}_i$ (where all AC symbols are considered as constants);
3. recursive application of the algorithm to P_{i+1} (the set of 'removed' terms during the computation of $\hat{P}_i$);
4. construction of a special edge between AC symbols appearing in A_i and the top automaton A_{i+1} associated to the sub-AC matching structure built during the recursive application of the algorithm.

Let us consider again the set of patterns in our running example, where F is the unique AC function symbol:

$$P_1 = P = \{F(z', z, f(a,x), g(a)), F(z', f(a,x), f(y, g(b)))\}$$

We have $\hat{P}_1 = \{F, F\}$ and the set of removed terms is $P_2 = \{z', z, f(a,x), f(y,g(b)), g(a)\}$. This decomposition leads us to build the AC discrimination net presented in Figure 2.

The resulting net is composed of two parts:

- a discrimination net for the top layers used to determine which rules can be applied, and a link to the sub-automaton used to match subterms of AC function symbols (here F);
- the sub-automaton itself that implements a many-to-one syntactic matching algorithm (Gräf, 1991; Christian, 1993; Nedjah *et al.*, 1997; Moreau, 1999) for the set of syntactic subterms: $f(a,x), f(y,g(b))$ and $g(a)$.

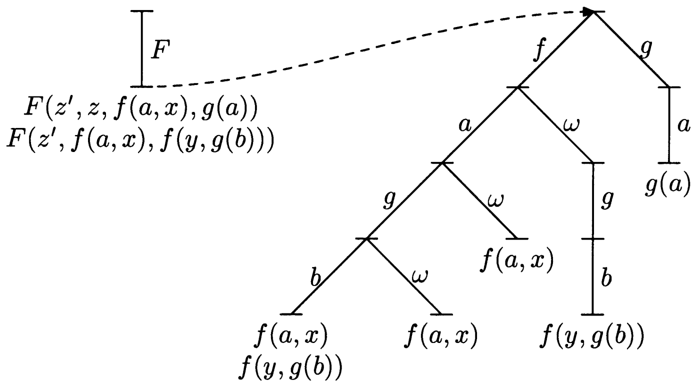


Fig. 2. Example of AC discrimination net.

3.1 Description of the algorithm

The skeleton of our many-to-one AC matching algorithm is similar to the algorithm presented ny Bachmair *et al.* (1993). Given a set of patterns P,

1. Transform rules to fit into a specific class of patterns. In particular, patterns are converted to their canonical forms, each rule with non-linear left-hand side is transformed into a conditional rule with a linear left-hand side and a condition expressing equality between variables (of the form $x = y$).

2. Compute the AC discrimination net associated to $P = \{p_1, \dots, p_n\}$ and the corresponding matching automata (as in Figure 2).

The previous steps only depend upon the set of rewrite rules. They can be performed once and for all at compile time.

At run-time the subject $s = F(s_1, \dots, s_p)$ is known and the matching automata are used to build bipartite graphs, where an edge between p_i and s_j is added if the subpattern p_i matches the subterm s_j. Given a ground term $s = F(s_1, \dots, s_p)$ in canonical form, the following steps are performed.

3. Build a hierarchy of bipartite graphs according to the given subject s in canonical form. For each subterm $p_{i|v}$, where v is a position of an AC function symbol in $\hat{p}_i$, an associated bipartite graph is built. Let us consider an AC matching problem from $F(t_1, \dots, t_m)$ to $F(s_1, \dots, s_p)$, where $t_1, \dots, t_m$ come from subterms of $p_{i|v}$, and where for some k, $0 \leqslant k \leqslant m$, no $t_1, \dots, t_k$ is a variable, and all $t_{k+1}, \dots, t_m$ are variables. The associated bipartite graph is $BG = (V_1 \cup V_2, E)$ whose sets of vertices are $V_1 = \{s_1, \dots, s_p\}$ and $V_2 = \{t_1, \dots, t_k\}$, and whose set of edges E consists of all pairs $[s_i, t_j]$ such that $t_j\sigma$ and s_i are equal modulo AC for some substitution σ.
This construction is done recursively for each subterm of $p_{i|v}$ whose root is an AC symbol. An example of recursive construction which leads to a hierarchy of bipartite graphs is given in Figure 3.

In the case of our running example, given the ground term $s = F(f(a,a), f(a, g(b)), f(g(c), g(b)), g(a))$, the compiled matching automaton is used to build the two bipartite graphs given in Figure 4 (one for each rule);

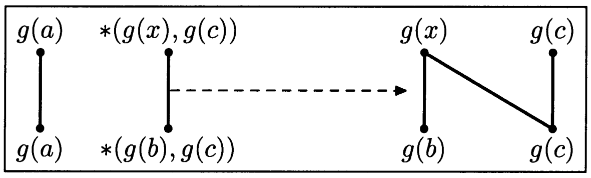


Fig. 3. To illustrate the notion of hierarchy of bipartite graphs, we consider the following AC matching problem: a pattern $+(g(a), *(g(x), g(c)))$ and a subject $+(g(a), *(g(b), g(c)))$, where $+$ and $*$ are AC symbols, x is a variable and a, b, c, f are syntactic. Following the main algorithm, a recursive call of the AC matching algorithm is needed to build the edge between $*(g(x), g(c))$ and $*(g(b), g(c))$. This call leads to the construction of a sub-bipartite graph whose satisfiability has to be checked: if the sub-graph has a solution, the edge between $*(g(x), g(c))$ and $*(g(b), g(c))$ can be built. Solving a hierarchy of bipartite graphs means that sub-graphs are solved in a bottom-up way.

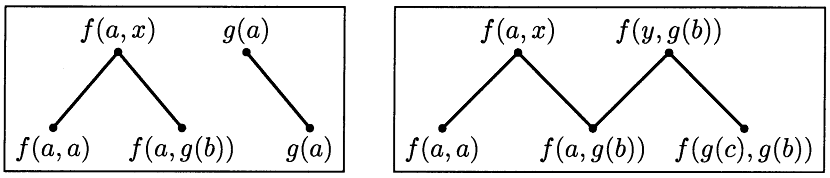


Fig. 4. Examples of bipartite graphs.

4. Find a set of solutions to the hierarchy of bipartite graphs and construct a linear Diophantine system which encodes the constraints on the remaining unbound variables (which appear directly under an AC symbol): to match m variables $x_1^{\alpha_1}, \ldots, x_m^{\alpha_m}$ to n remaining subterms $s_1^{\beta_1}, \ldots, s_n^{\beta_n}$ one looks for non-negative integer solutions of the system $\bigwedge_{i=1\ldots n} \beta_i = \alpha_1 X_i^1 + \cdots + \alpha_m X_i^m$ with the additional constraint $\Sigma_{i=1}^n X_i^j \geqslant 1$.

5. Solve the linear Diophantine system to get solutions of the form $x_k = F(s_1^{X_1^k}, \ldots, s_n^{X_n^k})$ for $k = 1 \ldots m$. This completes the definition of the matching substitution. As an example, consider the pattern $+(x_1, x_2^3)$ and the subject $+(a^3, b^2, c^5)$ where $+$ is an AC operator. Intuitively, this matching problem has three solutions:

$$\begin{aligned} S_1 &= \{x_1 \mapsto +(a^3, b^2, c^2), && x_2 \mapsto c\} \\ S_2 &= \{x_1 \mapsto +(b^2, c^5), && x_2 \mapsto a\} \\ S_3 &= \{x_1 \mapsto +(b^2, c^2), && x_2 \mapsto +(a, c)\} \end{aligned}$$

These solutions are found by solving the following linear Diophantine system:

$$\begin{aligned} \beta_1 = 3 &= 1 \times X_1^1 &+& \quad 3 \times X_1^2 \\ \beta_2 = 2 &= 1 \times X_2^1 &+& \quad 3 \times X_2^2 \\ \beta_3 = 5 &= \underbrace{1 \times X_3^1}_{\Sigma X_i^1 > 1} &+& \quad \underbrace{3 \times X_3^2}_{\Sigma X_i^2 > 1} \end{aligned}$$

This linear Diophantine system has three solutions:

$$\begin{aligned} &X_1^1 = 3 \wedge X_1^2 = 0 \wedge X_2^1 = 2 \wedge X_2^2 = 0 \wedge X_3^1 = 2 \wedge X_3^2 = 1 \\ &X_1^1 = 0 \wedge X_1^2 = 1 \wedge X_2^1 = 2 \wedge X_2^2 = 0 \wedge X_3^1 = 5 \wedge X_3^2 = 0 \\ &X_1^1 = 0 \wedge X_1^2 = 1 \wedge X_2^1 = 2 \wedge X_2^2 = 0 \wedge X_3^1 = 2 \wedge X_3^2 = 1 \end{aligned}$$

By computing $x_1 = +(a^{X_1^1}, b^{X_2^1}, c^{X_3^1})$ and $x_2 = +(a^{X_1^2}, b^{X_2^2}, c^{X_3^2})$, the first assignment leads to the first solution:

$$S_1 = \{x_1 \mapsto +(a^3, b^2, c^2), x_2 \mapsto c\}$$

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, building bipartite graphs, or solving linear Diophantine systems. The idea is to apply these costly steps on specific patterns for which they can be designed efficiently, or to simply skip these steps when they are useless. We identified classes of patterns, presented in the following section, for which the general algorithm described above can be efficiently implemented. These patterns already cover a large class of rewrite programs, and for the other cases, we propose a pre-processing of the rewrite program, that transforms it into a semantically equivalent program that belongs to the restricted classes of patterns. This is explained in section 3.5.

3.2 Classes of patterns

The classes of compiled patterns are defined on the whole set of rules together with their extensions, automatically added as described at the end of section 2. 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, defined below, respectively contain linear terms with no AC function symbol, at most one and at most two levels of AC function symbols with a maximum of two variables rooted by an AC function symbol. The motivation to select these patterns was first based on an empirical study of rewrite rules systems used in different applications. It appeared that, in practice, these restrictions are sufficiently weak to describe a large class of patterns occurring in specifications based on rewriting. On the other hand, they are sufficiently strong to allow us to design a specialised and efficient AC normalisation algorithm.

Definition 3 *Let $\mathscr{F}_\emptyset$ be the set of free function symbols, $\mathscr{F}_{AC}$ the set of AC function symbols and $\mathscr{X}$ the set of variables.*

- *The pattern class C_0 consists of linear terms $t \in \mathscr{T}(\mathscr{F}_\emptyset, \mathscr{X}) \backslash \mathscr{X}$.*
- *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 $t = F(x_1, x_2^{\alpha_2}, t_1, \ldots, t_n)$, with $F \in \mathscr{F}_{AC}$, $0 \leqslant n$, $t_1, \ldots, t_n \in C_0$, $x_1, x_2 \in \mathscr{X}$, $\alpha_2 \geqslant 0$, and all terms t of the form $f(t_1, \ldots, t_n)$, with $f \in \mathscr{F}_\emptyset$, $t_1, \ldots, t_n \in C_1 \cup \mathscr{X}$.*
- *The pattern class C_2 is the smallest set of almost linear terms in canonical form that contains C_1, all terms of the form $t = F(x_1, x_2^{\alpha_2}, G(x_3, x_4^{\alpha_4}))$ with $F, G \in \mathscr{F}_{AC}$, $x_1, x_2, x_3, x_4 \in \mathscr{X}$, $\alpha_2 \geqslant 0$, $\alpha_4 > 0$, and all terms t of the form $f(t_1, \ldots, t_n)$, with $f \in \mathscr{F}_\emptyset$, $t_1, \ldots, t_n \in C_2 \cup \mathscr{X}$.*

In our example, the patterns $F(z, f(a, x), g(a))$ and $F(f(a, x), f(y, g(b)))$, have been extended into patterns $F(z', z, f(a, x), g(a))$ and $F(z', f(a, x), f(y, g(b)))$ where z' is an extension variable. Only these two last patterns have to be considered for reduction, and they belong to the class C_1.

3.3 Many-to-one AC matching using compact bipartite graphs

Let us first emphasize that the AC matching techniques described in this section are restricted to the class of patterns presented in section 3.2, which leads to several improvements of the general approach described in section 3.1:

- Thanks to the restriction put on patterns, the hierarchy of bipartite graphs has at most two levels, and the second one is degenerate. Thus, the construction can be done without recursion.
- We use a new compact representation of bipartite graphs, which encodes, in only one data structure, all matching problems relative to the given set of rewrite rules.
- No linear Diophantine system is generated since there are at most two variables, with (restricted) multiplicity, under an AC function symbol in the patterns. Instantiating these variables can be done in a simple and efficient way.
- A preliminary syntactic analysis of rewrite rules can determine that only one solution of an AC matching problem has to be found to apply some rule. This is the case for unconditional rules or for rules whose conditions do not depend on a variable that occurs under an AC function symbol in the left-hand side. Taking advantage of the structure of compact bipartite graphs, a refined algorithm is presented to handle those particular (but frequent) cases.

3.3.1 Compact bipartite graph

Given a set of patterns with the same syntactic top layer, all subterms with the same AC top function symbol are grouped (at compiled time) to build (at runtime) a particular bipartite graph called a *compact bipartite graph* described below. Given a subject, the compact bipartite graph encodes all matching problems relative to the given set of rewrite rules. All bipartite graphs that the general algorithm would have to construct can be generated from this compact data structure. In general, the syntactic top layer may be not empty and several AC function symbols may occur. In this case, a compact bipartite graph has to be associated to each AC function symbol. Each graph is solved and the solutions have to be combined to solve the matching problem.

Such a decomposition leads us to focus our attention on sets of patterns $p_1, \dots, p_n$ defined as follows:

$$\begin{array}{llllll} p_1 & = F(& p_{1,1} & ,\dots, & p_{1,m_1} &) \\ \vdots & & \vdots & & \vdots & \\ p_n & = F(& p_{n,1} & ,\dots, & p_{n,m_n} &) \end{array}$$

where for some k_j, $0 \leqslant k_j \leqslant m_j$, no $p_{j,1}, \dots, p_{j,k_j}$ is a variable, and all $p_{j,k_j+1}, \dots, p_{j,m_j}$ are variables. Syntactic subterms $p_{j,k}$ are grouped together and given a subject $s = F(s_1^{\alpha_1}, \dots, s_p^{\alpha_p})$, a discrimination net that encodes a many-to-one syntactic matching automaton is built. This automaton is used to find pairs of terms $[s_i, p_{j,k}]$ that match each other and to build the associated Compact Bipartite Graph, defined as follows:

$CBG = (V_1 \cup V_2, E)$ where $V_1 = \{s_1, \dots, s_p\}$, $V_2 = \{p_{j,k} \mid 1 \leqslant j \leqslant n, 1 \leqslant k \leqslant k_j\}$, and E consists of all pairs $[s_i, p_{j,k}]$ such that $p_{j,k}\sigma = s_i$ for some substitution σ.

3.3.2 Solving a compact bipartite graph

Finding a pattern that matches the subject usually consists of selecting a pattern p_j, building the associated bipartite graph BG_j, finding a maximum bipartite matching (Hopcroft & Karp, 1973; Fukuda & Matsui, 1989) and finding assignments to remaining unbound variables. Instead of building a new bipartite graph BG_j each time a new pattern p_j is tried, in our approach, the bipartite graph BG_j is extracted from the compact bipartite graph $CBG = (V_1 \cup V_2, E)$ as follows:

$$BG_j = (V_1 \cup V_2', E') \text{ where } \left\{ \begin{array}{lcl} V_2' & = & \{p_{j,k} \mid p_{j,k} \in V_2 \text{ and } 1 \leqslant k \leqslant k_j\} \\ E' & = & \{[s_i, p_{j,k}] \mid [s_i, p_{j,k}] \in E \text{ and } p_{j,k} \in V_2'\} \end{array} \right.$$

The set V_2' contains only vertices associated to the pattern p_j and E' is the set of edges that consists of pairs $[s_i, p_{j,k}]$ matched by p_j.

Given the subject $s = F(s_1^{\alpha_1}, \dots, s_p^{\alpha_p})$ and a fixed j, S_j is a *solution* of BG_j if:

$$\left\{ \begin{array}{l} S_j \subseteq E' \text{ and } \forall k \in \{1, \dots, k_j\}, \exists!\ [s_i, p_{j,k}] \in S_j \\ card(\{[s_i, p_{j,k}] \in S_j \mid 1 \leqslant i \leqslant p\}) \leqslant \alpha_i \end{array} \right.$$

Roughly speaking, S_j is a solution of the bipartite graph BG_j if all patterns $p_{j,1}, \dots, p_{j,k_j}$ match a different ground subterm of $\{s_1, \dots, s_p\}$ (according to multiplicities $\alpha_1, \dots, \alpha_p$).

This solution corresponds to a maximum bipartite matching for BG_j. If S_j does not exist, the next bipartite graph BG_{j+1} (associated to p_{j+1}) has to be extracted. Note that common syntactic subterms are matched only once, even if they appear in several rules, since the information is saved once for all in the compact bipartite graph.

The main advantage of the many-to-one approach is that it is no longer necessary to inspect the subject more than once to build the compact bipartite graph: given a ground term s_i, the compiled version of the matching automaton traverses positions of s_i only once, and returns a list of $p_{j,k}$ that match s_i. This list of pairs $[s_i, p_{j,k}]$ is directly used to build the compact bipartite graph. Moreover, extraction can be performed efficiently with an adapted data structure: the compact bipartite graph can be represented by a set of bit vectors. A bit vector is associated to each subterm $p_{j,k}$ and the ith bit is set to 1 if $p_{j,k}$ matches to s_i. Encoding compact bipartite graphs by a list of bit vectors has two main advantages: the memory usage is low and the bipartite graph extraction operation is extremely cheap, since only selections of bit vectors are performed.

Considering our running example, an analysis of subterms with the same AC top function symbol F gives three distinct non-variable subterms up to variable renaming: $p_{1,1} = f(a, x)$ and $p_{1,2} = g(a)$ for $F(z', z, f(a, x), g(a)) \rightarrow F(z', r_1)$ **if** $z = x$, and $p_{2,1} = f(a, x)$, $p_{2,2} = f(y, g(b))$ for $F(z', f(a, x), f(y, g(b))) \rightarrow F(z', r_2)$.

As explained in section 3.3, given a set of patterns with the same syntactic top layer, all subterms with the same AC top function symbol are grouped. This initialisation step is compiled in the function `init_pattern_list_F` described in Program 1.

Initialising the list of patterns

```
void init_pattern_list_F() {
  /* F(z',z,f(a,x),g(a)) */
  pattern_tab[0]=0; pattern_tab[1]=1;
  MS_pattern_list_init(pattern_list_F,pattern_tab);
  /* F(z',f(y,g(b)),f(a,x)) */
  pattern_tab[0]=2; pattern_tab[1]=0;
  MS_pattern_list_init(pattern_list_F,pattern_tab);
}
```

Program 1: *This function (written in* `C`*) initialises the compact bipartite graph construction by assigning a number to each pattern and sub-pattern, and filling the global list:* `pattern_list_F` *(the local array* `pattern_tab` *is used to temporally store these numbers).* $F(z', z, f(a, x), g(a))$ *and* $F(z', f(y, g(b)), f(a, x))$ *are patterns number* 0 *and* 1 *(note that an extension variables* `z'` *has been added). The numbers* 0, 1 *and* 2 *are assigned, respectively, to sub-patterns* $f(a, x)$, $g(a)$ *and* $f(y, g(b))$.

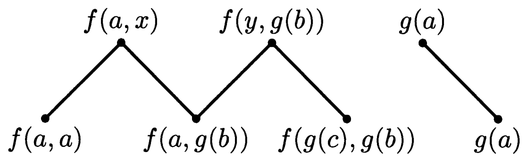
Variable subterms (z, z' in this example) are not involved in the compact bipartite graph construction. They are instantiated later in the substitution construction phase described in section 3.4.

Let us consider the subject:

$$s = F(f(a, a), f(a, g(b)), f(g(c), g(b)), g(a)).$$

The matching automaton, presented in Figure 2 and implemented by Program 2, is successively applied on $f(a, a)$, $f(a, g(b))$, $f(g(c), g(b))$ and $g(a)$ to find the pairs: $[f(a, a), p_{1,1} = p_{2,1}]$, $[f(a, g(b)), p_{1,1} = p_{2,1}]$, $[f(a, g(b)), p_{2,2}]$, $[f(g(c), g(b)), p_{2,2}]$ and $[g(a), p_{1,2}]$.

The matching automaton is used to build the following compact bipartite:



The compact bipartite graph is exploited as follows. A rule has to be selected in order to normalise the subject, for instance $F(z', f(a, x), f(y, g(b))) \to F(z', r_2)$. The bipartite graph that should have been created by the general method can be easily constructed by extracting edges that join $f(a, x)$ and $f(y, g(b))$ to subject subterms, which gives the 'classical' bipartite graph presented in the right part of Figure 4. To check if the selected rule can be applied, a maximum bipartite matching has to be

Compiling the deterministic discrimination tree

```
int match_subterm_F(struct term *subject, int *mask) {
  switch(getSymb(subject)) {
  case code_g: successor_g=subject->subterm[0];
    switch(getSymb(successor_g)) {
    case code_a:
      mask[nb_bit++]=1;
      break;
    }
    break;
  case code_f: successor_f=subject->subterm[0];
    switch(getSymb(successor_f)) {
    case code_a: successor_a=subject->subterm[1];
      switch(getSymb(successor_a)) {
      case code_g: successor_g=successor_a->subterm[0];
        switch(getSymb(successor_g)) {
        case code_b:
          mask[nb_bit++]=0;
          mask[nb_bit++]=2;
          break;
        default: goto label7;
        }
        break;
      default:
      label7:
        mask[nb_bit++]=0;
      }
 ...
  }
  return nb_bit;
}
```

Program 2: *This function implements a deterministic discrimination tree for a set of patterns. Given a ground term* `subject`*, the term is traversed according to the constructors (* `code_f`*,* `code_g` *or* `code_a` *for example) that occur in it. When a leaf is reached, the numbers assigned to the patterns that match the subject are stored in a vector* `mask`*. This vector is then used to build the compact bipartite graph.*

found. The bipartite graph has three solutions:

$$\begin{array}{rll} S = & \{[f(a,a), f(a,x)], & [f(a,g(b)), f(y,g(b))]\} \\ S' = & \{[f(a,a), f(a,x)], & [f(g(c),g(b)), f(y,g(b))]\} \\ S'' = & \{[f(a,g(b)), f(a,x)], & [f(g(c),g(b)), f(y,g(b))]\} \end{array}$$

The given example of compact bipartite graph is represented by only three bit vectors: `1100`, `0110` and `0001`. The first one: `1100`, means that the corresponding pattern $f(a,x)$ matches the two first subterms: $f(a,a)$ and $f(a,g(b))$, and similarly for the other ones. Extracting the bipartite graph is done by only selecting bit vectors associated to $f(a,x)$ and $f(y,g(b))$: `1100` and `0110`.

In this example, the bipartite graph has three solutions $\{S, S', S''\}$. With the first solution S, the rewrite rule $F(z', f(a,x), f(y, g(b))) \rightarrow F(z', r_2)$ can be applied, by instantiating z' to unbounded terms $(f(g(c), g(b))$ and $g(a)$.

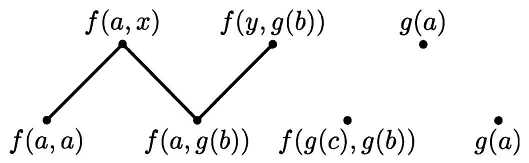
3.3.3 Eager matching

An AC matching problem usually has more than one solution. However, for applying a rule without conditions or whose conditions do not depend on a variable that occurs under an AC function symbol of the left-hand side, there is no need to compute a set of solutions: the first match which is found is used to apply the corresponding rule. Those rules are called *eager rules*. This remark leads us to further specialise our algorithm to get an *eager matching* algorithm, which tries to find a match for eager rules, as fast as possible. The idea consists of incrementally building the whole compact bipartite graph and adding to each step a test to check whether a bipartite graph associated to an eager rule has a solution. This test is not performed for non-eager rules and no check is necessary on a bipartite graph if no modification has occurred since the last applied satisfiability test (i.e. no edge has been added). Using those two remarks, the number of checked bipartite graphs is considerably reduced.

Let us consider our running example. With the first method presented above (called the main algorithm), four matching attempts were done to completely build the compact bipartite graph (corresponding to its five edges). Only after this building phase, bipartite graphs are extracted and solved. Assuming that subterms are matched from left to right, it is sufficient to match only two subterms (with the eager algorithm), to find the first suitable solution:

$$S = \{[f(a,a), f(a,x)], [f(a, g(b)), f(y, g(b))]\}.$$

This solution is found as soon as the following partial compact bipartite graph is built:



In practice, eager matching considerably reduces the number of matching attempts and there is only a small time overhead, due to the test, when no eager rule is applied. In the main algorithm, the number of matching attempts is linear in the number of subterms of the subject. In the eager algorithm, this number also depends on the pattern structure. Using two examples (Prop and Bool3) described in section 6.1, some experimental results that compare the number of matched subterms with and without the eager algorithm are given in Table 1 on the facing page.

Note, however, that eager matching is not compatible with the concept of priority

Table 1. *Number of matched subterms with and without the eager algorithm.*

No. of matched subterms	Main algorithm	Eager algorithm	Gain
Prop	50,108	32,599	17,509 (35%)
Bool3	44,861	8,050	36,811 (82%)

rewriting, 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.[2]

3.4 Construction of substitutions

Once matching is performed, the remaining tasks are to instantiate variables and to build the reduced term. In the construction of the reduced term, 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 Vittek (1996) for the syntactic case. Similar techniques have been developed in our compiler, but we do not discuss them here, and rather focus on the construction of substitutions. At this stage of description of the compiler, two problems have to be addressed: how to instantiate the remaining variables in AC patterns? How to optimise the substitution construction?

3.4.1 Variable instantiation

Variables that occur in patterns just below an AC function symbol 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 linear Diophantine system. Several cases can be distinguished according to the syntactic form of patterns.

- For $F(x_1, t_1, \ldots, t_n)$, once $t_1, \ldots, t_n$ are matched, all the unmatched subject subterms are captured by x_1.
- For $F(x_1, x_2^{\alpha_2}, t_1, \ldots, t_n)$, let us first consider the case where $\alpha_2 = 1$. 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.

  If $\alpha_2 > 1$, once $t_1, \ldots, t_n$ are matched, one tries to find in all possible ways α_2 identical remaining subjects to match x_2 and then, all the remaining unmatched subject subterms are captured by x_1.

Consider our running example, and the rule $F(z', z, f(a, x), g(a)) \to F(z', r_1)$ **if** $z = x$. Once matching have been performed, and the two solutions for x (a and $g(b)$) have

[2] In section 5, we introduce the strategy constructors (called first and first_one) that apply rules in a specific order: in this case, the eager matching algorithm cannot be used.

been found, the variables z and z' can be instantiated to $F(f(a,g(b)), f(g(c), g(b)))$ or $F(f(a,a), f(g(c), g(b)))$ or subterms directly under the two F symbols. The condition $z = x$ is never satisfied with those substitutions, so application of this rule fails.

3.4.2 *Compiling the substitution construction*

In the syntactic case, the matching substitution is easily performed by the discrimination net, since there is at most one solution. In the AC case, there may be many different instantiations for each variable. It would be too costly to store them in a data structure for possible backtracking. Furthermore, the construction of this dynamic data structure is not necessary when the first selected rule is applied, because all computed substitutions are deleted. Our approach consists of computing the substitution only when a solution of the bipartite graph is found. For each subterm $p_{j,k}$, variable positions are known at compile time and used to construct an access function $access\_p_{j,k}$ from terms to lists of terms. This function takes a ground term as argument and returns the list of instances of variables of $p_{j,k}$ as a result. Given $S_j = \{[s_i, p_{j,k}]\}$ a solution of BG_j, the set $I_j = \{access\_p_{j,k}(s_i) \mid [s_i, p_{j,k}] \in S_j\}$ of variable instantiations can be computed.

Given the rule $F(z', f(a,x), f(y,g(b))) \to F(z', r_2)$, the functions $access\_f(a,x)(t) = t_{|2}$ and $access\_f(y,g(b))(t) = t_{|1}$ are defined. Starting from $S_2 = \{[f(a,a), f(a,x)], [f(a,g(b)), f(y,g(b))]\}$, the set $I_2 = \{a, a\}$ is easily computed, and we get the substitution $\sigma = \{x \mapsto a, y \mapsto a\}$. An implementation of these access functions is described in Program 3.

Compiling the substitution construction

```
void variable_extract_F(struct term *subject, int id_pattern,
                        struct term *substitution[], int *index) {
  switch(id_pattern) {
  case 0: /* f(a,x) */
    substitution[*index]=subject->subterm[1];  (*index)++;
    break;
  case 1: /* g(a) */
    break;
  case 2: /* f(y,g(b)) */
    substitution[*index]=subject->subterm[0];  (*index)++;
    break;
  }
}
```

Program 3: *Given a number of pattern* `id_pattern` *(0, 1 or 2 on this example) and a ground term* `subject`*, this function retrieves and store (in the array* `substitution`*) the instances of variables that appear in the pattern.*

3.5 Handling other patterns

To handle rules whose patterns are not in C_2, a program transformation is applied. It transforms these rules into equivalent ones whose left-hand sides are in the class C_2.

Any rule can be transformed into a conditional rule with matching conditions and satisfying our pattern restrictions. The transformation preserves the semantics in the sense that a term is reducible by the initial rule if and only if it is reducible by the transformed rule.

Let l be a left-hand side of rule which does not belong to C_2, and Λ be an abstraction function that replaces non-variable subterms of l, say u_j, by new variables, say x_j, in such a way that $l' = \Lambda(l)$ is in the class C_2. Let k be the number of abstracted subterms. The new rule

$$\begin{aligned} l' \to r \quad & \textbf{where } u_1 := x_1 \\ & \vdots \\ & \textbf{where } u_k := x_k \end{aligned}$$

is equivalent to $l \to r$.

When using such a transformation approach, the efficiency of the resulting system partially depends on the one-to-one AC matching algorithm used for the matching conditions: simple rules that belong to the presented pattern classes are efficiently compiled, and complex rules that were not in C_2 are transformed and compiled.[3]

Example 2 *Let $\cup$, Eq be two AC operators,* e, solve *and* $\mathsf{simplify}$ *be three syntactic operators, and $r(x_1, x_2, x_3)$ any term where the variables x_1, x_2, x_3 occur. Let us consider the following rule:*

$$\mathsf{solve}(\mathsf{simplify}(x_1 \cup Eq(\mathsf{e}(x_2), \mathsf{e}(x_3)))) \to r(x_1, x_2, x_3)$$

A transformation has to be applied because the left-hand side of the rule does not belong to C_2. Let $\Lambda = \{\mathsf{e}(x_2) \mapsto y_2, \mathsf{e}(x_3) \mapsto y_3\}$ be the abstraction function. The following rule now belongs to the class C_2:

$$\begin{aligned} \mathsf{solve}(\mathsf{simplify}(x_1 \cup Eq(y_2, y_3))) \to r(x_1, x_2, x_3) \quad & \textbf{\textit{where}}\ \mathsf{e}(x_2) := y2 \\ & \textbf{\textit{where}}\ \mathsf{e}(x_3) := y3 \end{aligned}$$

It is worth recalling that when computing such matching conditions, only one-to-one matching problems occur. If a pattern u_j contains an AC function symbol, a general one-to-one AC matching procedure, such as the one described in Eker (1995), is called. In the worst case, our 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 the **where** parts, helped by a full indexing for the topmost free function symbol layer. This is also a frequently implemented matching technique, used in Maude (Clavel *et al.*, 1996), for instance.

[3] In the current version of the ELAN compiler, we use the algorithm presented in Eker (1995), which is also used in C*i*ME (Marché, 1996) and in the ELAN interpreter.

4 Optimisations

Before concluding the compilation of the normalisation process, let us mention a few useful optimisations. As illustrated in section 6.1, these optimisations are essential to get an efficient AC normalisation procedure: without any optimisation it would not be possible to get results as good as those obtained by Maude, Brute (a Cafe-OBJ compiler) and the ELAN compiler.

4.1 Maintaining canonical forms

The compact bipartite graph construction (and thus the matching phase) assumes that both pattern and subject are in canonical form. Instead of re-computing 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 $s = F(s_1^{\alpha_1}, \dots, s_p^{\alpha_p})$, if an equivalent subterm s_i already exists, its multiplicity is incremented, else, the subterm t (which is in canonical form by construction) is inserted in the list $s_1^{\alpha_1}, \dots, s_p^{\alpha_p}$ at a position compatible with the chosen ordering. If t has the same AC top symbol F, a flattening step is done and the two subterm lists are merged with a merge sort algorithm.

Definition 4 *Let us define the function* mcf *taking as arguments two terms* $s = F(s_1^{\alpha_1}, \dots, s_p^{\alpha_p})$ *and* $t = G(t_1^{\beta_1}, \dots, t_m^{\beta_m})$ *in canonical form, as follows:*

- *case* $F \neq G$ *(s and t have different top symbol)*
  – *if there exists i in* $[1 \dots p]$ *such that* $s_i = t$ *the multiplicity* α_i *is incremented:*
  $\mathrm{mcf}(F(s_1^{\alpha_1}, \dots, s_p^{\alpha_p}), t) = F(s_1^{\alpha_1}, \dots, s_i^{\alpha_i+1}, \dots, s_p^{\alpha_p})$
  – *else, there exists i in* $[1 \dots p]$ *such that* $\forall j \leqslant i, s_j > t$ *and* $\forall j > i, t > s_j$*:*
  $\mathrm{mcf}(F(s_1^{\alpha_1}, \dots, s_p^{\alpha_p}), t) = F(s_1^{\alpha_1}, \dots, s_i^{\alpha_i}, t, s_{i+1}^{\alpha_{i+1}}, \dots, s_p^{\alpha_p})$
- *case* $F = G$ *(s and t have same top symbol)*
  $\mathrm{mcf}(F(s_1^{\alpha_1}, \dots, s_p^{\alpha_p}), t) = F(u_1^{\gamma_1}, \dots, u_k^{\gamma_k})$ *such that* $(u_1^{\gamma_1}, \dots, u_k^{\gamma_k})$ *is the merged sort (without multiple occurrences) of* $(s_1^{\alpha_1}, \dots, s_p^{\alpha_p})$ *and* $(t_1^{\beta_1}, \dots, t_m^{\beta_m})$

From the definition of mcf, it is easy to get the following result: let $s = F(s_1^{\alpha_1}, \dots, s_p^{\alpha_p})$ and t be two terms in canonical form. The function mcf applied to s and t returns the canonical form of $F(s_1^{\alpha_1}, \dots, s_p^{\alpha_p}, t)$. As a consequence, the canonical form of a term can be obtained by a bottom-up construction using the mcf function.

4.2 Normalised substitutions

In the case of the leftmost-innermost reduction strategy, nested function calls are such that before a matching phase, each subterm is in normal form w.r.t. the rewrite rule system. In syntactic rewriting, when a pattern p matches a ground term s, all variables of p are assigned to a subterm of s which is irreducible by construction.

This is no longer the case in AC rewriting because variables that appear directly under an AC top symbol may be instantiated to a reducible combination of irreducible subterms. For instance, in our running example, the variable z can be

Table 2. *Number of performed normalisations with and without the 'colour' feature.*

No. of normalisations	Main algorithm	With colour	Gain
Prop	13,976	3,111	10,865 (78%)
Bool3	5,599	2,968	2,631 (47%)

instantiated to $F(f(a, g(b)), f(g(c), g(b)))$ which is reducible by the rule $F(z', f(a, x), f(y, g(b))) \to F(z', r_2)$.

To ensure that instances of variables occurring immediately under an AC top function symbol are irreducible, these instances 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 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. As illustrated in section 6.1, in practice, the number of applied rules is significantly reduced.

4.3 Using colors to avoid unnecessary normalised substitutions

Normalising reducible instances of variables considerably reduces the number of applied rules, but re-normalising a term already in normal form involves extra work that introduces an overhead. Let us note that all subterms of the subject are in normal form by construction and that an instance of a variable that appears immediately under an AC top function symbol is irreducible if this instance is a subterm of an irreducible term. This remark leads to further improve the algorithm used to build ground reduced terms:

- whenever a term rooted by an AC symbol is built with the mcf function, a different 'colour' is assigned to all its immediate subterms.
- whenever an irreducible term rooted by an AC symbol is reached by normalisation, a same 'colour' is assigned to all its immediate subterms.
- in the algorithm that computes the canonical form of a term, if α identical subterms t appear, they are replaced by a single instance of the subterm with multiplicity α and a special colour, say *bicolour*, is assigned to this subterm.

Coming back to the original problem of checking whether the term s assigned by the matching substitution to the variable x is irreducible, it is now possible to inspect the colours of immediate subterms of s: if all subterms have the same colour and none of them is *bicolour*, the term s is a subterm of the subject (irreducible by construction), so it is not necessary to normalise s again.

Using two examples (Prop and Bool3) described in section 6.1, some experimental results that compare the number of performed normalisations with and without the 'colour' feature are given in Table 2.

The compilation of the whole normalisation process, including the previous optimisations, is sketched on our running example in Program 4.

Compiling the normalisation process

```
struct term* normalise_F(struct term *subject ) {
  struct term *res;
  match_state *ms=NULL;
  /* Begin syntactical matching */
  bitSet32_set(mask32,0);
  bitSet32_set(mask32,1);
  /* Begin AC matching */
  MS_init(&ms, match_subterm_F, pattern_list_F);
  ...
  if(bitSet32_get(mask32,1)) {
      struct term *substitution[3];
      /* lhs: F(z',f(y,g(b)),f(a,x)) */
      substitution_build(subject,ms,substitution,variable_extract_F,1);
      /* rhs: F(z',h(x,y)) */
      if(!isMonoColor(substitution[0])) {
        substitution[0]=normalise_F( substitution[0] );
      }
      TERM_ALLOC(node_h,code_h);
      node_h->subterm[0] = substitution[2];                  // x
      node_h->subterm[1] = substitution[1];                  // y
      TERM_ALLOC(node_F,code_F);
      term_add_cf_term_color(node_F,substitution[0],color1); // z'
      term_add_cf_term_color(node_F,node_h         ,color2); // h(x,y)
      res = normalise_F( node_F );
      goto end;
    } else { ... }
    ...
match_fail:
  res=subject;
end:
  return res;
}
```

Program 4: *This figure shows how the whole normalisation process is compiled: after the syntactic matching phase (trivial in this example), the compact bipartite graph is built by the* `MS_init(&ms, match_subterm_F, pattern_list_F)` *instruction. Then, for a selected pattern (the second one* $F(z', f(y, g(b)), f(a, x))$ *for example), the substitution is built by* `substitution_build(subject, ms, substitution, variable_extract_F, 1)`. *As described in sections 4.2 and 4.3, a test (* `if(!isMonoColor(substitution[0]))`*) is performed to check if the instance is reducible or not. The last component of the function consists of building the reduced term and computing its ordered normal form with the function* `term_add_cf_term_color`, *as described in section 4.1.*

5 Determinism analysis

Let us now turn to the problem of non-determinism. The fact that a computation may have several results can be taken into account either by introducing explicitly sets of results or by a backtracking capability to enumerate the elements of this set. We adopt here the second approach and we introduce strategy constructors to specify whether a function call returns several, at least one or only one result.

For implementation of backtracking, 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 non-deterministic capabilities provide similar functions: for instance world+ and world- in Claire (Caseau & Laburthe, 1996), try and retry in WAM (Warren, 1983; Aït-Kaci, 1990), onfail, fail, createlog and replaylog in the Alma-0 Abstract Machine (Partington, 1997; Apt & Schaerf, 1997). Following Vittek (1996), two flow control functions, setChoicePoint and fail, have been implemented in assembly language. The setChoicePoint function sets a choice point, and the computation goes on. The fail function performs a jump into the last call of setChoicePoint. These functions can remind the pair of standard C functions `setjmp` and `longjmp`. However, the `longjmp` can be used only in a function called from the function setting `setjmp`. The two functions setChoicePoint and fail do not have such a limitation. Their implementation is described in Moreau (1998).

To take into account sets of results, we use the concept of strategy: a strategy is a function which, when applied to an initial term, returns a set of possible results (more precisely a multiset of results). The strategy fails if the set is empty. To precisely define how sets of results are handled, we introduce the following strategy constructors.

- A labelled rule is a primal strategy. The result of applying a rule labelled lab on a term t returns a multiset of terms. This primal strategy fails if the multiset of resulting terms is empty.
- Two strategies can be concatenated by the symbol ';', i.e. the second strategy is applied on all results of the first one. $S_1 ; S_2$ denotes the sequential composition of the two strategies. It fails if either S_1 fails or S_2 fails. Its results are all results of S_1 on which S_2 is applied and gives some results.
- dc($S_1, \ldots, S_n$) chooses one strategy S_i in the list that does not fail, and returns all its results. This strategy may return more than one result, or fails if all sub-strategies S_i fail.
- first($S_1, \ldots, S_n$) chooses the first strategy S_i in the list that does not fail, and returns all its results. Again, this strategy may return more than one result, or fails when all sub-strategies S_i fail.
- dc_one($S_1, \ldots, S_n$) chooses one strategy S_i in the list that does not fail, and returns its first result. This strategy returns at most one result or fails if all sub-strategies fail.
- first_one($S_1, \ldots, S_n$) chooses the first strategy S_i in the list that does not fail, and returns one of its first results. This strategy returns at most one result or fails if all sub-strategies fail.

- $\mathsf{dk}(S_1, \dots, S_n)$ chooses all strategies given in the list of arguments and for each of them returns all its results. This multiset of results may be empty, in which case the strategy fails.
- The strategy id is the identity that does nothing but never fails.
- fail is the strategy that always fails and never gives any result.
- $\mathsf{repeat}(S)$ applies repeatedly the strategy S until it fails and returns the results of the last unfailing application. This strategy may return more than one result but can never fail because zero applications of S is possible: in this case the initial term is returned.
- The strategy $\mathsf{iterate}(S)$ is similar to $\mathsf{repeat}(S)$ but returns all intermediate results of repeated applications.

The strategy constructors introduced here are quite close to other tactics languages used on proof systems designed in the LCF style (Plotkin, 1977; Gordon *et al.*, 1979), such as Isabelle (Paulson, 1994). They have been chosen to express main control constructions: concatenation, iteration and search. All these constructors are part of the ELAN language, and have been useful to design in ELAN theorem proving and constraint solving tools.

From now on, let us consider that not only rules but also strategies can be applied on terms. $[S](t)$ denotes the application of the strategy S on the term t that produces a multiset of results. Indeed, a rule itself may call a strategy in its matching conditions that are now of the form **where** $p := [S](t)$.

It may be interesting to remark that ELAN does not provide any strategy constructor for negation, simply because it may be expressed using others constructors. Let us consider the strategy $S' = \mathsf{first}(S;\mathsf{fail},\mathsf{id})$. This strategy S' fails when S succeeds and S' succeeds when S fails. This example also illustrates the use of the id constructor.

Example 3 *We have shown in the introduction a simple rule that removes an element from a set, and returns the element and the new set. This rule is now labelled by* extract*:*

$$[\mathit{extract}] \quad (i) \cup S \quad \to \quad [i, S]$$

The strategies $\mathit{dk}(\mathsf{extract})$*,* $\mathit{dc}(\mathsf{extract})$ *and* $\mathit{first}(\mathsf{extract})$ *are all equivalent in this case and enumerate the elements of the set. On the other hand,* $\mathit{dc\_one}(\mathsf{extract})$ *and* $\mathit{first\_one}(\mathsf{extract})$ *produce only one result which corresponds to the first match which is found.*

To implement the N-queens problem, we need two more rules, where set, sol, s_1, p_1 are variables, and check a predicate that is satisfied when a queen can be set in position p_1 without being attacked by a previously partial solution sol:

$$\begin{array}{llll} [\mathit{queens}] & state(set, sol) & \to & state(s_1, p_1 \cdot sol) \\ & & & \textbf{\textit{where}}\ [p_1, s_1] := [\mathit{dk}(\mathit{extract})](set) \\ & & & \textbf{\textit{if}}\ check(1, p_1, sol) \\ [\mathit{final}] & state(\emptyset, sol) & \to & sol \end{array}$$

We also need to define a strategy that controls the application of these two rules:

$$qStrat \quad \rightarrow \quad \mathsf{repeat^*}(\mathsf{dk}(\mathsf{queens}));\mathsf{final}$$

The most important statement is: ***where*** $[p_1, s_1]$:= $[\mathsf{dk}(\mathsf{extract})](set)$.

When applied on a set of integers, for instance set $= (1) \cup (2) \cup \ldots \cup (8)$*, the strategy* $\mathsf{dk}(\mathsf{extract})$ *non-deterministically applies the rule* extract*, and non-deterministically chooses an assignment for i and S. In our case, there are eight solutions. Each assignment of i and S is stored in a pair* $[p_1, s_1]$*, then the check predicate is evaluated. When it is satisfied, the position* p_1 *is added to the partial solution l, and the* queens *rule is applied again, thanks to the* $\mathsf{repeat^*}(\mathsf{dk}(\mathsf{queens}))$ *strategy. When the set of potential positions is empty (represented by the constructor* $\emptyset$*), each queen has a compatible assignment and the* final *rule is applied to return a solution of the N-queens problem. By using* $\mathsf{dk}(\mathsf{queens})$ *in the qStrat strategy, the search space is fully explored and we obtain the complete set of solutions to the N-queens problem. Using instead* dc_one(queens) *would result in searching for only one solution.*

To efficiently deal with strategies and this more general notion of rules, our compiler incorporates a static analysis phase that annotates every rule and strategy in the program with its determinism. This determinism information is used in later phases of the compiler: the matching phase, various optimisations on the generated code and detection of non termination. The determinism analysis runs after the type-checking analysis, the transformation of rules to fit into the restricted class of patterns described in section 3.2, and the linearisation of patterns (left-hand sides of rewrite rules), but before the many-to-one AC matching compilation phase.

To facilitate the determinism analysis, we introduce four primitive operators that allow us to classify the cases according to two different levels of control.

Controlling the number of results: given a rewrite rule or a strategy,

- the one operator builds a strategy that returns at most one result;
- the all operator builds a strategy that returns all possible results of the strategy or the rule.

Controlling the choice mechanism: given a list of strategies (possibly reduced to a singleton),

- the select_one operator chooses and returns a non-failing strategy among the list of strategies;
- the select_first operator chooses and returns the first (from left to right) non-failing strategy among the list of strategies;
- the select_all operator returns all unfailing strategies.

In the current version of ELAN, these five primitives are hidden from the user and are internally used to perform the determinism analysis. However, all strategy constructors dk, dc, first, dc_one and first_one can be expressed using these primitives, using the following axioms, where S_i stands for a rule or a strategy:

$\mathsf{dk}(S_1, \ldots, S_n)$	$=$	$\mathsf{select\_all}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n))$
$\mathsf{dc}(S_1, \ldots, S_n)$	$=$	$\mathsf{select\_one}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n))$
$\mathsf{first}(S_1, \ldots, S_n)$	$=$	$\mathsf{select\_first}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n))$
$\mathsf{dc\_one}(S_1, \ldots, S_n)$	$=$	$\mathsf{select\_one}(\mathsf{one}(S_1), \ldots, \mathsf{one}(S_n))$
$\mathsf{first\_one}(S_1, \ldots, S_n)$	$=$	$\mathsf{select\_first}(\mathsf{one}(S_1), \ldots, \mathsf{one}(S_n))$

Note that dk, dc and first operators are equivalent if they are applied on a unique argument: $\mathsf{dk}(S) = \mathsf{dc}(S) = \mathsf{first}(S) = S$.

5.1 Determinism

For each strategy, a determinism information is inferred according to the maximum number of results it can produce (one or more than one) and whether or not it can fail before producing its first result. We adopt the same terminology for determinism as in Mercury (Henderson *et al.*, 1996b; Henderson *et al.*, 1996a):

- if the strategy has exactly one result, its determinism is *deterministic* (det);
- if the strategy can fail and has at most one result, its determinism is *semi-deterministic* (semi);
- if the strategy cannot fail and has more than one result, its determinism is *multi-result* (multi);
- if the strategy can fail and may have more than one result, its determinism is *non-deterministic* (nondet);
- if the strategy always fail, i.e. has no result, its determinism is *failure* (fail).

A partial ordering on this determinism is defined as follows:

$$\mathsf{det} < \mathsf{semi}, \mathsf{multi} < \mathsf{nondet}$$

and intuitively corresponds to an inclusion ordering on the intervals which the number of results belongs to:

$$[1, 1] < [0, 1], [1, +\infty[< [0, +\infty[$$

The algorithm for inferring the determinism of strategies uses two operators *And* and *Or* that intuitively correspond to the composition and the union of two strategies (the union of two strategies is defined by the union of their results). Their values given in the following tables should be clear from the semantics given to the different determinisms. For instance, a conjunction of two strategies is semi-deterministic if any one can fail and none of them can return more than one result ($\mathit{And}(\mathsf{det}, \mathsf{semi}) = \mathit{And}(\mathsf{semi}, \mathsf{det}) = \mathit{And}(\mathsf{semi}, \mathsf{semi}) = \mathsf{semi}$). These values can be also computed with operations on boolean variables as, for instance, in Henderson *et al.* (1996a, b).

And	**det**	**semi**	**multi**	**nondet**	**fail**
det	det	semi	multi	nondet	fail
semi	semi	semi	nondet	nondet	fail
multi	multi	nondet	multi	nondet	fail
nondet	nondet	nondet	nondet	nondet	fail
fail	fail	fail	fail	fail	fail

Or	**det**	**semi**	**multi**	**nondet**	**fail**
det	multi	multi	multi	multi	det
semi	multi	nondet	multi	nondet	semi
multi	multi	multi	multi	multi	multi
nondet	multi	nondet	multi	nondet	nondet
fail	det	semi	multi	nondet	fail

5.2 *Determinism inference*

The algorithm for inferring the determinism is presented here in three steps: for a strategy, it uses the decomposed form of the strategy into the primitives introduced above. For a rule, it analyses the determinism of the matching conditions. Finally, it deals with the recursion problem due to the fact that strategies are built from rules and that rules call strategies in their matching conditions.

5.2.1 *Strategy* detism *inference*

The detism of a strategy is inferred from its expression using one, all, select_one and select_all.

- detism(one(S)) = semi if S is a rewrite rule, since application of a rewrite rule may fail; otherwise,
  $$\text{detism(one}(S)) = \begin{cases} \text{det} & \text{if detism(S) is det or multi} \\ \text{semi} & \text{if detism(S) is semi or nondet} \end{cases}$$
- detism(all(S)) = And(semi, detism(S)) if S is a rewrite rule, since application of a rewrite rule may fail; otherwise, detism(all(S)) = detism(S)
- $$\text{detism(repeat(S))} = \begin{cases} \text{det} & \text{if detism(S) is det or semi} \\ \text{multi} & \text{if detism(S) is multi or nondet} \end{cases}$$
  The repeat operator cannot fail because zero application of the strategy is allowed. Note that if S cannot fail, the repeat construction cannot terminate.
- detism(iterate(S)) = multi. The iterate operator cannot fail either. In general, it returns more than one result because all intermediate steps are considered as results. If S cannot fail, the iterate construction cannot terminate, but this is quite useful to represent infinite data structures, like infinite lists.

- detism($S_1 ; S_2$) = And(detism(S_1), detism(S_2)).
- detism(select_one($S_1, \ldots, S_n$)) = And(detism(S_1), ..., detism(S_n))
- detism(select_all($S_1, \ldots, S_n$)) = Or(detism(S_1), ..., detism(S_n))

5.2.2 *Rule detism inference*

Inferring the determinism of a rewrite rule R consists of analysing the determinism of its matching conditions:

- Let us first consider a matching condition **where** $p := c$ where c does not involve any strategy. The normalisation of c (with unlabelled rules) cannot fail. If p does not match the normalised term, the current rule cannot be applied, but this does not modify the detism of the rule. Such a condition is usually said to be deterministic (det is a neutral element for the *And* operator). The only different situation is when a variable of c occurs in the left-hand side of the rule or in a pattern of a previous matching condition with AC function symbols: if this variable is involved in an AC matching problem, it may have several possible instances, thus, an application of the rule may return more than one result. The matching condition is said to be multi.
- Let us now consider a matching condition **where** $p := [S](t)$ involving a strategy call. Then the matching condition has in general the determinism of the strategy S, except as before when a variable of t occurs in the left-hand side of the rule or in a pattern of a previous matching condition with AC function symbols: the detism of the matching condition is multi or nondet, and is computed as And(multi, detism(S)).

The determinism of the rewrite rule R is the conjunction (*And* operation) of the inferred determinisms of all its matching conditions.

5.2.3 *Recursion problem*

In general, strategy definitions may be (mutually) recursive. So the detism of a strategy may depend on itself. A similar problem arises in logic programming for finding the determinism of a predicate (Sawamura & Takeshima, 1985). To avoid non-termination of the determinism analysis algorithm, when the detism of a strategy depends on itself, a default determinism is given. On the strategy constructors, this default corresponds to the maximum of the determinism in the ordering $<$ that the strategy can have and is given in the following table:

constructor	**one**	**all**	**repeat**	**iterate**	**;**
default detism	semi	nondet	multi	multi	nondet

To refine this brute force approximation we plan to explore a fixpoint technique similar to the one used in Sictus Prolog (Sahlin, 1991).

5.3 Impact of determinism analysis

The determinism analysis enables us to design better compilation schemes for det or semi strategies. With this approach, the search space size, the memory usage, the number of necessary choice points, and the time spent in backtracking and memory management can be considerably reduced. We can also take benefit from the determinism analysis to improve the efficiency of AC matching and to detect some non-terminating strategies.

Several optimisations can be done to improve the backtracking management:

- When compiling a set of rules whose matching conditions are deterministic, no choice point is needed because no backtracking can occur between the matching conditions.
- When compiling a set of deterministic rules with some non-deterministic matching conditions, some choice points are needed to handle the backtracking. Note that all set choice points can be removed when the rule is applied, because at most one result is needed. For instance, when searching only one solution in a problem where several choice points are needed, one can delete them after finding the first solution.
- When dealing with non-deterministic strategies and the repeat constructor, a lot of choice points have to be set, because the strategy is recursively called in all branches of the computation space. The situation can be depicted as follows, where the bullet represents a set choice point.

  $$\begin{tikzcd}[column sep=small] & {} & & {} & & {} & & {} & \\ t \bullet \arrow[ur, "S"] \arrow[r, "S"] \arrow[dr, "S"'] & t_1 \bullet \arrow[ur, "S"] \arrow[r, "S"] \arrow[dr, "S"'] & \cdots \bullet \arrow[ur, "S"] \arrow[r, "S"] \arrow[dr, "S"'] & t_n \bullet \arrow[ur, "S"] \arrow[r, "S"] \arrow[dr, "S"'] & \mathsf{fail} \\ & {} & & {} & & {} & & {} & \end{tikzcd}$$

  One choice point per step is needed, and when a failure occurs, one choice point only is deleted and the process goes on.

  This is no longer the case when compiling a strategy repeat(S) where S is det or semi. The compilation scheme then consists of setting a single choice point and trying to apply the strategy S as many times as possible. Each time the strategy S is applied, the resulting term is saved in a special variable $lastTerm$. When a failure occurs, the choice point is deleted and the saved term $lastTerm$ is returned. The situation is depicted as follows:

  $$\bullet\, t \longrightarrow^{S} t_1 \longrightarrow^{S} \cdots \longrightarrow^{S} t_n \longrightarrow \mathsf{fail}$$

To illustrate this last point, let us consider the following example.

Example 4 *Let us consider a simple modeling of a game: a pawn on a chessboard can move in several directions (see Figure 5), each of them corresponding to one labelled rule d_i.*

Exploring all possibilities of moves for this pawn in one step can be expressed by a strategy `move` $\rightarrow$ dk($d_1, \ldots, d_n$)*, where $d_1, \ldots, d_n$ are basic moves. Once a move has been performed, in some situation, it may be considered as a definitive choice and the search space related to all other moves is forgotten. This is performed via a strategy* dc_one(`move`)*. To iterate this process, the strategy* repeat(dc_one(`move`)) *repeatedly*

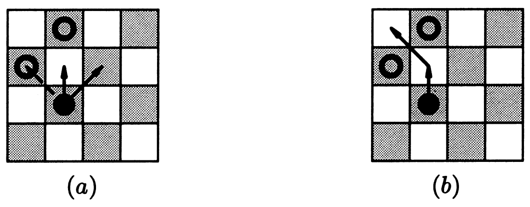


Fig. 5. (a) By applying `dk(move)` the pawn can move in three possible directions. (b) An external square can be reached if the strategy `repeat(dc_one(move))` is applied.

moves a given pawn up to a failure: in this example, a pawn cannot move when an external square is reached.

This simple game is an example of situation where the last presented impact of determinism analysis is crucial: by reducing to one the number of set choice point, it considerably improves the efficiency and reduces the memory needed.

Other advantages of determinism analysis are related to the rewriting process. To improve efficiency of rewriting, a well-known idea is to reuse parts of left-hand sides of rules to construct the right-hand sides (Didrich *et al.*, 1994; Vittek, 1996). This technique avoids memory cell copies and reduces the number of allocations. Unfortunately, the presence of non-deterministic strategies and rules limits its applicability, because backtracking requires access to structures that would otherwise be reused. The determinism information is then used to detect cases where reusing is possible.

The determinism analysis is also important to design more efficient AC matching algorithms: when a rule is deterministic, only the first match which is found is needed to apply a rewrite step. This remark has to be related to the design of the *eager matching* algorithm, described in section 3.3.3, which avoids building the whole compact bipartite graph before solving it. Experiments show a reduction of the number of matching attempts up to 50%, which significantly improves the overall performance of the system.

Finally, the determinism analysis is also useful to detect some non-terminating strategies, such as a strategy repeat(S), where S never fails. Detecting this non-termination problem at compile time allows the system to give a warning to the programmer and can help in improving the strategy design.

6 A compiler for ELAN

The techniques described in the previous sections have been implemented in the ELAN system (Kirchner *et al.*, 1995). ELAN provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. It offers a natural and simple logical framework for the combination of the computation and deduction paradigms, supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures, and it offers a modular framework for studying their combination.

ELAN takes from functional programming the concept of abstract data types and the function evaluation principle based on rewriting. In ELAN a rewrite rule may be labelled, may have boolean conditions introduced by the keyword **if**, and matching conditions introduced by the keyword **where**. The evaluation mechanism also involves backtracking since in ELAN, a computation may have several results. One of the original aspects of the language is that it provides a strategy language allowing the programmer to specify the control used during rule applications. This is in contrast to many existing rewriting-based languages where the term reduction strategy is hard-wired and not accessible to the designer of an application. The strategy language offers primitives for sequential composition, iteration, deterministic and non-deterministic choices of elementary strategies that are labelled rules. From these primitives, which correspond to the strategy constructors defined in section 5, more complex strategies can be expressed. In addition, the user can introduce new strategy operators and define them by rewrite rules. Evaluation of strategy application is itself based on rewriting. Moreover it should be emphasised that ELAN has logical foundations based on rewriting logic (Meseguer, 1992) and detailed in Borovanský *et al.* (1996, 1998a). So the simple and well-known paradigm of rewriting provides both the logical framework in which deduction systems can be expressed and combined, and the evaluation mechanism of the language.

The current version of ELAN includes an interpreter and a compiler written respectively in C++ and Java, a library of standard ELAN modules, a user manual and examples of applications. Among those, let us mention for instance the design of rules and strategies for constraint satisfaction problems (Castro, 1998), theorem proving tools in first-order logic with equality (Kirchner & Moreau, 1995; Cirstea & Kirchner, 1997), the combination of unification algorithms and of decision procedures in various equational theories (Ringeissen, 1997; Kirchner & Ringeissen, 1998). More information on the system can be found on the web site.[4]

A first ELAN compiler was designed and presented in Vittek (1996). Experimentations made clear that a higher-level of programming is achieved when some functions may be declared as associative and commutative. The new ELAN compiler has been implemented by the second author with approximatively 20,000 lines of Java. A runtime library has been implemented in C to handle basic terms and AC matching operations. This library contains more than 10,000 lines of code.

To give an intuition about the performance of the ELAN system compared to programming languages largely used in practice, we show in Table 3 on the next page the results of a brief comparison[5] with the Objective Caml (v 2.02) functional programming system and the GNU Prolog (v 1.1.2) logic programming system. We used the Fib benchmark which computes 100 times the Nth Fibonacci number, and the N-queens program illustrated in Example 3.

To complete this first comparison and to illustrate the power of our compilation techniques, it is interesting to compare ELAN to other systems implementing AC normalisation. As benchmarks, we consider below examples that make an heavy use of AC symbols and involve non-deterministic computations.

[4] http://www.loria.fr/ELAN.
[5] On a Sun Enterprise with Solaris 5.6.

Table 3. *Very small comparison of Objective Caml, GNU Prolog and* ELAN.

(Time in second)	OCaml	GNU Prolog	Elan
Fib(23)	0.67	12.08	1.12
Fib(25)	1.75	31.77	2.39
N-queens(10)	0.580	1.750	1.07
N-queens(12)	19	57	31.2

6.1 AC normalisation without strategies

In this section, we study four examples of rewriting rule systems, designed to represent different programming styles: the two first examples contain a few number of unconditional rules, the third one contains a large number of rules, and the last one contains some conditional rules that make AC matching more complex. From an execution point of view, different kinds of behaviour are experimented: normalisation of deep terms in which a lot of nested AC symbols occur, normalisation of large terms where many subterms appear under the same AC symbol, and computations that involve backtracking to extract all solutions of a given AC matching problem.

► The Prop example (Table 4) implements the two basic operators *and*, *xor*, that satisfy AC axioms, and four syntactic rules that transform *not*, *implies*, *or* and *iff* functions into nested calls of *xor* and *and*.

Table 4. *The* Prop *example*

$xor(x,\bot)$	$\to$	x
$xor(x,x)$	$\to$	$\bot$
$and(x,\top)$	$\to$	x
$and(x,\bot)$	$\to$	$\bot$
$and(x,x)$	$\to$	x
$and(x,xor(y,z))$	$\to$	$xor(and(x,y),and(x,z))$
$implies(x,y)$	$\to$	$not(xor(x,and(x,y)))$
$not(x)$	$\to$	$xor(x,\top)$
$or(x,y)$	$\to$	$xor(and(x,y),xor(x,y))$
$iff(x,y)$	$\to$	$not(xor(x,y))$

The benchmark consists of normalising the following term:

```
implies(and(iff(iff(or(a1,a2),or(not(a3),iff(xor(a4,a5),not(not(not(a6)))))),
not(and(and(a7,a8),not(xor(xor(or(a9,and(a10,a11)),a2),and(and(a11,xor(a2,iff(
a5,a5))),xor(xor(a7,a7),iff(a9,a4)))))))),implies(iff(iff(or(a1,a2),or(not(a3),
iff(xor(a4,a5),not(not(not(a6)))))),not(and(and(a7,a8),not(xor(xor(or(a9,and(
a10,a11)),a2),and(and(a11,xor(a2,iff(a5,a5))),xor(xor(a7,a7),iff(a9,a4))))))),
not(and(implies(and(a1,a2),not(xor(or(or(xor(implies(and(a3,a4),implies(a5,a6)),
or(a7,a8)),xor(iff(a9,a10),a11)),xor(xor(a2,a2),a7)),iff(or(a4,a9),xor(not(a6),
a6))))),not(iff(not(a11),not(a9))))))),not(and(implies(and(a1,a2),not(xor(or(
or(xor(implies(and(a3,a4),implies(a5,a6)),or(a7,a8)),xor(iff(a9,a10),a11)),xor(
xor(a2,a2),a7)),iff(or(a4,a9),xor(not(a6),a6))))),not(iff(not(a11),not(a9))))))
```

The normalisation of this term quickly produces a very large term that contains a lot of nested AC symbols. The expected result of the evaluation is $\top$.

► The Bool3 example (Table 5, designed by Steven Eker) implements computation in a three-valued logic where $+$ and $*$ are AC.

Table 5. *The Bool3 example.*

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

The benchmark consists of normalising the two following terms, and compare their normal forms:

and(and(and(a1,a2),and(a3,a4)),and(a5,a6))

and

not(or(or(or(not(a1),not(a2)),or(not(a3),not(a4))), or(not(a5),not(a6))))

► A rewrite system modulo AC for natural arithmetic, called Nat10, was presented in Contejean (1997). 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 used this rewrite system to compute the 16^{th} Fibonacci number.

► The Sum100 example (Table 6) uses the AC union operator: $\cup$ and three conditional rewrite rules in order to extract integers from a set and compute their sum ($\Sigma_{i=1}^{100} i$).

The benchmark consists of normalising the following term:

$$\mathsf{state}(\emptyset \cup \mathsf{set}(1) \cup \cdots \cup \mathsf{set}(100), \emptyset, 0)$$

The expected result of the evaluation is:

$$\mathsf{state}(\emptyset, \emptyset \cup \mathsf{set}(1) \cup \cdots \cup \mathsf{set}(100), 5050)$$

Table 6. *The Sum100 example.*

$x \in \emptyset$	$\rightarrow$	$\bot$
$x \in s$	$\rightarrow$	$\mathsf{check}(x \in' s) = \top$
$x \in' s \cup \mathsf{set}(y)$	$\rightarrow$	$\top$ **if** $x = y$
$\mathsf{check}(\top)$	$\rightarrow$	$\top$
$\mathsf{check}(x \in' s)$	$\rightarrow$	$\bot$
$\mathsf{state}(s_1 \cup \mathsf{set}(x), s_2, y)$	$\rightarrow$	error **if** $(x \in s_2) = \top$
$\mathsf{state}(s_1 \cup \mathsf{set}(x), s_2, y)$	$\rightarrow$	$\mathsf{state}(s_1, s_2 \cup \mathsf{set}(x), x + y)$ **if** $(x \in s_2) = \bot$

The first two benchmarks (Prop and Bool3) seem to be trivial because they contain a small number of rules. However, after several rewrite steps, the term to be reduced becomes very large (several MBytes) and contains a lot of AC symbols. It is not surprising to see a system spending several hours (on a fast machine) before finding the result, or running out memory.

The execution of the Nat10 example[6] does not generate such large terms, but the rewrite system contains a lot of rules. This illustrates the usefulness of many-to-one matching techniques.

The execution of the Sum100 example does not involve terms larger than the query (approximatively 100 subterms for this benchmark), but the rewrite system contains three conditional rewrite rules that involve AC matching. This benchmark tests the performance of the system when all solutions of a given AC matching problem have to be extracted. Let us consider, in the rewrite system Sum100, the rule:

$$\mathsf{state}(s_1 \cup \mathsf{set}(x), s_2, y) \rightarrow \mathsf{error}\ \textbf{if}\ x \in s_2$$

This rule applies only when the initial set contains at least two identical elements. When all elements are distinct, all possible instances of x have to be checked to verify that $x \in s_2$ is never satisfied.

Those four examples have been tested with Brute[7], C*i*ME (Marché, 1996), Maude (Clavel *et al.*, 1996), OBJ (Goguen & Winkler, 1988), RRL (Kapur & Zhang, 1988), Spike (Bouhoula & Rusinowitch, 1995) and the new ELAN compiler on a Sun Ultra-Sparc 1 (Solaris). The number of applied rewrite rules (rwr) and the time spent in seconds (sec) are given in Table 7. When for a given benchmark, no time is given in the second column (sec), this means that it has not been tested with the corresponding system. The tests were not exhaustively performed for theorem provers (C*i*ME, RRL and Spike) but the first experimental results clearly show that the problems considered are not easily solved by the rewriting techniques used in these provers.

[6] This last example was originally implemented in C*i*ME which is rather a theorem prover than a programming environment. To compute *Fib*(16), C*i*ME applies 10,599 rules in 16,400 seconds.

[7] Available at `ftp://ftp.sra.co.jb/pub/lang/CafeOBH/brute-X.Y.tar.gz`.

Table 7. *Experimental results – AC normalisation.*

	Prop		Bool3		Nat10		Sum100	
	rwr	sec	rwr	sec	rwr	sec	rwr	sec
CiME	-	-	?	> 24h	?	294	-	-
RRL	?	> 24h	?	> 4h[8]	-	-	-	-
Spike	?	> 24h	?	> 24h	-	-	?	> 24h
OBJ	12,837	1,164	?	> 24h	26,936	111	?	> 24h
Brute	23,284	1.78	34,407	2.25	26,648	0.36	177,595	6.25
Maude	12,281	0.47	4,854	0.15	25,314	0.17	177,252	16.77
ELAN	12,689	0.43	5,282	0.18	15,384	0.15	177,152	1.32

The aim of these experiments is not to find a 'winner' or to try to demonstrate that the ELAN compiler is faster than any other tool. It is rather to illustrate the usefulness of new developed techniques, such as interpreted greedy matching techniques for Maude, carefully designed brute force algorithms based on an abstract machine approach for Brute, and optimised many-to-one compilation techniques for ELAN. On the three first examples, Maude gives very interesting results. The last example tends to show that in a more realistic situation, the many-to-one approach may be an advantage when conditional rules are involved. What is important to note is that these new techniques allow us to design some improved tools that are significantly faster than several years ago. It is also important to remark that ELAN and Maude integrate some optimisations that reduce the number of needed applied rule: Maude is using a shared term approach to avoid redundant computations, whereas ELAN is using the normalised substitution approach described in section 4.2.

The statistics[9] presented in Table 8 give an overview of the time spent in specialised AC matching operations compared to the total execution time (the total is not equal to 100% because many other functions are involved in the normalisation process).

Table 8. *Time spent in specialised AC matching operations.*

	Prop	Bool3	Nat10	Sum100
CBG building	12.76%	14.59%	7.39%	5.71%
BG extraction	0.3%	0.39%	4.31%	0.1%
BG solving	3%	3.19%	9.38%	4.5%
substitution build	3.74%	3.77%	4.52%	38%
canonical form maintenance	24.1%	23.9%	1.7%	0.34%

On the four examples presented, less than 21% (resp. 16%, 18%, 21% and 10%) of the total execution time is spent in building and solving the bipartite graphs.

[8] More than 70 MBytes and 115 MBytes were respectively used.
[9] Measured with Quantify 3.1, (C) Rational Software.

When the number of AC rules increases (Nat10), the time spent in extracting and solving bipartite graphs slightly increases. The CBG construction is cheaper in Nat10 and Sum100 because the size of the subject is smaller, so the number of matching attempts is reduced. As illustrated by the Sum100 benchmark, the time spent in building substitutions mainly depends on the number of extracted solutions: in presence of conditional rewrite rules, all solutions and substitutions of AC matching problems have to be extracted and built. This explains why the time spent in building substitutions is larger than in other benchmarks, where only unconditional rules are applied. As expected, this time does not significantly depend on the complexity of the AC matching problem, nor on the theoretical number of solutions. This clearly shows the interest of our compiled substitution construction approach. To conclude, even with a well-suited term data structure and an optimised canonical form maintenance algorithm, the time spent in maintaining terms in canonical form can be really important on examples where very large terms are involved.

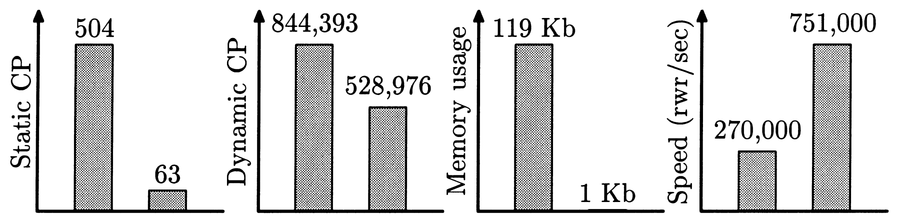
What is really interesting in the compiled approach is that we can combine interesting performances of the proposed AC rewriting process with extremely good results obtained with the syntactic compiler: up to 15,000,000 rewrite steps per second can be performed on simple specifications, and an average of 500,000 rwr/sec when dealing with large specifications that involve complex non-deterministic strategies. The best interpreters can perform up to 400,000 rwr/sec in the syntactic case. Compared with the compiled approach, this is a serious bottleneck when specifying a constraint solver or a theorem prover, for example.

6.2 Determinism analysis

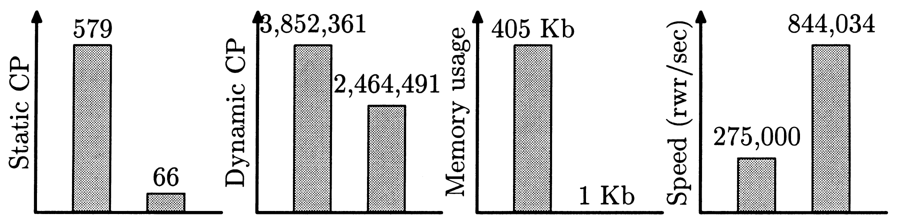
As mentioned in section 5.3, any ELAN program execution can benefit from the determinism analysis techniques described in this paper. However, to give a more concrete estimation of the practical impact of determinism analysis, let us consider experimental results obtained on a selection of programs in different areas of programming styles. Each program is executed twice: a first time without any optimisation, and a second time with the determinism analysis activated.

Figures show, for each program, the number of generated setChoicePoint instructions for creating a choice point (Static CP), the number of choice points created at runtime and removed by a fail instruction (Dynamic CP), the memory needed to save local environments (Memory usage) and the number of applied rewrite rules per second (rwr/sec) on a Dec Alpha Station.

- p5 and p8 correspond to the Knuth–Bendix completion of modified versions of the Group theory, with five (resp. eight) identity elements and five (resp. eight) inverse elements, together with the corresponding axioms. These theories are often benchmarks for theorem provers. The execution of p5 gives the following results:

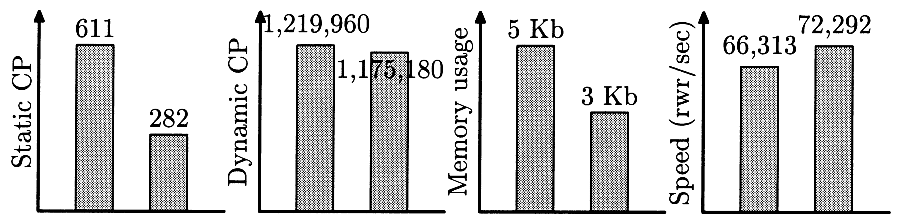


The execution of p8 involves more complex computations:

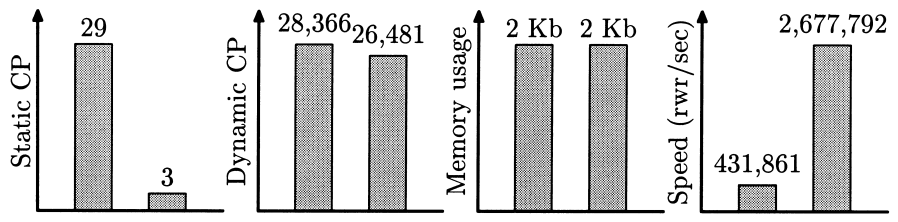


Note that without optimisation the completion program needs an amount of memory approximatively proportional to the number of choice points (Dynamic CP). When the determinism analysis is activated, this number becomes constant: 1 Kb.

- minela is a small ELAN interpreter written in ELAN itself; it executes an ELAN program composed of pure conditional rules on an input term, and outputs the result together with a proof term that represents the derivation from the input term.

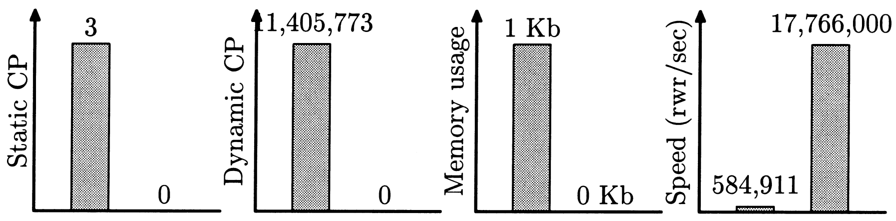


- queens is an implementation of the N-queens problem that searches for a solution with a chessboard of size $n = 14$. This is also a typical benchmark problem in logic programming.



This example shows a real speed improvement, even if the numbers of dynamic choice points seem to be approximatively equivalent with and without determinism analysis. In fact, without optimisation, much more choice points were set, but removed by a cut operator, which is not taken into account by the current statistics. When applying the determinism analysis, less choice points are created and all the cut operators are replaced by a goto statement. This explains why the speed is improved.

- fib is a functional program that computes the 33th Fibonacci number (using builtin integers). Again, this is a typical benchmark problem in functional programming.



When applying the determinism analysis, the generated program does not contain any setChoicePoint instruction and the control flow no longer involves any backtracking. This explains why the memory needed to save environments is reduced to 0 Kb and why the execution speed is improved in such a way: more than 15,000,000 applied rewrite rules per second.

These results show clearly that the determinism analysis significantly decreases the number of set choice points and improves the overall performance. It also considerably decreases the amount of memory needed to save local environments. Let us consider the completion process for instance. Without any optimisation, the memory needed depends on the input term to reduce (p5 or p8). When applying the determinism analysis, the memory needed is most often reduced to a constant independent from the query. This constant corresponds to the number of choice points that are simultaneously set during the computation. It happened that programs running out of memory without determinism analysis, eventually gave answers, once this improvement was activated.

7 Conclusion

In this paper we have presented the main techniques used in our ELAN compiler, which we hope will be useful for other rewriting-based languages. From the point of view of matching and rewriting, ELAN can be compared to other systems such as OBJ (Goguen & Winkler, 1988), ASF+SDF (Klint, 1993), Maude (Clavel *et al.*, 1996) or Cafe-OBJ (Futatsugi & Nakagawa, 1997). Maude also provides efficient AC rewriting and ASF+SDF performs list matching, a specific instance of AC matching. However, these languages do not involve non-deterministic strategy constructors. With respect to the determinism analysis, ELAN is closer to logic programming languages such as Alma-0 (Apt & Schaerf, 1997) or Mercury (Henderson *et al.*, 1996b). In our case, the determinism analysis simply makes possible to run programs that could not be executed before due to memory explosion. This analysis significantly decreased the number of set choice points and improved the performance.

It seems now that further improvements of the ELAN compiler rely on the backtracking management. The setChoicePoint and fail functions implemented in assembly language turned out to be very useful for designing complex compilation schemas. A deeper analysis reveals that useless information is also stored in local environments. So it should be possible to improve the low-level management of non-determinism and to combine this with an efficient garbage collector. Together with this re-design of the memory management, we think of using a shared terms library (van den Brand *et al.*, 1999; Van den Brand *et al.*, 2000), as in ASF+SDF, or using a generational garbage collection approach as is Haskell (Sansom & Peyton Jones, 1993).

Significant examples have been handled in ELAN that took benefit from the compilation methods developed in this paper. Let us mention three of them:

- Techniques used in solving constraint satisfaction problems are based on exploration of the space of all solutions with backtracking, and problem reduction techniques that reduce the set of values that the variables can take. As explained in Castro (1998), such techniques are expressible by rules and strategies and the system Colette implements them in ELAN.
- In the specification of authentication protocols (Cirstea, 1999), the protocol, the intruder and the attack are modeled by rules and strategies. ELAN behaves like a model checker by generating all possible situations and looking for situations revealing an attack.
- Planning and scheduling problems have been explored by Dubois & Kirchner (1998, 1999). In this case, ELAN is used as a decision support tool, which can simulate plan executions and explore consequences of decision-making during a planification.

These three application areas have in common to involve set data structures, to develop huge search spaces, and to need non-deterministic rules and strategies. For running such examples, the use of the ELAN compiler is essential. But the achievement of these significant programs comforts the affirmation that the rewriting paradigm can be promoted to the level of a realistic programming language.

We feel that the techniques presented in this paper could benefit the functional programming community in at least two directions. The first is to add built-in equational theories in higher-order matching and rewriting. This was already explored by several authors (Wadler, 1987; Jouannaud & Okada, 1991; Nipkow & Prehofer, 1998). The second direction is to increase higher-order features of rewriting based languages. A promising approach to bring closer rewriting-based languages and functional languages is the rewriting calculus (ρ-calculus) proposed by Cirstea & Kirchner (1999). This is a framework in which rule, rule application, and sets of results are explicit objects. The main intuition is that a rewrite rule is an abstractor generalising λ-abstraction: the left-hand side of a rule determines the bound variables and the contextual structure. The calculus handles non-determinism via sets of results. ELAN is actually an implementation of a large part of this calculus. To come closer to a functional language, the syntax should be extended by λ-expressions. The study of compilation techniques for such an extension, inspired from those used in the two classes of languages, is certainly a challenging research and development issue.

Acknowledgements

We sincerely thank M. Hermann and C. Ringeissen for their comments on the preliminary version of this paper and the anonymous referees for their valuable remarks and suggestions.

References

Aït-Kaci, H. (1990) *The WAM: a (real) tutorial*. Technical report 5. Digital Systems Research Center, Paris (France).

Apt, K. R. & Schaerf, A. (1997) Search and Imperative Programming. *24th POPL*, pp. 67–79.

Armstrong, J. (1997) The development of Erlang. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pp. 196–203.

Armstrong, J., Virding, R., Wikström, C. & M., Williams. (1996) *Concurrent programming in Erlang*. Prentice-Hall International.

Baader, F. & Nipkow, T. (1998) *Term rewriting and all that*. Cambridge University Press.

Bachmair, L., Chen, T. & Ramakrishnan, I. V. (1993) Associative-commutative discrimination nets. In: Gaudel, M.-C. & Jouannaud, J.-P., editors, *TAPSOFT'93: Theory and practice of software development, 4th international joint conference CAAP/FASE: Lecture Notes in Computer Science 668*, pp. 61–74. Springer-Verlag.

Benanav, D., Kapur, D. & Narendran, P. (1987) Complexity of matching problems. *J. Symbolic Computation*, **3**(1 & 2), 203–216.

Bird, R. & Wadler, P. (1988) *Introduction to functional programming*. Prentice-Hall International. (Japanese translation, 1991. Dutch translation, 1991. German translation, 1992.)

Borovanský, P., Kirchner, C. & Kirchner, H. (1996) Controlling rewriting by rewriting. In: Meseguer, J., editor, *Proceedings 1st International Workshop on Rewriting Logic. Electronic Notes in Theoretical Computer Science*, **4**.

Borovanský, P., Kirchner, C. & Kirchner, H. (1998a) A functional view of rewriting and strategies for a semantics of ELAN. In: Sato, M. & Toyama, Y., editors, *The 3rd Fuji International Symposium on Functional and Logic Programming*, pp. 143–167. World Scientific.

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E. & Ringeissen, C. (1998b) An overview of ELAN. In: Kirchner, C. & Kirchner, H., editors, *Proceedings 2nd International Workshop on Rewriting Logic and Applications*, **15**, pp. 143–167. Pont-à-Mousson (France). (`http://www.elsevier.nl/locate/entcs/volume15.html`)

Bouhoula, A. & Rusinowitch, M. (1995) Implicit induction in conditional theories. *J. Automated Reasoning*, **14**(2), 189–235.

Brus, T. H., van Eekelen, M. C. J. D., van Leer, M. O., Plasmeijer, M. J. & Barendregt, H. P. (1987) CLEAN – A language for functional graph rewriting. In: Kahn, editor, *Proceedings Conference on Functional Programming Languages and Computer Architecture (FPCA'87): Lecture Notes in Computer Science 274*, pp. 364–384. Springer-Verlag.

Caseau, Y. & Laburthe, F. (1996) *Introduction to the CLAIRE programming language*. Technical report 96-15. LIENS Technical.

Castro, C. (1998) Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, **34**(3), 263–293.

Christian, J. (1993) Flatterms, discrimination nets, and fast term rewriting. *J. Automated Reasoning*, **10**(1), 95–113.

Cirstea, H. (1999) Specifying authentication protocols using ELAN. *Workshop on Modelling and Verification.*

Cirstea, H. & Kirchner, C. (1997) Theorem proving using computational systems: The case of the B predicate prover. *Workshop CCL'97.*

Cirstea, H. & Kirchner, C. (1999) Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In: Gabbay, D. & de Rijke, M., editors, *Frontiers of Combining Systems 2*, pp. 95–120. Wiley.

Clavel, M., Eker, S., Lincoln, P. & Meseguer, J. (1996) Principles of Maude. In: Meseguer, J., editor, *Proceedings 1st International Workshop on Rewriting Logic*, **4**.

Contejean, E., Marché, C. & Rabehasaina, L. (1997) Rewrite systems for natural, integral, and rational arithmetic. In: Comon, H., editor, *Proceedings 8th International Conference on Rewriting Techniques and Applications: Lecture Notes in Computer Science*, pp. 98–112. Springer-Verlag.

Cousineau, G. & Mauny, M. (1998) *The Functional Approach to Programming*. Cambridge University Press.

Cousineau, G., Paulson, L. C., Huet, G., Milner, R., Gordon, M. & Wadsworth, C. (1985) *The ML handbook*. Rocquencourt: INRIA.

Dershowitz, N. & Jouannaud, J.-P. (1990) *Handbook of Theoretical Computer Science*, Vol. B. Elsevier.

Didrich, K., Fett, A., Gerke, C., Grieskamp, W. & Pepper, P. (1994) OPAL: Design and implementation of an algebraic programming language. In: Gutknecht, J., editor, *Programming Languages and System Architectures (PLSA'94): Lecture Notes in Computer Science 782*, pp. 228–244. Springer-Verlag.

Dubois, H. & Kirchner, H. (1998) Actions and plans in ELAN. *Proceedings Workshop on Strategies in Automated Deduction – CADE-15*, pp. 35–45. Lindau, Germany.

Dubois, H. & Kirchner, H. (1999) *Rule based programming with constraints and strategies.* Techical report 99-R-084. LORIA, Nancy, France.

Eker, S. (1995) Associative-commutative matching via bipartite graph matching. *Computer J.*, **38**(5), 381–399.

Fukuda, K. & Matsui, T. (1989) *Finding all the perfect matchings in bipartite graphs.* Technical Report B-225, Department of Information Sciences, Tokyo Institute of Technology, Japan.

Futatsugi, K. & Nakagawa, A. (1997) An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. *Proceedings 1st IEEE Int. Conference on Formal Engineering Methods.*

Goguen, J. A. & Winkler, T. (1988) *Introducing OBJ3*. Technical Report SRI-CSL-88-9, SRI International, CA, USA.

Gordon, M., Milner, A. & Wadsworth, C. (1979) *Edinburgh LCF: A mechanized logic of computation: Lecture Notes in Computer Science 78*. Springer-Verlag.

Gräf, A. (1991) Left-to-right tree pattern matching. In: Book, R. V., editor, *Proceedings 4th Conference on Rewriting Techniques and Applications: Lecture Notes in Computer Science 488*, pp. 323–334, Springer-Verlag.

Graf, P. (1996) *Term Indexing: Lecture Notes in Artificial Intelligence 1053*. Springer-Verlag.

Henderson, F., Somogyi, Z. & Conway, T. (1996a) Determinism analysis in the Mercury compiler. *Proceedings of the 19th Australian Computer Science Conference*, pp. 337–346.

Henderson, F., Conway, T. & Somogyi, Z. (1996b) The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Logic Programming*, **29**(October–December), 17–54.

Hermann, M. & Kolaitis, P. G. (1995) Computational complexity of simultaneous elementary AC-matching problems. In: Wiedermann, J. & Hájek, P., editors, *Proceedings 20th International Symposium on Mathematical Foundations of Computer Science: Lecture Notes in Computer Science 969*, pp. 359–370. Springer-Verlag.

Hopcroft, J. E. & Karp, R. M. (1973) An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, **2**(4), 225–231.

Hullot, J.-M. (1980) *Compilation de formes canoniques dans les théories équationelles.* Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France).

Jones, M. P. (1994) *The implementation of the Gofer functional programming system.* Technical Report Report YALEU/DCS/RR-1030, Yale University, New Haven, CT, USA.

Jones, M. P. (1996) *Hugs 1.3, The Haskell User's Gofer System: User Manual.* Technical Report NOTTCS-TR-96-2, Department of Computer Science, University of Nottingham, UK.

Jouannaud, J.-P. & Kirchner, H. (1986) Completion of a set of rules modulo a set of equations. *SIAM J. Computing*, **15**(4), 1155–1194. (Preliminary version in *Proceedings 11th ACM Symposium on Principles of Programming Languages*, Salt Lake City, UT, 1984.)

Jouannaud, J.-P. & Okada, M. (1991) Executable higher-order algebraic specification languages. *Proceedings 6th IEEE Symposium on Logic in Computer Science*, Amsterdam, The Netherlands.

Kapur, D. & Zhang, H. (1988) RRL: A rewrite rule laboratory. *Proceedings 9th International Conference on Automated Deduction: Lecture Notes in Computer Science 310*, pp. 768–769. Argonne IL. Springer-Verlag.

Kirchner, C. & Ringeissen, Ch. (1998) Rule-Based Constraint Programming. *Fundamenta Informaticae*, **34**(3), 225–262.

Kirchner, C., Kirchner, H. & Vittek, M. (1995) Designing constraint logic programming languages using computational systems. In: Van Hentenryck, P. & Saraswat, V., editors, *Principles and Practice of Constraint Programming. The Newport Papers*, p. 131–158. MIT Press.

Kirchner, H. & Moreau, P.-E. (1995) Prototyping completion with constraints using computational systems. In: Hsiang, J., editor, *Proceedings 6th Conference on Rewriting Techniques and Applications: Lecture Notes in Computer Science 914*, pp. 438–443. Kaiserslautern, Germany. Springer-Verlag.

Kirchner, H. & Moreau, P.-E. (1998) Non-deterministic computations in ELAN. In: Fiadeiro, J. L., editor, *Recent Developements in Algebraic Specification Techniques, Proceedings of the 13th WADT'98: Lecture Notes in Computer Science 1548*, pp. 168–182. Springer-Verlag.

Klint, P. (1993) A meta-environment for generating programming environments. *ACM Trans. Software Eng. & Methodology*, **2**, 176–201.

Kounalis, E. & Lugiez, D. (1991) Compilation of pattern matching with associative commutative functions. *16th Colloquium on Trees in Algebra and Programming: Lecture Notes in Computer Science 493*, pp. 57–73. Springer-Verlag.

Leroy, X. & Mauny, M. (1993) Dynamics in ML. *J. Functional Programming*, **3**(4), 431–463.

Lugiez, D. & Moysset, J.-L. (1994) Tree automata help one to solve equational formulae in AC-theories. *J. Symbolic Computation*, **18**(4), 297–318.

Marché, C. (1996) Normalized rewriting: an alternative to rewriting modulo a set of equations. *J. Symbolic Computation*, **21**(3), 253–288.

McCune, W. (1992) Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *J. Automated Reasoning*, **9**(2), 147–167.

Meseguer, J. (1992) Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, **96**(1), 73–155.

Moreau, P.-E. (1998) *A choice-point library for backtrack programming*. JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic.

Moreau, P.-E. (1999). *Compilation de règles de réécritures et de stratégies non-déterministes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France.

Moreau, P.-E. & Kirchner, H. (1998) A compiler for rewrite programs in associative-commutative theories. In: *'Principles of Declarative Programming': Lecture Notes in Computer Science 1490*, pp. 230–249. Springer-Verlag.

Nedjah, N., Walter, C. D. & Eldrige, E. (1997) Optimal left-to-right pattern-matching automata. In: Hanus, M., Heering, J. & Meinke, K., editors, *Proceedings 6th International Conference on Algebraic and Logic Programming: Lecture Notes in Computer Science 1298*, pp. 273–286. Springer-Verlag.

Nipkow, T. & Prehofer, C. (1998) Higher-order rewriting and equational reasoning. In: Bibel, W. & Schmitt, P., editors, *Automated Deduction – A Basis for Applications. Volume I: Foundations*. Kluwer.

Partington, V. (1997) *Implementation of an Imperative Programming Language with Backtracking*. Technical Report P9714, University of Amsterdam, Programming Research Group. (Available by anonymous ftp from ftp.wins.uva.nl, file pub/programming-research/reports/1997/P9712.ps.Z.)

Paulson, L. C. (1994) *Isabelle: A generic theorem prover: Lecture Notes in Computer Science 828*. Springer-Verlag.

Peterson, G. & Stickel, M. E. (1981) Complete sets of reductions for some equational theories. *J. ACM*, **28**, 233–264.

Peyton Jones, S. (1996) Compiling Haskell by program transformation: a report from the trenches. *Proceedings of the European Symposium on Programming (ESOP'96): Lecture Notes in Computer Science 1058*. Springer-Verlag.

Plasmeijer, M. J. & van Eekelen, M. C. J. D. (1993) *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley.

Plotkin, G. (1977) LCF considered as a programming language. *Theor. Comput. Sci.*, **5**, 223–255.

Ringeissen, C. (1997) Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. *Proceedings 8th Conference on Rewriting Techniques and Applications: Lecture Notes in Computer Science 1232*, pp. 323–326. Springer-Verlag.

Sahlin, D. (1991) Determinacy analysis for full prolog. *Proceeding of the ACM/IFIP Symposium on Partial Evaluation and Semantics based Program Manipulation*. ACM Press.

Sansom, P. & Peyton Jones, S. (1993) Generational garbage collection for Haskell. *Proceedings of Functional Programming Languages and Computer Architecture (FPCA'93)*.

Sawamura, H. & Takeshima, T. (1985) Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization. *Proceedings 2nd International Logic Programming Conference*, pp. 200–207.

Van den Brand, M., de Jong, H. A., Klint, P. & Olivier, P. A. (2000) Efficient annotated terms. *Software—Practice & Experience*, **30**, 259–291.

van den Brand, M. G. J., Klint, P. & Olivier, P. (1999) Compilation and Memory Management for ASF+SDF. *Compiler Construction: Lecture Notes in Computer Science 1575*, pp. 198–213. Springer-Verlag.

Vittek, M. (1996) A compiler for nondeterministic term rewriting systems. In: Ganzinger, H., editor, *Proceedings of RTA'96: Lecture Notes in Computer Science 1103*, pp. 154–168. Springer-Verlag.

Voronkov, A. (1995) The Anatomy of Vampire: Implementing Bottom-up Procedures with Code Trees. *J. Automated Reasoning*, **15**, 237–265.

Wadler, P. (1987) Views: a way for pattern matching to cohabit with data abstraction. *14'th ACM Symposium on Principles of Programming Languages.*

Warren, D. H. D. (1983) *An abstract Prolog instruction set*. Technical Report 309, SRI International.

Weis, P. & Leroy, X. (1993) *Le langage Caml*. InterEditions.