Non-deterministic Computations in ELAN

Hélène Kirchner and Pierre-Etienne Moreau

LORIA-CNRS & INRIA
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France

Abstract. The ELAN system is an environment for specifying and prototyping constraint solvers, theorem provers and deduction systems in general. It also provides a framework for experimenting their combination. The ELAN language is based on rewriting logic and evaluation of labelled conditional rewrite rules. ELAN has two originalities with respect to several other algebraic languages, namely to handle non-deterministic computations and to provide a user-defined strategy language for controlling rule application. We focus in this paper on these two related aspects and explain how non-determinism is used in ELAN programs and handled in the ELAN compiler.

1 Introduction

The ELAN system [KKV95] provides an environment for specifying and prototyping deduction systems in a language based on rules controlled by strategies. Its purpose is to support the design of theorem provers, logic programming languages, constraints solvers and decision procedures and to offer a 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 conditions and may introduce local variables. But rewriting is inherently non-deterministic since several rules can be applied simultaneously on a same term. So in ELAN, a computation may have several results. This aspect is taken into account through choice operations and a backtracking capability. One of the main originality of the language is to provide strategy constructors to specify whether a function call returns several, at-least one or only one result. Non-determinism is handled with two operators: dc standing for dont-care-choose and dk standing for dont-know-choose. Determinism is enforced by the operator dc one standing for dont-care-choose one result. This declarative handling of non-determinism is part of a strategy language allowing the programmer to specify the control on rules application. 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, more complex strategies can be expressed. In addition the user

© Springer-Verlag Berlin Heidelberg 1999

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 [Mes92] and detailed in [BKK96,BKK98]. 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 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 [Cas98], theorem proving tools in first-order logic with equality [KM95,CK97], the combination of unification algorithms and of decision procedures in various equational theories [Rin97,KR98]. More information on the system can be found on the WEB site[1].

A first ELAN compiler was designed and presented in [Vit96]. Experimentations made clear that a higher-level of programming is achieved when some functions may be declared as associative and commutative (AC for short). However rewriting in such theories is computationally difficult and the challenge is then to provide an efficient compiler for the language. The difficulty is both at the level of AC-matching and rewriting, addressed in [MK98], and at the level of non-deterministic computations, addressed in this paper. After a short presentation in Section 2 of ELAN programs and of the evaluation mechanism in ELAN, we explain in Section 3 the expressive power of non-deterministic features of the language and the related constructions of the strategy language. Then in Section 4, we detail how an analysis of non-determinism is performed and exploited in the ELAN compiler. We conclude in Section 5. We assume the reader familiar with basic definitions of term rewriting given for instance in [JK86].

2 Programs and evaluation mechanism of ELAN

An ELAN program is composed of a signature describing operators with their types, a list of rules and a list of strategies. A strategy provides a way to describe which computations the user is interested in, and specifies where a given rule should be applied in the term to be reduced. We describe informally here the evaluation mechanism and how it deals with rewrite rules and strategies.

In ELAN, rules are labelled rewrite rules with an optional sequence of conditions and/or local variable assignments:

$$[\mathsf{lab}] \quad l \Rightarrow r \quad \{\ \textbf{if}\ v \quad | \quad \textbf{where}\ y := (S)u\ \}^*$$

where lab is the label, l and r are terms respectively called left and right-hand sides, v is a boolean term called condition, and $y := (S)u$ is a local assignment, giving to the local variable y the results of the strategy S applied to the term u. Any sequence of **where** and **if** is allowed and their order is relevant for the evaluation. For applying such a rule on a term t at top position, first l is matched

[1] http://www.loria.fr/ELAN.

against t, the expressions introduced by **where** and **if** are instantiated with the matching substitution and evaluated in order. Instantiations of local variables (such as y) after **where** extend the matching substitution. When every condition is satisfied, the replacement by the instantiated right-hand side is performed.

The local assignment mechanism has been extended from variables to patterns. Rules of the following form are allowed too:

$$[\mathsf{lab}] \quad l \Rightarrow r \quad \{\ \mathbf{if}\ v \quad | \quad \mathbf{where}\ p := (S)u\ \}^*$$

where p is a term with variables. If p matches the result of the strategy S applied to the term u, its variables are instantiated and extend the main matching substitution, as before.

Unlabelled rules are applied with a leftmost-innermost strategy which is the default strategy built in the system. On the contrary, labelled rules are used in user-defined strategies and always applied at top position in the term to be reduced. Note that instantiations of local variables after **where** also invoke ELAN strategies.

A strategy is a function which, when applied to an initial term, returns a set of possible results. The strategy fails if the set is empty. A strategy is applied to a term, thanks to an application operator with the following type:

$$[\_]\_ : Strategy \times Term \to SetOfTerms$$

whose interpretation is given by labelled rewrite rules. The strategy interpreter is fully specified in ELAN. The strategy language and its semantics are described in [BKK98]. Let us explain now how to express a strategy, analyse when it fails and how is built its set of results.

– A labelled rule is a primal strategy. The result of applying a rule labelled lab on a term t returns a set of terms. This primal strategy fails if the set 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.
– $\mathsf{dc}(S_1, \ldots, S_n)$ chooses one strategy S_i in the list that does not fail, i.e. whose application gives a non-empty set of results, and returns all its results. This strategy may return more than one result, or fails if all sub-strategies S_i fail.
– $\mathsf{dc\ one}(S_1, \ldots, S_n)$ 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.
– $\mathsf{dk}(S_1, \ldots, S_n)$ chooses all strategies given in the list of arguments and for each of them returns all its results. This set 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.

- repeat*(S) applies repeatedly the strategy S until it fails and returns the results of the last unfailing application. This strategy can never fail (zero application of S is possible) and may return more than one result.
- The strategy iterate*(S) is similar to repeat*(S) but returns intermediate results of repeated applications.

Example 1. The following small program illustrates the syntax of ELAN. It designs a function `enum` that takes two integer arguments `i` and `j` and enumerates all numbers between them.

```
module example
 import global int ; end
 operators global
   next(@): (int) int ;
   enum(@,@): (int int) int ;
 end
 stratop global
   enumStrat : <int> ;
 end
 rules for int
   i,j,k,l: int ;
   global
   [] next(i) => i+1 end
   [r1] enum(i,j) => i end
   [r2] enum(i,j) => l
                  if i<j
                  where k:= () next(i)
                  where l:= (enumStrat) enum(k,j)
                  end
 end
 strategies for int
   [] enumStrat => dk(r1,r2) end
 end
```

This program is for instance executed via a request `enum(20,40)` and then counts from 20 to 40.

ELAN also gives the possibility to the user to define recursive and parameterised strategies with rewrite rules [BKK96,BKK98]. Given a set of strategy symbols, rewrite rules on user-defined strategies are of the form [lab] $S_1 \Rightarrow S_2$ where lab is a label, S_1 and S_2 are strategy terms built from all previously introduced strategy symbols. Such strategy rewrite rules are called *implicit* since they do not involve explicitly the application operator $[\_]\_$. However, it is sometimes useful to express a strategy rule depending on the argument on which it is applied. Such rules are also allowed and are called *explicit* strategy rules. They are of the form: [lab] $[S_1]\ t \Rightarrow t'$, where S_1 is a strategy term as before, t is a term and t' is built on function and strategy symbols and possibly the strategy application operator. From the evaluation point of view, these rules are just added to the strategy interpreter. An example of recursive strategies is given in Example 2 of Section 4.3.

3 Non-determinism in ELAN

Let us first explain on a simple example the expressive power of the non-deterministic features provided in ELAN. Consider the modelisation of a game: a pawn on a chessboard can move in several directions (see figure 1), 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 symbol `move` and a rule `move` $\Rightarrow$ `dk(`$d_1, \ldots, d_n$`)`. 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)`. In order to iterate this process, ELAN provides the constructor `repeat*` and the strategy `repeat*(dc one(move))` repeatedly moves a given pawn up to a failure: in this example, a pawn cannot move when an external square is reached.

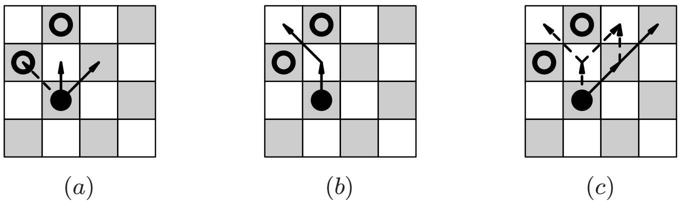

Fig. 1. Move a pawn on a chessboard. (a) – By applying `dk(move)` the pawn can move in three possible directions. (b) – An external square can be reached if the strategy `repeat*(dc one(move))` is applied. (c) – Suppose that a final position should not be near a white pawn: backtracking is needed when applying `repeat*(dk(move))`; the explored search tree is illustrated by dashed arrows.

Assume now that one looks for moves that lead the pawn in a specific final position characterised by some predicate `P`. In this situation, when a move has been performed, it is not a definitive choice since the final position is not yet known. To be able to explore the search space, one can then replace `repeat*(dc one(move))` by `repeat*(dk(move))` and check if the predicate `P` is satisfied by the last found position. This can be expressed in ELAN by the following rule, where the current state of the game is represented by a two component structure `<position,chessboard>` whose first argument in the pawn's position and the second is the chessboard.

```
[win] <initPos,chessboard> => <finalPos,chessboard'>
      where <finalPos,chessboard'>:=
                (repeat*(dk(move)))  <initPos,chessboard>
      if P(finalPos)
```

This sketch of a simple game can be instantiated by several other situations in the areas of constraint solving, rewriting and theorem proving. Let us take two more examples.

To simulate exploration of a derivation tree for instance, instead of defining move, one can define the strategy rewrite $\Rightarrow$ dk($r_1, \ldots, r_n$) that non-deterministically applies the n rules $r_1, \ldots, r_n$. The strategy dc one(rewrite) consists in a deterministic rewrite step by one of the rule that applies on top of the term to be reduced. The strategy repeat*(dc one(rewrite)) leads to a normal form, if the rewrite relation induced by the n rules terminates, while repeat*(dk(rewrite)) computes all normal forms. It can be specified that normal forms satisfy a given property, such as a size less than 20, for instance.

Another example is constraint solving which is now often formalised as a rewriting process that transforms a constraint into a set of solved forms providing the solutions. So computing one solution is reducing to a normal form, and finding all solutions is computing all normal forms. More specifically, for constraints on finite domains, constraint satisfaction techniques involve "generate and test" strategies which enumerate the finite set of possible values in the domain of each variable. More sophisticated strategies are developed for constraint satisfaction using the ELAN language in [Cas98].

The main difficulty when dealing with such real applications is first an efficiency problem. Thanks to new compilation techniques for matching and rewriting, as presented for instance in [MK98], rewrite rule systems can now lead to very efficient first-order functional programs, whose evaluation involves several millions of rewrite steps per second. The bottleneck is rather at the level of memory and backtracking management, related to handling non-determinism.

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 [CL96], try and retry in WAM [War83,AK90], onfail, fail, createlog and replaylog in the Alma-0 Abstract Machine [Par97,AS97]. Following [Vit96], two flow control functions, setChoicePoint and fail, have been implemented in assembly language. setChoicePoint 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 [Mor98a,Mor98b]. Such functions may be useful in other contexts, for instance to implement backtracking in an imperative language, along the lines of Alma-0 [AS97].

Let us consider again the game example and suppose that several successive moves are performed by the `repeat*(dc one(move))` strategy. In principle, when repeatedly applying a strategy S, the environment has to be saved between each application of the strategy, in order to be able to explore another branch in case of backtracking. In some situations, the strategy S can be known to return at most one result as it is the case in our example for `dc one(move)`. One can then deduce that no backtracking is possible when applying such strategy and, consequently that it is no longer needed to save the environment before applying S. This informal reasoning is formalised in the next section and is made fully automatic in the ELAN environment.

4 Determinism analysis

The determinism analysis phase of the ELAN compiler annotates every rule and strategy in the program with its determinism mode for use in later phases of the compiler: matching phase, various optimisations on the generated code and detection of non termination.

4.1 Determinism modes

For each strategy, a mode 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 modes as in Mercury [HCS96,HSC96]:
- if the strategy cannot fail and has at most one result its mode is *deterministic* (det).
- if the strategy can fail and has at most one result, its mode is *semi-deterministic* (semi).
- if the strategy cannot fail and has more than one results, its mode is *multi-result* (multi).
- if the strategy can fail and may have more than one results, its mode is *non-deterministic* (nondet).
- if the strategy always fail, i.e. has no result, its mode is *failure* (fail).
A partial ordering on these modes is defined as follows:

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

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

The algorithm for inferring the determinism mode 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 modes. 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 ($\mathit{And}(\mathsf{det}, \mathsf{semi}) = \mathit{And}(\mathsf{semi}, \mathsf{det}) = \mathit{And}(\mathsf{semi}, \mathsf{semi}) = \mathsf{semi}$). But they can be also computed with operations on boolean variables as for instance in [HSC96].

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

4.2 dc one, dc, dk constructors

The non-determinism of ELAN is handled by three operators: dc one, dc and dk. These operators are heavily overloaded: they can be applied on a single rewrite rule or a single strategy, on a list of rules or a list of strategies, or a list of both strategies and rules. To describe all those combinations, we introduce four primitive operators, not accessible to the programmer, 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 all operator returns all unfailing strategies.

If the list contains only failing strategies, the current operation fails.
Using these four primitives, the three ELAN strategy constructors dc one, dc, dk can be defined by the following axioms, where S_i stands for a rule or a strategy:

$$\boxed{\begin{array}{lcl}
\mathsf{dc\ one}(S_1, \ldots, S_n) & = & \mathsf{select\ one}(\mathsf{one}(S_1), \ldots, \mathsf{one}(S_n)) \\
\mathsf{dc}(S_1, \ldots, S_n) & = & \mathsf{select\ one}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n)) \\
\mathsf{dk}(S_1, \ldots, S_n) & = & \mathsf{select\ all}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n))
\end{array}}$$

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

4.3 Determinism mode inference

The algorithm for inferring the determinism mode is presented here in three steps: for a strategy, it uses the decomposed form of ELAN strategy operators into their primitives representation. For a rule, it analyses the modes of the conditions and local assignments. Finally it deals with the recursivity problem due to the fact that strategies are built from rules and that rules call strategies in their local assignments.

Strategy d-mode inference – The d-mode of a strategy is inferred from its expression using one, all, select one and select all.

- $\mathsf{d\text{-}mode}(\mathsf{one}(S)) = \mathsf{semi}$ if S is a rewrite rule, since application of a rewrite rule may fail; otherwise,
  $$\mathsf{d\text{-}mode}(\mathsf{one}(S)) = \begin{cases} \mathsf{det} & \text{if } \mathsf{d\text{-}mode}(\mathrm{S}) \text{ is } \mathsf{det} \text{ or } \mathsf{multi} \\ \mathsf{semi} & \text{if } \mathsf{d\text{-}mode}(\mathrm{S}) \text{ is } \mathsf{semi} \text{ or } \mathsf{nondet} \end{cases}$$
- $$\mathsf{d\text{-}mode}(\mathsf{all}(S)) = \begin{cases} And(\mathsf{semi}, \mathsf{d\text{-}mode}(S)) & \text{if S is a rewrite rule} \\ \mathsf{d\text{-}mode}(S) & \text{otherwise} \end{cases}$$
- $$\mathsf{d\text{-}mode}(\mathsf{repeat*}(\mathrm{S})) = \begin{cases} \mathsf{det} & \text{if } \mathsf{d\text{-}mode}(\mathrm{S}) \text{ is } \mathsf{det} \text{ or } \mathsf{semi} \\ \mathsf{multi} & \text{if } \mathsf{d\text{-}mode}(\mathrm{S}) \text{ is } \mathsf{multi} \text{ or } \mathsf{nondet} \end{cases}$$
  The repeat* operator cannot fail because zero application of the strategy is allowed. Note that if S cannot fail, the repeat* construction cannot terminate.
- $\mathsf{d\text{-}mode}(\mathsf{iterate*}(S)) = \mathsf{multi}$. 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.
- $\mathsf{d\text{-}mode}(S_1; S_2) = And(\mathsf{d\text{-}mode}(S_1), \mathsf{d\text{-}mode}(S_2))$.
- $\mathsf{d\text{-}mode}(\mathsf{select\ one}(S_1, \ldots, S_n)) = And(\mathsf{d\text{-}mode}(S_1), \ldots, \mathsf{d\text{-}mode}(S_n))$
- $\mathsf{d\text{-}mode}(\mathsf{select\ all}(S_1, \ldots, S_n)) = Or(\mathsf{d\text{-}mode}(S_1), \ldots, \mathsf{d\text{-}mode}(S_n))$

Rule d-mode inference – Inferring the mode of a rewrite rule R consists in analysing the determinism mode of its conditions and local assignments.

- Evaluating a *condition* consists in normalising (with unlabelled rules) a boolean term and comparing its normal form with the builtin boolean term *true*. So evaluating a condition can never fail. If the reduced term is not *true*, the current rule cannot be applied, but this does not modify the d-mode of the rule. This is why 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 the boolean term occurs in the left-hand side of the rule 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 condition is said to be multi.

– A *local assignment* has the determinism mode of the strategy used to compute its value. If no strategy occurs in the local assignment, it is said to be deterministic.

The mode of the rewrite rule R is the conjunction (*And* operation) of the inferred determinism modes of all conditions and local assignments.

Recursivity problem – In ELAN, strategy definitions may be (mutually) recursive. So the d-mode of a strategy may depend on itself. This is similar to the problem well-known in logic programming of finding the mode of a predicate [ST85]. To avoid non-termination of the determinism analysis algorithm, when the d-mode of a strategy depends on itself, a default mode is given. On the strategy constructors, this default corresponds to the maximum of the modes in the ordering $<$ that the strategy can have and is given in the next table.

constructor	one	all	repeat*	iterate*	;
default d-mode	semi	nondet	multi	multi	nondet

The default mode for the strategy selectors is computed as before from the default modes of the components.

Example 2. The following rule is used to implement the "N-horse problem" which consists in finding N consecutive moves for a horse on a chessboard, so that it does not visit the same square twice. The horse_strat searches all solutions for N consecutive moves.

```
rules for listOfPair
  [horse_n] horse(n) => position . listOfPositions
             if n>0
             where listOfPositions:=(horse_strat) horse(n-1)
             where position:=(move) head(listOfPositions)
             if not( position occurs in listOfPositions )
  end
strategies for listOfPair
  [] horse_strat     => dk(horse_n)      end
  [] det_horse_strat => dc_one(horse_n)  end
```

The horse_strat strategy is recursive, because it uses the rule horse_n in which a call to horse_strat is done. Its default d-mode is computed. Since dk(horse_n) = select all(all(horse_n)) = all(horse_n), the default d-mode is nondet. The second strategy det_horse_strat searches for only one solution of the horse problem.
Since dc one(horse_n) = one(horse_n), d-mode(det_horse_strat)=semi.

4.4 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 detect some non-terminating strategies.

Several optimisations have been done to improve the backtracking management:

– When compiling a set of rules whose local assignments are deterministic, no choice point is needed because no backtracking can occur between local assignments.
– When compiling a set of rules with non-deterministic local assignments, some choice points are needed to handle the backtracking. But these choice points can be removed when the rule is applied with a strategy that requires at most one result. For instance, when searching only one solution to the horse problem (see Example 2), several choice points are needed for the rule `horse_n`, but they can be deleted when this rule is applied with the strategy `dc one(horse_n)`.
– 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.

$$t \bullet \overset{\nearrow^{S}}{\underset{\searrow_{S}}{\longrightarrow^{S}}} t_1 \bullet \overset{\nearrow^{S}}{\underset{\searrow_{S}}{\longrightarrow^{S}}} \cdots \bullet \overset{\nearrow^{S}}{\underset{\searrow_{S}}{\longrightarrow^{S}}} t_n \bullet \overset{\nearrow^{S}}{\underset{\searrow_{S}}{\longrightarrow^{S}}} \mathsf{fail}$$

One choice point per step is needed, and when a failure occurs, one choice point only is deleted and the process goes on.
But this is no more the case when compiling a strategy repeat*(S) where S is det or semi. The compilation scheme then consists in 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:

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

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 [DFG$^+$94,Vit96]. 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 mode 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 found match is needed to apply a rewrite step. This remark leads to the design of an *eager AC matching* algorithm which optimizes the construction of the dynamic data structure used in [MK98] for matching the patterns to the subject subterms. 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 him to improve his strategy design.

Any ELAN program execution can benefit from the determinism analysis techniques described in this paper. However, in order 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.

– p5 and p8 correspond to the Knuth-Bendix completion of modified versions of the Group theory, with 5 (resp. 8) identity elements and 5 (resp. 8) inverse elements, together with the corresponding axioms. These theories are often benchmarks for theorem provers.
– 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.
– fib is a functional program that computes the 33^{th} Fibonacci number. Again, this is a typical benchmark problem in functional programming.

Figure 2 gives for each program the number of generated setChoicePoint instructions for creating a choice point (Static CP), the number of choice points created at runtime (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.

These results show clearly that the deterministic analysis significantly decreases the number of set choice points and improves the overall performance. But it also considerably decreases the amount of needed memory to save local environments. Let us consider the completion process for instance. Without any optimisation, the needed memory depends on the input term to reduce (p5 or p8). In practice, this memory area is fixed (1000 Kb for example) and for some other queries or programs, it is possible to get a “memory overflow” error message that stops the computation. When applying the deterministic analysis, the needed memory is usually reduced to a constant independent from the query. This constant corresponds to the number of choice points that are simultaneously

	Static CP	**Dynamic CP**	**Memory usage**	**rwr/sec**
p5 without DA	504	844,393	119 Kb	270,000
p5 with DA	63	528,976	**1 Kb**	751,389
p8 without DA	579	3,852,361	405 Kb	275,000
p8 with DA	66	2,464,491	**1 Kb**	844,034
minela without DA	611	1,219,960	5 Kb	66,313
minela with DA	282	1,175,180	3 Kb	72,292
queens without DA	29	28,366	2 Kb	431,861
queens with DA	3	26,481	2 Kb	**2,677,792**
fib without DA	3	11,405,773	1 Kb	584,911
fib with DA	0	**0**	0 Kb	**17,766,001**

Fig. 2. Impact of the deterministic analysis

set during the computation. It happened that programs running out of memory without deterministic analysis, eventually gave answers once this improvement was activated.

5 Conclusion

From the point of view of matching and rewriting, ELAN can be compared to other systems such as OBJ [GW88], ASF+SDF [Kli93], Maude [CELM96] or Cafe-OBJ [FN97]. But these languages do not involve non-deterministic strategy constructors. With respect to this feature, ELAN is closer to logic programming languages such as Alma-0 [AS97] or Mercury [HCS96]. In our case, the determinism analysis simply makes possible to run programs that could not be executed before due to memory explosion. This was the case in particular for constraint satisfaction problems. In other examples like completion processes, or the queens problem, this analysis significantly decreased the number of set choice points and improved the performance.

It seems now that further improvements of the 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. But a deeper analysis reveals that useless informations are 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, as in ASF+SDF, that would reduce the memory space needed for performing the computations.

Acknowledgements: We sincerely thank Peter Borovanský, Claude Kirchner, Christophe Ringeissen and Laurent Vigneron for helpful discussions, and the anonymous referees for their valuable comments.

References

AK90. H. Aït-Kaci. The WAM: a (real) tutorial. Technical report 5, Digital Systems Research Center, Paris (France), January 1990. 173

AS97. K. R. Apt and A. Schaerf. Search and imperative programming. In *24th POPL*, pages 67–79, 1997. 173, 174, 180

BKK96. P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. 169, 171

BKK98. P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific. Also report LORIA 98-R-165. 169, 170, 171

Cas98. C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998. 169, 173

CELM96. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science. 180

CK97. H. Cirstea and C. Kirchner. Theorem proving Using Computational Systems: The case of the B Predicate Prover. Available at `http://www.loria.fr/ ~cirstea/Papers/TheoremProver.ps`, 1997. 169

CL96. Y. Caseau and F. Laburthe. Introduction to the CLAIRE programming language. Technical report 96-15, LIENS Technical, September 1996. 173

DFG$^+$94. K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and implementation of an algebraic programming language. In J. Gutknecht, editor, *Programming Languages and System Architectures PLSA '94*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer-Verlag, March 1994. 178

FN97. K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proceedings of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997. 180

GW88. J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 333, Ravenswood Ave., Menlo Park, CA 94025, August 1988. 180

HCS96. F. Henderson, T. Conway, and Z. Somogyi. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–54, October-December 1996. 174, 180

HSC96. F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Nineteenth Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996. 174, 175

JK86. J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984. 169

KKV95. C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995. 168

Kli93. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993. 180

KM95. H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995. 169

KR98. C. Kirchner and C. Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998. 169

Mes92. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. 169

MK98. P.-E. Moreau and H. Kirchner. A compiler for rewrite programs in associative-commutative theories. In *"Principles of Declarative Programming"*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998. Report LORIA 98-R-226. 169, 173, 179

Mor98a. P.-E. Moreau. A choice-point library for backtrack programming. JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic, 1998. 173

Mor98b. P.-E. Moreau. Compiling nondeterministic computations. Technical Report 98-R-005, CRIN, 1998. 173

Par97. V. Partington. Implementation of an Imperative Programming Language with Backtracking. Technical Report P9714, University of Amsterdam, Programming Research Group, 1997. Available by anonymous ftp from ftp.wins.uva.nl, file pub/programming-research/reports/1997/P9712.ps.Z. 173

Rin97. C. Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997. 169

ST85. H. Sawamura and T. Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization. In *Proceedings of the Second International Logic Programming Conference*, pages 200–207, Boston, Massachusetts, 1985. 177

Vit96. M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag. 169, 173, 178

War83. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Artificial Intelligence Center, 1983. 173